HTML 5

A vocabulary and associated APIs for HTML and XHTML

W3C Working Draft 12 February 2009

This Version:
http://www.w3.org/TR/2009/WD-html5-20090212/
Latest Published Version:
http://www.w3.org/TR/html5/
Latest Editor's Draft:
http://www.w3.org/html/wg/html5/
Previous Versions:
http://www.w3.org/TR/2008/WD-html5-20080610/
http://www.w3.org/TR/2008/WD-html5-20080122/
Editors:
Ian Hickson, Google, Inc.
David Hyatt, Apple, Inc.

Copyright © 2009 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark and document use rules apply.

Abstract

This specification defines the 5th major revision of the core language of the World Wide Web: the Hypertext Markup Language (HTML). In this version, new features are introduced to help Web application authors, new elements are introduced based on research into prevailing authoring practices, and special attention has been given to defining clear conformance criteria for user agents in an effort to improve interoperability.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the most recently formally published revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

The WHATWG version of this specification is available under a license that permits reuse of the specification text.

If you wish to make comments regarding this document, please send them to public-html-comments@w3.org (subscribe, archives) or whatwg@whatwg.org (subscribe, archives), or submit them using our public bug database. All feedback is welcome.

We maintain a list of all e-mails that have not yet been considered and a list of all bug reports that have not yet been resolved.

Implementors should be aware that this specification is not stable. **Implementors who are not taking part in the discussions are likely to find the specification changing out from under them in incompatible ways.** Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation stage should join the aforementioned mailing lists and take part in the discussions.

The publication of this document by the W3C as a W3C Working Draft does not imply that all of the participants in the W3C HTML working group endorse the contents of the specification. Indeed, for any section of the specification, one can usually find many members of the working group or of the W3C as a whole who object strongly to the current text, the existence of the section at all, or the idea that the working group should even spend time discussing the concept of that section.

The latest stable version of the editor's draft of this specification is always available on the W3C CVS server and in the WHATWG Subversion repository. The latest editor's working copy (which may contain unfinished text in the process of being prepared) is also available.

There are various ways to follow the change history for the specification:

E-mail notifications of changes
: HTML-Diffs mailing list (diff-marked HTML versions for each change): http://lists.w3.org/Archives/Public/public-html-diffs/latest
: Commit-Watchers mailing list (complete source diffs): http://lists.whatwg.org/listinfo.cgi/commit-watchers-whatwg.org

Real-time notifications of changes:
: Generated diff-marked HTML versions for each change: http://twitter.com/HTML5
: All (non-editorial) changes to the spec source: http://twitter.com/WHATWG

Browsable version-control record of all changes:
: CVSWeb interface with side-by-side diffs: http://dev.w3.org/cvsweb/html5/spec/Overview.html
: Annotated summary with unified diffs: http://html5.org/tools/web-apps-tracker
: Raw Subversion interface: `svn checkout http://svn.whatwg.org/webapps/`

The W3C HTML Working Group is the W3C working group responsible for this specification's progress along the W3C Recommendation track. This specification is the 12 February 2009 Working Draft.

This specification is also being produced by the WHATWG. The two specifications are identical from the table of contents onwards.

This specification is intended to replace (be a new version of) what was previously the HTML4, XHTML 1.0, and DOM2 HTML specifications.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Stability

Different parts of this specification are at different levels of maturity.

Some of the more major known issues are marked like this. There are many other issues that have been raised as well; the issues given in this document are not the only known issues! Also, firing of events needs to be unified (right now some bubble, some don't, they all use different text to fire events, etc).

Table of contents

1 Introduction
- 1.1 Background
- 1.2 Audience
- 1.3 Scope
- 1.4 History
- 1.5 Relationships to other specifications
  - 1.5.1 Relationship to HTML 4.01 and DOM2 HTML
  - 1.5.2 Relationship to XHTML 1.x
  - 1.5.3 Relationship to XHTML2 and XForms
- 1.6 HTML vs XHTML
- 1.7 Structure of this specification
  - 1.7.1 How to read this specification
  - 1.7.2 Typographic conventions

2 Common infrastructure
- 2.1 Terminology
  - 2.1.1 XML
  - 2.1.2 DOM trees
  - 2.1.3 Scripting
  - 2.1.4 Plugins
  - 2.1.5 Character encodings
- 2.2 Conformance requirements
  - 2.2.1 Dependencies
  - 2.2.2 Features defined in other specifications
  - 2.2.3 Common conformance requirements for APIs exposed to JavaScript
- 2.3 Case-sensitivity and string comparison
- 2.4 Common microsyntaxes
  - 2.4.1 Common parser idioms
  - 2.4.2 Boolean attributes
  - 2.4.3 Numbers
    - 2.4.3.1 Non-negative integers
    - 2.4.3.2 Signed integers
    - 2.4.3.3 Real numbers
    - 2.4.3.4 Ratios
    - 2.4.3.5 Percentages and lengths
    - 2.4.3.6 Lists of integers
    - 2.4.3.7 Lists of dimensions
  - 2.4.4 Dates and times
    - 2.4.4.1 Months
    - 2.4.4.2 Dates
    - 2.4.4.3 Times
    - 2.4.4.4 Local dates and times
    - 2.4.4.5 Global dates and times
    - 2.4.4.6 Weeks
    - 2.4.4.7 Vaguer moments in time
  - 2.4.5 Colors

2.4.6 Space-separated tokens
2.4.7 Comma-separated tokens
2.4.8 Keywords and enumerated attributes
2.4.9 References
2.5 URLs
2.5.1 Terminology
2.5.2 Parsing URLs
2.5.3 Resolving URLs
2.5.4 Dynamic changes to base URLs
2.5.5 Interfaces for URL manipulation
2.6 Fetching resources
2.6.1 Protocol concepts
2.6.2 Encrypted HTTP and related security concerns
2.7 Determining the type of a resource
2.7.1 Content-Type metadata
2.7.2 Content-Type sniffing: Web pages
2.7.3 Content-Type sniffing: text or binary
2.7.4 Content-Type sniffing: unknown type
2.7.5 Content-Type sniffing: image
2.7.6 Content-Type sniffing: feed or HTML
2.8 Character encodings
2.9 Common DOM interfaces
2.9.1 Reflecting content attributes in DOM attributes
2.9.2 Collections
2.9.2.1 HTMLCollection
2.9.2.2 HTMLFormControlsCollection
2.9.2.3 HTMLOptionsCollection
2.9.3 DOMTokenList
2.9.4 Safe passing of structured data
2.9.5 DOMStringMap
2.9.6 DOM feature strings
2.9.7 Exceptions
2.9.8 Garbage collection

3 Semantics and structure of HTML documents
3.1 Introduction
3.2 Documents
3.2.1 Documents in the DOM
3.2.2 Security
3.2.3 Resource metadata management
3.2.4 DOM tree accessors
3.3 Elements
3.3.1 Semantics
3.3.2 Elements in the DOM
3.3.3 Global attributes
3.3.3.1 The `id` attribute
3.3.3.2 The `title` attribute
3.3.3.3 The `lang` and `xml:lang` attributes
3.3.3.4 The `xml:base` attribute (XML only)
3.3.3.5 The `dir` attribute
3.3.3.6 The `class` attribute
3.3.3.7 The `style` attribute

3.3.3.8 Embedding custom non-visible data
3.4 Content models
3.4.1 Kinds of content
3.4.1.1 Metadata content
3.4.1.2 Flow content
3.4.1.3 Sectioning content
3.4.1.4 Heading content
3.4.1.5 Phrasing content
3.4.1.6 Embedded content
3.4.1.7 Interactive content
3.4.2 Transparent content models
3.5 Paragraphs
3.6 APIs in HTML documents
3.7 Dynamic markup insertion
3.7.1 Controlling the input stream
3.7.2 `document.write()`
3.7.3 `document.writeln()`
3.7.4 `innerHTML`
3.7.5 `outerHTML`
3.7.6 `insertAdjacentHTML()`

4 The elements of HTML
4.1 The root element
4.1.1 The `html` element
4.2 Document metadata
4.2.1 The `head` element
4.2.2 The `title` element
4.2.3 The `base` element
4.2.4 The `link` element
4.2.5 The `meta` element
4.2.5.1 Standard metadata names
4.2.5.2 Other metadata names
4.2.5.3 Pragma directives
4.2.5.4 Other pragma directives
4.2.5.5 Specifying the document's character encoding
4.2.6 The `style` element
4.2.7 Styling
4.3 Scripting
4.3.1 The `script` element
4.3.1.1 Scripting languages
4.3.1.2 Inline documentation for external scripts
4.3.2 The `noscript` element
4.4 Sections
4.4.1 The `body` element
4.4.2 The `section` element
4.4.3 The `nav` element
4.4.4 The `article` element
4.4.5 The `aside` element
4.4.6 The `h1`, `h2`, `h3`, `h4`, `h5`, and `h6` elements
4.4.7 The `header` element
4.4.8 The `footer` element
4.4.9 The `address` element

4.4.10 Headings and sections
4.4.10.1 Creating an outline
4.4.10.2 Distinguishing site-wide headings from page headings
4.5 Grouping content
4.5.1 The p element
4.5.2 The hr element
4.5.3 The br element
4.5.4 The pre element
4.5.5 The dialog element
4.5.6 The blockquote element
4.5.7 The ol element
4.5.8 The ul element
4.5.9 The li element
4.5.10 The dl element
4.5.11 The dt element
4.5.12 The dd element
4.5.13 Common grouping idioms
4.5.13.1 Tag clouds
4.6 Text-level semantics
4.6.1 The a element
4.6.2 The q element
4.6.3 The cite element
4.6.4 The em element
4.6.5 The strong element
4.6.6 The small element
4.6.7 The mark element
4.6.8 The dfn element
4.6.9 The abbr element
4.6.10 The time element
4.6.11 The progress element
4.6.12 The meter element
4.6.13 The code element
4.6.14 The var element
4.6.15 The samp element
4.6.16 The kbd element
4.6.17 The sub and sup elements
4.6.18 The span element
4.6.19 The i element
4.6.20 The b element
4.6.21 The bdo element
4.6.22 The ruby element
4.6.23 The rt element
4.6.24 The rp element
4.6.25 Usage summary
4.6.26 Footnotes
4.7 Edits
4.7.1 The ins element
4.7.2 The del element
4.7.3 Attributes common to ins and del elements
4.7.4 Edits and paragraphs
4.7.5 Edits and lists
4.8 Embedded content

4.8.1 The `figure` element
4.8.2 The `img` element
4.8.2.1 Requirements for providing text to act as an alternative for images
4.8.2.1.1 A link or button containing nothing but the image
4.8.2.1.2 A phrase or paragraph with an alternative graphical representation: charts, diagrams, graphs, maps, illustrations
4.8.2.1.3 A short phrase or label with an alternative graphical representation: icons, logos
4.8.2.1.4 Text that has been rendered to a graphic for typographical effect
4.8.2.1.5 A graphical representation of some of the surrounding text
4.8.2.1.6 A purely decorative image that doesn't add any information
4.8.2.1.7 A group of images that form a single larger picture with no links
4.8.2.1.8 A group of images that form a single larger picture with links
4.8.2.1.9 A key part of the content
4.8.2.1.10 An image not intended for the user
4.8.2.1.11 An image in an e-mail or document intended for a specific person who is known to be able to view images
4.8.2.1.12 General guidelines
4.8.2.1.13 Guidance for markup generators
4.8.3 The `iframe` element
4.8.4 The `embed` element
4.8.5 The `object` element
4.8.6 The `param` element
4.8.7 The `video` element
4.8.7.1 Video and audio codecs for `video` elements
4.8.8 The `audio` element
4.8.8.1 Audio codecs for `audio` elements
4.8.9 The `source` element
4.8.10 Media elements
4.8.10.1 Error codes
4.8.10.2 Location of the media resource
4.8.10.3 Media types
4.8.10.4 Network states
4.8.10.5 Loading the media resource
4.8.10.6 Offsets into the media resource
4.8.10.7 The ready states
4.8.10.8 Cue ranges
4.8.10.9 Playing the media resource
4.8.10.10 Seeking
4.8.10.11 User interface
4.8.10.12 Time ranges
4.8.10.13 Event summary
4.8.10.14 Security and privacy considerations
4.8.11 The `canvas` element
4.8.11.1 The 2D context
4.8.11.1.1 The canvas state

4.8.11.1.2 Transformations
4.8.11.1.3 Compositing
4.8.11.1.4 Colors and styles
4.8.11.1.5 Line styles
4.8.11.1.6 Shadows
4.8.11.1.7 Simple shapes (rectangles)
4.8.11.1.8 Complex shapes (paths)
4.8.11.1.9 Text
4.8.11.1.10 Images
4.8.11.1.11 Pixel manipulation
4.8.11.1.12 Drawing model
4.8.11.2 Color spaces and color correction
4.8.11.3 Security with `canvas` elements
4.8.12 The `map` element
4.8.13 The `area` element
4.8.14 Image maps
4.8.14.1 Authoring
4.8.14.2 Processing model
4.8.15 MathML
4.8.16 SVG
4.8.17 Dimension attributes
4.9 Tabular data
4.9.1 Introduction
4.9.2 The `table` element
4.9.3 The `caption` element
4.9.4 The `colgroup` element
4.9.5 The `col` element
4.9.6 The `tbody` element
4.9.7 The `thead` element
4.9.8 The `tfoot` element
4.9.9 The `tr` element
4.9.10 The `td` element
4.9.11 The `th` element
4.9.12 Attributes common to `td` and `th` elements
4.9.13 Processing model
4.9.13.1 Forming a table
4.9.13.2 Forming relationships between data cells and header cells
4.10 Forms
4.10.1 The `form` element
4.10.2 The `fieldset` element
4.10.3 The `label` element
4.10.4 The `input` element
4.10.4.1 States of the `type` attribute
4.10.4.1.1 Hidden state
4.10.4.1.2 Text state and Search state
4.10.4.1.3 URL state
4.10.4.1.4 E-mail state
4.10.4.1.5 Password state
4.10.4.1.6 Date and Time state
4.10.4.1.7 Date state
4.10.4.1.8 Month state
4.10.4.1.9 Week state

4.10.4.1.10 Time state
4.10.4.1.11 Local Date and Time state
4.10.4.1.12 Number state
4.10.4.1.13 Range state
4.10.4.1.14 Color state
4.10.4.1.15 Checkbox state
4.10.4.1.16 Radio Button state
4.10.4.1.17 File Upload state
4.10.4.1.18 Submit Button state
4.10.4.1.19 Image Button state
4.10.4.1.20 Reset Button state
4.10.4.1.21 Button state
4.10.4.2 Common `input` element attributes
4.10.4.2.1 The `autocomplete` attribute
4.10.4.2.2 The `list` attribute
4.10.4.2.3 The `readonly` attribute
4.10.4.2.4 The `size` attribute
4.10.4.2.5 The `required` attribute
4.10.4.2.6 The `multiple` attribute
4.10.4.2.7 The `maxlength` attribute
4.10.4.2.8 The `pattern` attribute
4.10.4.2.9 The `min` and `max` attributes
4.10.4.2.10 The `step` attribute
4.10.4.2.11 The `placeholder` attribute
4.10.4.3 Common `input` element APIs
4.10.4.4 Common event behaviors
4.10.5 The `button` element
4.10.6 The `select` element
4.10.7 The `datalist` element
4.10.8 The `optgroup` element
4.10.9 The `option` element
4.10.10 The `textarea` element
4.10.11 The `output` element
4.10.12 Association of controls and forms
4.10.13 Attributes common to form controls
4.10.13.1 Naming form controls
4.10.13.2 Enabling and disabling form controls
4.10.13.3 A form control's value
4.10.13.4 Autofocusing a form control
4.10.13.5 Limiting user input length
4.10.13.6 Form submission
4.10.14 Constraints
4.10.14.1 Definitions
4.10.14.2 Constraint validation
4.10.14.3 The constraint validation API
4.10.14.4 Security
4.10.15 Form submission
4.10.15.1 Introduction
4.10.15.2 Implicit submission
4.10.15.3 Form submission algorithm
4.10.15.4 URL-encoded form data
4.10.15.5 Multipart form data

4.10.15.6 Plain text form data
4.10.16 Resetting a form
4.10.17 Event dispatch
4.11 Interactive elements
4.11.1 The `details` element
4.11.2 The `datagrid` element
4.11.2.1 The `datagrid` data model
4.11.2.2 How rows are identified
4.11.2.3 The data provider interface
4.11.2.4 The default data provider
4.11.2.4.1 Common default data provider method definitions for cells
4.11.2.5 Populating the `datagrid` element
4.11.2.6 Updating the `datagrid`
4.11.2.7 Requirements for interactive user agents
4.11.2.8 The selection
4.11.2.9 Columns and captions
4.11.3 The `command` element
4.11.4 The `bb` element
4.11.4.1 Browser button types
4.11.4.1.1 The *make application* state
4.11.5 The `menu` element
4.11.5.1 Introduction
4.11.5.2 Building menus and tool bars
4.11.5.3 Context menus
4.11.5.4 Tool bars
4.11.6 Commands
4.11.6.1 Using the `a` element to define a command
4.11.6.2 Using the `button` element to define a command
4.11.6.3 Using the `input` element to define a command
4.11.6.4 Using the `option` element to define a command
4.11.6.5 Using the `command` element to define a command
4.11.6.6 Using the `bb` element to define a command
4.12 Miscellaneous elements
4.12.1 The `legend` element
4.12.2 The `div` element
4.13 Matching HTML elements using selectors

5 Web browsers
5.1 Browsing contexts
5.1.1 Nested browsing contexts
5.1.1.1 Navigating nested browsing contexts in the DOM
5.1.2 Auxiliary browsing contexts
5.1.2.1 Navigating auxiliary browsing contexts in the DOM
5.1.3 Secondary browsing contexts
5.1.4 Security
5.1.5 Groupings of browsing contexts
5.1.6 Browsing context names
5.2 The `WindowProxy` object
5.3 The `Window` object
5.3.1 Security
5.3.2 APIs for creating and navigating browsing contexts by name

5.3.3 Accessing other browsing contexts
5.3.4 Named access on the `Window` object
5.3.5 Garbage collection and browsing contexts
5.3.6 Browser interface elements
5.4 Origin
5.4.1 Relaxing the same-origin restriction
5.5 Scripting
5.5.1 Introduction
5.5.2 Enabling and disabling scripting
5.5.3 Processing model
5.5.3.1 Definitions
5.5.3.2 Calling scripts
5.5.3.3 Creating scripts
5.5.3.4 Killing scripts
5.5.4 Event loops
5.5.4.1 Definitions
5.5.4.2 Processing model
5.5.4.3 Generic task sources
5.5.5 The `javascript:` protocol
5.5.6 Events
5.5.6.1 Event handler attributes
5.5.6.2 Event handler attributes on elements and on `Window` objects
5.5.6.3 Event firing
5.5.6.4 Events and the `Window` object
5.5.6.5 Runtime script errors
5.6 User prompts
5.6.1 Simple dialogs
5.6.2 Printing
5.6.3 Dialogs implemented using separate documents
5.7 System state and capabilities
5.7.1 Client identification
5.7.2 Custom protocol and content handlers
5.7.2.1 Security and privacy
5.7.2.2 Sample user interface
5.7.3 Manually releasing the storage mutex
5.8 Offline Web applications
5.8.1 Introduction
5.8.2 Application caches
5.8.3 The cache manifest syntax
5.8.3.1 A sample manifest
5.8.3.2 Writing cache manifests
5.8.3.3 Parsing cache manifests
5.8.4 Updating an application cache
5.8.5 Matching a fallback namespace
5.8.6 The application cache selection algorithm
5.8.7 Changes to the networking model
5.8.8 Application cache API
5.8.9 Browser state
5.9 Session history and navigation
5.9.1 The session history of browsing contexts
5.9.2 The `History` interface

5.9.3 Activating state object entries
5.9.4 The `Location` interface
5.9.4.1 Security
5.9.5 Implementation notes for session history
5.10 Browsing the Web
5.10.1 Navigating across documents
5.10.2 Page load processing model for HTML files
5.10.3 Page load processing model for XML files
5.10.4 Page load processing model for text files
5.10.5 Page load processing model for images
5.10.6 Page load processing model for content that uses plugins
5.10.7 Page load processing model for inline content that doesn't have a DOM
5.10.8 Navigating to a fragment identifier
5.10.9 History traversal
5.10.10 Unloading documents
5.10.10.1 Event definition
5.11 Links
5.11.1 Hyperlink elements
5.11.2 Following hyperlinks
5.11.2.1 Hyperlink auditing
5.11.3 Link types
5.11.3.1 Link type "`alternate`"
5.11.3.2 Link type "`archives`"
5.11.3.3 Link type "`author`"
5.11.3.4 Link type "`bookmark`"
5.11.3.5 Link type "`external`"
5.11.3.6 Link type "`feed`"
5.11.3.7 Link type "`help`"
5.11.3.8 Link type "`icon`"
5.11.3.9 Link type "`license`"
5.11.3.10 Link type "`nofollow`"
5.11.3.11 Link type "`noreferrer`"
5.11.3.12 Link type "`pingback`"
5.11.3.13 Link type "`prefetch`"
5.11.3.14 Link type "`search`"
5.11.3.15 Link type "`stylesheet`"
5.11.3.16 Link type "`sidebar`"
5.11.3.17 Link type "`tag`"
5.11.3.18 Hierarchical link types
5.11.3.18.1 Link type "`index`"
5.11.3.18.2 Link type "`up`"
5.11.3.19 Sequential link types
5.11.3.19.1 Link type "`first`"
5.11.3.19.2 Link type "`last`"
5.11.3.19.3 Link type "`next`"
5.11.3.19.4 Link type "`prev`"
5.11.3.20 Other link types

6 User Interaction
6.1 Introduction
6.2 The `hidden` attribute

6.3 Activation
6.4 Scrolling elements into view
6.5 Focus
6.5.1 Sequential focus navigation
6.5.2 Focus management
6.5.3 Document-level focus APIs
6.5.4 Element-level focus APIs
6.6 The text selection APIs
6.6.1 APIs for the browsing context selection
6.6.2 APIs for the text field selections
6.7 The `contenteditable` attribute
6.7.1 User editing actions
6.7.2 Making entire documents editable
6.8 Spelling and grammar checking
6.9 Drag and drop
6.9.1 Introduction
6.9.2 The `DragEvent` and `DataTransfer` interfaces
6.9.3 Events fired during a drag-and-drop action
6.9.4 Drag-and-drop processing model
6.9.4.1 When the drag-and-drop operation starts or ends in another document
6.9.4.2 When the drag-and-drop operation starts or ends in another application
6.9.5 The `draggable` attribute
6.9.6 Copy and paste
6.9.6.1 Copy to clipboard
6.9.6.2 Cut to clipboard
6.9.6.3 Paste from clipboard
6.9.6.4 Paste from selection
6.9.7 Security risks in the drag-and-drop model
6.10 Undo history
6.10.1 Introduction
6.10.2 Definitions
6.10.3 The `UndoManager` interface
6.10.4 Undo: moving back in the undo transaction history
6.10.5 Redo: moving forward in the undo transaction history
6.10.6 The `UndoManagerEvent` interface and the `undo` and `redo` events
6.10.7 Implementation notes
6.11 Command APIs

7 Communication
7.1 Event definitions
7.2 Cross-document messaging
7.2.1 Introduction
7.2.2 Security
7.2.2.1 Authors
7.2.2.2 User agents
7.2.3 Posting messages
7.2.4 Posting messages with message ports
7.3 Channel messaging
7.3.1 Introduction

7.3.2 Message channels
7.3.3 Message ports
7.3.3.1 Ports and garbage collection

8 The HTML syntax
8.1 Writing HTML documents
8.1.1 The DOCTYPE
8.1.2 Elements
8.1.2.1 Start tags
8.1.2.2 End tags
8.1.2.3 Attributes
8.1.2.4 Optional tags
8.1.2.5 Restrictions on content models
8.1.2.6 Restrictions on the contents of CDATA and RCDATA elements
8.1.3 Text
8.1.3.1 Newlines
8.1.4 Character references
8.1.5 CDATA sections
8.1.6 Comments
8.2 Parsing HTML documents
8.2.1 Overview of the parsing model
8.2.2 The input stream
8.2.2.1 Determining the character encoding
8.2.2.2 Preprocessing the input stream
8.2.2.3 Changing the encoding while parsing
8.2.3 Parse state
8.2.3.1 The insertion mode
8.2.3.2 The stack of open elements
8.2.3.3 The list of active formatting elements
8.2.3.4 The element pointers
8.2.3.5 Other parsing state flags
8.2.4 Tokenization
8.2.4.1 Data state
8.2.4.2 Character reference data state
8.2.4.3 Tag open state
8.2.4.4 Close tag open state
8.2.4.5 Tag name state
8.2.4.6 Before attribute name state
8.2.4.7 Attribute name state
8.2.4.8 After attribute name state
8.2.4.9 Before attribute value state
8.2.4.10 Attribute value (double-quoted) state
8.2.4.11 Attribute value (single-quoted) state
8.2.4.12 Attribute value (unquoted) state
8.2.4.13 Character reference in attribute value state
8.2.4.14 After attribute value (quoted) state
8.2.4.15 Self-closing start tag state
8.2.4.16 Bogus comment state
8.2.4.17 Markup declaration open state
8.2.4.18 Comment start state
8.2.4.19 Comment start dash state
8.2.4.20 Comment state

8.2.4.21 Comment end dash state
8.2.4.22 Comment end state
8.2.4.23 DOCTYPE state
8.2.4.24 Before DOCTYPE name state
8.2.4.25 DOCTYPE name state
8.2.4.26 After DOCTYPE name state
8.2.4.27 Before DOCTYPE public identifier state
8.2.4.28 DOCTYPE public identifier (double-quoted) state
8.2.4.29 DOCTYPE public identifier (single-quoted) state
8.2.4.30 After DOCTYPE public identifier state
8.2.4.31 Before DOCTYPE system identifier state
8.2.4.32 DOCTYPE system identifier (double-quoted) state
8.2.4.33 DOCTYPE system identifier (single-quoted) state
8.2.4.34 After DOCTYPE system identifier state
8.2.4.35 Bogus DOCTYPE state
8.2.4.36 CDATA section state
8.2.4.37 Tokenizing character references
8.2.5 Tree construction
8.2.5.1 Creating and inserting elements
8.2.5.2 Closing elements that have implied end tags
8.2.5.3 Foster parenting
8.2.5.4 The "initial" insertion mode
8.2.5.5 The "before html" insertion mode
8.2.5.6 The "before head" insertion mode
8.2.5.7 The "in head" insertion mode
8.2.5.8 The "in head noscript" insertion mode
8.2.5.9 The "after head" insertion mode
8.2.5.10 The "in body" insertion mode
8.2.5.11 The "in CDATA/RCDATA" insertion mode
8.2.5.12 The "in table" insertion mode
8.2.5.13 The "in caption" insertion mode
8.2.5.14 The "in column group" insertion mode
8.2.5.15 The "in table body" insertion mode
8.2.5.16 The "in row" insertion mode
8.2.5.17 The "in cell" insertion mode
8.2.5.18 The "in select" insertion mode
8.2.5.19 The "in select in table" insertion mode
8.2.5.20 The "in foreign content" insertion mode
8.2.5.21 The "after body" insertion mode
8.2.5.22 The "in frameset" insertion mode
8.2.5.23 The "after frameset" insertion mode
8.2.5.24 The "after after body" insertion mode
8.2.5.25 The "after after frameset" insertion mode
8.2.6 The end
8.2.7 Coercing an HTML DOM into an infoset
8.3 Namespaces
8.4 Serializing HTML fragments
8.5 Parsing HTML fragments
8.6 Named character references

9 The XHTML syntax
9.1 Writing XHTML documents

9.2 Parsing XHTML documents
9.3 Serializing XHTML fragments
9.4 Parsing XHTML fragments

10 Rendering
10.1 Introduction
10.2 The CSS user agent style sheet and presentational hints
10.2.1 Introduction
10.2.2 Display types
10.2.3 Margins and padding
10.2.4 Alignment
10.2.5 Fonts and colors
10.2.6 Punctuation and decorations
10.2.7 Resetting rules for inherited properties
10.2.8 The `hr` element
10.2.9 The `fieldset` element
10.3 Replaced elements
10.3.1 Embedded content
10.3.2 Images
10.3.3 Attributes for embedded content and images
10.3.4 Image maps
10.3.5 Tool bars
10.4 Bindings
10.4.1 Introduction
10.4.2 The `bb` element
10.4.3 The `button` element
10.4.4 The `datagrid` element
10.4.5 The `details` element
10.4.6 The `input` element as a text entry widget
10.4.7 The `input` element as domain-specific widgets
10.4.8 The `input` element as a range control
10.4.9 The `input` element as a color well
10.4.10 The `input` element as a check box and radio button widgets
10.4.11 The `input` element as a file upload control
10.4.12 The `input` element as a button
10.4.13 The `marquee` element
10.4.14 The `meter` element
10.4.15 The `progress` element
10.4.16 The `select` element
10.4.17 The `textarea` element
10.5 Frames and framesets
10.6 Interactive media
10.6.1 Links, forms, and navigation
10.6.2 The `mark` element
10.6.3 The `title` attribute
10.7 Print media
10.8 Interaction with CSS

11 Obsolete features
11.1 Self-contained features
11.1.1 The `applet` element
11.1.2 The `marquee` element

11.2 Other elements and attributes
11.3 Other DOM APIs
11.4 Conformance checkers

12 Things that you can't do with this specification because they are better handled using other technologies that are further described herein
12.1 Localization
12.2 Declarative 3D scenes
12.3 Timers
12.4 Rendering and the DOM

Index

References

Acknowledgements

1 Introduction

1.1 Background

This section is non-normative.

The World Wide Web's markup language has always been HTML. HTML was primarily designed as a language for semantically describing scientific documents, although its general design and adaptations over the years has enabled it to be used to describe a number of other types of documents.

The main area that has not been adequately addressed by HTML is a vague subject referred to as Web Applications. This specification attempts to rectify this, while at the same time updating the HTML specifications to address issues raised in the past few years.

1.2 Audience

This section is non-normative.

This specification is intended for authors of documents and scripts that use the features defined in this specification, and implementors of tools that are intended to conform to this specification, and individuals wishing to establish the correctness of documents or implementations with respect to the requirements of this specification.

This document is probably not suited to readers who do not already have at least a passing familiarity with Web technologies, as in places it sacrifices clarity for precision, and brevity for completeness. More approachable tutorials and authoring guides can provide a gentler introduction to the topic.

In particular, readers should be familiar with the basics of DOM Core and DOM Events before reading this specification. An understanding of WebIDL, HTTP, XML, Unicode, character encodings, JavaScript, and CSS will be helpful in places but is not essential.

1.3 Scope

This section is non-normative.

This specification is limited to providing a semantic-level markup language and associated semantic-level scripting APIs for authoring accessible pages on the Web ranging from static documents to dynamic applications.

The scope of this specification does not include providing mechanisms for media-specific customization of presentation (although default rendering rules for Web browsers are included at the end of this specification, and several mechanisms for hooking into CSS are provided as part of the language).

The scope of this specification does not include documenting every HTML or DOM feature supported by Web browsers. Browsers support many features that are considered to be very bad for accessibility or that are otherwise inappropriate. For example, the `blink` element is clearly presentational and authors wishing to cause text to blink should instead use CSS.

The scope of this specification is not to describe an entire operating system. In particular, hardware configuration software, image manipulation tools, and applications that users would be expected to use with high-end workstations on a daily basis are out of scope. In terms of applications, this specification is targeted specifically at applications that would be expected to be used by users on an occasional basis, or regularly but from disparate locations, with low CPU requirements. For instance online purchasing systems, searching systems, games (especially multiplayer online games), public telephone books or address books, communications software (e-mail clients, instant messaging clients, discussion software), document editing software, etc.

1.4 History

This section is non-normative.

Work on HTML5 originally started in late 2003, as a proof of concept to show that it was possible to extend HTML4's forms to provide many of the features that XForms 1.0 introduced, without requiring browsers to implement rendering engines that were incompatible with existing HTML Web pages. At this early stage, while the draft was already publicly available, and input was already being solicited from all sources, the specification was only under Opera Software's copyright.

In early 2004, some of the principles that underlie this effort, as well as an early draft proposal covering just forms-related features, were presented to the W3C jointly by Mozilla and Opera at a workshop discussing the future of Web Applications on the Web. The proposal was rejected on the grounds that the proposal conflicted with the previously chosen direction for the Web's evolution.

Shortly thereafter, Apple, Mozilla, and Opera jointly announced their intent to continue working on the effort. A public mailing list was created, and the drafts were moved to the WHATWG site. The copyright was subsequently amended to be jointly owned by all three vendors, and to allow reuse of the specifications.

In 2006, the W3C expressed interest in the specification, and created a working group chartered to work with the WHATWG on the development of the HTML5 specifications. The working group opened in 2007. Apple, Mozilla, and Opera allowed the W3C to publish the specifications under the W3C copyright, while keeping versions with the less restrictive license on the WHATWG site.

Since then, both groups have been working together.

1.5 Relationships to other specifications

1.5.1 Relationship to HTML 4.01 and DOM2 HTML

This section is non-normative.

This specification represents a new version of HTML4, along with a new version of the associated DOM2 HTML API. Migration from HTML4 to the format and APIs described in this specification should in most cases be straightforward, as care has been taken to ensure that backwards-compatibility is retained. [HTML4] [DOM2HTML]

1.5.2 Relationship to XHTML 1.x

This section is non-normative.

This specification is intended to replace XHTML 1.0 as the normative definition of the XML serialization of the HTML vocabulary. [XHTML10]

While this specification updates the semantics and requirements of the vocabulary defined by XHTML Modularization 1.1 and used by XHTML 1.1, it does not attempt to provide a replacement for the modularization scheme defined and used by those (and other) specifications, and therefore cannot be considered a complete replacement for them. [XHTMLMOD] [XHTML11]

Thus, authors and implementors who do not need such a modularization scheme can consider this specification a replacement for XHTML 1.x, but those who do need such a mechanism are encouraged to continue using the XHTML 1.1 line of specifications.

1.5.3 Relationship to XHTML2 and XForms

This section is non-normative.

XHTML2 defines a new vocabulary with features for hyperlinks, multimedia content, annotating document edits, rich metadata, declarative interactive forms, and describing the semantics of human literary works such as poems and scientific papers. [XHTML2]

XForms similarly defines a new vocabulary with features for complex data entry, such as tax forms or insurance forms.

However, XHTML2 and XForms lack features to express the semantics of many of the non-document types of content often seen on the Web. For instance, they are not well-suited for marking up forum sites, auction sites, search engines, online shops, mapping applications, e-mail applications, word processors, real-time strategy games, and the like.

This specification aims to extend HTML so that it is also suitable in these contexts.

XHTML2, XForms, and this specification all use different namespaces and therefore can all be implemented in the same XML processor.

1.6 HTML vs XHTML

This section is non-normative.

This specification defines an abstract language for describing documents and applications, and some APIs for interacting with in-memory representations of resources that use this language.

The in-memory representation is known as "DOM5 HTML", or "the DOM" for short.

There are various concrete syntaxes that can be used to transmit resources that use this abstract language, two of which are defined in this specification.

The first such concrete syntax is "HTML5". This is the format recommended for most authors. It is compatible with all legacy Web browsers. If a document is transmitted with the MIME type `text/html`, then it will be processed as an "HTML5" document by Web browsers.

The second concrete syntax uses XML, and is known as "XHTML5". When a document is transmitted with an XML MIME type, such as `application/xhtml+xml`, then it is processed by an XML processor by Web browsers, and treated as an "XHTML5" document. Authors are reminded that the processing for XML and HTML differs; in particular, even minor syntax errors will prevent an XML document from being rendered fully, whereas they would be ignored in the "HTML5" syntax.

The "DOM5 HTML", "HTML5", and "XHTML5" representations cannot all represent the same content. For example, namespaces cannot be represented using "HTML5", but they are supported in "DOM5 HTML" and "XHTML5". Similarly, documents that use the `noscript` feature can be represented using "HTML5", but cannot be represented with "XHTML5" and "DOM5 HTML". Comments that contain the string "`-->`" can be represented in "DOM5 HTML" but not in "HTML5" and "XHTML5". And so forth.

1.7 Structure of this specification

This section is non-normative.

This specification is divided into the following major sections:

Common Infrastructure
: The conformance classes, algorithms, definitions, and the common underpinnings of the rest of the specification.

The DOM
: Documents are built from elements. These elements form a tree using the DOM. This section defines the features of this DOM, as well as introducing the features common to all elements, and the concepts used in defining elements.

Elements
: Each element has a predefined meaning, which is explained in this section. Rules for authors on how to use the element, along with user agent requirements for how to handle each element, are also given.

Web Browsers
: HTML documents do not exist in a vacuum — this section defines many of the features that affect environments that deal with multiple pages, links between pages, and running scripts.

User Interaction
: HTML documents can provide a number of mechanisms for users to interact with and

modify content, which are described in this section.

The Communication APIs

Applications written in HTML often require mechanisms to communicate with remote servers, as well as communicating with other applications from different domains running on the same client.

The Language Syntax

All of these features would be for naught if they couldn't be represented in a serialized form and sent to other people, and so this section defines the syntax of HTML, along with rules for how to parse HTML.

There are also a couple of appendices, defining rendering rules for Web browsers and listing areas that are out of scope for this specification.

1.7.1 How to read this specification

This specification should be read like all other specifications. First, it should be read cover-to-cover, multiple times. Then, it should be read backwards at least once. Then it should be read by picking random sections from the contents list and following all the cross-references.

1.7.2 Typographic conventions

This is a definition, requirement, or explanation.

Note: This is a note.

> This is an example.

This is an open issue.

?Warning! ***This is a warning.***

```
interface Example {
  // this is an IDL definition
};
```

variable **=** ***object*** **.** `method` **(** **[** ***optionalArgument*** **]** **)**

This is a note to authors describing the usage of an interface.

```
/* this is a CSS fragment */
```

The defining instance of a term is marked up like **this**. Uses of that term are marked up like this or like *this*.

The defining instance of an element, attribute, or API is marked up like `this`. References to

that element, attribute, or API are marked up like `this`.

Other code fragments are marked up `like this`.

Variables are marked up like *this*.

This is an implementation requirement.

2 Common infrastructure

2.1 Terminology

This specification refers to both HTML and XML attributes and DOM attributes, often in the same context. When it is not clear which is being referred to, they are referred to as **content attributes** for HTML and XML attributes, and **DOM attributes** for those from the DOM. Similarly, the term "properties" is used for both ECMAScript object properties and CSS properties. When these are ambiguous they are qualified as object properties and CSS properties respectively.

The term HTML documents is sometimes used in contrast with XML documents to specifically mean documents that were parsed using an HTML parser (as opposed to using an XML parser or created purely through the DOM).

Generally, when the specification states that a feature applies to HTML or XHTML, it also includes the other. When a feature specifically only applies to one of the two languages, it is called out by explicitly stating that it does not apply to the other format, as in "for HTML, ... (this does not apply to XHTML)".

This specification uses the term *document* to refer to any use of HTML, ranging from short static documents to long essays or reports with rich multimedia, as well as to fully-fledged interactive applications.

For simplicity, terms such as *shown*, *displayed*, and *visible* might sometimes be used when referring to the way a document is rendered to the user. These terms are not meant to imply a visual medium; they must be considered to apply to other media in equivalent ways.

2.1.1 XML

To ease migration from HTML to XHTML, UAs conforming to this specification will place elements in HTML in the `http://www.w3.org/1999/xhtml` namespace, at least for the purposes of the DOM and CSS. The term "**elements in the HTML namespace**", or "**HTML elements**" for short, when used in this specification, thus refers to both HTML and XHTML elements.

Unless otherwise stated, all elements defined or mentioned in this specification are in the `http://www.w3.org/1999/xhtml` namespace, and all attributes defined or mentioned in this specification have no namespace (they are in the per-element partition).

When an XML name, such as an attribute or element name, is referred to in the form *`prefix`*`:`*`localName`*, as in `xml:id` or `svg:rect`, it refers to a name with the local name *localName* and the namespace given by the prefix, as defined by the following table:

`xml`
: `http://www.w3.org/XML/1998/namespace`

`html`

`http://www.w3.org/1999/xhtml`

svg

`http://www.w3.org/2000/svg`

Attribute names are said to be **XML-compatible** if they match the `Name` production defined in XML, they contain no U+003A COLON (:) characters, and their first three characters are not an ASCII case-insensitive match for the string "`xml`". [XML]

2.1.2 DOM trees

The term **root element**, when not explicitly qualified as referring to the document's root element, means the furthest ancestor element node of whatever node is being discussed, or the node itself if it has no ancestors. When the node is a part of the document, then that is indeed the document's root element; however, if the node is not currently part of the document tree, the root element will be an orphaned node.

A node's **home subtree** is the subtree rooted at that node's root element.

The `Document` of a `Node` (such as an element) is the `Document` that the `Node`'s `ownerDocument` DOM attribute returns.

An element is said to have been **inserted into a document** when its root element changes and is now the document's root element. If a `Node` is in a `Document` then that `Document` is always the `Node`'s `Document`, and the `Node`'s `ownerDocument` DOM attribute thus always returns that `Document`.

The term **tree order** means a pre-order, depth-first traversal of DOM nodes involved (through the `parentNode`/`childNodes` relationship).

When it is stated that some element or attribute is **ignored**, or treated as some other value, or handled as if it was something else, this refers only to the processing of the node after it is in the DOM. A user agent must not mutate the DOM in such situations.

The term **text node** refers to any `Text` node, including `CDATASection` nodes; specifically, any `Node` with node type `TEXT_NODE` (3) or `CDATA_SECTION_NODE` (4). [DOM3CORE]

2.1.3 Scripting

The construction "a `Foo` object", where `Foo` is actually an interface, is sometimes used instead of the more accurate "an object implementing the interface `Foo`".

A DOM attribute is said to be *getting* when its value is being retrieved (e.g. by author script), and is said to be *setting* when a new value is assigned to it.

If a DOM object is said to be **live**, then that means that any attributes returning that object must always return the same object (not a new object each time), and the attributes and methods on that object must operate on the actual underlying data, not a snapshot of the data.

The terms *fire* and *dispatch* are used interchangeably in the context of events, as in the DOM Events specifications. [DOM3EVENTS]

2.1.4 Plugins

The term **plugin** is used to mean any content handler, typically a third-party content handler, for Web content types that are not supported by the user agent natively, or for content types that do not expose a DOM, that supports rendering the content as part of the user agent's interface.

> One example of a plugin would be a PDF viewer that is instantiated in a browsing context when the user navigates to a PDF file. This would count as a plugin regardless of whether the party that implemented the PDF viewer component was the same as that which implemented the user agent itself. However, a PDF viewer application that launches separate from the user agent (as opposed to using the same interface) is not a plugin by this definition.

Note: This specification does not define a mechanism for interacting with plugins, as it is expected to be user-agent- and platform-specific. Some UAs might opt to support a plugin mechanism such as the Netscape Plugin API; others might use remote content converters or have built-in support for certain types. [NPAPI]

?Warning! Browsers should take extreme care when interacting with external content intended for plugins. When third-party software is run with the same privileges as the user agent itself, vulnerabilities in the third-party software become as dangerous as those in the user agent.

2.1.5 Character encodings

An **ASCII-compatible character encoding** is one that is a superset of US-ASCII (specifically, ANSI_X3.4-1968) for bytes in the set 0x09, 0x0A, 0x0C, 0x0D, 0x20 - 0x22, 0x26, 0x27, 0x2C - 0x3F, 0x41 - 0x5A, and 0x61 - 0x7A.

2.2 Conformance requirements

All diagrams, examples, and notes in this specification are non-normative, as are all sections explicitly marked non-normative. Everything else in this specification is normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC2119. For readability, these words do not appear in all uppercase letters in this specification. [RFC2119]

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.

This specification describes the conformance criteria for user agents (relevant to implementors) and documents (relevant to authors and authoring tool implementors).

Note: There is no implied relationship between document conformance requirements and implementation conformance requirements. User agents are not free to handle non-conformant documents as they please; the processing model described in this specification applies to implementations regardless of the conformity of the input documents.

User agents fall into several (overlapping) categories with different conformance requirements.

Web browsers and other interactive user agents

Web browsers that support XHTML must process elements and attributes from the HTML namespace found in XML documents as described in this specification, so that users can interact with them, unless the semantics of those elements have been overridden by other specifications.

> A conforming XHTML processor would, upon finding an XHTML `script` element in an XML document, execute the script contained in that element. However, if the element is found within a transformation expressed in XSLT (assuming the user agent also supports XSLT), then the processor would instead treat the `script` element as an opaque element that forms part of the transform.

Web browsers that support HTML must process documents labeled as `text/html` as described in this specification, so that users can interact with them.

User agents that support scripting must also be conforming implementations of the IDL fragments in this specification, as described in the WebIDL specification. [WebIDL]

Non-interactive presentation user agents

User agents that process HTML and XHTML documents purely to render non-interactive versions of them must comply to the same conformance criteria as Web browsers, except that they are exempt from requirements regarding user interaction.

Note: Typical examples of non-interactive presentation user agents are printers (static UAs) and overhead displays (dynamic UAs). It is expected that most static non-interactive presentation user agents will also opt to lack scripting support.

> A non-interactive but dynamic presentation UA would still execute scripts, allowing forms to be dynamically submitted, and so forth. However, since the concept of "focus" is irrelevant when the user cannot interact with the document, the UA would not need to support any of the focus-related DOM APIs.

User agents with no scripting support

Implementations that do not support scripting (or which have their scripting features disabled entirely) are exempt from supporting the events and DOM interfaces mentioned in this specification. For the parts of this specification that are defined in terms of an events model or in terms of the DOM, such user agents must still act as if events and the DOM were supported.

Note: Scripting can form an integral part of an application. Web browsers

that do not support scripting, or that have scripting disabled, might be unable to fully convey the author's intent.

Conformance checkers

Conformance checkers must verify that a document conforms to the applicable conformance criteria described in this specification. Automated conformance checkers are exempt from detecting errors that require interpretation of the author's intent (for example, while a document is non-conforming if the content of a `blockquote` element is not a quote, conformance checkers running without the input of human judgement do not have to check that `blockquote` elements only contain quoted material).

Conformance checkers must check that the input document conforms when parsed without a browsing context (meaning that no scripts are run, and that the parser's scripting flag is disabled), and should also check that the input document conforms when parsed with a browsing context in which scripts execute, and that the scripts never cause non-conforming states to occur other than transiently during script execution itself. (This is only a "SHOULD" and not a "MUST" requirement because it has been proven to be impossible. [HALTINGPROBLEM])

The term "HTML5 validator" can be used to refer to a conformance checker that itself conforms to the applicable requirements of this specification.

XML DTDs cannot express all the conformance requirements of this specification. Therefore, a validating XML processor and a DTD cannot constitute a conformance checker. Also, since neither of the two authoring formats defined in this specification are applications of SGML, a validating SGML system cannot constitute a conformance checker either.

To put it another way, there are three types of conformance criteria:

1. ***Criteria that can be expressed in a DTD.***
2. ***Criteria that cannot be expressed by a DTD, but can still be checked by a machine.***
3. ***Criteria that can only be checked by a human.***

A conformance checker must check for the first two. A simple DTD-based validator only checks for the first class of errors and is therefore not a conforming conformance checker according to this specification.

Data mining tools

Applications and tools that process HTML and XHTML documents for reasons other than to either render the documents or check them for conformance should act in accordance to the semantics of the documents that they process.

> A tool that generates document outlines but increases the nesting level for each paragraph and does not increase the nesting level for each section would not be conforming.

Authoring tools and markup generators

Authoring tools and markup generators must generate conforming documents.

Conformance criteria that apply to authors also apply to authoring tools, where appropriate.

Authoring tools are exempt from the strict requirements of using elements only for their specified purpose, but only to the extent that authoring tools are not yet able to determine author intent.

> For example, it is not conforming to use an `address` element for arbitrary contact information; that element can only be used for marking up contact information for the author of the document or section. However, since an authoring tool is likely unable to determine the difference, an authoring tool is exempt from that requirement.

Note: In terms of conformance checking, an editor is therefore required to output documents that conform to the same extent that a conformance checker will verify.

When an authoring tool is used to edit a non-conforming document, it may preserve the conformance errors in sections of the document that were not edited during the editing session (i.e. an editing tool is allowed to round-trip erroneous content). However, an authoring tool must not claim that the output is conformant if errors have been so preserved.

Authoring tools are expected to come in two broad varieties: tools that work from structure or semantic data, and tools that work on a What-You-See-Is-What-You-Get media-specific editing basis (WYSIWYG).

The former is the preferred mechanism for tools that author HTML, since the structure in the source information can be used to make informed choices regarding which HTML elements and attributes are most appropriate.

However, WYSIWYG tools are legitimate. WYSIWYG tools should use elements they know are appropriate, and should not use elements that they do not know to be appropriate. This might in certain extreme cases mean limiting the use of flow elements to just a few elements, like `div`, `b`, `i`, and `span` and making liberal use of the `style` attribute.

All authoring tools, whether WYSIWYG or not, should make a best effort attempt at enabling users to create well-structured, semantically rich, media-independent content.

Some conformance requirements are phrased as requirements on elements, attributes, methods or objects. Such requirements fall into two categories: those describing content model restrictions, and those describing implementation behavior. The former category of requirements are requirements on documents and authoring tools. The second category are requirements on user agents.

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

User agents may impose implementation-specific limits on otherwise unconstrained inputs, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.

For compatibility with existing content and prior specifications, this specification describes two authoring formats: one based on XML (referred to as **XHTML5**), and one using a custom format inspired by SGML (referred to as **HTML5**). Implementations may support only one of these two formats, although supporting both is encouraged.

XHTML documents (XML documents using elements from the HTML namespace) that use the new features described in this specification and that are served over the wire (e.g. by HTTP) must be sent using an XML MIME type such as `application/xml` or `application/xhtml+xml` and must not be served as `text/html`. [RFC3023]

HTML documents, if they are served over the wire (e.g. by HTTP) must be labeled with the `text/html` MIME type.

The language in this specification assumes that the user agent expands all entity references, and therefore does not include entity reference nodes in the DOM. If user agents do include entity reference nodes in the DOM, then user agents must handle them as if they were fully expanded when implementing this specification. For example, if a requirement talks about an element's child text nodes, then any text nodes that are children of an entity reference that is a child of that element would be used as well. Entity references to unknown entities must be treated as if they contained just an empty text node for the purposes of the algorithms defined in this specification.

2.2.1 Dependencies

This specification relies on several other underlying specifications.

XML

Implementations that support XHTML5 must support some version of XML, as well as its corresponding namespaces specification, because XHTML5 uses an XML serialization with namespaces. [XML] [XMLNAMES]

DOM

The Document Object Model (DOM) is a representation — a model — of a document and its content. The DOM is not just an API; the conformance criteria of HTML implementations are defined, in this specification, in terms of operations on the DOM. [DOM3CORE]

Implementations must support some version of DOM Core and DOM Events, because this specification is defined in terms of the DOM, and some of the features are defined as extensions to the DOM Core interfaces. [DOM3CORE] [DOM3EVENTS]

WebIDL

The IDL fragments in this specification must be interpreted as required for conforming IDL fragments, as described in the Web IDL specification. [WebIDL]

Media Queries

Implementations must support some version of the Media Queries language. [MQ]

This specification does not require support of any particular network transport protocols, style sheet language, scripting language, or any of the DOM and WebAPI specifications beyond those described above. However, the language described by this specification is biased

towards CSS as the styling language, ECMAScript as the scripting language, and HTTP as the network protocol, and several features assume that those languages and protocols are in use.

Note: This specification might have certain additional requirements on character encodings, image formats, audio formats, and video formats in the respective sections.

2.2.2 Features defined in other specifications

this section will be removed at some point

Some elements are defined in terms of their DOM **`textContent`** attribute. This is an attribute defined on the `Node` interface in DOM3 Core. [DOM3CORE]

The rules for handling alternative style sheets are defined in the CSS object model specification. [CSSOM]

See http://dev.w3.org/cvsweb/~checkout~/csswg/cssom/Overview.html?content-type=text/html;%20

2.2.3 Common conformance requirements for APIs exposed to JavaScript

This section will eventually be removed in favour of WebIDL.

A lot of arrays/lists/collections in this spec assume zero-based indexes but use the term "*index*th" liberally. We should define those to be zero-based and be clearer about this.

Unless otherwise specified, if a DOM attribute that is a floating point number type (`float`) is assigned an Infinity or Not-a-Number value, a `NOT_SUPPORTED_ERR` exception must be raised.

Unless otherwise specified, if a method with an argument that is a floating point number type (`float`) is passed an Infinity or Not-a-Number value, a `NOT_SUPPORTED_ERR` exception must be raised.

Unless otherwise specified, if a method is passed fewer arguments than is defined for that method in its IDL definition, a `NOT_SUPPORTED_ERR` exception must be raised.

Unless otherwise specified, if a method is passed more arguments than is defined for that method in its IDL definition, the excess arguments must be ignored.

2.3 Case-sensitivity and string comparison

This specification defines several comparison operators for strings.

Comparing two strings in a **case-sensitive** manner means comparing them exactly, codepoint for codepoint.

Comparing two strings in an **ASCII case-insensitive** manner means comparing them exactly, codepoint for codepoint, except that the characters in the range U+0041 .. U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) and the corresponding characters in the range U+0061 .. U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z) are considered to also match.

Comparing two strings in a **compatibility caseless** manner means using the Unicode *compatibility caseless match* operation to compare the two strings. [UNICODECASE]

Converting a string to uppercase means replacing all characters in the range U+0061 .. U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z) with the corresponding characters in the range U+0041 .. U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z).

Converting a string to lowercase means replacing all characters in the range U+0041 .. U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) with the corresponding characters in the range U+0061 .. U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z).

A string *pattern* is a **prefix match** for a string *s* when *pattern* is not longer than *s* and truncating *s* to *pattern*'s length leaves the two strings as matches of each other.

2.4 Common microsyntaxes

There are various places in HTML that accept particular data types, such as dates or numbers. This section describes what the conformance criteria for content in those formats is, and how to parse them.

Need to go through the whole spec and make sure all the attribute values are clearly defined either in terms of microsyntaxes or in terms of other specs, or as "Text" or some such.

2.4.1 Common parser idioms

The **space characters**, for the purposes of this specification, are U+0020 SPACE, U+0009 CHARACTER TABULATION (tab), U+000A LINE FEED (LF), U+000C FORM FEED (FF), and U+000D CARRIAGE RETURN (CR).

The **White_Space characters** are those that have the Unicode property "White_Space".

[UNICODE]

Some of the micro-parsers described below follow the pattern of having an *input* variable that holds the string being parsed, and having a *position* variable pointing at the next character to parse in *input*.

For parsers based on this pattern, a step that requires the user agent to **collect a sequence of characters** means that the following algorithm must be run, with *characters* being the set of characters that can be collected:

1. Let *input* and *position* be the same variables as those of the same name in the algorithm that invoked these steps.
2. Let *result* be the empty string.
3. While *position* doesn't point past the end of *input* and the character at *position* is one of the *characters*, append that character to the end of *result* and advance *position* to the next character in *input*.
4. Return *result*.

The step **skip whitespace** means that the user agent must collect a sequence of characters that are space characters. The step **skip White_Space characters** means that the user agent must collect a sequence of characters that are White_Space characters. In both cases, the collected characters are not used. [UNICODE]

When a user agent is to **strip line breaks** from a string, the user agent must remove any U+000A LINE FEED (LF) and U+000D CARRIAGE RETURN (CR) characters from that string.

The **codepoint length** of a string is the number of Unicode codepoints in that string.

2.4.2 Boolean attributes

A number of attributes in HTML5 are **boolean attributes**. The presence of a boolean attribute on an element represents the true value, and the absence of the attribute represents the false value.

If the attribute is present, its value must either be the empty string or a value that is an ASCII case-insensitive match for the attribute's canonical name, with no leading or trailing whitespace.

> ***Note: The values "true" and "false" are not allowed on boolean attributes. To represent a false value, the attribute has to be omitted altogether.***

2.4.3 Numbers

2.4.3.1 Non-negative integers

A string is a **valid non-negative integer** if it consists of one of more characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9).

A valid non-negative integer represents the number that is represented in base ten by that string of digits.

The **rules for parsing non-negative integers** are as given in the following algorithm. When invoked, the steps must be followed in the order given, aborting at the first step that returns a value. This algorithm will either return zero, a positive integer, or an error. Leading spaces are ignored. Trailing spaces and indeed any trailing garbage characters are ignored.

1. Let *input* be the string being parsed.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Let *value* have the value 0.
4. Skip whitespace.
5. If *position* is past the end of *input*, return an error.
6. If the next character is a U+002B PLUS SIGN character (+), advance *position* to the next character.
7. If *position* is past the end of *input*, return an error.
8. If the next character is not one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), then return an error.
9. If the next character is one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9):
    1. Multiply *value* by ten.
    2. Add the value of the current character (0..9) to *value*.
    3. Advance *position* to the next character.
    4. If *position* is not past the end of *input*, return to the top of step 7 in the overall algorithm (that's the step within which these substeps find themselves).
10. Return *value*.

2.4.3.2 Signed integers

A string is a **valid integer** if it consists of one of more characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), optionally prefixed with a U+002D HYPHEN-MINUS ("-") character.

A valid integer without a U+002D HYPHEN-MINUS ("-") prefix represents the number that is represented in base ten by that string of digits. A valid integer *with* a U+002D HYPHEN-MINUS ("-") prefix represents the number represented in base ten by the string of digits that follows the U+002D HYPHEN-MINUS, subtracted from zero.

The **rules for parsing integers** are similar to the rules for non-negative integers, and are as given in the following algorithm. When invoked, the steps must be followed in the order given,

aborting at the first step that returns a value. This algorithm will either return an integer or an error. Leading spaces are ignored. Trailing spaces and trailing garbage characters are ignored.

1. Let *input* be the string being parsed.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Let *value* have the value 0.
4. Let *sign* have the value "positive".
5. Skip whitespace.
6. If *position* is past the end of *input*, return an error.
7. If the character indicated by *position* (the first character) is a U+002D HYPHEN-MINUS ("-") character:
    1. Let *sign* be "negative".
    2. Advance *position* to the next character.
    3. If *position* is past the end of *input*, return an error.
8. If the next character is not one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), then return an error.
9. If the next character is one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9):
    1. Multiply *value* by ten.
    2. Add the value of the current character (0..9) to *value*.
    3. Advance *position* to the next character.
    4. If *position* is not past the end of *input*, return to the top of step 9 in the overall algorithm (that's the step within which these substeps find themselves).
10. If *sign* is "positive", return *value*, otherwise return 0-*value*.

2.4.3.3 Real numbers

A string is a **valid floating point number** if it consists of:

1. Optionally, a U+002D HYPHEN-MINUS ("-") character.
2. A series of one or more characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9).
3. Optionally:
    1. A single U+002E FULL STOP (".") character.
    2. A series of one or more characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9).
4. Optionally:
    1. Either a U+0065 LATIN SMALL LETTER E character or a U+0045 LATIN

CAPITAL LETTER E character.
2. Optionally, a U+002D HYPHEN-MINUS ("-") character or U+002B PLUS SIGN ("+") character.
3. A series of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9).

A valid floating point number represents the number obtained by multiplying the significand by ten raised to the power of the exponent, where the significand is the first number, interpreted as base ten (including the decimal point and the number after the decimal point, if any, and interpreting the significand as a negative number if the whole string starts with a U+002D HYPHEN-MINUS ("-") character and the number is not zero), and where the exponent is the number after the E, if any (interpreted as a negative number if there is a U+002D HYPHEN-MINUS ("-") character between the E and the number and the number is not zero, or else ignoring a U+002B PLUS SIGN ("+") character between the E and the number if there is one). If there is no E, then the exponent is treated as zero.

Note: The values ±Infinity and NaN are not valid floating point numbers.

The **best representation of the floating point number** *n* is the string obtained from applying the ECMAScript operator ToString to *n*.

The **rules for parsing floating point number values** are as given in the following algorithm. As with the previous algorithms, when this one is invoked, the steps must be followed in the order given, aborting at the first step that returns something. This algorithm will either return a number or an error. Leading spaces are ignored. Trailing spaces and garbage characters are ignored.

1. Let *input* be the string being parsed.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Let *value* have the value 1.
4. Let *divisor* have the value 1.
5. Let *exponent* have the value 1.
6. Skip whitespace.
7. If *position* is past the end of *input*, return an error.
8. If the character indicated by *position* is a U+002D HYPHEN-MINUS ("-") character:
    1. Change *value* and *divisor* to −1.
    2. Advance *position* to the next character.
    3. If *position* is past the end of *input*, return an error.
9. If the character indicated by *position* is not one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), then return an error.
10. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039

DIGIT NINE (9), and interpret the resulting sequence as a base-ten integer. Multiply *value* by that integer.

11. If *position* is past the end of *input*, return *value*.

12. If the character indicated by *position* is a U+002E FULL STOP ("."), run these substeps:

    1. Advance *position* to the next character.

    2. If *position* is past the end of *input*, or if the character indicated by *position* is not one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), then return *value*.

    3. *Fraction loop*: Multiply *divisor* by ten.

    4. Add the value of the current character interpreted as a base-ten digit (0..9) divided by *divisor*, to *value*.

    5. Advance *position* to the next character.

    6. If *position* is past the end of *input*, then return *value*.

    7. If the character indicated by *position* is one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), return to the step labeled *fraction loop* in these substeps.

13. If the character indicated by *position* is a U+0065 LATIN SMALL LETTER E character or a U+0045 LATIN CAPITAL LETTER E character, run these substeps:

    1. Advance *position* to the next character.

    2. If *position* is past the end of *input*, then return *value*.

    3. If the character indicated by *position* is a U+002D HYPHEN-MINUS ("-") character:

        1. Change *exponent* to −1.

        2. Advance *position* to the next character.

        3. If *position* is past the end of *input*, then return *value*.

        Otherwise, if the character indicated by *position* is a U+002B PLUS SIGN ("+") character:

        1. Advance *position* to the next character.

        2. If *position* is past the end of *input*, then return *value*.

    4. If the character indicated by *position* is not one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), then return *value*.

    5. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), and interpret the resulting sequence as a base-ten integer. Multiply *exponent* by that integer.

    6. Multiply *value* by ten raised to the *exponent*th power.

14. Return *value*.

2.4.3.4 Ratios

Note: The algorithms described in this section are used by the `progress` and `meter` elements.

A **valid denominator punctuation character** is one of the characters from the table below. There is **a value associated with each denominator punctuation character**, as shown in the table below.

Denominator Punctuation Character		Value
U+0025 PERCENT SIGN	%	100
U+066A ARABIC PERCENT SIGN	٪	100
U+FE6A SMALL PERCENT SIGN	﹪	100
U+FF05 FULLWIDTH PERCENT SIGN	％	100
U+2030 PER MILLE SIGN	‰	1000
U+2031 PER TEN THOUSAND SIGN	‱	10000

The **steps for finding one or two numbers of a ratio in a string** are as follows:

1. If the string is empty, then return nothing and abort these steps.
2. Find a number in the string according to the algorithm below, starting at the start of the string.
3. If the sub-algorithm in step 2 returned nothing or returned an error condition, return nothing and abort these steps.
4. Set *number1* to the number returned by the sub-algorithm in step 2.
5. Starting with the character immediately after the last one examined by the sub-algorithm in step 2, skip all White_Space characters in the string (this might match zero characters).
6. If there are still further characters in the string, and the next character in the string is a valid denominator punctuation character, set *denominator* to that character.
7. If the string contains any other characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE, but *denominator* was given a value in the step 6, return nothing and abort these steps.
8. Otherwise, if *denominator* was given a value in step 6, return *number1* and *denominator* and abort these steps.
9. Find a number in the string again, starting immediately after the last character that was examined by the sub-algorithm in step 2.
10. If the sub-algorithm in step 9 returned nothing or an error condition, return *number1* and

abort these steps.

11. Set *number2* to the number returned by the sub-algorithm in step 9.

12. Starting with the character immediately after the last one examined by the sub-algorithm in step 9, skip all White_Space characters in the string (this might match zero characters).

13. If there are still further characters in the string, and the next character in the string is a valid denominator punctuation character, return nothing and abort these steps.

14. If the string contains any other characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE, return nothing and abort these steps.

15. Otherwise, return *number1* and *number2*.

The algorithm to **find a number** is as follows. It is given a string and a starting position, and returns either nothing, a number, or an error condition.

1. Starting at the given starting position, ignore all characters in the given string until the first character that is either a U+002E FULL STOP or one of the ten characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE.

2. If there are no such characters, return nothing and abort these steps.

3. Starting with the character matched in step 1, collect all the consecutive characters that are either a U+002E FULL STOP or one of the ten characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE, and assign this string of one or more characters to *string*.

4. If *string* consists of just a single U+002E FULL STOP character or if it contains more than one U+002E FULL STOP character then return an error condition and abort these steps.

5. Parse *string* according to the rules for parsing floating point number values, to obtain *number*. This step cannot fail (*string* is guaranteed to be a valid floating point number).

6. Return *number*.

2.4.3.5 Percentages and lengths

The **rules for parsing dimension values** are as given in the following algorithm. When invoked, the steps must be followed in the order given, aborting at the first step that returns a value. This algorithm will either return a number greater than or equal to 1.0, or an error; if a number is returned, then it is further categorised as either a percentage or a length.

1. Let *input* be the string being parsed.

2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

3. Skip whitespace.

4. If *position* is past the end of *input*, return an error.

5. If the next character is a U+002B PLUS SIGN character (+), advance *position* to the next character.

6. Collect a sequence of characters that are U+0030 DIGIT ZERO (0) characters, and discard them.

7. If *position* is past the end of *input*, return an error.

8. If the next character is not one of U+0031 DIGIT ONE (1) .. U+0039 DIGIT NINE (9), then return an error.

9. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), and interpret the resulting sequence as a base-ten integer. Let *value* be that number.

10. If *position* is past the end of *input*, return *value* as an integer.

11. If the next character is a U+002E FULL STOP character (.):

    1. Advance *position* to the next character.

    2. If the next character is not one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), then return *value* as an integer.

    3. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). Let *length* be the number of characters collected. Let *fraction* be the result of interpreting the collected characters as a base-ten integer, and then dividing that number by 10^{length}.

    4. Increment *value* by *fraction*.

12. If *position* is past the end of *input*, return *value* as a length.

13. If the next character is a U+0025 PERCENT SIGN character (%), return *value* as a percentage.

14. Return *value* as a length.

2.4.3.6 Lists of integers

A **valid list of integers** is a number of valid integers separated by U+002C COMMA characters, with no other characters (e.g. no space characters). In addition, there might be restrictions on the number of integers that can be given, or on the range of values allowed.

The **rules for parsing a list of integers** are as follows:

1. Let *input* be the string being parsed.

2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

3. Let *numbers* be an initially empty list of integers. This list will be the result of this algorithm.

4. If there is a character in the string *input* at position *position*, and it is either a U+0020 SPACE, U+002C COMMA, or U+003B SEMICOLON character, then advance *position* to the next character in *input*, or to beyond the end of the string if there are no more characters.

5. If *position* points to beyond the end of *input*, return *numbers* and abort.

6. If the character in the string *input* at position *position* is a U+0020 SPACE, U+002C COMMA, or U+003B SEMICOLON character, then return to step 4.

7. Let *negated* be false.

8. Let *value* be 0.

9. Let *started* be false. This variable is set to true when the parser sees a number or a "-" character.

10. Let *got number* be false. This variable is set to true when the parser sees a number.

11. Let *finished* be false. This variable is set to true to switch parser into a mode where it ignores characters until the next separator.

12. Let *bogus* be false.

13. *Parser:* If the character in the string *input* at position *position* is:

    ↪ **A U+002D HYPHEN-MINUS character**
    Follow these substeps:

    1. If *got number* is true, let *finished* be true.
    2. If *finished* is true, skip to the next step in the overall set of steps.
    3. If *started* is true, let *negated* be false.
    4. Otherwise, if *started* is false and if *bogus* is false, let *negated* be true.
    5. Let *started* be true.

    ↪ **A character in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE**
    Follow these substeps:

    1. If *finished* is true, skip to the next step in the overall set of steps.
    2. Multiply *value* by ten.
    3. Add the value of the digit, interpreted in base ten, to *value*.
    4. Let *started* be true.
    5. Let *got number* be true.

    ↪ **A U+0020 SPACE character**
    ↪ **A U+002C COMMA character**
    ↪ **A U+003B SEMICOLON character**

Follow these substeps:

1. If *got number* is false, return the *numbers* list and abort. This happens if an entry in the list has no digits, as in "`1,2,x,4`".
2. If *negated* is true, then negate *value*.
3. Append *value* to the *numbers* list.
4. Jump to step 4 in the overall set of steps.

↪ **A character in the range U+0001 .. U+001F, U+0021 .. U+002B, U+002D .. U+002F, U+003A, U+003C .. U+0040, U+005B .. U+0060, U+007b .. U+007F (i.e. any other non-alphabetic ASCII character)**

Follow these substeps:

1. If *got number* is true, let *finished* be true.
2. If *finished* is true, skip to the next step in the overall set of steps.
3. Let *negated* be false.

↪ **Any other character**

Follow these substeps:

1. If *finished* is true, skip to the next step in the overall set of steps.
2. Let *negated* be false.
3. Let *bogus* be true.
4. If *started* is true, then return the *numbers* list, and abort. (The value in *value* is not appended to the list first; it is dropped.)

14. Advance *position* to the next character in *input*, or to beyond the end of the string if there are no more characters.
15. If *position* points to a character (and not to beyond the end of *input*), jump to the big *Parser* step above.
16. If *negated* is true, then negate *value*.
17. If *got number* is true, then append *value* to the *numbers* list.
18. Return the *numbers* list and abort.

2.4.3.7 Lists of dimensions

The **rules for parsing a list of dimensions** are as follows. These rules return a list of zero or more pairs consisting of a number and a unit, the unit being one of *percentage*, *relative*, and *absolute*.

1. Let *raw input* be the string being parsed.

2. If the last character in *raw input* is a U+002C COMMA character (","), then remove that character from *raw input*.

3. Split the string *raw input* on commas. Let *raw tokens* be the resulting list of tokens.

4. Let *result* be an empty list of number/unit pairs.

5. For each token in *raw tokens*, run the following substeps:

    1. Let *input* be the token.

    2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

    3. Let *value* be the number 0.

    4. Let *unit* be *absolute*.

    5. If *position* is past the end of *input*, set *unit* to *relative* and jump to the last substep.

    6. If the character at *position* is a character in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), interpret the resulting sequence as an integer in base ten, and increment *value* by that integer.

    7. If the character at *position* is a U+002E FULL STOP character (.), run these substeps:

        1. Collect a sequence of characters consisting of space characters and characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). Let *s* be the resulting sequence.

        2. Remove all space characters in *s*.

        3. If *s* is not the empty string, run these subsubsteps:

            1. Let *length* be the number of characters in *s* (after the spaces were removed).

            2. Let *fraction* be the result of interpreting *s* as a base-ten integer, and then dividing that number by 10^{length}.

            3. Increment *value* by *fraction*.

    8. Skip whitespace.

    9. If the character at *position* is a U+0025 PERCENT SIGN (%) character, then set *unit* to *percentage*.

        Otherwise, if the character at *position* is a U+002A ASTERISK character (*), then set *unit* to *relative*.

    10. Add an entry to *result* consisting of the number given by *value* and the unit given by *unit*.

6. Return the list *result*.

2.4.4 Dates and times

In the algorithms below, the **number of days in month *month* of year *year*** is: *31* if *month* is 1, 3, 5, 7, 8, 10, or 12; *30* if *month* is 4, 6, 9, or 11; *29* if *month* is 2 and *year* is a number divisible by 400, or if *year* is a number divisible by 4 but not by 100; and *28* otherwise. This takes into account leap years in the Gregorian calendar. [GREGORIAN]

The **digits** in the date and time syntaxes defined in this section must be characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE, used to express numbers in base ten.

2.4.4.1 Months

A **month** consists of a specific proleptic Gregorian date with no timezone information and no date information beyond a year and a month. [GREGORIAN]

A string is a **valid month string** representing a year *year* and month *month* if it consists of the following components in the given order:

1. Four or more digits, representing *year*, where *year* > 0
2. A U+002D HYPHEN-MINUS character (-)
3. Two digits, representing the month *month*, in the range 1 ≤ *month* ≤ 12

The rules to **parse a month string** are as follows. This will either return a year and month, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Let *input* be the string being parsed.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Parse a month component to obtain *year* and *month*. If this returns nothing, then fail.
4. If *position* is *not* beyond the end of *input*, then fail.
5. Return *year* and *month*.

The rules to **parse a month component**, given an *input* string and a *position*, are as follows. This will either return a year and a month, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not at least four characters long, then fail. Otherwise, interpret the resulting sequence as a base-ten integer. Let that number be the *year*.
2. If *year* is not a number greater than zero, then fail.
3. If *position* is beyond the end of *input* or if the character at *position* is not a U+002D

HYPHEN-MINUS character, then fail. Otherwise, move *position* forwards one character.

4. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not exactly two characters long, then fail. Otherwise, interpret the resulting sequence as a base-ten integer. Let that number be the *month*.

5. If *month* is not a number in the range 1 ≤ *month* ≤ 12, then fail.

6. Return *year* and *month*.

2.4.4.2 Dates

A **date** consists of a specific proleptic Gregorian date with no timezone information, consisting of a year, a month, and a day. [GREGORIAN]

A string is a **valid date string** representing a year *year*, month *month*, and day *day* if it consists of the following components in the given order:

1. A valid month string, representing *year* and *month*

2. A U+002D HYPHEN-MINUS character (-)

3. Two digits, representing *day*, in the range 1 ≤ *day* ≤ *maxday* where *maxday* is the number of days in the month *month* and year *year*

The rules to **parse a date string** are as follows. This will either return a date, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Let *input* be the string being parsed.

2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

3. Parse a date component to obtain *year*, *month*, and *day*. If this returns nothing, then fail.

4. If *position* is *not* beyond the end of *input*, then fail.

5. Let *date* be the date with year *year*, month *month*, and day *day*.

6. Return *date*.

The rules to **parse a date component**, given an *input* string and a *position*, are as follows. This will either return a year, a month, and a day, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Parse a month component to obtain *year* and *month*. If this returns nothing, then fail.

2. Let *maxday* be the number of days in month *month* of year *year*.

3. If *position* is beyond the end of *input* or if the character at *position* is not a U+002D HYPHEN-MINUS character, then fail. Otherwise, move *position* forwards one character.

4. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not exactly two characters long, then fail. Otherwise, interpret the resulting sequence as a base-ten integer. Let that number be the *day*.

5. If *day* is not a number in the range 1 ≤ *month* ≤ *maxday*, then fail.

6. Return *year*, *month*, and *day*.

2.4.4.3 Times

A **time** consists of a specific time with no timezone information, consisting of an hour, a minute, a second, and a fraction of a second.

A string is a **valid time string** representing an hour *hour*, a minute *minute*, and a second *second* if it consists of the following components in the given order:

1. Two digits, representing *hour*, in the range 0 ≤ *hour* ≤ 23

2. A U+003A COLON character (:)

3. Two digits, representing *minute*, in the range 0 ≤ *minute* ≤ 59

4. Optionally (required if *second* is non-zero):
    1. A U+003A COLON character (:)
    2. Two digits, representing the integer part of *second*, in the range 0 ≤ *s* ≤ 59
    3. Optionally (required if *second* is not an integer):
        1. A 002E FULL STOP character (.)
        2. One or more digits, representing the fractional part of *second*

Note: The second component cannot be 60 or 61; leap seconds cannot be represented.

The rules to **parse a time string** are as follows. This will either return a time, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Let *input* be the string being parsed.

2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

3. Parse a time component to obtain *hour*, *minute*, and *second*. If this returns nothing, then fail.

4. If *position* is *not* beyond the end of *input*, then fail.

5. Let *time* be the time with hour *hour*, minute *minute*, and second *second*.

6. Return *time*.

The rules to **parse a time component**, given an *input* string and a *position*, are as follows. This will either return an hour, a minute, and a second, or nothing. If at any point the algorithm

says that it "fails", this means that it is aborted at that point and returns nothing.

1. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not exactly two characters long, then fail. Otherwise, interpret the resulting sequence as a base-ten integer. Let that number be the *hour*.

2. If *hour* is not a number in the range 0 ≤ *hour* ≤ 23, then fail.

3. If *position* is beyond the end of *input* or if the character at *position* is not a U+003A COLON character, then fail. Otherwise, move *position* forwards one character.

4. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not exactly two characters long, then fail. Otherwise, interpret the resulting sequence as a base-ten integer. Let that number be the *minute*.

5. If *minute* is not a number in the range 0 ≤ *minute* ≤ 59, then fail.

6. Let *second* be a string with the value "0".

7. If *position* is not beyond the end of *input* and the character at *position* is a U+003A COLON, then run these substeps:

    1. Advance *position* to the next character in *input*.

    2. If *position* is beyond the end of *input*, or at the last character in *input*, or if the next *two* characters in *input* starting at *position* are not two characters both in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), then fail.

    3. Collect a sequence of characters that are either characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9) or U+002E FULL STOP characters. If the collected sequence has more than one U+002E FULL STOP characters, or if the last character in the sequence is a U+002E FULL STOP character, then fail. Otherwise, let the collected string be *second* instead of its previous value.

8. Interpret *second* as a base-ten number (possibly with a fractional part). Let *second* be that number instead of the string version.

9. If *second* is not a number in the range 0 ≤ *second* < 60, then fail.

10. Return *hour*, *minute*, and *second*.

2.4.4.4 Local dates and times

A **local date and time** consists of a specific proleptic Gregorian date, consisting of a year, a month, and a day, and a time, consisting of an hour, a minute, a second, and a fraction of a second, but expressed without a time zone. [GREGORIAN]

A string is a **valid local date and time string** representing a date and time if it consists of the following components in the given order:

1. A valid date string representing the date.

2. A U+0054 LATIN CAPITAL LETTER T character.

3. A valid time string representing the time.

The rules to **parse a local date and time string** are as follows. This will either return a date and time, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Let *input* be the string being parsed.

2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

3. Parse a date component to obtain *year*, *month*, and *day*. If this returns nothing, then fail.

4. If *position* is beyond the end of *input* or if the character at *position* is not a U+0054 LATIN CAPITAL LETTER T character then fail. Otherwise, move *position* forwards one character.

5. Parse a time component to obtain *hour*, *minute*, and *second*. If this returns nothing, then fail.

6. If *position* is *not* beyond the end of *input*, then fail.

7. Let *date* be the date with year *year*, month *month*, and day *day*.

8. Let *time* be the time with hour *hour*, minute *minute*, and second *second*.

9. Return *date* and *time*.

2.4.4.5 Global dates and times

A **global date and time** consists of a specific proleptic Gregorian date, consisting of a year, a month, and a day, and a time, consisting of an hour, a minute, a second, and a fraction of a second, expressed with a time zone, consisting of a number of hours and minutes. [GREGORIAN]

A string is a **valid global date and time string** representing a date, time, and a timezone offset if it consists of the following components in the given order:

1. A valid date string representing the date

2. A U+0054 LATIN CAPITAL LETTER T character

3. A valid time string representing the time

4. Either:
   - A U+005A LATIN CAPITAL LETTER Z character, allowed only if the time zone is UTC
   - Or:
     1. Either a U+002B PLUS SIGN character (+) or a U+002D HYPHEN-MINUS (-) character, representing the sign of the timezone offset
     2. Two digits, representing the hours component *hour* of the timezone offset, in the range $0 \leq hour \leq 23$

3. A U+003A COLON character (:)
4. Two digits, representing the minutes component *minute* of the timezone offset, in the range 0 ≤ *minute* ≤ 59

Note: This format allows for time zone offsets from -23:59 to +23:59. In practice, however, the range of actual time zones is -12:00 to +14:00, and the minutes component of actual time zones is always either 00, 30, or 45.

The following are some examples of dates written as valid global date and time strings.

"0037-12-13T00:00Z"
Midnight UTC on the birthday of Nero (the Roman Emperor).

"1979-10-14T12:00:00.001-04:00"
One millisecond after noon on October 14th 1979, in the time zone in use on the east coast of North America during daylight saving time.

"8592-01-01T02:09+02:09"
Midnight UTC on the 1st of January, 8592. The time zone associated with that time is two hours and nine minutes ahead of UTC, which is not a real time zone currently, but is nonetheless allowed.

Several things are notable about these dates:

- Years with fewer than four digits have to be zero-padded. The date "37-12-13" would not be a valid date.
- To unambiguously identify a moment in time prior to the introduction of the Gregorian calendar, the date has to be first converted to the Gregorian calendar from the calendar in use at the time (e.g. from the Julian calendar). The date of Nero's birth is the 15th of December 37, in the Julian Calendar, which is the 13th of December 37 in the proleptic Gregorian Calendar.
- The time and timezone components are not optional.
- Dates before the year zero can't be represented as a datetime in this version of HTML.
- Time zones differ based on daylight savings time.

The rules to **parse a global date and time string** are as follows. This will either return a time in UTC, with associated timezone information for round tripping or display purposes, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Let *input* be the string being parsed.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Parse a date component to obtain *year*, *month*, and *day*. If this returns nothing, then fail.
4. If *position* is beyond the end of *input* or if the character at *position* is not a U+0054 LATIN CAPITAL LETTER T character then fail. Otherwise, move *position* forwards one character.

5. Parse a time component to obtain *hour*, *minute*, and *second*. If this returns nothing, then fail.

6. If *position* is beyond the end of *input*, then fail.

7. Parse a timezone component to obtain $timezone_{hours}$ and $timezone_{minutes}$. If this returns nothing, then fail.

8. If *position* is *not* beyond the end of *input*, then fail.

9. Let *time* be the moment in time at year *year*, month *month*, day *day*, hours *hour*, minute *minute*, second *second*, subtracting $timezone_{hours}$ hours and $timezone_{minutes}$ minutes. That moment in time is a moment in the UTC timezone.

10. Let *timezone* be $timezone_{hours}$ hours and $timezone_{minutes}$ minutes from UTC.

11. Return *time* and *timezone*.

The rules to **parse a timezone component**, given an *input* string and a *position*, are as follows. This will either return timezone hours and timezone minutes, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. If the character at *position* is a U+005A LATIN CAPITAL LETTER Z, then:

    1. Let $timezone_{hours}$ be 0.

    2. Let $timezone_{minutes}$ be 0.

    3. Advance *position* to the next character in *input*.

    Otherwise, if the character at *position* is either a U+002B PLUS SIGN ("+") or a U+002D HYPHEN-MINUS ("-"), then:

    1. If the character at *position* is a U+002B PLUS SIGN ("+"), let *sign* be "positive". Otherwise, it's a U+002D HYPHEN-MINUS ("-"); let *sign* be "negative".

    2. Advance *position* to the next character in *input*.

    3. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not exactly two characters long, then fail. Otherwise, interpret the resulting sequence as a base-ten integer. Let that number be the $timezone_{hours}$.

    4. If $timezone_{hours}$ is not a number in the range $0 \leq timezone_{hours} \leq 23$, then fail.

    5. If *sign* is "negative", then negate $timezone_{hours}$.

    6. If *position* is beyond the end of *input* or if the character at *position* is not a U+003A COLON character, then fail. Otherwise, move *position* forwards one character.

    7. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not exactly two characters long, then fail. Otherwise, interpret the resulting sequence as a base-ten integer. Let that

number be the *timezone*$_{minutes}$.

8. If *timezone*$_{minutes}$ is not a number in the range 0 ≤ *timezone*$_{minutes}$ ≤ 59, then fail.

9. If *sign* is "negative", then negate *timezone*$_{minutes}$.

2. Return *timezone*$_{hours}$ and *timezone*$_{minutes}$.

2.4.4.6 Weeks

A **week** consists of a week-year number and a week number representing a seven day period. Each week-year in this calendaring system has either 52 weeks or 53 weeks, as defined below. A week is a seven-day period. The week starting on the Gregorian date Monday December 29th 1969 (1969-12-29) is defined as week number 1 in week-year 1970. Consecutive weeks are numbered sequentially. The week before the number 1 week in a week-year is the last week in the previous week-year, and vice versa. [GREGORIAN]

A week-year with a number *year* that corresponds to a year *year* in the proleptic Gregorian calendar that has a Thursday as its first day (January 1st), and a week-year *year* where *year* is a number divisible by 400, or a number divisible by 4 but not by 100, has 53 weeks. All other week-years have 52 weeks.

The **week number of the last day** of a week-year with 53 weeks is 53; the week number of the last day of a week-year with 52 weeks is 52.

> ***Note: The week-year number of a particular day can be different than the number of the year that contains that day in the proleptic Gregorian calendar. The first week in a week-year year is the week that contains the first Thursday of the Gregorian year year.***

A string is a **valid week string** representing a week-year *year* and week *week* if it consists of the following components in the given order:

1. Four or more digits, representing *year*, where *year* > 0

2. A U+002D HYPHEN-MINUS character (-)

3. A U+0057 LATIN CAPITAL LETTER W character

4. Two digits, representing the week *week*, in the range 1 ≤ *week* ≤ *maxweek*, where *maxweek* is the week number of the last day of week-year *year*

The rules to **parse a week string** are as follows. This will either return a week-year number and week number, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Let *input* be the string being parsed.

2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

3. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not at least four characters long, then fail.

Otherwise, interpret the resulting sequence as a base-ten integer. Let that number be the *year*.

4. If *year* is not a number greater than zero, then fail.

5. If *position* is beyond the end of *input* or if the character at *position* is not a U+002D HYPHEN-MINUS character, then fail. Otherwise, move *position* forwards one character.

6. If *position* is beyond the end of *input* or if the character at *position* is not a U+0057 LATIN CAPITAL LETTER W character, then fail. Otherwise, move *position* forwards one character.

7. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not exactly two characters long, then fail. Otherwise, interpret the resulting sequence as a base-ten integer. Let that number be the *week*.

8. Let *maxweek* be the week number of the last day of year *year*.

9. If *week* is not a number in the range 1 ≤ *week* ≤ *maxweek*, then fail.

10. If *position* is *not* beyond the end of *input*, then fail.

11. Return the week-year number *year* and the week number *week*.

2.4.4.7 Vaguer moments in time

A **date or time string** consists of either a date, a time, or a global date and time.

A string is a **valid date or time string** if it is also one of the following:

- A valid date string.
- A valid time string.
- A valid global date and time string.

A string is a **valid date or time string in content** if it consists of zero or more White_Space characters, followed by a valid date or time string, followed by zero or more further White_Space characters.

The rules to **parse a date or time string** are as follows. The algorithm is invoked with a flag indicating if the *in attribute* variant or the *in content* variant is to be used. The algorithm will either return a date, a time, a global date and time, or nothing. If at any point the algorithm says that it "fails", this means that it is aborted at that point and returns nothing.

1. Let *input* be the string being parsed.

2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

3. For the *in content* variant: skip White_Space characters.

4. Set *start position* to the same position as *position*.

5. Set the *date present* and *time present* flags to true.

6. Parse a date component to obtain *year*, *month*, and *day*. If this fails, then set the *date present* flag to false.

7. If *date present* is true, and *position* is not beyond the end of *input*, and the character at *position* is a U+0054 LATIN CAPITAL LETTER T character, then advance *position* to the next character in *input*.

   Otherwise, if *date present* is true, and either *position* is beyond the end of *input* or the character at *position* is not a U+0054 LATIN CAPITAL LETTER T character, then set *time present* to false.

   Otherwise, if *date present* is false, set *position* back to the same position as *start position*.

8. If the *time present* flag is true, then parse a time component to obtain *hour*, *minute*, and *second*. If this returns nothing, then set the *time present* flag to false.

9. If both the *date present* and *time present* flags are false, then fail.

10. If the *time present* flag is true, but *position* is beyond the end of *input*, then fail.

11. If the *date present* and *time present* flags are both true, parse a timezone component to obtain $timezone_{hours}$ and $timezone_{minutes}$. If this returns nothing, then fail.

12. For the *in content* variant: skip White_Space characters.

13. If *position* is *not* beyond the end of *input*, then fail.

14. If the *date present* flag is true and the *time present* flag is false, then let *date* be the date with year *year*, month *month*, and day *day*, and return *date*.

    Otherwise, if the *time present* flag is true and the *date present* flag is false, then let *time* be the time with hour *hour*, minute *minute*, and second *second*, and return *time*.

    Otherwise, let *time* be the moment in time at year *year*, month *month*, day *day*, hours *hour*, minute *minute*, second *second*, subtracting $timezone_{hours}$ hours and $timezone_{minutes}$ minutes, that moment in time being a moment in the UTC timezone; let *timezone* be $timezone_{hours}$ hours and $timezone_{minutes}$ minutes from UTC; and return *time* and *timezone*.

2.4.5 Colors

A **simple color** consists of three 8-bit numbers in the range 0..255, representing the red, green, and blue components of the color respectively, in the sRGB color space. [SRGB]

A string is a **valid simple color** if it is exactly seven characters long, and the first character is a U+0023 NUMBER SIGN (#) character, and the remaining six characters are all in the range U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F, U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F, with the first two digits representing the red component, the middle two

digits representing the green component, and the last two digits representing the blue component, in hexadecimal.

A string is a **valid lowercase simple color** if it is a valid simple color and doesn't use any characters in the range U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F.

The **rules for parsing simple color values** are as given in the following algorithm. When invoked, the steps must be followed in the order given, aborting at the first step that returns a value. This algorithm will either return a simple color or an error.

1. Let *input* be the string being parsed.
2. If *input* is not exactly seven characters long, then return an error.
3. If the first character in *input* is not a U+0023 NUMBER SIGN (#) character, then return an error.
4. If the last six characters of *input* are not all in the range U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F, U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F, then return an error.
5. Let *result* be a simple color.
6. Interpret the second and third characters as a hexadecimal number and let the result be the red component of *result*.
7. Interpret the fourth and fifth characters as a hexadecimal number and let the result be the green component of *result*.
8. Interpret the sixth and seventh characters as a hexadecimal number and let the result be the blue component of *result*.
9. Return *result*.

The **rules for serialising simple color values** given a simple color are as given in the following algorithm:

1. Let *result* be a string consisting of a single U+0023 NUMBER SIGN (#) character.
2. Convert the red, green, and blue components in turn to two-digit hexadecimal numbers using the digits U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9) and U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F, zero-padding if necessary, and append these numbers to *result*, in the order red, green, blue.
3. Return *result*, which will be a valid lowercase simple color.

Some obsolete legacy attributes parse colors in a more complicated manner, using the **rules for parsing a legacy color value**, which are given in the following algorithm. When invoked, the steps must be followed in the order given, aborting at the first step that returns a value. This algorithm will either return a simple color or an error.

1. Let *input* be the string being parsed.

2. If *input* is the empty string, then return an error.

3. If *input* is an ASCII case-insensitive match for the string "`transparent`", then return an error.

4. If *input* is an ASCII case-insensitive match for one of the keywords listed in the SVG color keywords or CSS2 System Colors sections of the CSS3 Color specification, then return the simple color corresponding to that keyword. [CSS3COLOR]

5. If *input* is four characters long, and the first character in *input* is a U+0023 NUMBER SIGN (#) character, and the the last three characters of *input* are all in the range U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F, and U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F, then run these substeps:

    1. Let *result* be a simple color.

    2. Interpret the second character of *input* as a hexadecimal digit; let the red component of *result* be the resulting number multiplied by 17.

    3. Interpret the third character of *input* as a hexadecimal digit; let the green component of *result* be the resulting number multiplied by 17.

    4. Interpret the fourth character of *input* as a hexadecimal digit; let the blue component of *result* be the resulting number multiplied by 17.

    5. Return *result*.

6. Replace any characters in *input* that have a Unicode codepoint greater than U+FFFF (i.e. any characters that are not in the basic multilingual plane) with the two-character string "`00`".

7. If *input* is longer than 128 characters, truncate *input*, leaving only the first 128 characters.

8. If the first character in *input* is a U+0023 NUMBER SIGN character (#), remove it.

9. Replace any character in *input* that is not in the range U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F, and U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F with the character U+0030 DIGIT ZERO (0).

10. While *input*'s length is zero or not a multiple of three, append a U+0030 DIGIT ZERO (0) character to *input*.

11. Split *input* into three strings of equal length, to obtain three components. Let *length* be the length of those components (one third the length of *input*).

12. If *length* is greater than 8, then remove the leading *length*-8 characters in each component, and let *length* be 8.

13. While *length* is greater than two and the first character in each component is a U+0030

DIGIT ZERO (0) character, remove that character and reduce *length* by one.

14. If *length* is *still* greater than two, truncate each component, leaving only the first two characters in each.

15. Let *result* be a simple color.

16. Interpret the first component as a hexadecimal number; let the red component of *result* be the resulting number.

17. Interpret the second component as a hexadecimal number; let the green component of *result* be the resulting number.

18. Interpret the third component as a hexadecimal number; let the blue component of *result* be the resulting number.

19. Return *result*.

Note: The 2D graphics context has a separate color syntax that also handles opacity.

2.4.6 Space-separated tokens

A **set of space-separated tokens** is a set of zero or more words separated by one or more space characters, where words consist of any string of one or more characters, none of which are space characters.

A string containing a set of space-separated tokens may have leading or trailing space characters.

An **unordered set of unique space-separated tokens** is a set of space-separated tokens where none of the words are duplicated.

An **ordered set of unique space-separated tokens** is a set of space-separated tokens where none of the words are duplicated but where the order of the tokens is meaningful.

Sets of space-separated tokens sometimes have a defined set of allowed values. When a set of allowed values is defined, the tokens must all be from that list of allowed values; other values are non-conforming. If no such set of allowed values is provided, then all values are conforming.

When a user agent has to **split a string on spaces**, it must use the following algorithm:

1. Let *input* be the string being parsed.

2. Let *position* be a pointer into *input*, initially pointing at the start of the string.

3. Let *tokens* be a list of tokens, initially empty.

4. Skip whitespace

5. While *position* is not past the end of *input*:
    1. Collect a sequence of characters that are not space characters.
    2. Add the string collected in the previous step to *tokens*.
    3. Skip whitespace
6. Return *tokens*.

When a user agent has to **remove a token from a string**, it must use the following algorithm:

1. Let *input* be the string being modified.
2. Let *token* be the token being removed. It will not contain any space characters.
3. Let *output* be the output string, initially empty.
4. Let *position* be a pointer into *input*, initially pointing at the start of the string.
5. If *position* is beyond the end of *input*, set the string being modified to *output*, and abort these steps.
6. If the character at *position* is a space character:
    1. Append the character at *position* to the end of *output*.
    2. Increment *position* so it points at the next character in *input*.
    3. Return to step 5 in the overall set of steps.
7. Otherwise, the character at *position* is the first character of a token. Collect a sequence of characters that are not space characters, and let that be *s*.
8. If *s* is exactly equal to *token*, then:
    1. Skip whitespace (in *input*).
    2. Remove any space characters currently at the end of *output*.
    3. If *position* is not past the end of *input*, and *output* is not the empty string, append a single U+0020 SPACE character at the end of *output*.
9. Otherwise, append *s* to the end of *output*.
10. Return to step 6 in the overall set of steps.

Note: This causes any occurrences of the token to be removed from the string, and any spaces that were surrounding the token to be collapsed to a single space, except at the start and end of the string, where such spaces are removed.

2.4.7 Comma-separated tokens

A **set of comma-separated tokens** is a set of zero or more tokens each separated from the next by a single U+002C COMMA character (,), where tokens consist of any string of zero or more characters, neither beginning nor ending with space characters, nor containing any U+002C COMMA characters (,), and optionally surrounded by space characters.

> For instance, the string " `a ,b,,d d` " consists of four tokens: "a", "b", the empty string, and "d d". Leading and trailing whitespace around each token doesn't count as part of the token, and the empty string can be a token.

Sets of comma-separated tokens sometimes have further restrictions on what consists a valid token. When such restrictions are defined, the tokens must all fit within those restrictions; other values are non-conforming. If no such restrictions are specified, then all values are conforming.

When a user agent has to **split a string on commas**, it must use the following algorithm:

1. Let *input* be the string being parsed.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Let *tokens* be a list of tokens, initially empty.
4. *Token*: If *position* is past the end of *input*, jump to the last step.
5. Collect a sequence of characters that are not U+002C COMMA characters (,). Let *s* be the resulting sequence (which might be the empty string).
6. Remove any leading or trailing sequence of space characters from *s*.
7. Add *s* to *tokens*.
8. If *position* is not past the end of *input*, then the character at *position* is a U+002C COMMA character (,); advance *position* past that character.
9. Jump back to the step labeled *token*.
10. Return *tokens*.

2.4.8 Keywords and enumerated attributes

Some attributes are defined as taking one of a finite set of keywords. Such attributes are called **enumerated attributes**. The keywords are each defined to map to a particular *state* (several keywords might map to the same state, in which case some of the keywords are synonyms of each other; additionally, some of the keywords can be said to be non-conforming, and are only in the specification for historical reasons). In addition, two default states can be given. The first is the *invalid value default*, the second is the *missing value default*.

If an enumerated attribute is specified, the attribute's value must be an ASCII case-insensitive match for one of the given keywords that are not said to be non-conforming, with no leading or trailing whitespace.

When the attribute is specified, if its value is an ASCII case-insensitively match for one of the given keywords then that keyword's state is the state that the attribute represents. If the attribute value matches none of the given keywords, but the attribute has an *invalid value default*, then the attribute represents that state. Otherwise, if the attribute value matches none of the keywords but there is a *missing value default* state defined, then *that* is the state represented by the attribute. Otherwise, there is no default, and invalid values must be ignored.

When the attribute is *not* specified, if there is a *missing value default* state defined, then that is the state represented by the (missing) attribute. Otherwise, the absence of the attribute means that there is no state represented.

> ***Note: The empty string can be one of the keywords in some cases. For example the `contenteditable` attribute has two states: true, matching the `true` keyword and the empty string, false, matching `false` and all other keywords (it's the invalid value default). It could further be thought of as having a third state inherit, which would be the default when the attribute is not specified at all (the missing value default), but for various reasons that isn't the way this specification actually defines it.***

2.4.9 References

A **valid hash-name reference** to an element of type *type* is a string consisting of a U+0023 NUMBER SIGN (`#`) character followed by a string which exactly matches the value of the `name` attribute of an element in the document with type *type*.

The **rules for parsing a hash-name reference** to an element of type *type* are as follows:

1. If the string being parsed does not contain a U+0023 NUMBER SIGN character, or if the first such character in the string is the last character in the string, then return null and abort these steps.

2. Let *s* be the string from the character immediately after the first U+0023 NUMBER SIGN character in the string being parsed up to the end of that string.

3. Return the first element of type *type* that has an `id` attribute whose value is a case-sensitive match for *s* or a `name` attribute whose value is a compatibility caseless match for *s*.

2.5 URLs

This specification defines the term URL, and defines various algorithms for dealing with URLs, because for historical reasons the rules defined by the URI and IRI specifications are not a complete description of what HTML user agents need to implement to be compatible with Web content.

2.5.1 Terminology

A **URL** is a string used to identify a resource.

A URL is a **valid URL** if at least one of the following conditions holds:

- The URL is a valid URI reference [RFC3986].
- The URL is a valid IRI reference and it has no query component. [RFC3987]
- The URL is a valid IRI reference and its query component contains no unescaped non-ASCII characters. [RFC3987]
- The URL is a valid IRI reference and the character encoding of the URL's `Document` is UTF-8 or UTF-16. [RFC3987]

A URL has an associated **URL character encoding**, determined as follows:

↪ **If the URL came from a script (e.g. as an argument to a method)**
The URL character encoding is the script's URL character encoding.

↪ **If the URL came from a DOM node (e.g. from an element)**
The node has a `Document`, and the URL character encoding is the document's character encoding.

↪ **If the URL had a character encoding defined when the URL was created or defined**
The URL character encoding is as defined.

Note: The term "URL" in this specification is used in a manner distinct from the precise technical meaning it is given in RFC 3986. Readers familiar with that RFC will find it easier to read this specification if they pretend the term "URL" as used herein is really called something else altogether.

2.5.2 Parsing URLs

To **parse a URL** *url* into its component parts, the user agent must use the following steps:

1. Strip leading and trailing space characters from *url*.
2. Parse *url* in the manner defined by RFC 3986, with the following exceptions:
   - Add all characters with codepoints less than or equal to U+0020 or greater than or equal to U+007F to the <unreserved> production.
   - Add the characters U+0022, U+003C, U+003E, U+005B .. U+005E, U+0060, and U+007B .. U+007D to the <unreserved> production.
   - Add a single U+0025 PERCENT SIGN character as a second alternative way of matching the <pct-encoded> production, except when the <pct-encoded> is used in the <reg-name> production.
   - Add the U+0023 NUMBER SIGN character to the characters allowed in the <fragment> production.
3. If *url* doesn't match the <URI-reference> production, even after the above changes are

made to the ABNF definitions, then parsing the URL fails with an error. [RFC3986]

Otherwise, parsing *url* was successful; the components of the URL are substrings of *url* defined as follows:

<scheme>
The substring matched by the <scheme> production, if any.

<host>
The substring matched by the <host> production, if any.

<port>
The substring matched by the <port> production, if any.

<hostport>
If there is a <scheme> component and a <port> component and the port given by the <port> component is different than the default port defined for the protocol given by the <scheme> component, then <hostport> is the substring that starts with the substring matched by the <host> production and ends with the substring matched by the <port> production, and includes the colon in between the two. Otherwise, it is the same as the <host> component.

<path>
The substring matched by one of the following productions, if one of them was matched:

- <path-abempty>
- <path-absolute>
- <path-noscheme>
- <path-rootless>
- <path-empty>

<query>
The substring matched by the <query> production, if any.

<fragment>
The substring matched by the <fragment> production, if any.

<host-specific>
The substring that *follows* the substring matched by the <authority> production, or the whole string if the <authority> production wasn't matched.

2.5.3 Resolving URLs

To **resolve a URL** to an absolute URL relative to either another absolute URL or an element, the user agent must use the following steps. Resolving a URL can result in an error, in which case the URL is not resolvable.

1. Let *url* be the URL being resolved.

2. Let *encoding* be the URL character encoding.

3. If *encoding* is UTF-16, then change it to UTF-8.

4. If the algorithm was invoked with an absolute URL to use as the base URL, let *base* be that absolute URL.

   Otherwise, let *base* be the *base URI of the element*, as defined by the XML Base specification, with *the base URI of the document entity* being defined as the document base URL of the `Document` that owns the element. [XMLBASE]

   For the purposes of the XML Base specification, user agents must act as if all `Document` objects represented XML documents.

   > ***Note: It is possible for `xml:base` attributes to be present even in HTML fragments, as such attributes can be added dynamically using script. (Such scripts would not be conforming, however, as `xml:base` attributes are not allowed in HTML documents.)***

   The **document base URL** of a `Document` is the absolute URL obtained by running these substeps:

   1. Let *fallback base url* be the document's address.

   2. If *fallback base url* is `about:blank`, and the `Document`'s browsing context has a creator browsing context, then let *fallback base url* be the document base URL of the creator `Document` instead.

   3. If there is no `base` element that is both a child of the `head` element and has an `href` attribute, then the document base URL is *fallback base url*.

   4. Otherwise, let *url* be the value of the `href` attribute of the first such element.

   5. Resolve *url* relative to *fallback base url* (thus, the `base` `href` attribute isn't affected by `xml:base` attributes).

   6. The document base URL is the result of the previous step if it was successful; otherwise it is *fallback base url*.

5. Parse *url* into its component parts.

6. If parsing *url* resulted in a <host> component, then replace the matching subtring of *url* with the string that results from expanding any sequences of percent-encoded octets in that component that are valid UTF-8 sequences into Unicode characters as defined by UTF-8.

   If any percent-encoded octets in that component are not valid UTF-8 sequences, then return an error and abort these steps.

   Apply the IDNA ToASCII algorithm to the matching substring, with both the AllowUnassigned and UseSTD3ASCIIRules flags set. Replace the matching substring with the result of the ToASCII algorithm.

   If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, then return an error and abort these steps. [RFC3490]

7. If parsing *url* resulted in a <path> component, then replace the matching substring of *url* with the string that results from applying the following steps to each character other than U+0025 PERCENT SIGN (%) that doesn't match the original <path> production defined in RFC 3986:

    1. Encode the character into a sequence of octets as defined by UTF-8.

    2. Replace the character with the percent-encoded form of those octets. [RFC3986]

    > For instance if *url* was "`//example.com/a^b☺c%FFd%z/?e`", then the <path> component's substring would be "`/a^b☺c%FFd%z/`" and the two characters that would have to be escaped would be "`^`" and "`☺`". The result after this step was applied would therefore be that *url* now had the value "`//example.com/a%5Eb%E2%98%BAc%FFd%z/?e`".

8. If parsing *url* resulted in a <query> component, then replace the matching substring of *url* with the string that results from applying the following steps to each character other than U+0025 PERCENT SIGN (%) that doesn't match the original <query> production defined in RFC 3986:

    1. If the character in question cannot be expressed in the encoding *encoding*, then replace it with a single 0x3F octet (an ASCII question mark) and skip the remaining substeps for this character.

    2. Encode the character into a sequence of octets as defined by the encoding *encoding*.

    3. Replace the character with the percent-encoded form of those octets. [RFC3986]

9. Apply the algorithm described in RFC 3986 section 5.2 Relative Resolution, using *url* as the potentially relative URI reference (*R*), and *base* as the base URI (*Base*). [RFC3986]

10. Apply any relevant conformance criteria of RFC 3986 and RFC 3987, returning an error and aborting these steps if appropriate. [RFC3986] [RFC3987]

    > For instance, if an absolute URI that would be returned by the above algorithm violates the restrictions specific to its scheme, e.g. a `data:` URI using the "`//`" server-based naming authority syntax, then user agents are to treat this as an error instead.

11. Let *result* be the target URI (*T*) returned by the Relative Resolution algorithm.

12. If *result* uses a scheme with a server-based naming authority, replace all U+005C REVERSE SOLIDUS (\) characters in *result* with U+002F SOLIDUS (/) characters.

13. Return *result*.

A URL is an **absolute URL** if resolving it results in the same URL without an error.

2.5.4 Dynamic changes to base URLs

When an `xml:base` attribute changes, the attribute's element, and all descendant elements, are affected by a base URL change.

When a document's document base URL changes, all elements in that document are affected by a base URL change.

When an element is moved from one document to another, if the two documents have different base URLs, then that element and all its descendants are affected by a base URL change.

When an element is **affected by a base URL change**, it must act as described in the following list:

↪ **If the element is a hyperlink element**
If the absolute URL identified by the hyperlink is being shown to the user, or if any data derived from that URL is affecting the display, then the `href` attribute should be reresolved relative to the element and the UI updated appropriately.

> For example, the CSS `:link`/`:visited` pseudo-classes might have been affected.

If the hyperlink has a `ping` attribute and its absolute URL(s) are being shown to the user, then the `ping` attribute's tokens should be reresolved relative to the element and the UI updated appropriately.

↪ **If the element is a `blockquote`, `q`, `ins`, or `del` element with a `cite` attribute**
If the absolute URL identified by the `cite` attribute is being shown to the user, or if any data derived from that URL is affecting the display, then the URL should be reresolved relative to the element and the UI updated appropriately.

↪ **Otherwise**
The element is not directly affected.

> Changing the base URL doesn't affect the image displayed by `img` elements, although subsequent accesses of the `src` DOM attribute from script will return a new absolute URL that might no longer correspond to the image being shown.

2.5.5 Interfaces for URL manipulation

An interface that has a complement of **URL decomposition attributes** will have seven attributes with the following definitions:

```
           attribute DOMString protocol;
           attribute DOMString host;
           attribute DOMString hostname;
           attribute DOMString port;
           attribute DOMString pathname;
           attribute DOMString search;
           attribute DOMString hash;
```

o.`protocol` [= *value*]

Returns the current scheme of the underlying URL.

Can be set, to change the underlying URL's scheme.

o.`host` [= *value*]

Returns the current host and port (if it's not the default port) in the underlying URL.

Can be set, to change the underlying URL's host and port.

The host and the port are separated by a colon. The port part, if omitted, will be assumed to be the current scheme's default port.

o.`hostname` [= *value*]

Returns the current host in the underlying URL.

Can be set, to change the underlying URL's host.

o.`port` [= *value*]

Returns the current port in the underlying URL.

Can be set, to change the underlying URL's port.

o.`pathname` [= *value*]

Returns the current path in the underlying URL.

Can be set, to change the underlying URL's path.

o.`search` [= *value*]

Returns the current query component in the underlying URL.

Can be set, to change the underlying URL's query component.

o.`hash` [= *value*]

Returns the current fragment identifier in the underlying URL.

Can be set, to change the underlying URL's fragment identifier.

The attributes defined to be URL decomposition attributes must act as described for the attributes with the same corresponding names in this section.

In addition, an interface with a complement of URL decomposition attributes will define an **input**, which is a URL that the attributes act on, and a **common setter action**, which is a set of steps invoked when any of the attributes' setters are invoked.

The seven URL decomposition attributes have similar requirements.

On getting, if the input fulfills the condition given in the "getter condition" column corresponding to the attribute in the table below, the user agent must return the part of the

input URL given in the "component" column, with any prefixes specified in the "prefix" column appropriately added to the start of the string and any suffixes specified in the "suffix" column appropriately added to the end of the string. Otherwise, the attribute must return the empty string.

On setting, the new value must first be mutated as described by the "setter preprocessor" column, then mutated by %-escaping any characters in the new value that are not valid in the relevant component as given by the "component" column. Then, if the resulting new value fulfills the condition given in the "setter condition" column, the user agent must make a new string *output* by replacing the component of the URL given by the "component" column in the input URL with the new value; otherwise, the user agent must let *output* be equal to the input. Finally, the user agent must invoke the common setter action with the value of *output*.

When replacing a component in the URL, if the component is part of an optional group in the URL syntax consisting of a character followed by the component, the component (including its prefix character) must be included even if the new value is the empty string.

> ***Note: The previous paragraph applies in particular to the ":" before a <port> component, the "?" before a <query> component, and the "#" before a <fragment> component.***

For the purposes of the above definitions, URLs must be parsed using the URL parsing rules defined in this specification.

Attribute	Component	Getter Condition	Prefix	Suffix	Setter Preprocessor	Setter Condition
`protocol`	<scheme>	—	—	U+003A COLON (":")	Remove all trailing U+003A COLON (":") characters	The new value is not the empty string
`host`	<hostport>	input is hierarchical and uses a server-based naming authority	—	—	—	—
`hostname`	<host>	input is hierarchical and uses a server-based naming authority	—	—	Remove all leading U+002F SOLIDUS ("/") characters	—
`port`	<port>	input is hierarchical, uses a server-based naming authority, and contained a	—	—	Remove any characters in the new value that are not in the range U+0030 DIGIT ZERO .. U+0039 DIGIT	—

Attribute	Component	Getter Condition	Prefix	Suffix	Setter Preprocessor	Setter Condition
		<port> component (possibly an empty one)			NINE. If the resulting string is empty, set it to a single U+0030 DIGIT ZERO character ('0').	
pathname	<path>	input is hierarchical	—	—	If it has no leading U+002F SOLIDUS ("/") character, prepend a U+002F SOLIDUS ("/") character to the new value	—
search	<query>	input is hierarchical, and contained a <query> component (possibly an empty one)	U+003F QUESTION MARK ("?")	—	Remove one leading U+003F QUESTION MARK ("?") character, if any	—
hash	<fragment>	input contained a <fragment> component (possibly an empty one)	U+0023 NUMBER SIGN ("#")	—	Remove one leading U+0023 NUMBER SIGN ("#") character, if any	—

The table below demonstrates how the getter condition for `search` results in different results depending on the exact original syntax of the URL:

Input URL	**search** value	Explanation
`http://example.com/`	*empty string*	No <query> component in input URL.
`http://example.com/?`	`?`	There is a <query> component, but it is empty. The question mark in the resulting value is the prefix.
`http://example.com/?test`	`?test`	The <query> component has the value "`test`".
`http://example.com/?test#`	`?test`	The (empty) <fragment> component is not part of the <query> component.

2.6 Fetching resources

When a user agent is to **fetch** a resource, the following steps must be run:

1. If the resource is identified by the URL `about:blank`, then return the empty string and abort these steps.

2. Perform the remaining steps asynchronously.

3. If the resource is identified by an absolute URL, and the resource is to be obtained using a idempotent action (such as an HTTP GET or equivalent), and it is already being downloaded for other reasons (e.g. another invocation of this algorithm), and the user agent is configured such that it is to reuse the data from the existing download instead of initiating a new one, then use the results of the existing download instead of starting a new one.

   Otherwise, at a time convenient to the user and the user agent, download (or otherwise obtain) the resource, applying the semantics of the relevant specifications (e.g. performing an HTTP GET or POST operation, or reading the file from disk, following redirects, dereferencing `javascript:` URLs, etc).

4. If there are cookies to be set, then the user agent must run the following substeps:

   1. Wait until ownership of the storage mutex can be taken by this instance of the fetching algorithm.

   2. Take ownership of the storage mutex.

   3. Update the cookies. [COOKIES]

   4. Release the storage mutex so that it is once again free.

5. When the resource is available, queue a task that uses the resource as appropriate. If the resource can be processed incrementally, as, for instance, with a progressively interlaced JPEG or an HTML file, multiple tasks may be queued to process the data as it is downloaded. The task source for these tasks is the networking task source.

Note: The application cache processing model introduces some changes to the networking model to handle the returning of cached resources.

Note: The navigation processing model handles redirects itself, overriding the redirection handling that would be done by the fetching algorithm.

Note: Whether the type sniffing rules apply to the fetched resource depends on the algorithm that invokes the rules — they are not always applicable.

2.6.1 Protocol concepts

User agents can implement a variety of transfer protocols, but this specification mostly defines behavior in terms of HTTP. [HTTP]

The **HTTP GET method** is equivalent to the default retrieval action of the protocol. For example, RETR in FTP. Such actions are idempotent and safe, in HTTP terms.

The **HTTP response codes** are equivalent to statuses in other protocols that have the same basic meanings. For example, a "file not found" error is equivalent to a 404 code, a server error is equivalent to a 5xx code, and so on.

The **HTTP headers** are equivalent to fields in other protocols that have the same basic meaning. For example, the the HTTP authentication headers are equivalent to the authentication aspects of the FTP protocol.

If there are any specific questions with what should be considered equivalent to what, let me know, and I'll make it more explicit for those cases.

2.6.2 Encrypted HTTP and related security concerns

Anything in this specification that refers to HTTP also applies to HTTP-over-TLS, as represented by URLs representing the `https` scheme.

?Warning! ***User agents should report certificate errors to the user and must either refuse to download resources sent with erroneous certificates or must act as if such resources were in fact served with no encryption.***

Not doing so can result in users not noticing man-in-the-middle attacks.

> If a user connects to a server with a self-signed certificate, the user agent could allow the connection but just act as if there had been no encryption. If the user agent instead allowed the user to override the problem and then displayed the page as if it was fully and safely encrypted, the user could be easily tricked into accepting man-in-the-middle connections.
>
> If a user connects to a server with full encryption, but the page then refers to an external resource that has an expired certificate, then the user agent will act as if the resource was unavailable, possibly also reporting the problem to the user. If the user agent instead allowed the resource to be used, then an attacker could just look for "secure" sites that used resources from a different host and only apply man-in-the-middle attacks to that host, for example taking over scripts in the page.

2.7 Determining the type of a resource

?Warning! ***It is imperative that the rules in this section be followed exactly. When a user agent uses different heuristics for content type detection than the server expects, security problems can occur. For example, if a server believes that the client will treat a contributed file as an image (and thus treat it as benign), but a Web browser believes the content to be HTML (and thus execute any scripts contained therein), the end user can be exposed to malicious content, making the user vulnerable to cookie theft attacks and other cross-site scripting attacks.***

2.7.1 Content-Type metadata

What explicit **Content-Type metadata** is associated with the resource (the resource's type

information) depends on the protocol that was used to fetch the resource.

For HTTP resources, only the first Content-Type HTTP header, if any, contributes any type information; the explicit type of the resource is then the value of that header, interpreted as described by the HTTP specifications. If the Content-Type HTTP header is present but the value of the first such header cannot be interpreted as described by the HTTP specifications (e.g. because its value doesn't contain a U+002F SOLIDUS ('/') character), then the resource has no type information (even if there are multiple Content-Type HTTP headers and one of the other ones is syntactically correct). [HTTP]

For resources fetched from the file system, user agents should use platform-specific conventions, e.g. operating system extension/type mappings.

Extensions must not be used for determining resource types for resources fetched over HTTP.

For resources fetched over most other protocols, e.g. FTP, there is no type information.

The **algorithm for extracting an encoding from a Content-Type**, given a string *s*, is as follows. It either returns an encoding or nothing.

1. Find the first seven characters in *s* that are an ASCII case-insensitive match for the word "charset". If no such match is found, return nothing.

2. Skip any U+0009, U+000A, U+000C, U+000D, or U+0020 characters that immediately follow the word 'charset' (there might not be any).

3. If the next character is not a U+003D EQUALS SIGN ('='), return nothing.

4. Skip any U+0009, U+000A, U+000C, U+000D, or U+0020 characters that immediately follow the equals sign (there might not be any).

5. Process the next character as follows:

   - **If it is a U+0022 QUOTATION MARK ('"') and there is a later U+0022 QUOTATION MARK ('"') in *s***
   - **If it is a U+0027 APOSTROPHE ("'") and there is a later U+0027 APOSTROPHE ("'") in *s***

     Return the string between this character and the next earliest occurrence of this character.

   - **If it is an unmatched U+0022 QUOTATION MARK ('"')**
   - **If it is an unmatched U+0027 APOSTROPHE ("')**
   - **If there is no next character**

     Return nothing.

   - **Otherwise**

     Return the string from this character to the first U+0009, U+000A, U+000C, U+000D, U+0020, or U+003B character or the end of *s*, whichever comes first.

Note: The above algorithm is a willful violation of the HTTP specification. [RFC2616]

2.7.2 Content-Type sniffing: Web pages

The **sniffed type of a resource** must be found as follows:

1. If the user agent is configured to strictly obey Content-Type headers for this resource, then jump to the last step in this set of steps.

2. If the resource was fetched over an HTTP protocol and there is an HTTP Content-Type header and the value of the first such header has bytes that exactly match one of the following lines:

Bytes in Hexadecimal	Textual representation
74 65 78 74 2f 70 6c 61 69 6e	`text/plain`
74 65 78 74 2f 70 6c 61 69 6e 3b 20 63 68 61 72 73 65 74 3d 49 53 4f 2d 38 38 35 39 2d 31	`text/plain; charset=ISO-8859-1`
74 65 78 74 2f 70 6c 61 69 6e 3b 20 63 68 61 72 73 65 74 3d 69 73 6f 2d 38 38 35 39 2d 31	`text/plain; charset=iso-8859-1`
74 65 78 74 2f 70 6c 61 69 6e 3b 20 63 68 61 72 73 65 74 3d 55 54 46 2d 38	`text/plain; charset=UTF-8`

   ...then jump to the *text or binary* section below.

3. Let *official type* be the type given by the Content-Type metadata for the resource, ignoring parameters. If there is no such type, jump to the *unknown type* step below. Comparisons with this type, as defined by MIME specifications, are done in an ASCII case-insensitive manner. [RFC2046]

4. If *official type* is "unknown/unknown" or "application/unknown", jump to the *unknown type* step below.

5. If *official type* ends in "+xml", or if it is either "text/xml" or "application/xml", then the sniffed type of the resource is *official type*; return that and abort these steps.

6. If *official type* is an image type supported by the user agent (e.g. "image/png", "image/gif", "image/jpeg", etc), then jump to the *images* section below, passing it the *official type*.

7. If *official type* is "text/html", then jump to the *feed or HTML* section below.

8. The sniffed type of the resource is *official type*.

2.7.3 Content-Type sniffing: text or binary

1. The user agent may wait for 512 or more bytes of the resource to be available.

2. Let *n* be the smaller of either 512 or the number of bytes already available.

3. If *n* is 4 or more, and the first bytes of the resource match one of the following byte sets:

Bytes in Hexadecimal	Description

Bytes in Hexadecimal	Description
FE FF	UTF-16BE BOM
FF FE	UTF-16LE BOM
EF BB BF	UTF-8 BOM

...then the sniffed type of the resource is "text/plain". Abort these steps.

4. If none of the first *n* bytes of the resource are binary data bytes then the sniffed type of the resource is "text/plain". Abort these steps.

5. If the first bytes of the resource match one of the byte sequences in the "pattern" column of the table in the *unknown type* section below, ignoring any rows whose cell in the "security" column says "scriptable" (or "n/a"), then the sniffed type of the resource is the type given in the corresponding cell in the "sniffed type" column on that row; abort these steps.

   ?Warning! ***It is critical that this step not ever return a scriptable type (e.g. text/html), as otherwise that would allow a privilege escalation attack.***

6. Otherwise, the sniffed type of the resource is "application/octet-stream".

Bytes covered by the following ranges are **binary data bytes**:

- 0x00 - 0x08
- 0x0B
- 0x0E - 0x1A
- 0x1C - 0x1F

2.7.4 Content-Type sniffing: unknown type

1. The user agent may wait for 512 or more bytes of the resource to be available.

2. Let *stream length* be the smaller of either 512 or the number of bytes already available.

3. For each row in the table below:

   ↪ **If the row has no "*WS*" bytes:**

   1. Let *pattern length* be the length of the pattern (number of bytes described by the cell in the second column of the row).

   2. If *stream length* is smaller than *pattern length* then skip this row.

   3. Apply the "and" operator to the first *pattern length* bytes of the resource and the given mask (the bytes in the cell of first column of that row), and let the result be the *data*.

   4. If the bytes of the *data* matches the given pattern bytes exactly, then the sniffed type of the resource is the type given in the cell of the third column in that row; abort these steps.

↪ **If the row has a "*WS*" byte:**

1. Let $index_{pattern}$ be an index into the mask and pattern byte strings of the row.

2. Let $index_{stream}$ be an index into the byte stream being examined.

3. *Loop*: If $index_{stream}$ points beyond the end of the byte stream, then this row doesn't match, skip this row.

4. Examine the $index_{stream}$th byte of the byte stream as follows:

   ↪ **If the $index_{pattern}$th byte of the pattern is a normal hexadecimal byte and not a "*WS*" byte:**

   If the "and" operator, applied to the $index_{stream}$th byte of the stream and the $index_{pattern}$th byte of the mask, yield a value different that the $index_{pattern}$th byte of the pattern, then skip this row.

   Otherwise, increment $index_{pattern}$ to the next byte in the mask and pattern and $index_{stream}$ to the next byte in the byte stream.

   ↪ **Otherwise, if the $index_{pattern}$th byte of the pattern is a "*WS*" byte:**

   "*WS*" means "whitespace", and allows insignificant whitespace to be skipped when sniffing for a type signature.

   If the $index_{stream}$th byte of the stream is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space), then increment only the $index_{stream}$ to the next byte in the byte stream.

   Otherwise, increment only the $index_{pattern}$ to the next byte in the mask and pattern.

5. If $index_{pattern}$ does not point beyond the end of the mask and pattern byte strings, then jump back to the *loop* step in this algorithm.

6. Otherwise, the sniffed type of the resource is the type given in the cell of the third column in that row; abort these steps.

4. If none of the first *n* bytes of the resource are binary data bytes then the sniffed type of the resource is "text/plain". Abort these steps.

5. Otherwise, the sniffed type of the resource is "application/octet-stream".

The table used by the above algorithm is:

Bytes in Hexadecimal		Sniffed type	Security	Comment
Mask	**Pattern**			
FF FF DF DF DF DF	3C 21 44 4F 43 54 59 50	text/html	Scriptable	The string "`<!DOCTYPE HTML`" in US-ASCII or compatible

Bytes in Hexadecimal		Sniffed type	Security	Comment
Mask	**Pattern**			
DF DF DF DF DF DF DF DF	45 20 48 54 4D 4C			encodings, case-insensitively.
FF FF DF DF DF DF	*WS* 3C 48 54 4D 4C	text/html	Scriptable	The string "`<HTML`" in US-ASCII or compatible encodings, case-insensitively, possibly with leading spaces.
FF FF DF DF DF DF	*WS* 3C 48 45 41 44	text/html	Scriptable	The string "`<HEAD`" in US-ASCII or compatible encodings, case-insensitively, possibly with leading spaces.
FF FF DF DF DF DF DF DF	*WS* 3C 53 43 52 49 50 54	text/html	Scriptable	The string "`<SCRIPT`" in US-ASCII or compatible encodings, case-insensitively, possibly with leading spaces.
FF FF FF FF FF	25 50 44 46 2D	application/pdf	Scriptable	The string "`%PDF-`", the PDF signature.
FF FF FF FF FF FF FF FF FF FF	25 21 50 53 2D 41 64 6F 62 65 2D	application/postscript	Safe	The string "`%!PS-Adobe-`", the PostScript signature.
FF FF 00 00	FE FF 00 00	text/plain	n/a	UTF-16BE BOM
FF FF 00 00	FF FF 00 00	text/plain	n/a	UTF-16LE BOM
FF FF FF 00	EF BB BF 00	text/plain	n/a	UTF-8 BOM
FF FF FF FF FF FF	47 49 46 38 37 61	image/gif	Safe	The string "`GIF87a`", a GIF signature.
FF FF FF FF FF FF	47 49 46 38 39 61	image/gif	Safe	The string "`GIF89a`", a GIF signature.
FF FF FF FF FF FF FF FF	89 50 4E 47 0D 0A 1A 0A	image/png	Safe	The PNG signature.
FF FF FF	FF D8 FF	image/jpeg	Safe	A JPEG SOI marker followed by the first byte of another marker.
FF FF	42 4D	image/bmp	Safe	The string "`BM`", a BMP signature.
FF FF FF FF	00 00 01 00	image/vnd.microsoft.icon	Safe	A 0 word following by a 1 word, a Windows Icon file format signature.

I'd like to add types like MPEG, AVI, Flash, Java, etc, to the above table.

User agents may support further types if desired, by implicitly adding to the above table. However, user agents should not use any other patterns for types already mentioned in the table above, as this could then be used for privilege escalation (where, e.g., a server uses the above table to determine that content is not HTML and thus safe from XSS attacks, but then a user agent detects it as HTML anyway and allows script to execute).

The column marked "security" is used by the algorithm in the "text or binary" section, to avoid sniffing `text/plain` content as a type that can be used for a privilege escalation attack.

2.7.5 Content-Type sniffing: image

If the resource's *official type* is "image/svg+xml", then the sniffed type of the resource is its *official type* (an XML type).

Otherwise, if the first bytes of the resource match one of the byte sequences in the first column of the following table, then the sniffed type of the resource is the type given in the corresponding cell in the second column on the same row:

Bytes in Hexadecimal	Sniffed type	Comment
47 49 46 38 37 61	image/gif	The string "`GIF87a`", a GIF signature.
47 49 46 38 39 61	image/gif	The string "`GIF89a`", a GIF signature.
89 50 4E 47 0D 0A 1A 0A	image/png	The PNG signature.
FF D8 FF	image/jpeg	A JPEG SOI marker followed by the first byte of another marker.
42 4D	image/bmp	The string "`BM`", a BMP signature.
00 00 01 00	image/vnd.microsoft.icon	A 0 word following by a 1 word, a Windows Icon file format signature.

Otherwise, the sniffed type of the resource is the same as its *official type*.

2.7.6 Content-Type sniffing: feed or HTML

1. The user agent may wait for 512 or more bytes of the resource to be available.

2. Let *s* be the stream of bytes, and let *s*[*i*] represent the byte in *s* with position *i*, treating *s* as zero-indexed (so the first byte is at *i*=0).

3. If at any point this algorithm requires the user agent to determine the value of a byte in *s* which is not yet available, or which is past the first 512 bytes of the resource, or which is beyond the end of the resource, the user agent must stop this algorithm, and assume that the sniffed type of the resource is "text/html".

Note: User agents are allowed, by the first step of this algorithm, to wait until the first 512 bytes of the resource are available.

4. Initialize *pos* to 0.

5. If *s*[0] is 0xEF, *s*[1] is 0xBB, and *s*[2] is 0xBF, then set *pos* to 3. (This skips over a leading UTF-8 BOM, if any.)

6. *Loop start:* Examine *s*[*pos*].

   ↳ **If it is 0x09 (ASCII tab), 0x20 (ASCII space), 0x0A (ASCII LF), or 0x0D (ASCII CR)**
   Increase *pos* by 1 and repeat this step.

   ↳ **If it is 0x3C (ASCII "<")**
   Increase *pos* by 1 and go to the next step.

   ↳ **If it is anything else**
   The sniffed type of the resource is "text/html". Abort these steps.

7. If the bytes with positions *pos* to *pos*+2 in *s* are exactly equal to 0x21, 0x2D, 0x2D respectively (ASCII for "!--"), then:

   1. Increase *pos* by 3.

   2. If the bytes with positions *pos* to *pos*+2 in *s* are exactly equal to 0x2D, 0x2D, 0x3E respectively (ASCII for "-->"), then increase *pos* by 3 and jump back to the previous step (the step labeled *loop start*) in the overall algorithm in this section.

   3. Otherwise, increase *pos* by 1.

   4. Return to step 2 in these substeps.

8. If *s*[*pos*] is 0x21 (ASCII "!"):

   1. Increase *pos* by 1.

   2. If *s*[*pos*] equal 0x3E, then increase *pos* by 1 and jump back to the step labeled *loop start* in the overall algorithm in this section.

   3. Otherwise, return to step 1 in these substeps.

9. If *s*[*pos*] is 0x3F (ASCII "?"):

   1. Increase *pos* by 1.

   2. If *s*[*pos*] and *s*[*pos*+1] equal 0x3F and 0x3E respectively, then increase *pos* by 1 and jump back to the step labeled *loop start* in the overall algorithm in this section.

   3. Otherwise, return to step 1 in these substeps.

10. Otherwise, if the bytes in *s* starting at *pos* match any of the sequences of bytes in the first column of the following table, then the user agent must follow the steps given in the corresponding cell in the second column of the same row.

Bytes in Hexadecimal	Requirement	Comment

72 73 73	The sniffed type of the resource is "application/rss+xml"; abort these steps	The three ASCII characters "`rss`"
66 65 65 64	The sniffed type of the resource is "application/atom+xml"; abort these steps	The four ASCII characters "`feed`"
72 64 66 3A 52 44 46	Continue to the next step in this algorithm	The ASCII characters "`rdf:RDF`"

If none of the byte sequences above match the bytes in *s* starting at *pos*, then the sniffed type of the resource is "text/html". Abort these steps.

11. If, before the next ">", you find two xmlns* attributes with http://www.w3.org/1999/02/22-rdf-syntax-ns# and http://purl.org/rss/1.0/ as the namespaces, then the sniffed type of the resource is "application/rss+xml", abort these steps. (maybe we only need to check for http://purl.org/rss/1.0/ actually)

12. Otherwise, the sniffed type of the resource is "text/html".

Note: For efficiency reasons, implementations may wish to implement this algorithm and the algorithm for detecting the character encoding of HTML documents in parallel.

2.8 Character encodings

User agents must at a minimum support the UTF-8 and Windows-1252 encodings, but may support more.

Note: It is not unusual for Web browsers to support dozens if not upwards of a hundred distinct character encodings.

User agents must support the preferred MIME name of every character encoding they support that has a preferred MIME name, and should support all the IANA-registered aliases. [IANACHARSET]

When comparing a string specifying a character encoding with the name or alias of a character encoding to determine if they are equal, user agents must use the Charset Alias Matching rules defined in Unicode Technical Standard #22. [UTS22]

> For instance, "GB_2312-80" and "g.b.2312(80)" are considered equivalent names.

When a user agent would otherwise use an encoding given in the first column of the following table to either convert content to Unicode characters or convert Unicode characters to bytes, it must instead use the encoding given in the cell in the second column of the same row. When a byte or sequence of bytes is treated differently due to this encoding aliasing, it is said to have been **misinterpreted for compatibility**.

Character encoding overrides

Input encoding	Replacement encoding	References
EUC-KR	Windows-949	[EUCKR] [WIN949]
GB2312	GBK	[GB2312] [GBK]
GB_2312-80	GBK	[RFC1345] [GBK]
ISO-8859-1	Windows-1252	[RFC1345] [WIN1252]
ISO-8859-9	Windows-1254	[RFC1345] [WIN1254]
ISO-8859-11	Windows-874	[ISO885911] [WIN874]
KS_C_5601-1987	Windows-949	[RFC1345] [WIN949]
TIS-620	Windows-874	[TIS620] [WIN874]
US-ASCII	Windows-1252	[RFC1345] [WIN1252]
x-x-big5	Big5	[BIG5]

Note: The requirement to treat certain encodings as other encodings according to the table above is a willful violation of the W3C Character Model specification. [CHARMOD]

User agents must not support the CESU-8, UTF-7, BOCU-1 and SCSU encodings. [CESU8] [UTF7] [BOCU1] [SCSU]

Support for encodings based on EBCDIC is not recommended. This encoding is rarely used for publicly-facing Web content.

Support for UTF-32 is not recommended. This encoding is rarely used, and frequently misimplemented.

Note: This specification does not make any attempt to support EBCDIC-based encodings and UTF-32 in its algorithms; support and use of these encodings can thus lead to unexpected behavior in implementations of this specification.

2.9 Common DOM interfaces

2.9.1 Reflecting content attributes in DOM attributes

Some DOM attributes are defined to **reflect** a particular content attribute. This means that on getting, the DOM attribute returns the current value of the content attribute, and on setting, the DOM attribute changes the value of the content attribute to the given value.

Note: A list of reflecting DOM attributes and their corresponding content attributes is given in the index.

If a reflecting DOM attribute is a `DOMString` attribute whose content attribute is defined to contain a URL, then on getting, the DOM attribute must resolve the value of the content

attribute relative to the element and return the resulting absolute URL if that was successful, or the empty string otherwise; and on setting, must set the content attribute to the specified literal value. If the content attribute is absent, the DOM attribute must return the default value, if the content attribute has one, or else the empty string.

If a reflecting DOM attribute is a `DOMString` attribute whose content attribute is defined to contain one or more URLs, then on getting, the DOM attribute must split the content attribute on spaces and return the concatenation of resolving each token URL to an absolute URL relative to the element, with a single U+0020 SPACE character between each URL, ignoring any tokens that did not resolve successfully. If the content attribute is absent, the DOM attribute must return the default value, if the content attribute has one, or else the empty string. On setting, the DOM attribute must set the content attribute to the specified literal value.

If a reflecting DOM attribute is a `DOMString` whose content attribute is an enumerated attribute, and the DOM attribute is **limited to only known values**, then, on getting, the DOM attribute must return the conforming value associated with the state the attribute is in (in its canonical case), or the empty string if the attribute is in a state that has no associated keyword value; and on setting, if the new value is an ASCII case-insensitive match for one of the keywords given for that attribute, then the content attribute must be set to the conforming value associated with the state that the attribute would be in if set to the given new value, otherwise, if the new value is the empty string, then the content attribute must be removed, otherwise, the setter must raise a `SYNTAX_ERR` exception.

If a reflecting DOM attribute is a `DOMString` but doesn't fall into any of the above categories, then the getting and setting must be done in a transparent, case-preserving manner.

If a reflecting DOM attribute is a boolean attribute, then on getting the DOM attribute must return true if the attribute is set, and false if it is absent. On setting, the content attribute must be removed if the DOM attribute is set to false, and must be set to have the same value as its name if the DOM attribute is set to true. (This corresponds to the rules for boolean content attributes.)

If a reflecting DOM attribute is a signed integer type (`long`) then, on getting, the content attribute must be parsed according to the rules for parsing signed integers, and if that is successful, and the value is in the range of the DOM attribute's type, the resulting value must be returned. If, on the other hand, it fails or returns an out of range value, or if the attribute is absent, then the default value must be returned instead, or 0 if there is no default value. On setting, the given value must be converted to the shortest possible string representing the number as a valid integer in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is an *unsigned* integer type (`unsigned long`) then, on getting, the content attribute must be parsed according to the rules for parsing non-negative integers, and if that is successful, and the value is in the range of the DOM attribute's type, the resulting value must be returned. If, on the other hand, it fails or returns an out of range value, or if the attribute is absent, the default value must be returned instead, or 0 if there is no default value. On setting, the given value must be converted to the shortest possible string representing the number as a valid non-negative integer in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is an unsigned integer type (`unsigned long`) that is **limited to only positive non-zero numbers**, then the behavior is similar to the previous case, but zero

is not allowed. On getting, the content attribute must first be parsed according to the rules for parsing non-negative integers, and if that is successful, and the value is in the range of the DOM attribute's type, the resulting value must be returned. If, on the other hand, it fails or returns an out of range value, or if the attribute is absent, the default value must be returned instead, or 1 if there is no default value. On setting, if the value is zero, the user agent must fire an `INDEX_SIZE_ERR` exception. Otherwise, the given value must be converted to the shortest possible string representing the number as a valid non-negative integer in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is a floating point number type (`float`) and it doesn't fall into one of the earlier categories, then, on getting, the content attribute must be parsed according to the rules for parsing floating point number values, and if that is successful, and the value is in the range of the DOM attribute's type, the resulting value must be returned. If, on the other hand, it fails or returns an out of range value, or if the attribute is absent, the default value must be returned instead, or 0.0 if there is no default value. On setting, the given value must be converted to the best representation of the floating point number and then that string must be used as the new content attribute value.

> ***Note: The values ±Infinity and NaN throw an exception on setting, as defined by WebIDL. [WEBIDL]***

If a reflecting DOM attribute is of the type `DOMTokenList`, then on getting it must return a `DOMTokenList` object whose underlying string is the element's corresponding content attribute. When the `DOMTokenList` object mutates its underlying string, the content attribute must itself be immediately mutated. When the attribute is absent, then the string represented by the `DOMTokenList` object is the empty string; when the object mutates this empty string, the user agent must first add the corresponding content attribute, and then mutate that attribute instead. `DOMTokenList` attributes are always read-only. The same `DOMTokenList` object must be returned every time for each attribute.

If a reflecting DOM attribute has the type `HTMLElement`, or an interface that descends from `HTMLElement`, then, on getting, it must run the following algorithm (stopping at the first point where a value is returned):

1. If the corresponding content attribute is absent, then the DOM attribute must return null.
2. Let *candidate* be the element that the `document.getElementById()` method would find if it was passed as its argument the current value of the corresponding content attribute.
3. If *candidate* is null, or if it is not type-compatible with the DOM attribute, then the DOM attribute must return null.
4. Otherwise, it must return *candidate*.

On setting, if the given element has an `id` attribute, then the content attribute must be set to the value of that `id` attribute. Otherwise, the DOM attribute must be set to the empty string.

2.9.2 Collections

The `HTMLCollection`, `HTMLFormControlsCollection`, and `HTMLOptionsCollection` interfaces represent various lists of DOM nodes. Collectively, objects implementing these interfaces are

called **collections**.

When a collection is created, a filter and a root are associated with the collection.

> For example, when the `HTMLCollection` object for the `document.images` attribute is created, it is associated with a filter that selects only `img` elements, and rooted at the root of the document.

The collection then **represents** a live view of the subtree rooted at the collection's root, containing only nodes that match the given filter. The view is linear. In the absence of specific requirements to the contrary, the nodes within the collection must be sorted in tree order.

Note: The `rows` list is not in tree order.

An attribute that returns a collection must return the same object every time it is retrieved.

2.9.2.1 HTMLCollection

The `HTMLCollection` interface represents a generic collection of elements.

```
[Callable=namedItem]
interface HTMLCollection {
  readonly attribute unsigned long length;
  [IndexGetter] Element item(in unsigned long index);
  [NameGetter] Element namedItem(in DOMString name);
};
```

collection **`.length`**

Returns the number of elements in the collection.

element* = *collection **`.item`**(***index***)
***collection*[*index*]**

Returns the item with index *index* from the collection. The items are sorted in tree order.

Returns null if *index* is out of range.

element* = *collection **`.namedItem`**(***name***)
***collection*[*name*]**

Returns the first item with ID or name *name* from the collection.

Returns null if no element with that ID or name could be found.

Only `a`, `applet`, `area`, `embed`, `form`, `frame`, `frameset`, `iframe`, `img`, and `object` elements can have a name for the purpose of this method; their name is given by the value of their `name` attribute.

The object's indices of the supported indexed properties are the numbers in the range zero to one less than the number of nodes represented by the collection. If there are no such

elements, then there are no supported indexed properties.

The **`length`** attribute must return the number of nodes represented by the collection.

The **`item(`*`index`*`)`** method must return the *index*th node in the collection. If there is no *index*th node in the collection, then the method must return null.

The names of the supported named properties consist of the values of the `name` attributes of each `a`, `applet`, `area`, `embed`, `form`, `frame`, `frameset`, `iframe`, `img`, and `object` element represented by the collection with a `name` attribute, plus the list of IDs that the elements represented by the collection have.

The **`namedItem(`*`key`*`)`** method must return the first node in the collection that matches the following requirements:

- It is an `a`, `applet`, `area`, `embed`, `form`, `frame`, `frameset`, `iframe`, `img`, or `object` element with a `name` attribute equal to *key*, or,
- It is an element with an ID *key*.

If no such elements are found, then the method must return null.

2.9.2.2 HTMLFormControlsCollection

The `HTMLFormControlsCollection` interface represents a collection of listed elements in `form` and `fieldset` elements.

```
[Callable=namedItem]
interface HTMLFormControlsCollection {
  readonly attribute unsigned long length;
  [IndexGetter] HTMLElement item(in unsigned long index);
  [NameGetter] Object namedItem(in DOMString name);
};
```

collection **.** **`length`**

Returns the number of elements in the collection.

element **=** ***collection*** **.** **`item`**(***index***)
collection[***index***]

Returns the item with index *index* from the collection. The items are sorted in tree order.

Returns null if *index* is out of range.

element **=** ***collection*** **.** **`namedItem`**(***name***)
collection[***name***]

Returns the item with ID or `name` *name* from the collection.

If there are multiple matching items, then a `NodeList` object containing all those elements is returned.

Returns null if no element with that ID or `name` could be found.

The object's indices of the supported indexed properties are the numbers in the range zero to one less than the number of nodes represented by the collection. If there are no such elements, then there are no supported indexed properties.

The **`length`** attribute must return the number of nodes represented by the collection.

The **`item(`*`index`*`)`** method must return the *index*th node in the collection. If there is no *index*th node in the collection, then the method must return null.

The names of the supported named properties consist of the values of all the `id` and `name` attributes of all the elements represented by the collection.

The **`namedItem(`*`name`*`)`** method must act according to the following algorithm:

1. If, at the time the method is called, there is exactly one node in the collection that has either an `id` attribute or a `name` attribute equal to *name*, then return that node and stop the algorithm.

2. Otherwise, if there are no nodes in the collection that have either an `id` attribute or a `name` attribute equal to *name*, then return null and stop the algorithm.

3. Otherwise, create a `NodeList` object representing a live view of the `HTMLFormControlsCollection` object, further filtered so that the only nodes in the `NodeList` object are those that have either an `id` attribute or a `name` attribute equal to *name*. The nodes in the `NodeList` object must be sorted in tree order.

4. Return that `NodeList` object.

2.9.2.3 HTMLOptionsCollection

The `HTMLOptionsCollection` interface represents a list of `option` elements. It is always rooted on a `select` element and has attributes and methods that manipulate that element's descendants.

```
[Callable=namedItem]
interface HTMLOptionsCollection {
           attribute unsigned long length;
  [IndexGetter] HTMLOptionElement item(in unsigned long index);
  [NameGetter] Object namedItem(in DOMString name);
  void add(in HTMLElement element, [Optional] in HTMLElement before);
  void add(in HTMLElement element, in long before);
  void remove(in long index);
};
```

collection **.** **`length`** **[= *value*]**

Returns the number of elements in the collection.

When set to a smaller number, truncates the number of `option` elements in the corresponding container.

When set to a greater number, adds new blank `option` elements to that container.

element* = *collection **.** **`item`**(***index***)
***collection*[*index*]**

Returns the item with index *index* from the collection. The items are sorted in tree order.

Returns null if *index* is out of range.

element* = *collection **.** **`namedItem`**(***name***)
***collection*[*name*]**

Returns the item with ID or `name` *name* from the collection.

If there are multiple matching items, then a `NodeList` object containing all those elements is returned.

Returns null if no element with that ID could be found.

collection **.** **`add`**(***element*** **[,** ***before*** **])**

Inserts *element* before the node given by *before*.

The *before* argument can be a number, in which case *element* is inserted before the item with that number, or an element from the collection, in which case *element* is inserted before that element.

If *before* is omitted, null, or a number out of range, then *element* will be added at the end of the list.

This method will throw a `HIERARCHY_REQUEST_ERR` exception if *element* is an ancestor of the element into which it is to be inserted. If *element* is not an `option` or `optgroup` element, then the method does nothing.

The object's indices of the supported indexed properties are the numbers in the range zero to one less than the number of nodes represented by the collection. If there are no such elements, then there are no supported indexed properties.

On getting, the **`length`** attribute must return the number of nodes represented by the collection.

On setting, the behavior depends on whether the new value is equal to, greater than, or less than the number of nodes represented by the collection at that time. If the number is the same, then setting the attribute must do nothing. If the new value is greater, then *n* new `option` elements with no attributes and no child nodes must be appended to the `select` element on which the `HTMLOptionsCollection` is rooted, where *n* is the difference between the two numbers (new value minus old value). If the new value is lower, then the last *n* nodes in the collection must be removed from their parent nodes, where *n* is the difference between the two numbers (old value minus new value).

> ***Note: Setting `length` never removes or adds any `optgroup` elements, and never adds new children to existing `optgroup` elements (though it can remove children from them).***

The **`item(`*`index`*`)`** method must return the *index*th node in the collection. If there is no *index*th node in the collection, then the method must return null.

The names of the supported named properties consist of the values of all the `id` and `name` attributes of all the elements represented by the collection.

The **`namedItem(`*`name`*`)`** method must act according to the following algorithm:

1. If, at the time the method is called, there is exactly one node in the collection that has either an `id` attribute or a `name` attribute equal to *name*, then return that node and stop the algorithm.

2. Otherwise, if there are no nodes in the collection that have either an `id` attribute or a `name` attribute equal to *name*, then return null and stop the algorithm.

3. Otherwise, create a `NodeList` object representing a live view of the `HTMLOptionsCollection` object, further filtered so that the only nodes in the `NodeList` object are those that have either an `id` attribute or a `name` attribute equal to *name*. The nodes in the `NodeList` object must be sorted in tree order.

4. Return that `NodeList` object.

The **`add(`*`element, before`*`)`** method must act according to the following algorithm:

1. If *element* is not an `option` or `optgroup` element, then return and abort these steps.

2. If *element* is an ancestor of the `select` element element on which the `HTMLOptionsCollection` is rooted, then throw a `HIERARCHY_REQUEST_ERR` exception.

3. If *before* is an element, but that element isn't a descendant of the `select` element element on which the `HTMLOptionsCollection` is rooted, then throw a `NOT_FOUND_ERR` exception.

4. If *element* and *before* are the same element, then return and abort these steps.

5. If *before* is a node, then let *reference* be that node. Otherwise, if *before* is an integer, and there is a *before*th node in the collection, let *reference* be that node. Otherwise, let *reference* be null.

6. If *reference* is not null, let *parent* be the parent node of *reference*. Otherwise, let *parent* be the `select` element element on which the `HTMLOptionsCollection` is rooted.

7. Act as if the DOM Core `insertBefore()` method was invoked on the *parent* node, with *element* as the first argument and *reference* as the second argument.

The **`remove(`*`index`*`)`** method must act according to the following algorithm:

1. If the number of nodes represented by the collection is zero, abort these steps.

2. If *index* is not a number greater than or equal to 0 and less than the number of nodes represented by the collection, let *element* be the first element in the collection. Otherwise, let *element* be the *index*th element in the collection.

3. Remove *element* from its parent node.

2.9.3 DOMTokenList

The `DOMTokenList` interface represents an interface to an underlying string that consists of an unordered set of unique space-separated tokens.

Which string underlies a particular `DOMTokenList` object is defined when the object is created. It might be a content attribute (e.g. the string that underlies the `classList` object is the `class` attribute), or it might be an anonymous string (e.g. when a `DOMTokenList` object is passed to an author-implemented callback in the `datagrid` APIs).

```
[Stringifies] interface DOMTokenList {
  readonly attribute unsigned long length;
  [IndexGetter] DOMString item(in unsigned long index);
  boolean has(in DOMString token);
  void add(in DOMString token);
  void remove(in DOMString token);
  boolean toggle(in DOMString token);
};
```

tokenlist **.** **`length`**

Returns the number of tokens in the string.

element **=** ***tokenlist*** **.** **`item`**(***index***)
tokenlist[***index***]

Returns the token with index *index*. The tokens are sorted alphabetically.

Returns null if *index* is out of range.

has-p **=** ***tokenlist*** **.** **`has`**(***token***)

Returns true if the *token* is present; false otherwise.

Throws an `INVALID_CHARACTER_ERR` exception if *token* contains any spaces.

***tokenlist* . `add`(*token*)**

Adds *token*, unless it is already present.

Throws an `INVALID_CHARACTER_ERR` exception if *token* contains any spaces.

***tokenlist* . `remove`(*token*)**

Removes *token* if it is present.

Throws an `INVALID_CHARACTER_ERR` exception if *token* contains any spaces.

***has-p* = *tokenlist* . `toggle`(*token*)**

Adds *token* if it is not present, or removes it if it is.

Throws an `INVALID_CHARACTER_ERR` exception if *token* contains any spaces.

The **`length`** attribute must return the number of *unique* tokens that result from splitting the underlying string on spaces. This is the *length*.

The object's indices of the supported indexed properties are the numbers in the range zero to *length*-1, unless the *length* is zero, in which case there are no supported indexed properties.

The **`item(`*`index`*`)`** method must split the underlying string on spaces, sort the resulting list of tokens by Unicode codepoint, remove exact duplicates, and then return the *index*th item in this list. If *index* is equal to or greater than the number of tokens, then the method must return null.

The **`has(`*`token`*`)`** method must run the following algorithm:

1. If the *token* argument contains any space characters, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.
2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.
3. If the token indicated by *token* is one of the tokens in the object's underlying string then return true and stop this algorithm.
4. Otherwise, return false.

The **`add(`*`token`*`)`** method must run the following algorithm:

1. If the *token* argument contains any space characters, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.
2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.
3. If the given *token* is already one of the tokens in the `DOMTokenList` object's underlying

string then stop the algorithm.

4. Otherwise, if the `DOMTokenList` object's underlying string is not the empty string and the last character of that string is not a space character, then append a U+0020 SPACE character to the end of that string.

5. Append the value of *token* to the end of the `DOMTokenList` object's underlying string.

The **`remove(`*`token`*`)`** method must run the following algorithm:

1. If the *token* argument contains any space characters, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.

2. Otherwise, remove the given *token* from the underlying string.

The **`toggle(`*`token`*`)`** method must run the following algorithm:

1. If the *token* argument contains any space characters, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.

2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.

3. If the given *token* is already one of the tokens in the `DOMTokenList` object's underlying string then remove the given *token* from the underlying string, and stop the algorithm, returning false.

4. Otherwise, if the `DOMTokenList` object's underlying string is not the empty string and the last character of that string is not a space character, then append a U+0020 SPACE character to the end of that string.

5. Append the value of *token* to the end of the `DOMTokenList` object's underlying string.

6. Return true.

Objects implementing the `DOMTokenList` interface must **stringify** to the object's underlying string representation.

2.9.4 Safe passing of structured data

When a user agent is required to obtain a **structured clone** of an object, it must run the following algorithm, which either returns a separate object, or throws an exception.

1. Let *input* be the object being cloned.

2. Let *memory* be a list of objects, initially empty. (This is used to catch cycles.)

3. Let *output* be the object resulting from calling the internal structured cloning algorithm with *input* and *memory*.

4. Return *output*.

The **internal structured cloning algorithm** is always called with two arguments, *input* and

memory, and its behavior depends on the type of *input*, as follows:

↪ **If *input* is the undefined value**
Return the undefined value.

↪ **If *input* is the null value**
Return the null value.

↪ **If *input* is the false value**
Return the false value.

↪ **If *input* is the true value**
Return the true value.

↪ **If *input* is a Number object**
Return a newly constructed Number object with the same value as *input*.

↪ **If *input* is a String object**
Return a newly constructed String object with the same value as *input*.

↪ **If *input* is a `Date` object**
Return a newly constructed `Date` object with the same value as *input*.

↪ **If *input* is a `RegExp` object**
Return a newly constructed `RegExp` object with the same pattern and flags as *input*.

Note: The value of the `lastIndex` property is not copied.

↪ **If *input* is a `ImageData` object**
Return a newly constructed `ImageData` object with the same `width` and `height` as *input*, and with a newly constructed `CanvasPixelArray` for its `data` attribute, with the same `length` and pixel values as the *input*'s.

↪ **If *input* is a host object (e.g. a DOM node)**
Return the null value.

↪ **If *input* is an Array object**
↪ **If *input* is an Object object**

1. If *input* is in *memory*, then throw a `NOT_SUPPORTED_ERR` exception and abort the overall structured clone algorithm.

2. Otherwise, let *new memory* be a list consisting of the items in *memory* with the addition of *input*.

3. Create a new object, *output*, of the same type as *input*: either an Array or an Object.

4. For each enumerable property in *input*, add a corresponding property to *output* having the same name, and having a value created from invoking the internal structured cloning algorithm recursively with the value of the property as the "*input*" argument and *new memory* as the "*memory*" argument. The order of the properties in the *input* and *output* objects must be the same.

Note: This does not walk the prototype chain.

5. Return *output*.

↪ **If *input* is another native object type (e.g. `Error`)**
Return the null value.

2.9.5 DOMStringMap

The `DOMStringMap` interface represents a set of name-value pairs. It exposes these using the scripting language's native mechanisms for property access.

When a `DOMStringMap` object is instantiated, it is associated with three algorithms, one for getting getting the list of name-value pairs, one for setting names to certain values, and one for deleting names.

```
[NameCreator, NameDeleter, NameGetter, NameSetter]
interface DOMStringMap {};
```

The names of the supported named properties on a `DOMStringMap` object at any instant are the names of each pair returned from the algorithm for getting the list of name-value pairs at that instant.

When a `DOMStringMap` object is indexed to retrieve a named property *name*, the value returned must be the value component of the name-value pair whose name component is *name* in the list returned by the algorithm for getting the list of name-value pairs.

When a `DOMStringMap` object is indexed to create or modify a named property *name* with value *value*, the algorithm for setting names to certain values must be run, passing *name* as the name and the result of converting *value* to a `DOMString` as the value.

When a `DOMStringMap` object is indexed to delete a named property named *name*, the algorithm for deleting names must be run, passing *name* as the name.

Note: The `DOMStringMap` interface definition here is only intended for JavaScript environments. Other language bindings will need to define how `DOMStringMap` is to be implemented for those languages.

The `dataset` attribute on elements exposes the `data-*` attributes on the element.

Given the following fragment and elements with similar constructions:

```
<img class="tower" id="tower5" data-x="12" data-y="5"
     data-ai="robotarget" data-hp="46" data-ability="flames"
     src="towers/rocket.png alt="Rocket Tower">
```

...one could imagine a function `splashDamage()` that takes some arguments, the first of which is the element to process:

```
function splashDamage(node, x, y, damage) {
  if (node.classList.has('tower') && // checking the 'class' attribute
      node.dataset.x == x && // reading the 'data-x' attribute
```

```
          node.dataset.y == y) { // reading the 'data-y' attribute
        var hp = parseInt(node.dataset.hp); // reading the 'data-hp' attribute
        hp = hp - damage;
        if (hp < 0) {
          hp = 0;
          node.dataset.ai = 'dead'; // setting the 'data-ai' attribute
          delete node.dataset.ability; // removing the 'data-ability' attribute
        }
        node.dataset.hp = hp; // setting the 'data-hp' attribute
      }
    }
```

2.9.6 DOM feature strings

DOM3 Core defines mechanisms for checking for interface support, and for obtaining implementations of interfaces, using feature strings. [DOM3CORE]

A DOM application can use the **`hasFeature(`*`feature, version`*`)`** method of the `DOMImplementation` interface with parameter values "`HTML`" and "`5.0`" (respectively) to determine whether or not this module is supported by the implementation. In addition to the feature string "`HTML`", the feature string "`XHTML`" (with version string "`5.0`") can be used to check if the implementation supports XHTML. User agents should respond with a true value when the `hasFeature` method is queried with these values. Authors are cautioned, however, that UAs returning true might not be perfectly compliant, and that UAs returning false might well have support for features in this specification; in general, therefore, use of this method is discouraged.

The values "`HTML`" and "`XHTML`" (both with version "`5.0`") should also be supported in the context of the `getFeature()` and `isSupported()` methods, as defined by DOM3 Core.

> ***Note: The interfaces defined in this specification are not always supersets of the interfaces defined in DOM2 HTML; some features that were formerly deprecated, poorly supported, rarely used or considered unnecessary have been removed. Therefore it is not guaranteed that an implementation that supports "`HTML`" "`5.0`" also supports "`HTML`" "`2.0`".***

2.9.7 Exceptions

The following `DOMException` codes are defined in DOM Core. [DOMCORE]

1. **`INDEX_SIZE_ERR`**
2. **`DOMSTRING_SIZE_ERR`**
3. **`HIERARCHY_REQUEST_ERR`**
4. **`WRONG_DOCUMENT_ERR`**
5. **`INVALID_CHARACTER_ERR`**
6. **`NO_DATA_ALLOWED_ERR`**
7. **`NO_MODIFICATION_ALLOWED_ERR`**
8. **`NOT_FOUND_ERR`**
9. **`NOT_SUPPORTED_ERR`**
10. **`INUSE_ATTRIBUTE_ERR`**
11. **`INVALID_STATE_ERR`**
12. **`SYNTAX_ERR`**

13. **`INVALID_MODIFICATION_ERR`**
14. **`NAMESPACE_ERR`**
15. **`INVALID_ACCESS_ERR`**
16. **`VALIDATION_ERR`**
17. **`TYPE_MISMATCH_ERR`**
18. **`SECURITY_ERR`**
19. **`NETWORK_ERR`**
20. **`ABORT_ERR`**
21. **`URL_MISMATCH_ERR`**
22. **`QUOTA_EXCEEDED_ERR`**
23. **`UNAVAILABLE_SCRIPT_ERR`**
81. **`PARSE_ERR`**
82. **`SERIALISE_ERR`**

2.9.8 Garbage collection

There is an **implied strong reference** from any DOM attribute that returns a pre-existing object to that object.

> For example, the `document.location` attribute means that there is a strong reference from a `Document` object to its `Location` object. Similarly, there is always a strong reference from a `Document` to any descendant nodes, and from any node to its owner `Document`.

3 Semantics and structure of HTML documents

3.1 Introduction

This section is non-normative.

An introduction to marking up a document.

3.2 Documents

Every XML and HTML document in an HTML UA is represented by a `Document` object. [DOM3CORE]

The document's address is an absolute URL that is set when the `Document` is created.

When a `Document` is created by a script using the `createDocument()` API, the document's address is the same as the document's address of the active document of the script's browsing context.

`Document` objects are assumed to be **XML documents** unless they are flagged as being **HTML documents** when they are created. Whether a document is an HTML document or an XML document affects the behavior of certain APIs, as well as a few CSS rendering rules. [CSS21]

> ***Note: A `Document` object created by the `createDocument()` API on the `DOMImplementation` object is initially an XML document, but can be made into an HTML document by calling `document.open()` on it.***

3.2.1 Documents in the DOM

All `Document` objects (in user agents implementing this specification) must also implement the `HTMLDocument` interface, available using binding-specific methods. (This is the case whether or not the document in question is an HTML document or indeed whether it contains any HTML elements at all.) `Document` objects must also implement the document-level interface of any other namespaces found in the document that the UA supports.

> For example, if an HTML implementation also supports SVG, then the `Document` object implements both `HTMLDocument` and `SVGDocument`.

> ***Note: Because the `HTMLDocument` interface is now obtained using binding-specific casting methods instead of simply being the primary interface of the document object, it is no longer defined as inheriting from `Document`.***

```
[NameGetter=OverrideBuiltins, ImplementedOn=Document]
interface HTMLDocument {
  // resource metadata management
  [PutForwards=href] readonly attribute Location location;
  readonly attribute DOMString URL;
           attribute DOMString domain;
  readonly attribute DOMString referrer;
           attribute DOMString cookie;
  readonly attribute DOMString lastModified;
  readonly attribute DOMString compatMode;
           attribute DOMString charset;
  readonly attribute DOMString characterSet;
  readonly attribute DOMString defaultCharset;
  readonly attribute DOMString readyState;

  // DOM tree accessors
           attribute DOMString title;
           attribute DOMString dir;
           attribute HTMLElement body;
  readonly attribute HTMLCollection images;
  readonly attribute HTMLCollection embeds;
  readonly attribute HTMLCollection plugins;
  readonly attribute HTMLCollection links;
  readonly attribute HTMLCollection forms;
  readonly attribute HTMLCollection anchors;
  readonly attribute HTMLCollection scripts;
  NodeList getElementsByName(in DOMString elementName);
  NodeList getElementsByClassName(in DOMString classNames);

  // dynamic markup insertion
           attribute DOMString innerHTML;
  HTMLDocument open([Optional] in DOMString type, [Optional] in DOMString replace)
  WindowProxy open(in DOMString url, in DOMString name, in DOMString features, [Op
  void close();
  void write([Variadic] in DOMString text);
  void writeln([Variadic] in DOMString text);

  // user interaction
  Selection getSelection();
  readonly attribute Element activeElement;
  boolean hasFocus();
           attribute DOMString designMode;
  boolean execCommand(in DOMString commandId);
  boolean execCommand(in DOMString commandId, in boolean showUI);
  boolean execCommand(in DOMString commandId, in boolean showUI, in DOMString valu
  boolean queryCommandEnabled(in DOMString commandId);
  boolean queryCommandIndeterm(in DOMString commandId);
  boolean queryCommandState(in DOMString commandId);
  boolean queryCommandSupported(in DOMString commandId);
  DOMString queryCommandValue(in DOMString commandId);
  readonly attribute HTMLCollection commands;
};
```

Since the `HTMLDocument` interface holds methods and attributes related to a number of disparate features, the members of this interface are described in various different sections.

3.2.2 Security

User agents must raise a `SECURITY_ERR` exception whenever any of the members of an `HTMLDocument` object are accessed by scripts whose effective script origin is not the same as the `Document`'s effective script origin.

3.2.3 Resource metadata management

document **.** **`URL`**

Returns the document's address.

document **.** **`referrer`**

Returns the address of the `Document` from which the user navigated to this one, unless it was blocked or there was no such document, in which case it returns the empty string.

The `noreferrer` link type can be used to block the referrer.

The **`URL`** attribute must return the document's address.

The **`referrer`** attribute must return either the address of the active document of the source browsing context at the time the navigation was started (that is, the page which navigated the browsing context to the current document), or the empty string if there is no such originating page, or if the UA has been configured not to report referrers in this case, or if the navigation was initiated for a hyperlink with a `noreferrer` keyword.

Note: In the case of HTTP, the `referrer` DOM attribute will match the `Referer` (sic) header that was sent when fetching the current page.

Note: Typically user agents are configured to not report referrers in the case where the referrer uses an encrypted protocol and the current page does not (e.g. when navigating from an `https:` page to an `http:` page).

document **.** **`cookie`** **[=** ***value*** **]**

Returns the HTTP cookies that apply to the `Document`. If there are no cookies or cookies can't be applied to this resource, the empty string will be returned.

Can be set, to add a new cookie to the element's set of HTTP cookies.

If the `Document` has no browsing context an `INVALID_STATE_ERR` exception will be thrown. If the contents are sandboxed into a unique origin, a `SECURITY_ERR` exception will be thrown.

The **`cookie`** attribute represents the cookies of the resource.

On getting, if the document is not associated with a browsing context then the user agent must raise an `INVALID_STATE_ERR` exception. Otherwise, if the sandboxed origin browsing context flag was set on the browsing context of the `Document` when the `Document` was created, the user agent must raise a `SECURITY_ERR` exception. Otherwise, if the document's address

does not use a server-based naming authority, it must return the empty string. Otherwise, it must first obtain the storage mutex and then return the same string as the value of the `Cookie` HTTP header it would include if fetching the resource indicated by the document's address over HTTP, as per RFC 2109 section 4.3.4 or later specifications, excluding HTTP-only cookies. [RFC2109] [RFC2965]

On setting, if the document is not associated with a browsing context then the user agent must raise an `INVALID_STATE_ERR` exception. Otherwise, if the sandboxed origin browsing context flag was set on the browsing context of the `Document` when the `Document` was created, the user agent must raise a `SECURITY_ERR` exception. Otherwise, if the document's address does not use a server-based naming authority, it must do nothing. Otherwise, the user agent must obtain the storage mutex and then act as it would when processing cookies if it had just attempted to fetch the document's address over HTTP, and had received a response with a `Set-Cookie` header whose value was the specified value, as per RFC 2109 sections 4.3.1, 4.3.2, and 4.3.3 or later specifications, but without overwriting the values of HTTP-only cookies. [RFC2109] [RFC2965]

Note: This specification does not define what makes an HTTP-only cookie, and at the time of publication the editor is not aware of any reference for HTTP-only cookies. They are a feature supported by some Web browsers wherein an "`httponly`" parameter added to the cookie string causes the cookie to be hidden from script.

Note: Since the `cookie` attribute is accessible across frames, the path restrictions on cookies are only a tool to help manage which cookies are sent to which parts of the site, and are not in any way a security feature.

document **`.lastModified`**

Returns the date of the last modification to the document, as reported by the server, in the form "`MM/DD/YYYY hh:mm:ss`".

If the last modification date is not known, it is reported as "`01/01/1970 00:00:00`"

The **`lastModified`** attribute, on getting, must return the date and time of the `Document`'s source file's last modification, in the user's local timezone, in the following format:

1. The month component of the date.
2. A U+002F SOLIDUS character ('/').
3. The day component of the date.
4. A U+002F SOLIDUS character ('/').
5. The year component of the date.
6. A U+0020 SPACE character.

7. The hours component of the time.
8. A U+003A COLON character (':').
9. The minutes component of the time.
10. A U+003A COLON character (':').
11. The seconds component of the time.

All the numeric components above, other than the year, must be given as two digits in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE representing the number in base ten, zero-padded if necessary. The year must be given as four or more digits in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE representing the number in base ten, zero-padded if necessary.

The `Document`'s source file's last modification date and time must be derived from relevant features of the networking protocols used, e.g. from the value of the HTTP `Last-Modified` header of the document, or from metadata in the file system for local files. If the last modification date and time are not known, the attribute must return the string "`01/01/1970 00:00:00`".

document **.** **`compatMode`**

In a conforming document, returns the string "`CSS1Compat`". (In quirks mode documents, returns the string "`BackCompat`", but a conforming document can never trigger quirks mode.)

A `Document` is always set to one of three modes: **no quirks mode**, the default; **quirks mode**, used typically for legacy documents; and **limited quirks mode**, also known as "almost standards" mode. The mode is only ever changed from the default by the HTML parser, based on the presence, absence, or value of the DOCTYPE string.

The **`compatMode`** DOM attribute must return the literal string "`CSS1Compat`" unless the document has been set to quirks mode by the HTML parser, in which case it must instead return the literal string "`BackCompat`".

As far as parsing goes, the quirks I know of are:

- Comment parsing is different.
- `p` can contain `table`
- Safari and IE have special parsing rules for <% ... %> (even in standards mode, though clearly this should be quirks-only).

document **.** **`charset`** **[= *value*]**

Returns the document's character encoding.

Can be set, to dynamically change the document's character encoding.

New values that are not IANA-registered aliases are ignored.

document **. `characterSet`**

Returns the document's character encoding.

document **. `defaultCharset`**

Returns what might be the user agent's default character encoding.

Documents have an associated **character encoding**. When a `Document` object is created, the document's character encoding must be initialized to UTF-16. Various algorithms during page loading affect this value, as does the `charset` setter. [IANACHARSET]

The **`charset`** DOM attribute must, on getting, return the preferred MIME name of the document's character encoding. On setting, if the new value is an IANA-registered alias for a character encoding, the document's character encoding must be set to that character encoding. (Otherwise, nothing happens.)

The **`characterSet`** DOM attribute must, on getting, return the preferred MIME name of the document's character encoding.

The **`defaultCharset`** DOM attribute must, on getting, return the preferred MIME name of a character encoding, possibly the user's default encoding, or an encoding associated with the user's current geographical location, or any arbitrary encoding name.

document **. `readyState`**

Returns "loading" while the `Document` is loading, and "complete" once it has loaded.

The `readystatechange` event fires on the `Document` object when this value changes.

Each document has a **current document readiness**. When a `Document` object is created, it must have its current document readiness set to the string "loading" if the document is associated with an HTML parser or an XML parser, or to the string "complete" otherwise. Various algorithms during page loading affect this value. When the value is set, the user agent must fire a simple event called `readystatechange` at the `Document` object.

The **`readyState`** DOM attribute must, on getting, return the current document readiness.

3.2.4 DOM tree accessors

The `html` element of a document is the document's root element, if there is one and it's an `html` element, or null otherwise.

The `head` element of a document is the first `head` element that is a child of the `html` element, if there is one, or null otherwise.

document **.** `title` **[=** ***value*** **]**

Returns the document's title, as given by the `title` element.

Can be set, to update the document's title. If there is no `head` element, the new value is ignored.

In SVG documents, the `SVGDocument` interface's `title` attribute takes precedence.

The `title` element of a document is the first `title` element in the document (in tree order), if there is one, or null otherwise.

The `title` attribute must, on getting, run the following algorithm:

1. If the root element is an `svg` element in the "`http://www.w3.org/2000/svg`" namespace, and the user agent supports SVG, then the getter must return the value that would have been returned by the DOM attribute of the same name on the `SVGDocument` interface.

2. Otherwise, it must return a concatenation of the data of all the child text nodes of the `title` element, in tree order, or the empty string if the `title` element is null.

On setting, the following algorithm must be run:

1. If the root element is an `svg` element in the "`http://www.w3.org/2000/svg`" namespace, and the user agent supports SVG, then the setter must defer to the setter for the DOM attribute of the same name on the `SVGDocument` interface (if it is readonly, then this will raise an exception). Stop the algorithm here.

2. If the `title` element is null and the `head` element is null, then the attribute must do nothing. Stop the algorithm here.

3. If the `title` element is null, then a new `title` element must be created and appended to the `head` element.

4. The children of the `title` element (if any) must all be removed.

5. A single `Text` node whose data is the new value being assigned must be appended to the `title` element.

The `title` attribute on the `HTMLDocument` interface should shadow the attribute of the same name on the `SVGDocument` interface when the user agent supports both HTML and SVG.

document **.** `body` **[=** ***value*** **]**

Returns the body element.

Can be set, to replace the body element.

If the new value is not a `body` or `frameset` element, this will throw a `HIERARCHY_REQUEST_ERR` exception.

The body element of a document is the first child of the `html` element that is either a `body` element or a `frameset` element. If there is no such element, it is null. If the body element is null, then when the specification requires that events be fired at "the body element", they must instead be fired at the `Document` object.

The **`body`** attribute, on getting, must return the body element of the document (either a `body` element, a `frameset` element, or null). On setting, the following algorithm must be run:

1. If the new value is not a `body` or `frameset` element, then raise a `HIERARCHY_REQUEST_ERR` exception and abort these steps.
2. Otherwise, if the new value is the same as the body element, do nothing. Abort these steps.
3. Otherwise, if the body element is not null, then replace that element with the new value in the DOM, as if the root element's `replaceChild()` method had been called with the new value and the incumbent body element as its two arguments respectively, then abort these steps.
4. Otherwise, the the body element is null. Append the new value to the root element.

document **`.images`**

Returns an `HTMLCollection` of the `img` elements in the `Document`.

document **`.embeds`**
document **`.plugins`**

Return an `HTMLCollection` of the `embed` elements in the `Document`.

document **`.links`**

Returns an `HTMLCollection` of the `a` and `area` elements in the `Document` that have `href` attributes.

document **`.forms`**

Return an `HTMLCollection` of the `form` elements in the `Document`.

document **`.scripts`**

Return an `HTMLCollection` of the `script` elements in the `Document`.

The **`images`** attribute must return an `HTMLCollection` rooted at the `Document` node, whose filter matches only `img` elements.

The **`embeds`** attribute must return an `HTMLCollection` rooted at the `Document` node, whose filter

matches only embed elements.

The **plugins** attribute must return the same object as that returned by the embeds attribute.

The **links** attribute must return an HTMLCollection rooted at the Document node, whose filter matches only a elements with href attributes and area elements with href attributes.

The **forms** attribute must return an HTMLCollection rooted at the Document node, whose filter matches only form elements.

The **anchors** attribute must return an HTMLCollection rooted at the Document node, whose filter matches only a elements with name attributes.

The **scripts** attribute must return an HTMLCollection rooted at the Document node, whose filter matches only script elements.

***collection* = *document* . getElementsByName(*name*)**

Returns a NodeList of a, applet, button, form, frame, frameset, iframe, img, input, map, meta, object, select, and textarea elements in the Document that have a name attribute with the value *name*.

***collection* = *document* . getElementsByClassName(*classes*)**
***collection* = *element* . getElementsByClassName(*classes*)**

Returns a NodeList of the elements in the object on which the method was invoked (a Document or an Element) that have all the classes given by *classes*.

The *classes* argument is interpreted as a space-separated list of classes.

The **getElementsByName(*name*)** method takes a string *name*, and must return a live NodeList containing all the a, applet, button, form, frame, frameset, iframe, img, input, map, meta, object, select, and textarea elements in that document that have a name attribute whose value is equal to the *name* argument (in a case-sensitive manner), in tree order.

The **getElementsByClassName(*classNames*)** method takes a string that contains an unordered set of unique space-separated tokens representing classes. When called, the method must return a live NodeList object containing all the elements in the document, in tree order, that have all the classes specified in that argument, having obtained the classes by splitting a string on spaces. If there are no tokens specified in the argument, then the method must return an empty NodeList. If the document is in quirks mode, then the comparisons for the classes must be done in an ASCII case-insensitive manner, otherwise, the comparisons must be done in a case-sensitive manner.

The **getElementsByClassName(*classNames*)** method on the HTMLElement interface must return a live NodeList with the nodes that the HTMLDocument getElementsByClassName() method would return when passed the same argument(s), excluding any elements that are not descendants of the HTMLElement object on which the method was invoked.

HTML, SVG, and MathML elements define which classes they are in by having an attribute in the per-element partition with the name class containing a space-separated list of classes to

which the element belongs. Other specifications may also allow elements in their namespaces to be labeled as being in specific classes.

> Given the following XHTML fragment:
>
> ```
> 
>  <p id="p1" class="aaa bbb"/>
>  <p id="p2" class="aaa ccc"/>
>  <p id="p3" class="bbb ccc"/>
> 
> ```
>
> A call to `document.getElementById('example').getElementsByClassName('aaa')` would return a `NodeList` with the two paragraphs `p1` and `p2` in it.
>
> A call to `getElementsByClassName('ccc bbb')` would only return one node, however, namely `p3`. A call to `document.getElementById('example').getElementsByClassName('bbb  ccc ')` would return the same thing.
>
> A call to `getElementsByClassName('aaa,bbb')` would return no nodes; none of the elements above are in the "aaa,bbb" class.

The `HTMLDocument` interface supports named properties. The names of the supported named properties at any moment consist of the values of the `name` content attributes of all the `applet`, `embed`, `form`, `iframe`, `img`, and fallback-free `object` elements in the `Document` that have `name` content attributes, and the values of the `id` content attributes of all the `applet` and fallback-free `object` elements in the `Document` that have `id` content attributes, and the values of the `id` content attributes of all the `img` elements in the `Document` that have both `name` content attributes and `id` content attributes.

When **the `HTMLDocument` object is indexed for property retrieval** using a name *name*, then the user agent must return the value obtained using the following steps:

1. Let *elements* be the list of named elements with the name *name* in the `Document`.

   Note: There will be at least one such element, by definition.

2. If *elements* has only one element, and that element is an `iframe` element, then return the `WindowProxy` object of the nested browsing context represented by that `iframe` element, and abort these steps.

3. Otherwise, if *elements* has only one element, return that element and abort these steps.

4. Otherwise return an `HTMLCollection` rooted at the `Document` node, whose filter matches only named elements with the name *name*.

Named elements with the name *name*, for the purposes of the above algorithm, are those that are either:

- `applet`, `embed`, `form`, `iframe`, `img`, or fallback-free `object` elements that have a `name` content attribute whose value is *name*, or

- `applet` or fallback-free `object` elements that have an `id` content attribute whose value is *name*, or

- `img` elements that have an `id` content attribute whose value is *name*, and that have a `name` content attribute present also.

An `object` element is said to be **fallback-free** if it has no element descendants other than `param` elements, and no text node descendants that are not inter-element whitespace.

Note: The `dir` attribute on the `HTMLDocument` interface is defined along with the `dir` content attribute.

3.3 Elements

3.3.1 Semantics

Elements, attributes, and attribute values in HTML are defined (by this specification) to have certain meanings (semantics). For example, the `ol` element represents an ordered list, and the `lang` attribute represents the language of the content.

Authors must not use elements, attributes, and attribute values for purposes other than their appropriate intended semantic purpose.

> For example, the following document is non-conforming, despite being syntactically correct:
>
> ```
> <!DOCTYPE html>
> <html lang="en-GB">
>  <head> <title> Demonstration </title> </head>
>  <body>
>   <table>
>    <tr> <td> My favourite animal is the cat. </td> </tr>
>    <tr>
>     <td>
>      —<a href="http://example.org/~ernest/"><cite>Ernest</cite></a>,
>      in an essay from 1992
>     </td>
>    </tr>
>   </table>
>  </body>
> </html>
> ```
>
> ...because the data placed in the cells is clearly not tabular data (and the `cite` element mis-used). A corrected version of this document might be:
>
> ```
> <!DOCTYPE html>
> <html lang="en-GB">
>  <head> <title> Demonstration </title> </head>
>  <body>
>   <blockquote>
>     My favourite animal is the cat. 
>   </blockquote>
>   
>    —<a href="http://example.org/~ernest/">Ernest</a>,
>    in an essay from 1992
>   
>  </body>
> </html>
> ```
>
> This next document fragment, intended to represent the heading of a corporate site, is

similarly non-conforming because the second line is not intended to be a heading of a subsection, but merely a subheading or subtitle (a subordinate heading for the same section).

```
<body>
 <h1>ABC Company</h1>
 <h2>Leading the way in widget design since 1432</h2>
 ...
```

The `header` element should be used in these kinds of situations:

```
<body>
 <header>
  <h1>ABC Company</h1>
  <h2>Leading the way in widget design since 1432</h2>
 </header>
 ...
```

Through scripting and using other mechanisms, the values of attributes, text, and indeed the entire structure of the document may change dynamically while a user agent is processing it. The semantics of a document at an instant in time are those represented by the state of the document at that instant in time, and the semantics of a document can therefore change over time. User agents must update their presentation of the document as this occurs.

> HTML has a `progress` element that describes a progress bar. If its "value" attribute is dynamically updated by a script, the UA would update the rendering to show the progress changing.

3.3.2 Elements in the DOM

The nodes representing HTML elements in the DOM must implement, and expose to scripts, the interfaces listed for them in the relevant sections of this specification. This includes HTML elements in XML documents, even when those documents are in another context (e.g. inside an XSLT transform).

Elements in the DOM represent things; that is, they have intrinsic *meaning*, also known as semantics.

> For example, an `ol` element represents an ordered list.

The basic interface, from which all the HTML elements' interfaces inherit, and which must be used by elements that have no additional requirements, is the `HTMLElement` interface.

```
interface HTMLElement : Element {
  // DOM tree accessors
  NodeList getElementsByClassName(in DOMString classNames);

  // dynamic markup insertion
           attribute DOMString innerHTML;
           attribute DOMString outerHTML;
  void insertAdjacentHTML(in DOMString position, in DOMString text);

  // metadata attributes
           attribute DOMString id;
           attribute DOMString title;
           attribute DOMString lang;
           attribute DOMString dir;
           attribute DOMString className;
```

```
  readonly attribute DOMTokenList classList;
  readonly attribute DOMStringMap dataset;

  // user interaction
           attribute boolean hidden;
  void click();
  void scrollIntoView();
  void scrollIntoView(in boolean top);
           attribute long tabIndex;
  void focus();
  void blur();
           attribute boolean draggable;
           attribute DOMString contentEditable;
  readonly attribute boolean isContentEditable;
           attribute HTMLMenuElement contextMenu;
           attribute boolean spellcheck;

  // styling
  readonly attribute CSSStyleDeclaration style;

  // event handler DOM attributes
           attribute Function onabort;
           attribute Function onblur;
           attribute Function onchange;
           attribute Function onclick;
           attribute Function oncontextmenu;
           attribute Function ondblclick;
           attribute Function ondrag;
           attribute Function ondragend;
           attribute Function ondragenter;
           attribute Function ondragleave;
           attribute Function ondragover;
           attribute Function ondragstart;
           attribute Function ondrop;
           attribute Function onerror;
           attribute Function onfocus;
           attribute Function onkeydown;
           attribute Function onkeypress;
           attribute Function onkeyup;
           attribute Function onload;
           attribute Function onmousedown;
           attribute Function onmousemove;
           attribute Function onmouseout;
           attribute Function onmouseover;
           attribute Function onmouseup;
           attribute Function onmousewheel;
           attribute Function onscroll;
           attribute Function onselect;
           attribute Function onsubmit;
};
```

The `HTMLElement` interface holds methods and attributes related to a number of disparate features, and the members of this interface are therefore described in various different sections of this specification.

3.3.3 Global attributes

The following attributes are common to and may be specified on all HTML elements (even those not defined in this specification):

- `class`
- `contenteditable`
- `contextmenu`

- dir
- draggable
- id
- hidden
- lang
- style
- spellcheck
- tabindex
- title

In addition, unless otherwise specified, the following event handler content attributes may be specified on any HTML element:

- onabort
- onblur
- onchange
- onclick
- oncontextmenu
- ondblclick
- ondrag
- ondragend
- ondragenter
- ondragleave
- ondragover
- ondragstart
- ondrop
- onerror*
- onfocus
- onkeydown
- onkeypress
- onkeyup
- onload*
- onmousedown
- onmousemove
- onmouseout
- onmouseover
- onmouseup
- onmousewheel
- onscroll
- onselect
- onsubmit

Note: The attributes marked with an asterisk cannot be specified on `body` elements as those elements expose event handler attributes of the `Window` object with the same names.

Also, custom data attributes (e.g. `data-foldername` or `data-msgid`) can be specified on any HTML element, to store custom data specific to the page.

In HTML documents, elements in the HTML namespace may have an `xmlns` attribute specified, if, and only if, it has the exact value "`http://www.w3.org/1999/xhtml`". This does not apply to XML documents.

Note: In HTML, the `xmlns` attribute has absolutely no effect. It is basically a talisman. It is allowed merely to make migration to and from XHTML mildly easier. When parsed by an HTML parser, the attribute ends up in no namespace, not the "`http://www.w3.org/2000/xmlns/`" namespace like namespace declaration attributes in XML do.

Note: In XML, an `xmlns` attribute is part of the namespace declaration mechanism, and an element cannot actually have an `xmlns` attribute in no namespace specified.

3.3.3.1 The `id` attribute

The `id` attribute represents its element's unique identifier. The value must be unique in the element's home subtree and must contain at least one character. The value must not contain any space characters.

If the value is not the empty string, user agents must associate the element with the given value (exactly, including any space characters) for the purposes of ID matching within the element's home subtree (e.g. for selectors in CSS or for the `getElementById()` method in the DOM).

Identifiers are opaque strings. Particular meanings should not be derived from the value of the `id` attribute.

This specification doesn't preclude an element having multiple IDs, if other mechanisms (e.g. DOM Core methods) can set an element's ID in a way that doesn't conflict with the `id` attribute.

The **`id`** DOM attribute must reflect the `id` content attribute.

3.3.3.2 The `title` attribute

The `title` attribute represents advisory information for the element, such as would be appropriate for a tooltip. On a link, this could be the title or a description of the target resource; on an image, it could be the image credit or a description of the image; on a paragraph, it could be a footnote or commentary on the text; on a citation, it could be further information about the source; and so forth. The value is text.

If this attribute is omitted from an element, then it implies that the `title` attribute of the nearest ancestor HTML element with a `title` attribute set is also relevant to this element. Setting the attribute overrides this, explicitly stating that the advisory information of any ancestors is not relevant to this element. Setting the attribute to the empty string indicates that the element has no advisory information.

If the `title` attribute's value contains U+000A LINE FEED (LF) characters, the content is split into multiple lines. Each U+000A LINE FEED (LF) character represents a line break.

Some elements, such as `link`, `abbr`, and `input`, define additional semantics for the `title`

attribute beyond the semantics described above.

The **`title`** DOM attribute must reflect the `title` content attribute.

3.3.3.3 The `lang` and `xml:lang` attributes

The `lang` attribute specifies the primary **language** for the element's contents and for any of the element's attributes that contain text. Its value must be a valid RFC 3066 language code, or the empty string. [RFC3066]

The `xml:lang` attribute (that is, the `lang` attribute with the `xml` prefix in the `http://www.w3.org/XML/1998/namespace` namespace) is defined in XML. [XML]

If these attributes are omitted from an element, then the language of this element is the same as the language of its parent element, if any. Setting the attribute to the empty string indicates that the primary language is unknown.

The `lang` attribute may be used on any HTML element.

The `xml:lang` attribute may be used on HTML elements in XML documents, as well as elements in other namespaces if the relevant specifications allow it (in particular, MathML and SVG allow `xml:lang` attributes to be specified on their elements). If both the `lang` attribute and the `xml:lang` attribute are specified on the same element, they must have exactly the same value when compared in an ASCII case-insensitive manner.

Authors must not use the `xml:lang` attribute (that is, the `lang` attribute with the `xml` prefix in the `http://www.w3.org/XML/1998/namespace` namespace) in HTML documents. To ease migration to and from XHTML, authors may specify an attribute in no namespace with no prefix and with the localname `xml:lang` on HTML elements in HTML documents, but such attributes must only be specified if a `lang` attribute is also specified, and both attributes must have the same value when compared in an ASCII case-insensitive manner.

To determine the language of a node, user agents must look at the nearest ancestor element (including the element itself if the node is an element) that has an `xml:lang` attribute set or is an HTML element and has a `lang` attribute set. That attribute specifies the language of the node.

If both the `xml:lang` attribute and the `lang` attribute are set on an element, user agents must use the `xml:lang` attribute, and the `lang` attribute must be ignored for the purposes of determining the element's language.

If no explicit language is given for the root element, but there is a document-wide default language set, then that is the language of the node.

If there is no document-wide default language, then language information from a higher-level protocol (such as HTTP), if any, must be used as the final fallback language. In the absence of any language information, the default value is unknown (the empty string).

If the resulting value is not a recognised language code, then it must be treated as an unknown language (as if the value was the empty string).

User agents may use the element's language to determine proper processing or rendering (e.g. in the selection of appropriate fonts or pronunciations, or for dictionary selection).

The **lang** DOM attribute must reflect the `lang` content attribute.

3.3.3.4 The `xml:base` attribute (XML only)

The `xml:base` attribute is defined in XML Base. [XMLBASE]

The `xml:base` attribute may be used on elements of XML documents. Authors must not use the `xml:base` attribute in HTML documents.

3.3.3.5 The `dir` attribute

The `dir` attribute specifies the element's text directionality. The attribute is an enumerated attribute with the keyword `ltr` mapping to the state *ltr*, and the keyword `rtl` mapping to the state *rtl*. The attribute has no defaults.

The processing of this attribute is primarily performed by the presentation layer. For example, the rendering section in this specification defines a mapping from this attribute to the CSS 'direction' and 'unicode-bidi' properties, and CSS defines rendering in terms of those properties.

The directionality of an element, which is used in particular by the `canvas` element's text rendering API, is either 'ltr' or 'rtl'. If the user agent supports CSS and the 'direction' property on this element has a computed value of either 'ltr' or 'rtl', then that is the directionality of the element. Otherwise, if the element is being rendered, then the directionality of the element is the directionality used by the presentation layer, potentially determined from the value of the `dir` attribute on the element. Otherwise, if the element's `dir` attribute has the state *ltr*, the element's directionality is 'ltr' (left-to-right); if the attribute has the state *rtl*, the element's directionality is 'rtl' (right-to-left); and oherwise, the element's directionality is the same as its parent element, or 'ltr' if there is no parent element.

***document* . `dir` [= *value*]**

Returns the `html` element's `dir` attribute's value, if any.

Can be set, to either "`ltr`" or "`rtl`", to replace the `html` element's `dir` attribute's value.

If there is no `html` element, returns the empty string and ignores new values.

The **dir** DOM attribute on an element must reflect the `dir` content attribute of that element, limited to only known values.

The **dir** DOM attribute on `HTMLDocument` objects must reflect the `dir` content attribute of the `html` element, if any, limited to only known values. If there is no such element, then the attribute must return the empty string and do nothing on setting.

Note: Authors are strongly encouraged to use the `dir` attribute to indicate text direction rather than using CSS, since that way their documents will continue to render correctly even in the absence of CSS (e.g. as interpreted by search engines).

3.3.3.6 The `class` attribute

Every HTML element may have a `class` attribute specified.

The attribute, if specified, must have a value that is an unordered set of unique space-separated tokens representing the various classes that the element belongs to.

The classes that an HTML element has assigned to it consists of all the classes returned when the value of the `class` attribute is split on spaces.

Note: Assigning classes to an element affects class matching in selectors in CSS, the `getElementsByClassName()` method in the DOM, and other such features.

Authors may use any value in the `class` attribute, but are encouraged to use the values that describe the nature of the content, rather than values that describe the desired presentation of the content.

The **`className`** and **`classList`** DOM attributes must both reflect the `class` content attribute.

3.3.3.7 The `style` attribute

All elements may have the `style` content attribute set. If specified, the attribute must contain only a list of zero or more semicolon-separated (;) CSS declarations. [CSS21]

In user agents that support CSS, the attribute's value must be parsed when the attribute is added or has its value changed, with its value treated as the body (the part inside the curly brackets) of a declaration block in a rule whose selector matches just the element on which the attribute is set. All URLs in the value must be resolved relative to the element when the attribute is parsed. For the purposes of the CSS cascade, the attribute must be considered to be a 'style' attribute at the author level.

Documents that use `style` attributes on any of their elements must still be comprehensible and usable if those attributes were removed.

Note: In particular, using the `style` attribute to hide and show content, or to convey meaning that is otherwise not included in the document, is non-conforming. (To hide and show content, use the `hidden` attribute.)

element **. `style`**

Returns a `CSSStyleDeclaration` object for the element's `style` attribute.

The **`style`** DOM attribute must return a `CSSStyleDeclaration` whose value represents the declarations specified in the attribute, if present. Mutating the `CSSStyleDeclaration` object must create a `style` attribute on the element (if there isn't one already) and then change its value to be a value representing the serialized form of the `CSSStyleDeclaration` object. [CSSOM]

> In the following example, the words that refer to colors are marked up using the `span` element and the `style` attribute to make those words show up in the relevant colors in visual media.
>
> ```
> My sweat suit is <span style="color: green; background:
> transparent">green and my eyes are <span style="color: blue;
> background: transparent">blue.
> ```

3.3.3.8 Embedding custom non-visible data

A **custom data attribute** is an attribute whose name starts with the string "**`data-`**", has at least one character after the hyphen, is XML-compatible, has no namespace, and contains no characters in the range U+0041 .. U+005A (LATIN CAPITAL LETTER A .. LATIN CAPITAL LETTER Z).

> ***Note: All attributes in HTML documents get lowercased automatically, so the restriction on uppercase letters doesn't affect such documents.***

Custom data attributes are intended to store custom data private to the page or application, for which there are no more appropriate attributes or elements.

These attributes are not intended for use by software that is independent of the site that uses the attributes.

> For instance, a site about music could annotate list items representing tracks in an album with custom data attributes containing the length of each track. This information could then be used by the site itself to allow the user to sort the list by track length, or to filter the list for tracks of certain lengths.
>
> ```
> <ol>
>  <li data-length="2m11s">Beyond The Sea</li>
>  ...
> </ol>
> ```
>
> It would be inappropriate, however, for the user to use generic software not associated with that music site to search for tracks of a certain length by looking at this data.
>
> This is because these attributes are intended for use by the site's own scripts, and are not a generic extension mechanism for publicly-usable metadata.

Every HTML element may have any number of custom data attributes specified, with any value.

element **. `dataset`**

Returns a `DOMStringMap` object for the element's `data-*` attributes.

The **`dataset`** DOM attribute provides convenient accessors for all the `data-*` attributes on an element. On getting, the `dataset` DOM attribute must return a `DOMStringMap` object, associated with the following algorithms, which expose these attributes on their element:

The algorithm for getting the list of name-value pairs

1. Let *list* be an empty list of name-value pairs.
2. For each content attribute on the element whose first five characters are the string "`data-`", add a name-value pair to *list* whose name is the attribute's name with the first five character removed and whose value is the attribute's value.
3. Return *list*.

The algorithm for setting names to certain values

1. Let *name* be the concatenation of the string `data-` and the name passed to the algorithm.
2. Let *value* be the value passed to the algorithm.
3. Set the value of the attribute with the name *name*, to the value *value*, replacing any previous value if the attribute already existed. If `setAttribute()` would have raised an exception when setting an attribute with the name *name*, then this must raise the same exception.

The algorithm for deleting names

1. Let *name* be the concatenation of the string `data-` and the name passed to the algorithm.
2. Remove the attribute with the name *name*, if such an attribute exists. Do nothing otherwise.

> If a Web page wanted an element to represent a space ship, e.g. as part of a game, it would have to use the `class` attribute along with `data-*` attributes:
>
> ```
> <div class="spaceship" data-id="92432"
>      data-weapons="laser 2" data-shields="50%"
>      data-x="30" data-y="10" data-z="90">
>  <button class="fire"
>          onclick="spaceships[this.parentNode.dataset.id].fire()">
>   Fire
>  </button>
> 
> ```

Authors should carefully design such extensions so that when the attributes are ignored and any associated CSS dropped, the page is still usable.

User agents must not derive any implementation behavior from these attributes or values. Specifications intended for user agents must not define these attributes to have any meaningful values.

3.4 Content models

All the elements in this specification have a defined content model, which describes what nodes are allowed inside the elements, and thus what the structure of an HTML document or fragment must look like.

> ***Note: As noted in the conformance and terminology sections, for the purposes of determining if an element matches its content model or not, `CDATASection` nodes in the DOM are treated as equivalent to `Text` nodes, and entity reference nodes are treated as if they were expanded in place.***

The space characters are always allowed between elements. User agents represent these characters between elements in the source markup as text nodes in the DOM. Empty text nodes and text nodes consisting of just sequences of those characters are considered **inter-element whitespace**.

Inter-element whitespace, comment nodes, and processing instruction nodes must be ignored when establishing whether an element matches its content model or not, and must be ignored when following algorithms that define document and element semantics.

An element *A* is said to be **preceded or followed** by a second element *B* if *A* and *B* have the same parent node and there are no other element nodes or text nodes (other than inter-element whitespace) between them.

Authors must not use elements in the HTML namespace anywhere except where they are explicitly allowed, as defined for each element, or as explicitly required by other specifications. For XML compound documents, these contexts could be inside elements from other namespaces, if those elements are defined as providing the relevant contexts.

> The Atom specification defines the Atom `content` element, when its `type` attribute has the value `xhtml`, as requiring that it contains a single HTML `div` element. Thus, a `div` element is allowed in that context, even though this is not explicitly normatively stated by this specification. [ATOM]

In addition, elements in the HTML namespace may be orphan nodes (i.e. without a parent node).

> For example, creating a `td` element and storing it in a global variable in a script is conforming, even though `td` elements are otherwise only supposed to be used inside `tr` elements.
>
> ```
> var data = {
>   name: "Banana",
>   cell: document.createElement('td'),
> ```

```
    };
```

3.4.1 Kinds of content

Each element in HTML falls into zero or more categories that group elements with similar characteristics together. The following broad categories are used in this specification:

- Metadata content
- Flow content
- Sectioning content
- Heading content
- Phrasing content
- Embedded content
- Interactive content

These categories are related as follows:

In addition, certain elements are categorised as form-associated elements and further subcategorised so so to define their role in various form-related processing models.

Some elements have unique requirements and do not fit into any particular category.

3.4.1.1 Metadata content

Metadata content is content that sets up the presentation or behavior of the rest of the content, or that sets up the relationship of the document with other documents, or that conveys other "out of band" information.

Elements from other namespaces whose semantics are primarily metadata-related (e.g. RDF) are also metadata content.

> Thus, in the XML serialisation, one can use RDF, like this:
>
> ```
> <html xmlns="http://www.w3.org/1999/xhtml"
>       xmlns:r="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
> ```

```
     <head>
      <title>Hedral's Home Page</title>
      <r:RDF>
       <Person xmlns="http://www.w3.org/2000/10/swap/pim/contact#"
               r:about="http://hedral.example.com/#">
        <fullName>Cat Hedral</fullName>
        <mailbox r:resource="mailto:hedral@damowmow.com"/>
        <personalTitle>Sir</personalTitle>
       </Person>
      </r:RDF>
     </head>
     <body>
      <h1>My home page</h1>
      <p>I like playing with string, I guess. Sister says squirrels are fun
      too so sometimes I follow her to play with them.</p>
     </body>
    </html>
```

This isn't possible in the HTML serialisation, however.

3.4.1.2 Flow content

Most elements that are used in the body of documents and applications are categorized as **flow content**.

As a general rule, elements whose content model allows any flow content should have either at least one descendant text node that is not inter-element whitespace, or at least one descendant element node that is embedded content. For the purposes of this requirement, `del` elements and their descendants must not be counted as contributing to the ancestors of the `del` element.

This requirement is not a hard requirement, however, as there are many cases where an element can be empty legitimately, for example when it is used as a placeholder which will later be filled in by a script, or when the element is part of a template and would on most pages be filled in but on some pages is not relevant.

3.4.1.3 Sectioning content

Sectioning content is content that defines the scope of headers, footers, and contact information.

Each sectioning content element potentially has a heading and an outline. See the section on headings and sections for further details.

> ***Note: There are also certain elements that are sectioning roots. These are distinct from sectioning content, but they can also have an outline.***

3.4.1.4 Heading content

Heading content defines the header of a section (whether explicitly marked up using sectioning content elements, or implied by the heading content itself).

3.4.1.5 Phrasing content

Phrasing content is the text of the document, as well as elements that mark up that text at the intra-paragraph level. Runs of phrasing content form paragraphs.

As a general rule, elements whose content model allows any phrasing content should have either at least one descendant text node that is not inter-element whitespace, or at least one descendant element node that is embedded content. For the purposes of this requirement, nodes that are descendants of `del` elements must not be counted as contributing to the ancestors of the `del` element.

> ***Note: Most elements that are categorized as phrasing content can only contain elements that are themselves categorized as phrasing content, not any flow content.***

Text nodes that are not inter-element whitespace are phrasing content.

3.4.1.6 Embedded content

Embedded content is content that imports another resource into the document, or content from another vocabulary that is inserted into the document.

Elements that are from namespaces other than the HTML namespace and that convey content but not metadata, are embedded content for the purposes of the content models defined in this specification. (For example, MathML, or SVG.)

Some embedded content elements can have **fallback content**: content that is to be used when the external resource cannot be used (e.g. because it is of an unsupported format). The element definitions state what the fallback is, if any.

3.4.1.7 Interactive content

Interactive content is content that is specifically intended for user interaction.

Certain elements in HTML have an activation behavior, which means that the user can activate them. This triggers a sequence of events dependent on the activation mechanism, and normally culminating in a `click` event followed by a `DOMActivate` event, as described below.

The user agent should allow the user to manually trigger elements that have an activation behavior, for instance using keyboard or voice input, or through mouse clicks. When the user triggers an element with a defined activation behavior in a manner other than clicking it, the default action of the interaction event must be to run synthetic click activation steps on the element.

When a user agent is to **run synthetic click activation steps** on an element, the user agent must run pre-click activation steps on the element, then fire a `click` event at the element. The default action of this click event must be to run post-click activation steps on the element. If

the event is canceled, the user agent must run canceled activation steps on the element instead.

Given an element *target*, the **nearest activatable element** is the element returned by the following algorithm:

1. If *target* has a defined activation behavior, then return *target* and abort these steps.
2. If *target* has a parent element, then set *target* to that parent element and return to the first step.
3. Otherwise, there is no nearest activatable element.

When a pointing device is clicked, the user agent must run these steps:

1. Let *e* be the nearest activatable element of the element designated by the user, if any.
2. If there is an element *e*, run pre-click activation steps on it.
3. Dispatching the required `click` event.

   Another specification presumably requires the firing of the click event?

   If there is an element *e*, then the default action of the click event must be to run post-click activation steps on element *e*.

   If there is an element *e* but the event is canceled, the user agent must run canceled activation steps on element *e*.

Note: The above doesn't happen for arbitrary synthetic events dispatched by author script. However, the `click()` method can be used to make it happen programmatically.

When a user agent is to **run post-click activation steps** on an element, the user agent must fire a simple event called `DOMActivate` at that element. The default action of this event must be to run final activation steps on that element. If the event is canceled, the user agent must run canceled activation steps on the element instead.

When a user agent is to **run pre-click activation steps** on an element, it must run the **pre-click activation steps** defined for that element, if any.

When a user agent is to **run canceled activation steps** on an element, it must run the **canceled activation steps** defined for that element, if any.

When a user agent is to **run final activation steps** on an element, it must run the **activation behavior** defined for that element. Activation behaviors can refer to the `click` and `DOMActivate` events that were fired by the steps above leading up to this point.

3.4.2 Transparent content models

Some elements are described as **transparent**; they have "transparent" in the description of

their content model.

When a content model includes a part that is "transparent", those parts must not contain content that would not be conformant if all transparent elements in the tree were replaced, in their parent element, by the children in the "transparent" part of their content model, retaining order.

When a transparent element has no parent, then the part of its content model that is "transparent" must instead be treated as accepting any flow content.

3.5 Paragraphs

A **paragraph** is typically a block of text with one or more sentences that discuss a particular topic, as in typography, but can also be used for more general thematic grouping. For instance, an address is also a paragraph, as is a part of a form, a byline, or a stanza in a poem.

Paragraphs in flow content are defined relative to what the document looks like without the `a`, `ins` and `del` elements complicating matters, since those elements, with their hybrid content models, can straddle paragraph boundaries.

Let *view* be a view of the DOM that replaces all `a`, `ins` and `del` elements in the document with their contents. Then, in *view*, for each run of phrasing content uninterrupted by other types of content, in an element that accepts content other than phrasing content, let *first* be the first node of the run, and let *last* be the last node of the run. For each run, a paragraph exists in the original DOM from immediately before *first* to immediately after *last*. (Paragraphs can thus span across `a`, `ins` and `del` elements.)

A paragraph is also formed explicitly by `p` elements.

Note: The `p` element can be used to wrap individual paragraphs when there would otherwise not be any content other than phrasing content to separate the paragraphs from each other.

In the following example, there are two paragraphs in a section. There is also a header, which contains phrasing content that is not a paragraph. Note how the comments and intra-element whitespace do not form paragraphs.

```
<section>
  <h1>Example of paragraphs</h1>
  This is the <em>first</em> paragraph in this example.
  <p>This is the second.</p>
  <!-- This is not a paragraph. -->
</section>
```

The following example takes that markup and puts `ins` and `del` elements around some of the markup to show that the text was changed (though in this case, the changes don't really make much sense, admittedly). Notice how this example has exactly the same paragraphs as the previous one, despite the `ins` and `del` elements.

```
<section>
  <ins><h1>Example of paragraphs</h1>
  This is the <em>first</em> paragraph in</ins> this example<del>.
```

```
      <p>This is the second.</p></del>
      <!-- This is not a paragraph. -->
    </section>
```

In the following example, the link spans half of the first paragraph, all of the header separating the two paragraphs, and half of the second paragraph.

```
<aside>
 Welcome!
 <a href="about.html">
  This is home of...
  <h1>The Falcons!</h1>
  The Lockheed Martin multirole jet fighter aircraft!
 </a>
 This page discusses the F-16 Fighting Falcon's innermost secrets.
</aside>
```

Here is another way of marking this up, this time showing the paragraphs explicitly, and splitting the one link element into three:

```
<aside>
 <p>Welcome! <a href="about.html">This is home of...</a></p>
 <h1><a href="about.html">The Falcons!</a></h1>
 <p><a href="about.html">The Lockheed Martin multirole jet
 fighter aircraft!</a> This page discusses the F-16 Fighting
 Falcon's innermost secrets.</p>
</aside>
```

Note: Generally, having elements straddle paragraph boundaries is best avoided. Maintaining such markup can be difficult.

3.6 APIs in HTML documents

For HTML documents, and for HTML elements in HTML documents, certain APIs defined in DOM3 Core become case-insensitive or case-changing, as sometimes defined in DOM3 Core, and as summarized or required below. [DOM3CORE].

This does not apply to XML documents or to elements that are not in the HTML namespace despite being in HTML documents.

`Element.tagName` and `Node.nodeName`

These attributes must return element names converted to uppercase, regardless of the case with which they were created.

`Document.createElement()`

The canonical form of HTML markup is all-lowercase; thus, this method will lowercase the argument before creating the requisite element. Also, the element created must be in the HTML namespace.

Note: This doesn't apply to `Document.createElementNS()`. Thus, it is possible, by passing this last method a tag name in the wrong case, to create an element that claims to have the tag name of an element defined in this specification, but doesn't support its interfaces, because it really has another tag name not accessible from the DOM APIs.

`Element.setAttribute()`
`Element.setAttributeNode()`

Attribute names are converted to lowercase.

Specifically: when an attribute is set on an HTML element using `Element.setAttribute()`, the name argument must be converted to lowercase before the element is affected; and when an `Attr` node is set on an HTML element using `Element.setAttributeNode()`, it must have its name converted to lowercase before the element is affected.

> ***Note: This doesn't apply to `Document.setAttributeNS()` and `Document.setAttributeNodeNS()`.***

`Document.getElementsByTagName()`
`Element.getElementsByTagName()`

These methods (but not their namespaced counterparts) must compare the given argument in an ASCII case-insensitive manner when looking at HTML elements, and in a case-sensitive manner otherwise.

> ***Note: Thus, in an HTML document with nodes in multiple namespaces, these methods will be both case-sensitive and case-insensitive at the same time.***

`Document.renameNode()`

HTML element names are converted to lowercase when the elements are created. Thus, if the new namespace passed to the `Document.renameNode()` method is the HTML namespace, the new qualified name must be converted to lowercase before the rename takes place.

3.7 Dynamic markup insertion

> ***Note: APIs for dynamically inserting markup into the document interact with the parser, and thus their behavior, varies depending on whether they are used with HTML documents (and the HTML parser) or XHTML in XML documents (and the XML parser).***

3.7.1 Controlling the input stream

The **`open()`** method comes in several variants with different numbers of arguments.

***document* = *document* . open([*type* [, *replace*]])**

Causes the `Document` to be replaced in-place, as if it was a new `Document` object, but reusing the previous object, which is then returned.

If the *type* argument is omitted or has the value "`text/html`", then the resulting

`Document` has an HTML parser associated with it, which can be given data to parse using `document.write()`. Otherwise, all content passed to `document.write()` will be parsed as plain text.

If the *replace* argument is absent or false, a new entry is added to the session history to represent this entry, and the previous entries for this `Document` are all collapsed into one entry with a new `Document` object.

The method has no effect if the `Document` is still being parsed.

***window* = *document* . `open`(*url*, *name*, *features* [, *replace*])**

Works like the `window.open()` method.

***document* . `close`()**

Closes the input stream that was opened by the `document.open()` method.

When called with two or fewer arguments, the method must act as follows:

1. Let *type* be the value of the first argument, if there is one, or "`text/html`" otherwise.
2. Let *replace* be true if there is a second argument and it is an ASCII case-insensitive match for the value "replace", and false otherwise.
3. If the document has an active parser that isn't a script-created parser, and the insertion point associated with that parser's input stream is not undefined (that is, it *does* point to somewhere in the input stream), then the method does nothing. Abort these steps and return the `Document` object on which the method was invoked.

   Note: This basically causes `document.open()` to be ignored when it's called in an inline script found during the parsing of data sent over the network, while still letting it have an effect when called asynchronously or on a document that is itself being spoon-fed using these APIs.

4. Unload the `Document` object, with the *recycle* parameter set to true. If the user refused to allow the document to be unloaded, then these steps must be aborted.
5. If the document has an active parser, then stop that parser, and throw away any pending content in the input stream. what about if it doesn't, because it's either like a text/plain, or Atom, or PDF, or XHTML, or image document, or something?
6. Unregister all event listeners registered on the `Document` node and its descendants.
7. Remove all child nodes of the document, without firing any mutation events.
8. Replace the `Document`'s singleton objects with new instances of those objects. (This includes in particular the `Window`, `Location`, `History`, `ApplicationCache`, `UndoManager`, `Navigator`, and `Selection` objects, the various `BarProp` objects, the two `Storage` objects, and the various `HTMLCollection` objects. It also includes all the WebIDL prototypes in

the ECMAScript binding, including the `Document` object's prototype.)

9. Change the document's character encoding to UTF-16.

10. Change the document's address to the first script's browsing context's active document's address.

11. Create a new HTML parser and associate it with the document. This is a **script-created parser** (meaning that it can be closed by the `document.open()` and `document.close()` methods, and that the tokeniser will wait for an explicit call to `document.close()` before emitting an end-of-file token). The encoding confidence is *irrelevant*.

12. Mark the document as being an HTML document (it might already be so-marked).

13. If the *type* string contains a U+003B SEMICOLON (;) character, remove the first such character and all characters from it up to the end of the string.

    Strip all leading and trailing space characters from *type*.

    If *type* is *not* now an ASCII case-insensitive match for the string "`text/html`", then act as if the tokeniser had emitted a start tag token with the tag name "pre", then set the HTML parser's tokenization stage's content model flag to *PLAINTEXT*.

14. If *replace* is false, then:

    1. Remove all the entries in the browsing context's session history after the current entry in its `Document`'s `History` object

    2. Remove any earlier entries that share the same `Document`

    3. Add a new entry just before the last entry that is associated with the text that was parsed by the previous parser associated with the `Document` object, as well as the state of the document at the start of these steps. (This allows the user to step backwards in the session history to see the page before it was blown away by the `document.open()` call.)

15. Finally, set the insertion point to point at just before the end of the input stream (which at this point will be empty).

16. Return the `Document` on which the method was invoked.

When called with three or more arguments, the `open()` method on the `HTMLDocument` object must call the `open()` method on the `Window` object of the `HTMLDocument` object, with the same arguments as the original call to the `open()` method, and return whatever that method returned. If the `HTMLDocument` object has no `Window` object, then the method must raise an `INVALID_ACCESS_ERR` exception.

The **`close()`** method must do nothing if there is no script-created parser associated with the document. If there is such a parser, then, when the method is called, the user agent must insert an explicit "EOF" character at the end of the parser's input stream.

3.7.2 `document.write()`

document **.** `write`**(*text...*)**

Adds the given string(s) to the `Document`'s input stream. If necessary, calls the `open()` method implicitly first.

This method throws an `INVALID_ACCESS_ERR` exception when invoked on XML documents.

The **`document.write(...)`** method must act as follows:

1. If the method was invoked on an XML document, throw an `INVALID_ACCESS_ERR` exception and abort these steps.
2. If the insertion point is undefined, the `open()` method must be called (with no arguments) on the `document` object. If the user refused to allow the document to be unloaded, then these steps must be aborted. Otherwise, the insertion point will point at just before the end of the (empty) input stream.
3. The string consisting of the concatenation of all the arguments to the method must be inserted into the input stream just before the insertion point.
4. If there is a pending external script, then the method must now return without further processing of the input stream.
5. Otherwise, the tokeniser must process the characters that were inserted, one at a time, processing resulting tokens as they are emitted, and stopping when the tokeniser reaches the insertion point or when the processing of the tokeniser is aborted by the tree construction stage (this can happen if a `script` end tag token is emitted by the tokeniser).

   Note: If the `document.write()` method was called from script executing inline (i.e. executing because the parser parsed a set of `script` tags), then this is a reentrant invocation of the parser.

6. Finally, the method must return.

3.7.3 `document.writeln()`

document **.** `writeln`**(*text...*)**

Adds the given string(s) to the `Document`'s input stream, followed by a newline character. If necessary, calls the `open()` method implicitly first.

This method throws an `INVALID_ACCESS_ERR` exception when invoked on XML documents.

The **`document.writeln(...)`** method, when invoked, must act as if the `document.write()` method had been invoked with the same argument(s), plus an extra argument consisting of a string containing a single line feed character (U+000A).

3.7.4 `innerHTML`

The **`innerHTML`** DOM attribute represents the markup of the node's contents.

document **`.innerHTML`** **[= *value*]**

Returns a fragment of HTML or XML that represents the `Document`.

Can be set, to replace the `Document`'s contents with the result of parsing the given string.

In the case of XML documents, will throw a `SYNTAX_ERR` if the `Document` cannot be serialized to XML, or if the given string is not well-formed.

element **`.innerHTML`** **[= *value*]**

Returns a fragment of HTML or XML that represents the element's contents.

Can be set, to replace the contents of the element with nodes parsed from the given string.

In the case of XML documents, will throw a `SYNTAX_ERR` if the element cannot be serialized to XML, or if the given string is not well-formed.

On getting, if the node's document is an HTML document, then the attribute must return the result of running the HTML fragment serialization algorithm on the node; otherwise, the node's document is an XML document, and the attribute must return the result of running the XML fragment serialization algorithm on the node instead (this might raise an exception instead of returning a string).

On setting, the following steps must be run:

1. If the node's document is an HTML document: Invoke the HTML fragment parsing algorithm.

   If the node's document is an XML document: Invoke the XML fragment parsing algorithm.

   In either case, the algorithm must be invoked with the string being assigned into the `innerHTML` attribute as the *input*. If the node is an `Element` node, then, in addition, that element must be passed as the *context* element.

   If this raises an exception, then abort these steps.

   Otherwise, let *new children* be the nodes returned.

2. If the attribute is being set on a `Document` node, and that document has an active HTML

parser or XML parser, then stop that parser.

what about if it doesn't, because it's either like a text/plain, or Atom, or PDF, or XHTML, or image document, or something?

3. Remove the child nodes of the node whose `innerHTML` attribute is being set.

4. If the attribute is being set on a `Document` node, let *target document* be that `Document` node. Otherwise, the attribute is being set on an `Element` node; let *target document* be the `ownerDocument` of that `Element`.

5. Set the `ownerDocument` of all the nodes in *new children* to the *target document*.

6. Append all the *new children* nodes to the node whose `innerHTML` attribute is being set, preserving their order, without firing any mutation events.

3.7.5 `outerHTML`

The **`outerHTML`** DOM attribute represents the markup of the element and its contents.

element **`.outerHTML`** **[= *value*]**

Returns a fragment of HTML or XML that represents the element and its contents.

Can be set, to replace the element with nodes parsed from the given string.

In the case of XML documents, will throw a `SYNTAX_ERR` if the element cannot be serialized to XML, or if the given string is not well-formed.

On getting, if the node's document is an HTML document, then the attribute must return the result of running the HTML fragment serialization algorithm on a fictional node whose only child is the node on which the attribute was invoked; otherwise, the node's document is an XML document, and the attribute must return the result of running the XML fragment serialization algorithm on that fictional node instead (this might raise an exception instead of returning a string).

On setting, the following steps must be run:

1. Let *target* be the element whose `outerHTML` attribute is being set.

2. If *target* has no parent node, then abort these steps. There would be no way to obtain a reference to the nodes created even if the remaining steps were run.

3. If *target*'s parent node is a `Document` object, throw a `NO_MODIFICATION_ALLOWED_ERR` exception and abort these steps.

4. Let *parent* be *target*'s parent node, unless that is a `DocumentFragment` node, in which case let *parent* be an arbitrary `body` element.

5. If *target*'s document is an HTML document: Invoke the HTML fragment parsing algorithm.

   If *target*'s document is an XML document: Invoke the XML fragment parsing algorithm.

   In either case, the algorithm must be invoked with the string being assigned into the `outerHTML` attribute as the *input*, and *parent* as the *context* element.

   If this raises an exception, then abort these steps.

   Otherwise, let *new children* be *target*s returned.

6. Set the `ownerDocument` of all the nodes in *new children* to *target*'s document.

7. Remove *target* from its parent node and insert in its place all the *new children* nodes, preserving their order, without firing any mutation events.

3.7.6 `insertAdjacentHTML()`

element **`.insertAdjacentHTML(`*position*`, `*text*`)`**

Parsed the given string *text* as HTML or XML and inserts the resulting nodes into the tree in the position given by the *position* argument, as follows:

"beforebegin"

Before the element itself.

"afterbegin"

Just inside the element, before its first child.

"beforeend"

Just inside the element, after its last child.

"afterend"

After the element itself.

Throws a `SYNTAX_ERR` exception the arguments have invalid values (e.g., in the case of XML documents, if the given string is not well-formed).

Throws a `NO_MODIFICATION_ALLOWED_ERR` exception if the given position isn't possible (e.g. inserting elements after the root element of a `Document`).

The **`insertAdjacentHTML(`*`position, text`*`)`** method, when invoked, must run the following algorithm:

1. Let *position* and *text* be the method's first and second arguments, respectively.

2. Let *target* be the element on which the method was invoked.

3. Use the first matching item from this list:

   If *position* is an ASCII case-insensitive match for the string "beforebegin"
   If *position* is an ASCII case-insensitive match for the string "afterend"
   If *target* has no parent node, then abort these steps.

   If *target*'s parent node is a `Document` object, then throw a `NO_MODIFICATION_ALLOWED_ERR` exception and abort these steps.

   Otherwise, let *context* be the parent node of *target*.

   If *position* is an ASCII case-insensitive match for the string "afterbegin"
   If *position* is an ASCII case-insensitive match for the string "beforeend"
   Let *context* be the same as *target*.

   Otherwise
   Throw a `SYNTAX_ERR` exception.

4. If *target*'s document is an HTML document: Invoke the HTML fragment parsing algorithm.

   If *target*'s document is an XML document: Invoke the XML fragment parsing algorithm.

   In either case, the algorithm must be invoked with *text* as the *input*, and the element selected in by the previous step as the *context* element.

   If this raises an exception, then abort these steps.

   Otherwise, let *new children* be *target*s returned.

5. Set the `ownerDocument` of all the nodes in *new children* to *target*'s document.

6. Use the first matching item from this list:

   If *position* is an ASCII case-insensitive match for the string "beforebegin"
   Insert all the *new children* nodes immediately before *target*, preserving their order.

   If *position* is an ASCII case-insensitive match for the string "afterbegin"
   Insert all the *new children* nodes before the first child of *target*, if there is one, preserving their order. If there is no such child, append them all to *target*, preserving their order.

   If *position* is an ASCII case-insensitive match for the string "beforeend"
   Append all the *new children* nodes to *target*, preserving their order.

   If *position* is an ASCII case-insensitive match for the string "afterend"
   Insert all the *new children* nodes immediately after *target*, preserving their order.

   The above mutations must be performed without firing any mutation events.

4 The elements of HTML

4.1 The root element

4.1.1 The `html` element

Categories
None.
Contexts in which this element may be used:
As the root element of a document.
Wherever a subdocument fragment is allowed in a compound document.
Content model:
A `head` element followed by a `body` element.
Content attributes:
Global attributes
`manifest`
DOM interface:
Uses `HTMLElement`.

The `html` element represents the root of an HTML document.

The **`manifest`** attribute gives the address of the document's application cache manifest, if there is one. If the attribute is present, the attribute's value must be a valid URL.

The `manifest` attribute only has an effect during the early stages of document load. Changing the attribute dynamically thus has no effect (and thus, no DOM API is provided for this attribute).

> ***Note: For the purposes of application cache selection, later `base` elements cannot affect the resolving of relative URLs in `manifest` attributes, as the attributes are processed before those elements are seen.***

4.2 Document metadata

4.2.1 The `head` element

Categories
None.
Contexts in which this element may be used:
As the first element in an `html` element.
Content model:
One or more elements of metadata content, of which exactly one is a `title` element.
Content attributes:

Global attributes

DOM interface:

Uses `HTMLElement`.

The `head` element represents a collection of metadata for the `Document`.

4.2.2 The `title` element

Categories

Metadata content.

Contexts in which this element may be used:

In a `head` element containing no other `title` elements.

Content model:

Text.

Content attributes:

Global attributes

DOM interface:

Uses `HTMLElement`.

The `title` element represents the document's title or name. Authors should use titles that identify their documents even when they are used out of context, for example in a user's history or bookmarks, or in search results. The document's title is often different from its first header, since the first header does not have to stand alone when taken out of context.

There must be no more than one `title` element per document.

The `title` element must not contain any elements.

> Here are some examples of appropriate titles, contrasted with the top-level headers that might be used on those same pages.
>
> ```
> <title>Introduction to The Mating Rituals of Bees</title>
>   ...
> <h1>Introduction</h1>
> <p>This companion guide to the highly successful
> <cite>Introduction to Medieval Bee-Keeping</cite> book is...
> ```
>
> The next page might be a part of the same site. Note how the title describes the subject matter unambiguously, while the first header assumes the reader knowns what the context is and therefore won't wonder if the dances are Salsa or Waltz:
>
> ```
> <title>Dances used during bee mating rituals</title>
>   ...
> <h1>The Dances</h1>
> ```

The string to use as the document's title is given by the `document.title` DOM attribute. User agents should use the document's title when referring to the document in their user interface.

4.2.3 The `base` element

Categories

Metadata content.

Contexts in which this element may be used:
In a `head` element containing no other `base` elements.
Content model:
Empty.
Content attributes:
Global attributes
`href`
`target`
DOM interface:

```
interface HTMLBaseElement : HTMLElement {
           attribute DOMString href;
           attribute DOMString target;
};
```

The `base` element allows authors to specify the document base URL for the purposes of resolving relative URLs, and the name of the default browsing context for the purposes of following hyperlinks. The element does not represent any content beyond this information.

There must be no more than one `base` element per document.

A `base` element must have either an `href` attribute, a `target` attribute, or both.

The **`href`** content attribute, if specified, must contain a valid URL.

A `base` element, if it has an `href` attribute, must come before any other elements in the tree that have attributes defined as taking URLs, except the `html` element (its `manifest` attribute isn't affected by `base` elements).

Note: If there are multiple `base` elements with `href` attributes, all but the first are ignored.

The **`target`** attribute, if specified, must contain a valid browsing context name or keyword, which specifies which browsing context is to be used as the default when hyperlinks and forms in the `Document` cause navigation.

A `base` element, if it has a `target` attribute, must come before any elements in the tree that represent hyperlinks.

Note: If there are multiple `base` elements with `target` attributes, all but the first are ignored.

The **`href`** and **`target`** DOM attributes must reflect the respective content attributes of the same name.

4.2.4 The `link` element

Categories
Metadata content.

Contexts in which this element may be used:

Where metadata content is expected.

In a `noscript` element that is a child of a `head` element.

Content model:

Empty.

Content attributes:

Global attributes

`href`

`rel`

`media`

`hreflang`

`type`

`sizes`

Also, the `title` attribute has special semantics on this element.

DOM interface:

```
interface HTMLLinkElement : HTMLElement {
           attribute boolean disabled;
           attribute DOMString href;
           attribute DOMString rel;
  readonly attribute DOMTokenList relList;
           attribute DOMString media;
           attribute DOMString hreflang;
           attribute DOMString type;
           attribute DOMString sizes;
};
```

The `LinkStyle` interface must also be implemented by this element, the styling processing model defines how. [CSSOM]

The `link` element allows authors to link their document to other resources.

The destination of the link(s) is given by the **`href`** attribute, which must be present and must contain a valid URL. If the `href` attribute is absent, then the element does not define a link.

The types of link indicated (the relationships) are given by the value of the **`rel`** attribute, which must be present, and must have a value that is a set of space-separated tokens. The allowed values and their meanings are defined in a later section. If the `rel` attribute is absent, or if the values used are not allowed according to the definitions in this specification, then the element does not define a link.

Two categories of links can be created using the `link` element. **Links to external resources** are links to resources that are to be used to augment the current document, and **hyperlink links** are links to other documents. The link types section defines whether a particular link type is an external resource or a hyperlink. One element can create multiple links (of which some might be external resource links and some might be hyperlinks); exactly which and how many links are created depends on the keywords given in the `rel` attribute. User agents must process the links on a per-link basis, not a per-element basis.

Note: Each link is handled separately. For instance, if there are two `link` elements with `rel="stylesheet"`, they each count as a separate external resource, and each is affected by its own attributes independently.

The exact behavior for links to external resources depends on the exact relationship, as defined for the relevant link type. Some of the attributes control whether or not the external resource is to be applied (as defined below). For external resources that are represented in the DOM (for example, style sheets), the DOM representation must be made available even if the resource is not applied. To obtain the resource, the user agent must resolve the URL given by the `href` attribute, relative to the element, and then fetch the resulting absolute URL. User agents may opt to only fetch such resources when they are needed, instead of pro-actively fetching all the external resources that are not applied.

The semantics of the protocol used (e.g. HTTP) must be followed when fetching external resources. (For example, redirects must be followed and 404 responses must cause the external resource to not be applied.)

Once a resource *and any subresources it uses* have been fetched, if the loads were successful, the user agent must queue a task to fire a simple event called `load` at the `link` element. If the resource or one of its subresources fails to completely load for any reason (e.g. DNS error, HTTP 404 response, the connection being prematurely closed, unsupported Content-Type), the user agent must instead queue a task to fire a simple event called `error` at the `link` element. Non-network errors in processing the resource or its subresources (e.g. CSS parse errors, PNG decoding errors) are not failures for the purposes of this paragraph.

Interactive user agents should provide users with a means to follow the hyperlinks created using the `link` element, somewhere within their user interface. The exact interface is not defined by this specification, but it should include the following information (obtained from the element's attributes, again as defined below), in some form or another (possibly simplified), for each hyperlink created with each `link` element in the document:

- The relationship between this document and the resource (given by the `rel` attribute)
- The title of the resource (given by the `title` attribute).
- The address of the resource (given by the `href` attribute).
- The language of the resource (given by the `hreflang` attribute).
- The optimum media for the resource (given by the `media` attribute).

User agents may also include other information, such as the type of the resource (as given by the `type` attribute).

> ***Note: Hyperlinks created with the `link` element and its `rel` attribute apply to the whole page. This contrasts with the `rel` attribute of `a` and `area` elements, which indicates the type of a link whose context is given by the link's location within the document.***

The **`media`** attribute says which media the resource applies to. The value must be a valid media query. [MQ]

If the link is a hyperlink then the `media` attribute is purely advisory, and describes for which media the document in question was designed.

However, if the link is an external resource link, then the `media` attribute is prescriptive. The user agent must apply the external resource to views while their state match the listed media and the other relevant conditions apply, and must not apply them otherwise.

> ***Note: The external resource might have further restrictions defined within that limit its applicability. For example, a CSS style sheet might have some `@media` blocks. This specification does not override such further restrictions or requirements.***

The default, if the `media` attribute is omitted, is `all`, meaning that by default links apply to all media.

The **`hreflang`** attribute on the `link` element has the same semantics as the `hreflang` attribute on hyperlink elements.

The **`type`** attribute gives the MIME type of the linked resource. It is purely advisory. The value must be a valid MIME type, optionally with parameters. [RFC2046]

For external resource links, the `type` attribute is used as a hint to user agents so that they can avoid fetching resources they do not support. If the attribute is present, then the user agent must assume that the resource is of the given type. If the attribute is omitted, but the external resource link type has a default type defined, then the user agent must assume that the resource is of that type. If the UA does not support the given MIME type for the given link relationship, then the UA should not fetch the resource; if the UA does support the given MIME type for the given link relationship, then the UA should fetch the resource. If the attribute is omitted, and the external resource link type does not have a default type defined, but the user agent would fetch the resource if the type was known and supported, then the user agent should fetch the resource under the assumption that it will be supported.

User agents must not consider the `type` attribute authoritative — upon fetching the resource, user agents must not use the `type` attribute to determine its actual type. Only the actual type (as defined in the next paragraph) is used to determine whether to *apply* the resource, not the aforementioned assumed type.

If the resource is expected to be an image, user agents may apply the image sniffing rules, with the *official type* being the type determined from the resource's Content-Type metadata, and use the resulting sniffed type of the resource as if it was the actual type. Otherwise, if the resource is not expected to be an image, or if the user agent opts not to apply those rules, then the user agent must use the resource's Content-Type metadata to determine the type of the resource. If there is no type metadata, but the external resource link type has a default type defined, then the user agent must assume that the resource is of that type.

Once the user agent has established the type of the resource, the user agent must apply the resource if it is of a supported type and the other relevant conditions apply, and must ignore the resource otherwise.

> If a document contains style sheet links labeled as follows:
>
> ```
> <link rel="stylesheet" href="A" type="text/plain">
> <link rel="stylesheet" href="B" type="text/css">
> <link rel="stylesheet" href="C">
> ```
>
> ...then a compliant UA that supported only CSS style sheets would fetch the B and C

files, and skip the A file (since `text/plain` is not the MIME type for CSS style sheets).

For files B and C, it would then check the actual types returned by the server. For those that are sent as `text/css`, it would apply the styles, but for those labeled as `text/plain`, or any other type, it would not.

If one the two files was returned without a Content-Type metadata, or with a syntactically incorrect type like `Content-Type: "null"`, then the default type for `stylesheet` links would kick in. Since that default type is `text/css`, the style sheet *would* nonetheless be applied.

The **`title`** attribute gives the title of the link. With one exception, it is purely advisory. The value is text. The exception is for style sheet links, where the `title` attribute defines alternative style sheet sets.

Note: The `title` attribute on `link` elements differs from the global `title` attribute of most other elements in that a link without a title does not inherit the title of the parent element: it merely has no title.

The `sizes` attribute is used with the `icon` link type. The attribute must not be specified on `link` elements that do not have a `rel` attribute that specifies the `icon` keyword.

Some versions of HTTP defined a `Link:` header, to be processed like a series of `link` elements. If supported, for the purposes of ordering links defined by HTTP headers must be assumed to come before any links in the document, in the order that they were given in the HTTP entity header. (URIs in these headers are to be processed and resolved according to the rules given in HTTP; the rules of *this* specification don't apply.) [RFC2616] [RFC2068]

The DOM attributes **`href`**, **`rel`**, **`media`**, **`hreflang`**, and **`type`**, and **`sizes`** each must reflect the respective content attributes of the same name.

The DOM attribute **`relList`** must reflect the `rel` content attribute.

The DOM attribute **`disabled`** only applies to style sheet links. When the `link` element defines a style sheet link, then the `disabled` attribute behaves as defined for the alternative style sheets DOM. For all other `link` elements it always return false and does nothing on setting.

4.2.5 The `meta` element

Categories
Metadata content.
Contexts in which this element may be used:
If the `charset` attribute is present, or if the element is in the Encoding declaration state: in a `head` element.
If the `http-equiv` attribute is present, and the element is not in the Encoding declaration state: in a `head` element.
If the `http-equiv` attribute is present, and the element is not in the Encoding declaration state: in a `noscript` element that is a child of a `head` element.
If the `name` attribute is present: where metadata content is expected.
Content model:
Empty.

Content attributes:

Global attributes
`name`
`http-equiv`
`content`
`charset`

DOM interface:

```
interface HTMLMetaElement : HTMLElement {
           attribute DOMString content;
           attribute DOMString name;
           attribute DOMString httpEquiv;
};
```

The `meta` element represents various kinds of metadata that cannot be expressed using the `title`, `base`, `link`, `style`, and `script` elements.

The `meta` element can represent document-level metadata with the `name` attribute, pragma directives with the `http-equiv` attribute, and the file's character encoding declaration when an HTML document is serialized to string form (e.g. for transmission over the network or for disk storage) with the `charset` attribute.

Exactly one of the `name`, `http-equiv`, and `charset` attributes must be specified.

If either `name` or `http-equiv` is specified, then the `content` attribute must also be specified. Otherwise, it must be omitted.

The **`charset`** attribute specifies the character encoding used by the document. This is a character encoding declaration. If the attribute is present in an XML document, its value must be an ASCII case-insensitive match for the string "`UTF-8`" (and the document is therefore required to use UTF-8 as its encoding).

> ***Note: The `charset` attribute on the `meta` element has no effect in XML documents, and is only allowed in order to facilitate migration to and from XHTML.***

There must not be more than one `meta` element with a `charset` attribute per document.

The **`content`** attribute gives the value of the document metadata or pragma directive when the element is used for those purposes. The allowed values depend on the exact context, as described in subsequent sections of this specification.

If a `meta` element has a **`name`** attribute, it sets document metadata. Document metadata is expressed in terms of name/value pairs, the `name` attribute on the `meta` element giving the name, and the `content` attribute on the same element giving the value. The name specifies what aspect of metadata is being set; valid names and the meaning of their values are described in the following sections. If a `meta` element has no `content` attribute, then the value part of the metadata name/value pair is the empty string.

If a `meta` element has the `http-equiv` attribute specified, it must be either in a `head` element or in a `noscript` element that itself is in a `head` element. If a `meta` element does not have the `http-equiv` attribute specified, it must be in a `head` element.

The DOM attributes **`name`** and **`content`** must reflect the respective content attributes of the same name. The DOM attribute **`httpEquiv`** must reflect the content attribute `http-equiv`.

4.2.5.1 Standard metadata names

This specification defines a few names for the `name` attribute of the `meta` element.

application-name
: The value must be a short free-form string that giving the name of the Web application that the page represents. If the page is not a Web application, the `application-name` metadata name must not be used. User agents may use the application name in UI in preference to the page's `title`, since the title might include status messages and the like relevant to the status of the page at a particular moment in time instead of just being the name of the application.

description
: The value must be a free-form string that describes the page. The value must be appropriate for use in a directory of pages, e.g. in a search engine.

generator
: The value must be a free-form string that identifies the software used to generate the document. This value must not be used on hand-authored pages.

4.2.5.2 Other metadata names

Extensions to the predefined set of metadata names may be registered in the WHATWG Wiki MetaExtensions page.

Anyone is free to edit the WHATWG Wiki MetaExtensions page at any time to add a type. These new names must be specified with the following information:

Keyword
: The actual name being defined. The name should not be confusingly similar to any other defined name (e.g. differing only in case).

Brief description
: A short description of what the metadata name's meaning is, including the format the value is required to be in.

Link to more details
: A link to a more detailed description of the metadata name's semantics and requirements. It could be another page on the Wiki, or a link to an external page.

Synonyms
: A list of other names that have exactly the same processing requirements. Authors should not use the names defined to be synonyms, they are only intended to allow user agents to support legacy content.

Status
: One of the following:

Proposal
: The name has not received wide peer review and approval. Someone has proposed it and is using it.

Accepted
: The name has received wide peer review and approval. It has a specification that unambiguously defines how to handle pages that use the name, including when they use it in incorrect ways.

Unendorsed
: The metadata name has received wide peer review and it has been found wanting. Existing pages are using this keyword, but new pages should avoid it. The "brief description" and "link to more details" entries will give details of what authors should use instead, if anything.

If a metadata name is added with the "proposal" status and found to be redundant with existing values, it should be removed and listed as a synonym for the existing value.

Conformance checkers must use the information given on the WHATWG Wiki MetaExtensions page to establish if a value not explicitly defined in this specification is allowed or not. When an author uses a new type not defined by either this specification or the Wiki page, conformance checkers should offer to add the value to the Wiki, with the details described above, with the "proposal" status.

This specification does not define how new values will get approved. It is expected that the Wiki will have a community that addresses this.

Metadata names whose values are to be URLs must not be proposed or accepted. Links must be represented using the `link` element, not the `meta` element.

4.2.5.3 Pragma directives

When the **`http-equiv`** attribute is specified on a `meta` element, the element is a pragma directive.

The `http-equiv` attribute is an enumerated attribute. The following table lists the keywords defined for this attribute. The states given in the first cell of the rows with keywords give the states to which those keywords map. Some of the keywords are non-conforming, as noted in the last column.

State	Keywords	Notes
Content Language	`content-language`	Non-conforming
Encoding declaration	`content-type`	
Default style	`default-style`	
Refresh	`refresh`	

When a `meta` element is inserted into the document, if its `http-equiv` attribute is present and represents one of the above states, then the user agent must run the algorithm appropriate for that state, as described in the following list:

Content language
: This non-conforming pragma sets the **document-wide default language**. Until the

pragma is successfully processed, there is no document-wide default language.

1. If another `meta` element in the Content Language state has already been successfully processed (i.e. when it was inserted the user agent processed it and reached the last step of this list of steps), then abort these steps.

2. If the `meta` element has no `content` attribute, or if that attribute's value is the empty string, then abort these steps.

3. Let *input* be the value of the element's `content` attribute.

4. Let *position* point at the first character of *input*.

5. Skip whitespace.

6. Collect a sequence of characters that are neither space characters nor a U+002C COMMA character (",").

7. Let the document-wide default language be the string that resulted from the previous step.

For `meta` elements in the Content Language state, the `content` attribute must have a value consisting of a valid RFC 3066 language code. [RFC3066]

Note: This pragma is not exactly equivalent to the HTTP `Content-Language` header, for instance it only supports one language. [RFC2616]

Encoding declaration state

The Encoding declaration state is just an alternative form of setting the `charset` attribute: it is a character encoding declaration. This state's user agent requirements are all handled by the parsing section of the specification.

For `meta` elements in the Encoding declaration state, the `content` attribute must have a value that is an ASCII case-insensitive match for a string that consists of: the literal string "`text/html;`", optionally followed by any number of space characters, followed by the literal string "`charset=`", followed by the character encoding name of the character encoding declaration.

If the document contains a `meta` element in the Encoding declaration state, then the document must not contain a `meta` element with the `charset` attribute present.

The Encoding declaration state may be used in HTML documents only, elements in that state must not be used in XML documents.

Default style state

This pragma sets the name of the default alternative style sheet set.

1. ...

Refresh state

This pragma acts as timed redirect.

1. If another `meta` element in the Refresh state has already been successfully

processed (i.e. when it was inserted the user agent processed it and reached the last step of this list of steps), then abort these steps.

2. If the `meta` element has no `content` attribute, or if that attribute's value is the empty string, then abort these steps.

3. Let *input* be the value of the element's `content` attribute.

4. Let *position* point at the first character of *input*.

5. Skip whitespace.

6. Collect a sequence of characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE, and parse the resulting string using the rules for parsing non-negative integers. If the sequence of characters collected is the empty string, then no number will have been parsed; abort these steps. Otherwise, let *time* be the parsed number.

7. Collect a sequence of characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE and U+002E FULL STOP ("."). Ignore any collected characters.

8. Skip whitespace.

9. Let *url* be the address of the current page.

10. If the character in *input* pointed to by *position* is a U+003B SEMICOLON (";"), then advance *position* to the next character. Otherwise, jump to the last step.

11. Skip whitespace.

12. If the character in *input* pointed to by *position* is one of U+0055 LATIN CAPITAL LETTER U or U+0075 LATIN SMALL LETTER U, then advance *position* to the next character. Otherwise, jump to the last step.

13. If the character in *input* pointed to by *position* is one of U+0052 LATIN CAPITAL LETTER R or U+0072 LATIN SMALL LETTER R, then advance *position* to the next character. Otherwise, jump to the last step.

14. If the character in *input* pointed to by *position* is one of U+004C LATIN CAPITAL LETTER L or U+006C LATIN SMALL LETTER L, then advance *position* to the next character. Otherwise, jump to the last step.

15. Skip whitespace.

16. If the character in *input* pointed to by *position* is a U+003D EQUALS SIGN ("="), then advance *position* to the next character. Otherwise, jump to the last step.

17. Skip whitespace.

18. Let *url* be equal to the substring of *input* from the character at *position* to the end of the string.

19. Strip any trailing space characters from the end of *url*.

20. Strip any U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), and

U+000D CARRIAGE RETURN (CR) characters from *url*.

21. Resolve the *url* value to an absolute URL, relative to the `meta` element. If this fails, abort these steps.

22. Perform one or more of the following steps:

    - Set a timer so that in *time* seconds, adjusted to take into account user or user agent preferences, if the user has not canceled the redirect, the user agent navigates the document's browsing context to *url*, with replacement enabled, and with the document's browsing context as the source browsing context.

    - Provide the user with an interface that, when selected, navigates a browsing context to *url*, with the document's browsing context as the source browsing context.

    - Do nothing.

    In addition, the user agent may, as with anything, inform the user of any and all aspects of its operation, including the state of any timers, the destinations of any timed redirects, and so forth.

For `meta` elements in the Refresh state, the `content` attribute must have a value consisting either of:

- just a valid non-negative integer, or

- a valid non-negative integer, followed by a U+003B SEMICOLON (;), followed by one or more space characters, followed by either a U+0055 LATIN CAPITAL LETTER U or a U+0075 LATIN SMALL LETTER U, a U+0052 LATIN CAPITAL LETTER R or a U+0072 LATIN SMALL LETTER R, a U+004C LATIN CAPITAL LETTER L or a U+006C LATIN SMALL LETTER L, a U+003D EQUALS SIGN (=), and then a valid URL.

In the former case, the integer represents a number of seconds before the page is to be reloaded; in the latter case the integer represents a number of seconds before the page is to be replaced by the page at the given URL.

There must not be more than one `meta` element with any particular state in the document at a time.

4.2.5.4 Other pragma directives

Extensions to the predefined set of pragma directives may, under certain conditions, be registered in the WHATWG Wiki PragmaExtensions page.

Such extensions must use a name that is identical to a previously-registered HTTP header defined in an RFC, and must have behavior identical to that described for the HTTP header. Pragma directions corresponding to headers describing metadata, or not requiring specific user agent processing, must not be registered; instead, use metadata names. Pragma directions corresponding to headers that affect the HTTP processing model (e.g. caching) must not be registered, as they would result in HTTP-level behavior being different for user

agents that implement HTML than for user agents that do not.

Anyone is free to edit the WHATWG Wiki PragmaExtensions page at any time to add a pragma directive satisfying these conditions. Such registrations must specify the following information:

Keyword
The actual name being defined.

Brief description
A short description of the purpose of the pragma directive.

Specification
A link to an IETF RFC defining the corresponding HTTP header.

Conformance checkers must use the information given on the WHATWG Wiki PragmaExtensions page to establish if a value not explicitly defined in this specification is allowed or not.

4.2.5.5 Specifying the document's character encoding

A **character encoding declaration** is a mechanism by which the character encoding used to store or transmit a document is specified.

The following restrictions apply to character encoding declarations:

- The character encoding name given must be the name of the character encoding used to serialize the file.
- The value must be a valid character encoding name, and must be the preferred name for that encoding. [IANACHARSET]
- The character encoding declaration must be serialized without the use of character references or character escapes of any kind.
- The element containing the character encoding declaration must be serialised completely within the first 512 bytes of the document.

If an HTML document does not start with a BOM, and if its encoding is not explicitly given by Content-Type metadata, then the character encoding used must be an ASCII-compatible character encoding, and, in addition, if that encoding isn't US-ASCII itself, then the encoding must be specified using a `meta` element with a `charset` attribute or a `meta` element in the Encoding declaration state.

If an HTML document contains a `meta` element with a `charset` attribute or a `meta` element in the Encoding declaration state, then the character encoding used must be an ASCII-compatible character encoding.

Authors should not use JIS_X0212-1990, x-JIS0208, and encodings based on EBCDIC. Authors should not use UTF-32. Authors must not use the CESU-8, UTF-7, BOCU-1 and SCSU encodings. [CESU8] [UTF7] [BOCU1] [SCSU]

Authors are encouraged to use UTF-8. Conformance checkers may advise against authors

using legacy encodings.

In XHTML, the XML declaration should be used for inline character encoding information, if necessary.

4.2.6 The `style` element

Categories

Metadata content.

If the `scoped` attribute is present: flow content.

Contexts in which this element may be used:

If the `scoped` attribute is absent: where metadata content is expected.

If the `scoped` attribute is absent: in a `noscript` element that is a child of a `head` element.

If the `scoped` attribute is present: where flow content is expected, but before any other flow content other than other `style` elements and inter-element whitespace.

Content model:

Depends on the value of the `type` attribute.

Content attributes:

Global attributes

`media`

`type`

`scoped`

Also, the `title` attribute has special semantics on this element.

DOM interface:

```
interface HTMLStyleElement : HTMLElement {
           attribute boolean disabled;
           attribute DOMString media;
           attribute DOMString type;
           attribute boolean scoped;
};
```

The `LinkStyle` interface must also be implemented by this element, the styling processing model defines how. [CSSOM]

The `style` element allows authors to embed style information in their documents. The `style` element is one of several inputs to the styling processing model. The element does not represent content for the user.

If the **`type`** attribute is given, it must contain a valid MIME type, optionally with parameters, that designates a styling language. [RFC2046] If the attribute is absent, the type defaults to `text/css`. [RFC2138]

When examining types to determine if they support the language, user agents must not ignore unknown MIME parameters — types with unknown parameters must be assumed to be unsupported.

The **`media`** attribute says which media the styles apply to. The value must be a valid media query. [MQ] User agents must apply the styles to views while their state match the listed media, and must not apply them otherwise.

Note: The styles might be further limited in scope, e.g. in CSS with the use of `@media` blocks. This specification does not override such further restrictions or requirements.

The default, if the `media` attribute is omitted, is `all`, meaning that by default styles apply to all media.

The **`scoped`** attribute is a boolean attribute. If set, it indicates that the styles are intended just for the subtree rooted at the `style` element's parent element, as opposed to the whole `Document`.

If the `scoped` attribute is present, then the user agent must apply the specified style information only to the `style` element's parent element (if any), and that element's child nodes. Otherwise, the specified styles must, if applied, be applied to the entire document.

The **`title`** attribute on `style` elements defines alternative style sheet sets. If the `style` element has no `title` attribute, then it has no title; the `title` attribute of ancestors does not apply to the `style` element.

Note: The `title` attribute on `style` elements, like the `title` attribute on `link` elements, differs from the global `title` attribute in that a `style` block without a title does not inherit the title of the parent element: it merely has no title.

All descendant elements must be processed, according to their semantics, before the `style` element itself is evaluated. For styling languages that consist of pure text, user agents must evaluate `style` elements by passing the concatenation of the contents of all the text nodes that are direct children of the `style` element (not any other nodes such as comments or elements), in tree order, to the style system. For XML-based styling languages, user agents must pass all the child nodes of the `style` element to the style system.

All URLs found by the styling language's processor must be resolved, relative to the element (or as defined by the styling language), when the processor is invoked.

Once the element has been evaluated, if it had no subresources or once all the subresources it uses have been fetched, the user agent must queue a task to fire a simple event called `load` at the `style` element. If the resource has a subresource that fails to completely load for any reason (e.g. DNS error, HTTP 404 response, the connection being prematurely closed, unsupported Content-Type), the user agent must instead queue a task to fire a simple event called `error` at the `style` element. Non-network errors in the processing of the element's contents or its subresources (e.g. CSS parse errors) are not failures for the purposes of this paragraph.

Note: This specification does not specify a style system, but CSS is expected to be supported by most Web browsers. [CSS21]

The **`media`**, **`type`** and **`scoped`** DOM attributes must reflect the respective content attributes of the same name.

The DOM **`disabled`** attribute behaves as defined for the alternative style sheets DOM.

4.2.7 Styling

The `link` and `style` elements can provide styling information for the user agent to use when rendering the document. The DOM Styling specification specifies what styling information is to be used by the user agent and how it is to be used. [CSSOM]

The `style` and `link` elements implement the `LinkStyle` interface. [CSSOM]

For `style` elements, if the user agent does not support the specified styling language, then the `sheet` attribute of the element's `LinkStyle` interface must return null. Similarly, `link` elements that do not represent external resource links that contribute to the styling processing model (i.e. that do not have a `stylesheet` keyword in their `rel` attribute), and `link` elements whose specified resource has not yet been fetched, or is not in a supported styling language, must have their `LinkStyle` interface's `sheet` attribute return null.

Otherwise, the `LinkStyle` interface's `sheet` attribute must return a `StyleSheet` object with the attributes implemented as follows: [CSSOM]

The content type (`type` DOM attribute)
: The content type must be the same as the style's specified type. For `style` elements, this is the same as the `type` content attribute's value, or `text/css` if that is omitted. For `link` elements, this is the Content-Type metadata of the specified resource.

The location (`href` DOM attribute)
: For `link` elements, the location must be the result of resolving the URL given by the element's `href` content attribute, relative to the element, or the empty string if that fails. For `style` elements, there is no location.

The intended destination media for style information (`media` DOM attribute)
: The media must be the same as the value of the element's `media` content attribute.

The style sheet title (`title` DOM attribute)
: The title must be the same as the value of the element's `title` content attribute. If the attribute is absent, then the style sheet does not have a title. The title is used for defining **alternative style sheet sets**.

The **`disabled`** DOM attribute on `link` and `style` elements must return false and do nothing on setting, if the `sheet` attribute of their `LinkStyle` interface is null. Otherwise, it must return the value of the `StyleSheet` interface's `disabled` attribute on getting, and forward the new value to that same attribute on setting.

4.3 Scripting

Scripts allow authors to add interactivity to their documents.

Authors are encouraged to use declarative alternatives to scripting where possible, as declarative mechanisms are often more maintainable, and many users disable scripting.

> For example, instead of using script to show or hide a section to show more details, the

> `details` element could be used.

Authors are also encouraged to make their applications degrade gracefully in the absence of scripting support.

> For example, if an author provides a link in a table header to dynamically resort the table, the link could also be made to function without scripts by requesting the sorted table from the server.

4.3.1 The `script` element

Categories

Metadata content.
Flow content.
Phrasing content.

Contexts in which this element may be used:

Where metadata content is expected.
Where phrasing content is expected.

Content model:

If there is no `src` attribute, depends on the value of the `type` attribute.
If there *is* a `src` attribute, the element must be either empty or contain only script documentation.

Content attributes:

Global attributes
`src`
`async`
`defer`
`type`
`charset`

DOM interface:

```
interface HTMLScriptElement : HTMLElement {
           attribute DOMString src;
           attribute boolean async;
           attribute boolean defer;
           attribute DOMString type;
           attribute DOMString charset;
           attribute DOMString text;
};
```

The `script` element allows authors to include dynamic script and data blocks in their documents. The element does not represent content for the user.

When used to include dynamic scripts, the scripts may either be embedded inline or may be imported from an external file using the `src` attribute. If the language is not that described by "`text/javascript`", then the `type` attribute must be present. If the `type` attribute is present, its value must be the type of the script's language.

When used to include data blocks, the data must be embedded inline, the format of the data must be given using the `type` attribute, and the `src` attribute must not be specified.

The **`type`** attribute gives the language of the script or format of the data. If the attribute is

present, its value must be a valid MIME type, optionally with parameters. The `charset` parameter must not be specified. (The default, which is used if the attribute is absent, is "`text/javascript`".) [RFC2046]

The **`src`** attribute, if specified, gives the address of the external script resource to use. The value of the attribute must be a valid URL identifying a script resource of the type given by the `type` attribute, if the attribute is present, or of the type "`text/javascript`", if the attribute is absent.

The **`charset`** attribute gives the character encoding of the external script resource. The attribute must not be specified if the `src` attribute is not present. If the attribute is set, its value must be a valid character encoding name, must be the preferred name for that encoding, and must match the encoding given in the `charset` parameter of the Content-Type metadata of the external file, if any. [IANACHARSET]

The **`async`** and **`defer`** attributes are boolean attributes that indicate how the script should be executed.

There are three possible modes that can be selected using these attributes. If the `async` attribute is present, then the script will be executed asynchronously, as soon as it is available. If the `async` attribute is not present but the `defer` attribute is present, then the script is executed when the page has finished parsing. If neither attribute is present, then the script is fetched and executed immediately, before the user agent continues parsing the page. The exact processing details for these attributes is described below.

The `defer` attribute may be specified even if the `async` attribute is specified, to cause legacy Web browsers that only support `defer` (and not `async`) to fall back to the `defer` behavior instead of the synchronous blocking behavior that is the default.

Changing the `src`, `type`, `charset`, `async`, and `defer` attributes dynamically has no direct effect; these attribute are only used at specific times described below (namely, when the element is inserted into the document).

`script` elements have four associated pieces of metadata. The first is a flag indicating whether or not the script block has been **"already executed"**. Initially, `script` elements must have this flag unset (script blocks, when created, are not "already executed"). When a `script` element is cloned, the "already executed" flag, if set, must be propagated to the clone when it is created. The second is a flag indicating whether the element was **"parser-inserted"**. This flag is set by the HTML parser and is used to handle `document.write()` calls. The third and fourth pieces of metadata are ***the script block's type*** and ***the script block's character encoding***. They are determined when the script is run, based on the attributes on the element at that time.

When a `script` element that is neither marked as having "already executed" nor marked as being "parser-inserted" experiences one of the events listed in the following list, the user agent must run the `script` element:

- The `script` element gets inserted into a document.
- The `script` element's child nodes are changed.
- The `script` element has a `src` attribute set where previously the elment had no such attribute.

Running a script: When a `script` element is to be run, the user agent must act as follows:

1. If either:

   - the `script` element has a `type` attribute and its value is the empty string, or
   - the `script` element has no `type` attribute but it has a `language` attribute and *that* attribute's value is the empty string, or
   - the `script` element has neither a `type` attribute nor a `language` attribute, then

   ...let *the script block's type* for this `script` element be "`text/javascript`".

   Otherwise, if the `script` element has a `type` attribute, let *the script block's type* for this `script` element be the value of that attribute.

   Otherwise, the element has a non-empty `language` attribute; let *the script block's type* for this `script` element be the concatenation of the string "`text/`" followed by the value of the `language` attribute.

   Note: The `language` attribute is never conforming, and is always ignored if there is a `type` attribute present.

2. If the `script` element has a `charset` attribute, then let *the script block's character encoding* for this `script` element be the encoding given by the `charset` attribute.

   Otherwise, let *the script block's character encoding* for this `script` element be the same as the encoding of the document itself.

3. If scripting is disabled for the `script` element, or if the user agent does not support the scripting language given by *the script block's type* for this `script` element, then the user agent must abort these steps at this point. The script is not executed.

4. If the element has no `src` attribute, and its child nodes consist only of comment nodes and empty text nodes, then the user agent must abort these steps at this point. The script is not executed.

5. The user agent must set the element's "already executed" flag.

6. If the element has a `src` attribute, then the value of that attribute must be resolved relative to the element, and if that is successful, the specified resource must then be fetched.

   For historical reasons, if the URL is a `javascript:` URL, then the user agent must not, despite the requirements in the definition of the fetching algorithm, actually execute the given script; instead the user agent must act as if it had received an empty HTTP 400 response.

   Once the resource's Content Type metadata is available, if it ever is, apply the algorithm for extracting an encoding from a Content-Type to it. If this returns an encoding, and the user agent supports that encoding, then let *the script block's character encoding* be that encoding.

   Once the fetching process has completed, and the script has **completed loading**, the user agent will have to complete the steps described below. (If the parser is still active at

that time, those steps defer to the parser to handle the execution of pending scripts.)

For performance reasons, user agents may start fetching the script as soon as the attribute is set, instead, in the hope that the element will be inserted into the document. Either way, once the element is inserted into the document, the load must have started. If the UA performs such prefetching, but the element is never inserted in the document, or the `src` attribute is dynamically changed, then the user agent will not execute the script, and the fetching process will have been effectively wasted.

7. Then, the first of the following options that describes the situation must be followed:

   ↪ **If the document is still being parsed, and the element has a `defer` attribute, and the element does not have an `async` attribute**
   The element must be added to the end of the list of scripts that will execute when the document has finished parsing.

   > This isn't compatible with IE for inline deferred scripts, but then what IE does is pretty hard to pin down exactly. Do we want to keep this like it is? Be more compatible?

   ↪ **If the element has an `async` attribute and a `src` attribute**
   The element must be added to the end of the list of scripts that will execute asynchronously.

   ↪ **If the element has an `async` attribute but no `src` attribute, and the list of scripts that will execute asynchronously is not empty**
   The element must be added to the end of the list of scripts that will execute asynchronously.

   ↪ **If the element has a `src` attribute and has been flagged as "parser-inserted"**
   The element is the **pending external script**. (There can only be one such script at a time.)

   ↪ **If the element has a `src` attribute**
   The element must be added to the end of the list of scripts that will execute as soon as possible.

   ↪ **Otherwise**
   The user agent must immediately execute the script block, even if other scripts are already executing.

When a script completes loading: If the `script` element was added to one of the lists mentioned above and the document is still being parsed, then the parser handles it. Otherwise, the UA must run the following steps as the task that the networking task source places on the task queue:

↪ **If the `script` element was added to the *list of scripts that will execute when the document has finished parsing*:**

1. If the `script` element is not the first element in the list, then do nothing yet. Stop going through these steps.
2. Otherwise, execute the script block (the first element in the list).
3. Remove the `script` element from the list (i.e. shift out the first entry in the list).

4. If there are any more entries in the list, and if the script associated with the element that is now the first in the list is already loaded, then jump back to step two to execute it.

↪ **If the `script` element was added to the *list of scripts that will execute asynchronously*:**

1. If the script is not the first element in the list, then do nothing yet. Stop going through these steps.

2. Execute the script block (the first element in the list).

3. Remove the `script` element from the list (i.e. shift out the first entry in the list).

4. If there are any more scripts in the list, and the element now at the head of the list had no `src` attribute when it was added to the list, or had one, but its associated script has finished loading, then jump back to step two to execute the script associated with this element.

↪ **If the `script` element was added to the *list of scripts that will execute as soon as possible*:**

1. Execute the script block.

2. Remove the `script` element from the list.

Fetching an external script must delay the `load` event.

Executing a script block: When the steps above require that the script block be executed, the user agent must act as follows:

↪ **If the load resulted in an error (for example a DNS error, or an HTTP 404 error)**
Executing the script block must just consist of firing an `error` event at the element.

↪ **If the load was successful**

1. Initialize ***the script block's source*** as follows:

   ↪ **If the script is from an external file**
   The contents of that file, interpreted as string of Unicode characters, are the script source.

   For each of the rows in the following table, starting with the first one and going down, if the file has as many or more bytes available than the number of bytes in the first column, and the first bytes of the file match the bytes given in the first column, then set *the script block's character encoding* to the encoding given in the cell in the second column of that row, irrespective of any previous value:

Bytes in Hexadecimal	Encoding
FE FF	UTF-16BE
FF FE	UTF-16LE

Bytes in Hexadecimal	Encoding
EF BB BF	UTF-8

Note: This step looks for Unicode Byte Order Marks (BOMs).

The file must then be converted to Unicode using the character encoding given by *the script block's character encoding*.

↪ **If the script is inline and *the script block's type* is a text-based language**

The value of the DOM `text` attribute at the time the "running a script" algorithm was first invoked is the script source.

↪ **If the script is inline and *the script block's type* is an XML-based language**

The child nodes of the `script` element at the time the "running a script" algorithm was first invoked are the script source.

2. Create a script from the `script` element node, using the *the script block's source* and the *the script block's type*.

   Note: This is where the script is compiled and actually executed.

3. Fire a `load` event at the `script` element.

The DOM attributes **`src`**, **`type`**, **`charset`**, **`async`**, and **`defer`**, each must reflect the respective content attributes of the same name.

script **`.text`** **[= *value*]**

Returns the contents of the element, ignoring child nodes that aren't text nodes.

Can be set, to replace the element's children with the given value.

The DOM attribute **`text`** must return a concatenation of the contents of all the text nodes that are direct children of the `script` element (ignoring any other nodes such as comments or elements), in tree order. On setting, it must act the same way as the `textContent` DOM attribute.

In this example, two `script` elements are used. One embeds an external script, and the other includes some data.

```
<script src="game-engine.js"></script>
<script type="text/x-game-map">
........U.........e
o............A....e
.....A.....AAA....e
.A..AAA...AAAAA...e
```

```
    </script>
```

The data in this case might be used by the script to generate the map of a video game. The data doesn't have to be used that way, though; maybe the map data is actually embedded in other parts of the page's markup, and the data block here is just used by the site's search engine to help users who are looking for particular features in their game maps.

Note: When inserted using the `document.write()` method, `script` elements execute (typically synchronously), but when inserted using `innerHTML` and `outerHTML` attributes, they do not execute at all.

4.3.1.1 Scripting languages

A user agent is said to **support the scripting language** if *the script block's type* matches the MIME type of a scripting language that the user agent implements.

The following lists some MIME types and the languages to which they refer:

`text/javascript`
`text/javascript1.1`
`text/javascript1.2`
`text/javascript1.3`
: ECMAScript. [ECMA262]

`text/javascript;e4x=1`
: ECMAScript with ECMAScript for XML. [ECMA357]

User agents may support other MIME types and other languages.

When examining types to determine if they support the language, user agents must not ignore unknown MIME parameters — types with unknown parameters must be assumed to be unsupported.

4.3.1.2 Inline documentation for external scripts

If a `script` element's `src` attribute is specified, then the contents of the `script` element, if any, must be such that the value of the DOM `text` attribute, which is derived from the element's contents, matches the `documentation` production in the following ABNF, the character set for which is Unicode. [ABNF]

```
documentation = *( *( space / tab / comment ) [ line-comment ] newline )
comment       = slash star *( not-star / star not-slash ) 1*star slash
line-comment  = slash slash *not-newline

; characters
tab           = %x0009 ; U+0009 TAB
newline       = %x000A ; U+000A LINE FEED
space         = %x0020 ; U+0020 SPACE
star          = %x002A ; U+002A ASTERISK
slash         = %x002F ; U+002F SOLIDUS
not-newline   = %x0000-0009 / %x000B-%10FFFF
                ; a Unicode character other than U+000A LINE FEED
not-star      = %x0000-0029 / %x002B-%10FFFF
                ; a Unicode character other than U+002A ASTERISK
```

```
not-slash      = %x0000-002E / %x0030-%10FFFF
                   ; a Unicode character other than U+002F SOLIDUS
```

This allows authors to include documentation, such as license information or API information, inside their documents while still referring to external script files. The syntax is constrained so that authors don't accidentally include what looks like valid script while also providing a `src` attribute.

```
<script src="cool-effects.js">
 // create new instances using:
 //    var e = new Effect();
 // start the effect using .play, stop using .stop:
 //    e.play();
 //    e.stop();
</script>
```

4.3.2 The `noscript` element

Categories

Metadata content.
Flow content.
Phrasing content.

Contexts in which this element may be used:

In a `head` element of an HTML document, if there are no ancestor `noscript` elements.
Where phrasing content is expected in HTML documents, if there are no ancestor `noscript` elements.

Content model:

When scripting is disabled, in a `head` element: in any order, zero or more `link` elements, zero or more `style` elements, and zero or more `meta` elements.
When scripting is disabled, not in a `head` element: transparent, but there must be no `noscript` element descendants.
Otherwise: text that conforms to the requirements given in the prose.

Content attributes:

Global attributes

DOM interface:

Uses `HTMLElement`.

The `noscript` element represents nothing if scripting is enabled, and represents its children if scripting is disabled. It is used to present different markup to user agents that support scripting and those that don't support scripting, by affecting how the document is parsed.

When used in HTML documents, the allowed content model is as follows:

In a `head` element, if scripting is disabled for the `noscript` element

The `noscript` element must contain only `link`, `style`, and `meta` elements.

In a `head` element, if scripting is enabled for the `noscript` element

The `noscript` element must contain only text, except that invoking the HTML fragment parsing algorithm with the `noscript` element as the *context* element and the text contents as the *input* must result in a list of nodes that consists only of `link`, `style`, and `meta` elements, and no parse errors.

Outside of `head` elements, if scripting is disabled for the `noscript` element

The noscript element's content model is transparent, with the additional restriction that a noscript element must not have a noscript element as an ancestor (that is, noscript can't be nested).

Outside of head elements, if scripting is enabled for the noscript element

The noscript element must contain only text, except that the text must be such that running the following algorithm results in a conforming document with no noscript elements and no script elements, and such that no step in the algorithm causes an HTML parser to flag a parse error:

1. Remove every script element from the document.
2. Make a list of every noscript element in the document. For every noscript element in that list, perform the following steps:
    1. Let the *parent element* be the parent element of the noscript element.
    2. Take all the children of the *parent element* that come before the noscript element, and call these elements *the before children*.
    3. Take all the children of the *parent element* that come *after* the noscript element, and call these elements *the after children*.
    4. Let *s* be the concatenation of all the text node children of the noscript element.
    5. Set the innerHTML attribute of the *parent element* to the value of *s*. (This, as a side-effect, causes the noscript element to be removed from the document.)
    6. Insert *the before children* at the start of the *parent element*, preserving their original relative order.
    7. Insert *the after children* at the end of the *parent element*, preserving their original relative order.

Note: All these contortions are required because, for historical reasons, the noscript element is handled differently by the HTML parser based on whether scripting was enabled or not when the parser was invoked. The element is not allowed in XML, because in XML the parser is not affected by such state, and thus the element would not have the desired effect.

The noscript element must not be used in XML documents.

Note: The noscript element is only effective in the HTML serialization, it has no effect in the XML serialization.

The noscript element has no other requirements. In particular, children of the noscript element are not exempt from form submission, scripting, and so forth, even when scripting is enabled for the element.

4.4 Sections

Some elements, for example address elements, are scoped to their nearest ancestor sectioning content. For such elements *x*, the elements that apply to a sectioning content element *e* are all the *x* elements whose nearest sectioning content ancestor is *e*.

4.4.1 The body element

Categories

Sectioning root.

Contexts in which this element may be used:

As the second element in an html element.

Content model:

Flow content.

Content attributes:

Global attributes
onbeforeunload
onerror
onhashchange
onload
onmessage
onoffline
ononline
onpopstate
onresize
onstorage
onunload

DOM interface:

```
interface HTMLBodyElement : HTMLElement {
           attribute Function onbeforeunload;
           attribute Function onerror;
           attribute Function onhashchange;
           attribute Function onload;
           attribute Function onmessage;
           attribute Function onoffline;
           attribute Function ononline;
           attribute Function onpopstate;
           attribute Function onresize;
           attribute Function onstorage;
           attribute Function onunload;
};
```

The body element represents the main content of the document.

In conforming documents, there is only one body element. The document.body DOM attribute provides scripts with easy access to a document's body element.

Note: Some DOM operations (for example, parts of the drag and drop model)

are defined in terms of "the body element". This refers to a particular element in the DOM, as per the definition of the term, and not any arbitrary `body` element.

The `body` element exposes as event handler content attributes a number of the event handler attributes of the `Window` object. It also mirrors their event handler DOM attributes.

The `onerror` and `onload` event handler attributes of the `Window` object, exposed on the `body` element, shadow the generic `onerror` and `onload` event handler attributes normally supported by HTML elements.

> Thus, for example, a bubbling `error` event fired on a child of the body element of a `Document` would first trigger the `onerror` event handler content attributes of that element, then that of the root `html` element, and only *then* would it trigger the `onerror` event handler content attribute on the `body` element. This is because the event would bubble from the target, to the `body`, to the `html`, to the `Document`, to the `Window`, and the event handler attribute on the `body` is watching the `Window` not the `body`. A regular event listener attached to the `body` using `addEventListener()`, however, would fire when the event bubbled through the `body` and not when it reaches the `Window` object.

4.4.2 The `section` element

Categories
Flow content.
Sectioning content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Flow content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `section` element represents a generic document or application section. A section, in this context, is a thematic grouping of content, typically with a header, possibly with a footer.

> Examples of sections would be chapters, the various tabbed pages in a tabbed dialog box, or the numbered sections of a thesis. A Web site's home page could be split into sections for an introduction, news items, contact information.

> In the following example, we see an article (part of a larger Web page) about apples, containing two short sections.

```
<article>
 <header>
  <h1>Apples</h1>
  <p>Tasty, delicious fruit!</p>
 </header>
 <p>The apple is the pomaceous fruit of the apple tree.</p>
 <section>
  <h1>Red Delicious</h1>
  <p>These bright red apples are the most common found in many
```

```
      supermarkets.</p>
     </section>
     <section>
      <h1>Granny Smith</h1>
      <p>These juicy, green apples and make a great filling for
      apple pies.</p>
     </section>
    </article>
```

4.4.3 The `nav` element

Categories
Flow content.
Sectioning content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Flow content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `nav` element represents a section of a page that links to other pages or to parts within the page: a section with navigation links. Not all groups of links on a page need to be in a `nav` element — only sections that consist of primary navigation blocks are appropriate for the `nav` element. In particular, it is common for footers to have a list of links to various key parts of a site, but the `footer` element is more appropriate in such cases.

In the following example, the page has several places where links are present, but only one of those places is considered a navigation section.

```
<body>
 <header>
  <h1>Wake up sheeple!</h1>
  <p><a href="news.html">News</a> -
     <a href="blog.html">Blog</a> -
     <a href="forums.html">Forums</a></p>
 </header>
 <nav>
  <h1>Navigation</h1>
  <ul>
   <li><a href="articles.html">Index of all articles</a><li>
   <li><a href="today.html">Things sheeple need to wake up for today</a><li>
   <li><a href="successes.html">Sheeple we have managed to wake</a><li>
  </ul>
 </nav>
 <article>
  <p>...page content would be here...</p>
 </article>
 <footer>
  <p>Copyright © 2006 The Example Company</p>
  <p><a href="about.html">About</a> -
     <a href="policy.html">Privacy Policy</a> -
     <a href="contact.html">Contact Us</a></p>
 </footer>
</body>
```

4.4.4 The `article` element

Categories
Flow content.
Sectioning content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Flow content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `article` element represents a section of a page that consists of a composition that forms an independent part of a document, page, or site. This could be a forum post, a magazine or newspaper article, a Web log entry, a user-submitted comment, or any other independent item of content.

> ***Note: An `article` element is "independent" in that its contents could stand alone, for example in syndication. However, the element is still associated with its ancestors; for instance, contact information that applies to a parent `body` element still covers the `article` as well.***

When `article` elements are nested, the inner `article` elements represent articles that are in principle related to the contents of the outer article. For instance, a Web log entry on a site that accepts user-submitted comments could represent the comments as `article` elements nested within the `article` element for the Web log entry.

Author information associated with an `article` element (q.v. the `address` element) does not apply to nested `article` elements.

4.4.5 The `aside` element

Categories
Flow content.
Sectioning content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Flow content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `aside` element represents a section of a page that consists of content that is tangentially

related to the content around the aside element, and which could be considered separate from that content. Such sections are often represented as sidebars in printed typography.

> The following example shows how an aside is used to mark up background material on Switzerland in a much longer news story on Europe.
>
> ```
> <aside>
>  <h1>Switzerland</h1>
>  Switzerland, a land-locked country in the middle of geographic
>  Europe, has not joined the geopolitical European Union, though it is
>  a signatory to a number of European treaties.
> </aside>
> ```

> The following example shows how an aside is used to mark up a pull quote in a longer article.
>
> ```
> ...
>
> He later joined a large company, continuing on the same work.
> <q>I love my job. People ask me what I do for fun when I'm not at
> work. But I'm paid to do my hobby, so I never know what to
> answer. Some people wonder what they would do if they didn't have to
> work... but I know what I would do, because I was unemployed for a
> year, and I filled that time doing exactly what I do
> now.</q>
>
> <aside>
>  <q> People ask me what I do for fun when I'm not at work. But I'm
>  paid to do my hobby, so I never know what to answer. </q>
> </aside>
>
> Of course his work — or should that be hobby? —
> isn't his only passion. He also enjoys other pleasures.
>
> ...
> ```

4.4.6 The h1, h2, h3, h4, h5, and h6 elements

Categories
Flow content.
Heading content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses HTMLElement.

These elements define headers for their sections.

The semantics and meaning of these elements are defined in the section on headings and sections.

These elements have a **rank** given by the number in their name. The h1 element is said to have the highest rank, the h6 element has the lowest rank, and two elements with the same name have equal rank.

4.4.7 The `header` element

Categories
Flow content.
Heading content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Flow content, including at least one descendant that is heading content, but no sectioning content descendants, no `header` element descendants, and no `footer` element descendants.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `header` element represents the header of a section. The element is typically used to group a set of `h1`–`h6` elements to mark up a page's title with its subtitle or tagline. However, `header` elements may contain more than just the section's headings and subheadings — for example it would be reasonable for the header to include version history information.

For the purposes of document summaries, outlines, and the like, the text of `header` elements is defined to be the text of the highest ranked `h1`–`h6` element descendant of the `header` element, if there are any such elements, and the first such element if there are multiple elements with that rank. If there are no such elements, then the text of the `header` element is the empty string.

Other heading elements in the `header` element indicate subheadings or subtitles.

The rank of a `header` element is the same as for an `h1` element (the highest rank).

The section on headings and sections defines how `header` elements are assigned to individual sections.

> Here are some examples of valid headers. In each case, the emphasised text represents the text that would be used as the header in an application extracting header data and ignoring subheadings.
>
> ```
> <header>
>  <h1>The reality dysfunction</h1>
>  <h2>Space is not the only void</h2>
> </header>
>
> <header>
>  <h1>Dr. Strangelove</h1>
>  <h2>Or: How I Learned to Stop Worrying and Love the Bomb</h2>
> </header>
>
> <header>
>  Welcome to...
>  <h1>Voidwars!</h1>
> </header>
>
> <header>
>  <h1>Scalable Vector Graphics (SVG) 1.2</h1>
> ```

```
 <h2>W3C Working Draft 27 October 2004</h2>
 <dl>
  <dt>This version:</dt>
  <dd><a href="http://www.w3.org/TR/2004/WD-SVG12-20041027/">http://www.w3.org
  <dt>Previous version:</dt>
  <dd><a href="http://www.w3.org/TR/2004/WD-SVG12-20040510/">http://www.w3.org
  <dt>Latest version of SVG 1.2:</dt>
  <dd><a href="http://www.w3.org/TR/SVG12/">http://www.w3.org/TR/SVG12/</a></
  <dt>Latest SVG Recommendation:</dt>
  <dd><a href="http://www.w3.org/TR/SVG/">http://www.w3.org/TR/SVG/</a></dd>
  <dt>Editor:</dt>
  <dd>Dean Jackson, W3C, <a href="mailto:dean@w3.org">dean@w3.org</a></dd>
  <dt>Authors:</dt>
  <dd>See <a href="#authors">Author List</a></dd>
 </dl>
 <p class="copyright"><a href="http://www.w3.org/Consortium/Legal/ipr-notic .
</header>
```

4.4.8 The `footer` element

Categories
Flow content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Flow content, but with no heading content descendants, no sectioning content descendants, and no `footer` element descendants.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `footer` element represents a footer for the section it applies to. A footer typically contains information about its section such as who wrote it, links to related documents, copyright data, and the like.

Contact information for the section given in a `footer` should be marked up using the `address` element.

Footers don't necessarily have to appear at the end of a section, though they usually do.

Here is a page with two footers, one at the top and one at the bottom, with the same content:

```
<body>
 <footer><a href="../">Back to index...</a></footer>
 <header>
  <h1>Lorem ipsum</h1>
  <h2>The ipsum of all lorems</h2>
 </header>
 <p>A dolor sit amet, consectetur adipisicing elit, sed do eiusmod
 tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
 veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
 ea commodo consequat. Duis aute irure dolor in reprehenderit in
 voluptate velit esse cillum dolore eu fugiat nulla
 pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
 culpa qui officia deserunt mollit anim id est laborum.</p>
 <footer><a href="../">Back to index...</a></footer>
</body>
```

4.4.9 The `address` element

Categories
Flow content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Flow content, but with no heading content descendants, no sectioning content descendants, no `footer` element descendants, and no `address` element descendants.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `address` element represents the contact information for the section it applies to. If it applies to the body element, then it instead applies to the document as a whole.

> For example, a page at the W3C Web site related to HTML might include the following contact information:
>
> ```
> <ADDRESS>
>  <A href="../People/Raggett/">Dave Raggett</A>,
>  <A href="../People/Arnaud/">Arnaud Le Hors</A>,
>  contact persons for the <A href="Activity">W3C HTML Activity</A>
> </ADDRESS>
> ```

The `address` element must not be used to represent arbitrary addresses (e.g. postal addresses), unless those addresses are contact information for the section. (The `p` element is the appropriate element for marking up such addresses.)

The `address` element must not contain information other than contact information.

> For example, the following is non-conforming use of the `address` element:
>
> ```
> <ADDRESS>Last Modified: 1999/12/24 23:37:50</ADDRESS>
> ```

Typically, the `address` element would be included with other information in a `footer` element.

To determine the contact information for a sectioning content element (such as a document's `body` element, which would give the contact information for the page), UAs must collect all the `address` elements that apply to that sectioning content element and its ancestor sectioning content elements. The contact information is the collection of all the information given by those elements.

> ***Note: Contact information for one sectioning content element, e.g. an `aside` element, does not apply to its ancestor elements, e.g. the page's `body`.***

4.4.10 Headings and sections

The h1–h6 elements and the header element are headings.

The first element of heading content in an element of sectioning content represents the header for that section. Subsequent headers of equal or higher rank start new (implied) sections, headers of lower rank start implied subsections that are part of the previous one. In both cases, the element represents the header of the implied section.

Sectioning content elements are always considered subsections of their nearest ancestor element of sectioning content, regardless of what implied sections other headings may have created.

Certain elements are said to be **sectioning roots**, including blockquote and td elements. These elements can have their own outlines, but the sections and headers inside these elements do not contribute to the outlines of their ancestors.

> For the following fragment:
>
> ```
> <body>
>  <h1>Foo</h1>
>  <h2>Bar</h2>
>  <blockquote>
>   <h3>Bla</h3>
>  </blockquote>
>  Baz
>  <h2>Quux</h2>
>  <section>
>   <h3>Thud</h3>
>  </section>
>  Grunt
> </body>
> ```
>
> ...the structure would be:
>
> 1. Foo (heading of explicit body section, containing the "Grunt" paragraph)
>   1. Bar (heading starting implied section, containing a block quote and the "Baz" paragraph)
>   2. Quux (heading starting implied section)
>   3. Thud (heading of explicit section section)
>
> Notice how the section ends the earlier implicit section so that a later paragraph ("Grunt") is back at the top level.

Sections may contain headers of any rank, but authors are strongly encouraged to either use only h1 elements, or to use elements of the appropriate rank for the section's nesting level.

Authors are also encouraged to explicitly wrap sections in elements of sectioning content, instead of relying on the implicit sections generated by having multiple heading in one element of sectioning content.

> For example, the following is correct:
>
> ```
> <body>
>  <h4>Apples</h4>
>  Apples are fruit.
>  <section>
>   <h2>Taste</h2>
>   They taste lovely.
>   <h6>Sweet</h6>
> ```

```
    <p>Red apples are sweeter than green ones.</p>
    <h1>Color</h1>
    <p>Apples come in various colors.</p>
   </section>
  </body>
```

However, the same document would be more clearly expressed as:

```
  <body>
   <h1>Apples</h1>
   <p>Apples are fruit.</p>
   <section>
    <h2>Taste</h2>
    <p>They taste lovely.</p>
    <section>
     <h3>Sweet</h3>
     <p>Red apples are sweeter than green ones.</p>
    </section>
   </section>
   <section>
    <h2>Color</h2>
    <p>Apples come in various colors.</p>
   </section>
  </body>
```

Both of the documents above are semantically identical and would produce the same outline in compliant user agents.

4.4.10.1 Creating an outline

This section defines an algorithm for creating an outline for a sectioning content element or a sectioning root element. It is defined in terms of a walk over the nodes of a DOM tree, in tree order, with each node being visited when it is *entered* and when it is *exited* during the walk.

The **outline** for a sectioning content element or a sectioning root element consists of a list of one or more potentially nested sections. A **section** is a container that corresponds to some nodes in the original DOM tree. Each section can have one heading associated with it, and can contain any number of further nested sections. The algorithm for the outline also associates each node in the DOM tree with a particular section and potentially a heading. (The sections in the outline aren't `section` elements, though some may correspond to such elements — they are merely conceptual sections.)

The following markup fragment:

```
  <body>
   <h1>A</h1>
   <p>B</p>
   <h2>C</h2>
   <p>D</p>
   <h2>E</h2>
   <p>F</p>
  </body>
```

...results in the following outline being created for the `body` node (and thus the entire document):

1. Section created for `body` node.
   Associated with heading "A".
   Also associated with paragraph "B".

Nested sections:

1. Section implied for first h2 element.
   Associated with heading "C".
   Also associated with paragraph "D".
   No nested sections.
2. Section implied for second h2 element.
   Associated with heading "E".
   Also associated with paragraph "F".
   No nested sections.

The algorithm that must be followed during a walk of a DOM subtree rooted at a sectioning content element or a sectioning root element to determine that element's outline is as follows:

1. Let *current outlinee* be null. (It holds the element whose outline is being created.)

2. Let *current section* be null. (It holds a pointer to a section, so that elements in the DOM can all be associated with a section.)

3. Create a stack to hold elements, which is used to handle nesting. Initialize this stack to empty.

4. As you walk over the DOM in tree order, trigger the first relevant step below for each element as you enter and exit it.

   ↪ **If the top of the stack is an element, and you are exiting that element**

   Note: The element being exited is a heading content element.

   Pop that element from the stack.

   ↪ **If the top of the stack is a heading content element**
   Do nothing.

   ↪ **When entering a sectioning content element or a sectioning root element**
   If *current outlinee* is not null, push *current outlinee* onto the stack.

   Let *current outlinee* be the element that is being entered.

   Let *current section* be a newly created section for the *current outlinee* element.

   Let there be a new outline for the new *current outlinee*, initialized with just the new *current section* as the only section in the outline.

   ↪ **When exiting a sectioning content element, if the stack is not empty**
   Pop the top element from the stack, and let the *current outlinee* be that element.

   Let *current section* be the last section in the outline of the *current outlinee* element.

   Append the outline of the sectioning content element being exited to the *current section*. (This does not change which section is the last section in the

outline.)

↪ **When exiting a sectioning root element, if the stack is not empty**

Run these steps:

1. Pop the top element from the stack, and let the *current outlinee* be that element.
2. Let *current section* be the last section in the outline of the *current outlinee* element.
3. *Finding the deepest child:* If *current section* has no child sections, stop these steps.
4. Let *current section* be the last child section of the current *current section*.
5. Go back to the substep labeled *finding the deepest child*.

↪ **When exiting a sectioning content element or a sectioning root element**

Note: The current outlinee is the element being exited.

Let *current section* be the first section in the outline of the *current outlinee* element.

Skip to the next step in the overall set of steps. (The walk is over.)

↪ **If the *current outlinee* is null.**

Do nothing.

↪ **When entering a heading content element**

If the *current section* has no heading, let the element being entered be the heading for the *current section*.

Otherwise, if the element being entered has a rank equal to or greater than the heading of the last section of the outline of the *current outlinee*, then create a new section and append it to the outline of the *current outlinee* element, so that this new section is the new last section of that outline. Let *current section* be that new section. Let the element being entered be the new heading for the *current section*.

Otherwise, run these substeps:

1. Let *candidate section* be *current section*.
2. If the element being entered has a rank lower than the rank of the heading of the *candidate section*, then create a new section, and append it to *candidate section*. (This does not change which section is the last section in the outline.) Let *current section* be this new section. Let the element being entered be the new heading for the *current section*. Abort these substeps.
3. Let *new candidate section* be the section that contains *candidate*

section in the outline of *current outlinee*.

4. Let *candidate section* be *new candidate section*.

5. Return to step 2.

Push the element being entered onto the stack. (This causes the algorithm to skip any descendants of the element.)

Note: Recall that h1 has the highest rank, and h6 has the lowest rank.

↪ **Otherwise**
Do nothing.

In addition, whenever you exit a node, after doing the steps above, if *current section* is not null, associate the node with the section *current section*.

5. If the *current outlinee* is null, then there was no sectioning content element or sectioning root element in the DOM. There is no outline. Abort these steps.

6. Associate any nodes that were not associated with a section in the steps above with *current outlinee* as their section.

7. Associate all nodes with the heading of the section with which they are associated, if any.

8. If *current outlinee* is the body element, then the outline created for that element is the outline of the entire document.

The tree of sections created by the algorithm above, or a proper subset thereof, must be used when generating document outlines, for example when generating tables of contents.

When creating an interactive table of contents, entries should jump the user to the relevant sectioning content element, if the section was created for a real element in the original document, or to the relevant heading content element, if the section in the tree was generated for a heading in the above process.

Note: Selecting the first section of the document therefore always takes the user to the top of the document, regardless of where the first header in the body is to be found.

The following JavaScript function shows how the tree walk could be implemented. The root argument is the root of the tree to walk, and the enter and exit arguments are callbacks that are called with the nodes as they are entered and exited. [ECMA262]

```
function (root, enter, exit) {
  var node = root;
  start: while (node) {
    enter(node);
    if (node.firstChild) {
```

```
        node = node.firstChild;
        continue start;
      }
      while (node) {
        exit(node);
        if (node.nextSibling) {
          node = node.nextSibling;
          continue start;
        }
        if (node == root)
          node = null;
        else
          node = node.parentNode;
      }
    }
  }
```

4.4.10.2 Distinguishing site-wide headings from page headings

Given the outline of a document, but ignoring any sections created for `nav` and `aside` elements, and any of their descendants, if the only root of the tree is the body element's section, and it has only a single subsection which is created by an `article` element, then the heading of the body element should be assumed to be a site-wide heading, and the heading of the `article` element should be assumed to be the page's heading.

If a page starts with a heading that is common to the whole site, the document must be authored such that, in the document's outline, ignoring any sections created for `nav` and `aside` elements and any of their descendants, the tree has only one root section, the body element's section, its heading is the site-wide heading, the body element has just one subsection, that subsection is created by an `article` element, and that `article`'s heading is the page heading.

If a page does not contain a site-wide heading, then the page must be authored such that, in the document's outline, ignoring any sections created for `nav` and `aside` elements and any of their descendants, either the body element has no subsections, or it has more than one subsection, or it has a single subsection but that subsection is not created by an `article` element, or there is more than one section at the root of the outline.

> ***Note: Conceptually, a site is thus a document with many articles — when those articles are split into many pages, the heading of the original single page becomes the heading of the site, repeated on every page.***

4.5 Grouping content

4.5.1 The `p` element

Categories
Flow content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:

Phrasing content.

Content attributes:

Global attributes

DOM interface:

Uses `HTMLElement`.

The `p` element represents a paragraph.

> The following examples are conforming HTML fragments:
>
> ```
> The little kitten gently seated himself on a piece of
> carpet. Later in his life, this would be referred to as the time the
> cat sat on the mat.
>
> <fieldset>
>  <legend>Personal information</legend>
>  
>    <label>Name: <input name="n"></label>
>    <label><input name="anon" type="checkbox"> Hide from other users</label>
>  
>  <label>Address: <textarea name="a"></textarea></label>
> </fieldset>
>
> There was once an example from Femley,<br>
> Whose markup was of dubious quality.<br>
> The validator complained,<br>
> So the author was pained,<br>
> To move the error from the markup to the rhyming.
> ```

The `p` element should not be used when a more specific element is more appropriate.

> The following example is technically correct:
>
> ```
> <section>
>  <!-- ... -->
>  Last modified: 2001-04-23
>  Author: fred@example.com
> </section>
> ```
>
> However, it would be better marked-up as:
>
> ```
> <section>
>  <!-- ... -->
>  <footer>Last modified: 2001-04-23</footer>
>  <address>Author: fred@example.com</address>
> </section>
> ```
>
> Or:
>
> ```
> <section>
>  <!-- ... -->
>  <footer>
>   Last modified: 2001-04-23
>   <address>Author: fred@example.com</address>
>  </footer>
> </section>
> ```

4.5.2 The `hr` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Empty.

Content attributes:

Global attributes

DOM interface:

Uses `HTMLElement`.

The `hr` element represents a paragraph-level thematic break, e.g. a scene change in a story, or a transition to another topic within a section of a reference book.

4.5.3 The `br` element

Categories

Flow content.

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Empty.

Content attributes:

Global attributes

DOM interface:

Uses `HTMLElement`.

The `br` element represents a line break.

`br` elements must be empty. Any content inside `br` elements must not be considered part of the surrounding text.

`br` elements must be used only for line breaks that are actually part of the content, as in poems or addresses.

The following example is correct usage of the `br` element:

```
<p>P. Sherman<br>
42 Wallaby Way<br>
Sydney</p>
```

`br` elements must not be used for separating thematic groups in a paragraph.

The following examples are non-conforming, as they abuse the `br` element:

```
<p><a ...>34 comments.</a><br>
<a ...>Add a comment.<a></p>

<p>Name: <input name="name"><br>
Address: <input name="address"></p>
```

Here are alternatives to the above, which are correct:

```
<p><a ...>34 comments.</a></p>
```

```
    <p><a ...>Add a comment.<a></p>

    <p>Name: <input name="name"></p>
    <p>Address: <input name="address"></p>
```

If a paragraph consists of nothing but a single `br` element, it represents a placeholder blank line (e.g. as in a template). Such blank lines must not be used for presentation purposes.

4.5.4 The `pre` element

Categories
Flow content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `pre` element represents a block of preformatted text, in which structure is represented by typographic conventions rather than by elements.

Note: In the HTML serialization, a leading newline character immediately following the `pre` element start tag is stripped.

Some examples of cases where the `pre` element could be used:

- Including an e-mail, with paragraphs indicated by blank lines, lists indicated by lines prefixed with a bullet, and so on.
- Including fragments of computer code, with structure indicated according to the conventions of that language.
- Displaying ASCII art.

To represent a block of computer code, the `pre` element can be used with a `code` element; to represent a block of computer output the `pre` element can be used with a `samp` element. Similarly, the `kbd` element can be used within a `pre` element to indicate text that the user is to enter.

In the following snippet, a sample of computer code is presented.

```
<p>This is the <code>Panel</code> constructor:</p>
<pre><code>function Panel(element, canClose, closeHandler) {
  this.element = element;
  this.canClose = canClose;
  this.closeHandler = function () { if (closeHandler) closeHandler() };
}</code></pre>
```

In the following snippet, `samp` and `kbd` elements are mixed in the contents of a `pre` element to show a session of Zork I.

```
<pre><samp>You are in an open field west of a big white house with a boarded
front door.
There is a small mailbox here.

></samp> <kbd>open mailbox</kbd>

<samp>Opening the mailbox reveals:
A leaflet.

></samp></pre>
```

The following shows a contemporary poem that uses the `pre` element to preserve its unusual formatting, which forms an intrinsic part of the poem itself.

```
<pre>                maxling

it is with a          heart
               heavy

that i admit loss of a feline
        so           loved

a friend lost to the
        unknown
                                  (night)

~cdr 11dec07</pre>
```

4.5.5 The `dialog` element

Categories
: Flow content.

Contexts in which this element may be used:
: Where flow content is expected.

Content model:
: Zero or more pairs of one `dt` element followed by one `dd` element.

Content attributes:
: Global attributes

DOM interface:
: Uses `HTMLElement`.

The `dialog` element represents a conversation, meeting minutes, a chat transcript, a dialog in a screenplay, an instant message log, or some other construct in which different players take turns in discourse.

Each part of the conversation must have an explicit talker (or speaker) given by a `dt` element, and a discourse (or quote) given by a `dd` element.

This example demonstrates this using an extract from Abbot and Costello's famous sketch, *Who's on first*:

```
<dialog>
 <dt> Costello
 <dd> Look, you gotta first baseman?
 <dt> Abbott
 <dd> Certainly.
 <dt> Costello
 <dd> Who's playing first?
 <dt> Abbott
```

```
      <dd> That's right.
      <dt> Costello
      <dd> When you pay off the first baseman every month, who gets the money?
      <dt> Abbott
      <dd> Every dollar of it.
     </dialog>
```

Note: Text in a `dt` element in a `dialog` element is implicitly the source of the text given in the following `dd` element, and the contents of the `dd` element are implicitly a quote from that speaker. There is thus no need to include `cite`, `q`, or `blockquote` elements in this markup. Indeed, a `q` element inside a `dd` element in a conversation would actually imply the people talking were themselves quoting another work. See the `cite`, `q`, and `blockquote` elements for other ways to cite or quote.

4.5.6 The `blockquote` element

Categories

Flow content.
Sectioning root.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Content attributes:

Global attributes
`cite`

DOM interface:

```
interface HTMLQuoteElement : HTMLElement {
           attribute DOMString cite;
};
```

Note: The `HTMLQuoteElement` interface is also used by the `q` element.

The `blockquote` element represents a section that is quoted from another source.

Content inside a `blockquote` must be quoted from another source, whose address, if it has one, should be cited in the **`cite`** attribute.

If the `cite` attribute is present, it must be a valid URL. To obtain the corresponding citation link, the value of the attribute must be resolved relative to the element. User agents should allow users to follow such citation links.

The **`cite`** DOM attribute must reflect the element's `cite` content attribute.

Note: The best way to represent a conversation is not with the `cite` and `blockquote` elements, but with the `dialog` element.

This next example shows the use of `cite` alongside `blockquote`:

```
    <p>His next piece was the aptly named <cite>Sonnet 130</cite>:</p>
    <blockquote cite="http://quotes.example.org/s/sonnet130.html">
      <p>My mistress' eyes are nothing like the sun,<br>
      Coral is far more red, than her lips red,<br>
      ...
```

4.5.7 The `ol` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Zero or more `li` elements.

Content attributes:

Global attributes
`reversed`
`start`

DOM interface:

```
interface HTMLOListElement : HTMLElement {
           attribute boolean reversed;
           attribute long start;
};
```

The `ol` element represents a list of items, where the items have been intentionally ordered, such that changing the order would change the meaning of the document.

The items of the list are the `li` element child nodes of the `ol` element, in tree order.

The **`reversed`** attribute is a boolean attribute. If present, it indicates that the list is a descending list (..., 3, 2, 1). If the attribute is omitted, the list is an ascending list (1, 2, 3, ...).

The **`start`** attribute, if present, must be a valid integer giving the ordinal value of the first list item.

If the `start` attribute is present, user agents must parse it as an integer, in order to determine the attribute's value. The default value, used if the attribute is missing or if the value cannot be converted to a number according to the referenced algorithm, is 1 if the element has no `reversed` attribute, and is the number of child `li` elements otherwise.

The first item in the list has the ordinal value given by the `ol` element's `start` attribute, unless that `li` element has a `value` attribute with a value that can be successfully parsed, in which case it has the ordinal value given by that `value` attribute.

Each subsequent item in the list has the ordinal value given by its `value` attribute, if it has one, or, if it doesn't, the ordinal value of the previous item, plus one if the `reversed` is absent, or minus one if it is present.

The **`reversed`** DOM attribute must reflect the value of the `reversed` content attribute.

The **`start`** DOM attribute must reflect the value of the `start` content attribute.

The following markup shows a list where the order matters, and where the `ol` element is therefore appropriate. Compare this list to the equivalent list in the `ul` section to see an example of the same items using the `ul` element.

```
<p>I have lived in the following countries (given in the order of when
I first lived there):</p>
<ol>
 <li>Switzerland
 <li>United Kingdom
 <li>United States
 <li>Norway
</ol>
```

Note how changing the order of the list changes the meaning of the document. In the following example, changing the relative order of the first two items has changed the birthplace of the author:

```
<p>I have lived in the following countries (given in the order of when
I first lived there):</p>
<ol>
 <li>United Kingdom
 <li>Switzerland
 <li>United States
 <li>Norway
</ol>
```

4.5.8 The `ul` element

Categories
Flow content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Zero or more `li` elements.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `ul` element represents a list of items, where the order of the items is not important — that is, where changing the order would not materially change the meaning of the document.

The items of the list are the `li` element child nodes of the `ul` element.

The following markup shows a list where the order does not matter, and where the `ul` element is therefore appropriate. Compare this list to the equivalent list in the `ol` section to see an example of the same items using the `ol` element.

```
<p>I have lived in the following countries:</p>
<ul>
 <li>Norway
 <li>Switzerland
 <li>United Kingdom
 <li>United States
</ul>
```

Note that changing the order of the list does not change the meaning of the document. The items in the snippet above are given in alphabetical order, but in the snippet below they are given in order of the size of their current account balance in 2007, without changing the meaning of the document whatsoever:

```
<p>I have lived in the following countries:</p>
<ul>
 <li>Switzerland
 <li>Norway
 <li>United Kingdom
 <li>United States
</ul>
```

4.5.9 The `li` element

Categories

None.

Contexts in which this element may be used:

Inside `ol` elements.
Inside `ul` elements.
Inside `menu` elements.

Content model:

Flow content.

Content attributes:

Global attributes
If the element is a child of an `ol` element: `value`

DOM interface:

```
interface HTMLLIElement : HTMLElement {
           attribute long value;
};
```

The `li` element represents a list item. If its parent element is an `ol`, `ul`, or `menu` element, then the element is an item of the parent element's list, as defined for those elements. Otherwise, the list item has no defined list-related relationship to any other `li` element.

The **`value`** attribute, if present, must be a valid integer giving the ordinal value of the list item.

If the `value` attribute is present, user agents must parse it as an integer, in order to determine the attribute's value. If the attribute's value cannot be converted to a number, the attribute must be treated as if it was absent. The attribute has no default value.

The `value` attribute is processed relative to the element's parent `ol` element (q.v.), if there is one. If there is not, the attribute has no effect.

The **`value`** DOM attribute must reflect the value of the `value` content attribute.

The following example, the top ten movies are listed (in reverse order). Note the way the list is given a title by using a `figure` element and its `legend`.

```
<figure>
 <legend>The top 10 movies of all time</legend>
 <ol>
  <li value="10"><cite>Josie and the Pussycats</cite>, 2001</li>
```

```
      <li value="9"><cite lang="sh">Црна мачка, бели мачор</cite>, 1998</li>
      <li value="8"><cite>A Bug's Life</cite>, 1998</li>
      <li value="7"><cite>Toy Story</cite>, 1995</li>
      <li value="6"><cite>Monsters, Inc</cite>, 2001</li>
      <li value="5"><cite>Cars</cite>, 2006</li>
      <li value="4"><cite>Toy Story 2</cite>, 1999</li>
      <li value="3"><cite>Finding Nemo</cite>, 2003</li>
      <li value="2"><cite>The Incredibles</cite>, 2004</li>
      <li value="1"><cite>Ratatouille</cite>, 2007</li>
     </ol>
    </figure>
```

The markup could also be written as follows, using the `reversed` attribute on the `ol` element:

```
    <figure>
     <legend>The top 10 movies of all time</legend>
     <ol reversed>
      <li><cite>Josie and the Pussycats</cite>, 2001</li>
      <li><cite lang="sh">Црна мачка, бели мачор</cite>, 1998</li>
      <li><cite>A Bug's Life</cite>, 1998</li>
      <li><cite>Toy Story</cite>, 1995</li>
      <li><cite>Monsters, Inc</cite>, 2001</li>
      <li><cite>Cars</cite>, 2006</li>
      <li><cite>Toy Story 2</cite>, 1999</li>
      <li><cite>Finding Nemo</cite>, 2003</li>
      <li><cite>The Incredibles</cite>, 2004</li>
      <li><cite>Ratatouille</cite>, 2007</li>
     </ol>
    </figure>
```

Note: If the `li` element is the child of a `menu` element and itself has a child that defines a command, then the `li` element will match the `:enabled` and `:disabled` pseudo-classes in the same way as the first such child element does.

4.5.10 The `dl` element

Categories
Flow content.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Zero or more groups each consisting of one or more `dt` elements followed by one or more `dd` elements.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `dl` element represents an association list consisting of zero or more name-value groups (a description list). Each group must consist of one or more names (`dt` elements) followed by one or more values (`dd` elements).

Name-value groups may be terms and definitions, metadata topics and values, or any other groups of name-value data.

The values within a group are alternatives; multiple paragraphs forming part of the same

value must all be given within the same dd element.

The order of the list of groups, and of the names and values within each group, may be significant.

If a dl element is empty, it contains no groups.

If a dl element contains non-whitespace text nodes, or elements other than dt and dd, then those elements or text nodes do not form part of any groups in that dl.

If a dl element contains only dt elements, then it consists of one group with names but no values.

If a dl element contains only dd elements, then it consists of one group with values but no names.

If a dl element starts with one or more dd elements, then the first group has no associated name.

If a dl element ends with one or more dt elements, then the last group has no associated value.

Note: When a dl element doesn't match its content model, it is often due to accidentally using dd elements in the place of dt elements and vice versa. Conformance checkers can spot such mistakes and might be able to advise authors how to correctly use the markup.

In the following example, one entry ("Authors") is linked to two values ("John" and "Luke").

```
<dl>
 <dt> Authors
 <dd> John
 <dd> Luke
 <dt> Editor
 <dd> Frank
</dl>
```

In the following example, one definition is linked to two terms.

```
<dl>
 <dt lang="en-US"> <dfn>color</dfn> </dt>
 <dt lang="en-GB"> <dfn>colour</dfn> </dt>
 <dd> A sensation which (in humans) derives from the ability of
 the fine structure of the eye to distinguish three differently
 filtered analyses of a view. </dd>
</dl>
```

The following example illustrates the use of the dl element to mark up metadata of sorts. At the end of the example, one group has two metadata labels ("Authors" and "Editors") and two values ("Robert Rothman" and "Daniel Jackson").

```
<dl>
 <dt> Last modified time </dt>
 <dd> 2004-12-23T23:33Z </dd>
 <dt> Recommended update interval </dt>
 <dd> 60s </dd>
 <dt> Authors </dt>
```

```
 <dt> Editors </dt>
 <dd> Robert Rothman </dd>
 <dd> Daniel Jackson </dd>
</dl>
```

The following example shows the `dl` element used to give a set of instructions. The order of the instructions here is important (in the other examples, the order of the blocks was not important).

```
<p>Determine the victory points as follows (use the
first matching case):</p>
<dl>
 <dt> If you have exactly five gold coins </dt>
 <dd> You get five victory points </dd>
 <dt> If you have one or more gold coins, and you have one or more silver coi
 <dd> You get two victory points </dd>
 <dt> If you have one or more silver coins </dt>
 <dd> You get one victory point </dd>
 <dt> Otherwise </dt>
 <dd> You get no victory points </dd>
</dl>
```

The following snippet shows a `dl` element being used as a glossary. Note the use of `dfn` to indicate the word being defined.

```
<dl>
 <dt><dfn>Apartment</dfn>, n.</dt>
 <dd>An execution context grouping one or more threads with one or
 more COM objects.</dd>
 <dt><dfn>Flat</dfn>, n.</dt>
 <dd>A deflated tire.</dd>
 <dt><dfn>Home</dfn>, n.</dt>
 <dd>The user's login directory.</dd>
</dl>
```

Note: The `dl` element is inappropriate for marking up dialogue, since dialogue is ordered (each speaker/line pair comes after the next). For an example of how to mark up dialogue, see the `dialog` element.

4.5.11 The `dt` element

Categories
None.
Contexts in which this element may be used:
Before `dd` or `dt` elements inside `dl` elements.
Before a `dd` element inside a `dialog` element.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `dt` element represents the term, or name, part of a term-description group in a description list (`dl` element), and the talker, or speaker, part of a talker-discourse pair in a conversation (`dialog` element).

Note: The `dt` element itself, when used in a `dl` element, does not indicate that its contents are a term being defined, but this can be indicated using the `dfn` element.

If the `dt` element is the child of a `dialog` element, and it further contains a `time` element, then that `time` element represents a timestamp for when the associated discourse (`dd` element) was said, and is not part of the name of the talker.

The following extract shows how an IM conversation log could be marked up.

```
<dialog>
 <dt> <time>14:22</time> egof
 <dd> I'm not that nerdy, I've only seen 30% of the star trek episodes
 <dt> <time>14:23</time> kaj
 <dd> if you know what percentage of the star trek episodes you have seen, you
 <dt> <time>14:23</time> egof
 <dd> it's unarguably
 <dt> <time>14:24</time> kaj
 <dd> you are not helping your case
</dialog>
```

4.5.12 The `dd` element

Categories
None.
Contexts in which this element may be used:
After `dt` or `dd` elements inside `dl` elements.
After a `dt` element inside a `dialog` element.
Content model:
Flow content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `dd` element represents the description, definition, or value, part of a term-description group in a description list (`dl` element), and the discourse, or quote, part in a conversation (`dialog` element).

A `dl` can be used to define a vocabulary list, like in a dictionary. In the following example, each entry, given by a `dt` with a `dfn`, has several `dd`s, showing the various parts of the definition.

```
<dl>
 <dt><dfn>happiness</dfn></dt>
 <dd class="pronunciation">/'hæ p. nes/</dd>
 <dd class="part-of-speech"><i><abbr>n.</abbr></i></dd>
 <dd>The state of being happy.</dd>
 <dd>Good fortune; success. <q>Oh <b>happiness</b>! It worked!</q></dd>
 <dt><dfn>rejoice</dfn></dt>
 <dd class="pronunciation">/ri jois'/</dd>
 <dd><i class="part-of-speech"><abbr>v.intr.</abbr></i> To be delighted onese
 <dd><i class="part-of-speech"><abbr>v.tr.</abbr></i> To cause one to be deli
</dl>
```

4.5.13 Common grouping idioms

4.5.13.1 Tag clouds

This specification does not define any markup specifically for marking up lists of keywords that apply to a group of pages (also known as *tag clouds*). In general, authors are encouraged to either mark up such lists using `ul` elements with explicit inline counts that are then hidden and turned into a presentational effect using a style sheet, or to use SVG.

Here, three tags are included in a short tag cloud:

```
<style>
@media screen, print, handheld, tv {
  /* should be ignored by non-visual browsers */
  .tag-cloud > li > span { display: none; }
  .tag-cloud > li { display: inline; }
  .tag-cloud-1 { font-size: 0.7em; }
  .tag-cloud-2 { font-size: 0.9em; }
  .tag-cloud-3 { font-size: 1.1em; }
  .tag-cloud-4 { font-size: 1.3em; }
  .tag-cloud-5 { font-size: 1.5em; }
}
</style>
...
<ul class="tag-cloud">
 <li class="tag-cloud-4"><a title="28 instances" href="/t/apple">apple</a> <s
 <li class="tag-cloud-2"><a title="6 instances"  href="/t/kiwi">kiwi</a> <spa
 <li class="tag-cloud-5"><a title="41 instances" href="/t/pear">pear</a> <spa
</ul>
```

The actual frequency of each tag is given using the `title` attribute. A CSS style sheet is provided to convert the markup into a cloud of differently-sized words, but for user agents that do not support CSS or are not visual, the markup contains annotations like "(popular)" or "(rare)" to categorise the various tags by frequency, thus enabling all users to benefit from the information.

The `ul` element is used (rather than `ol`) because the order is not particular important: while the list is in fact ordered alphabetically, it would convey the same information if ordered by, say, the length of the tag.

The `tag` `rel`-keyword is *not* used on these `a` elements because they do not represent tags that apply to the page itself; they are just part of an index listing the tags themselves.

4.6 Text-level semantics

4.6.1 The `a` element

Categories

Interactive content.

Flow content.

When the element only contains phrasing content: phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Transparent, but there must be no interactive content descendant.

Content attributes:

Global attributes
href
target
ping
rel
media
hreflang
type

DOM interface:

```
[Stringifies=href] interface HTMLAnchorElement : HTMLElement {
           attribute DOMString href;
           attribute DOMString target;
           attribute DOMString ping;
           attribute DOMString rel;
  readonly attribute DOMTokenList relList;
           attribute DOMString media;
           attribute DOMString hreflang;
           attribute DOMString type;
};
```

The Command interface must also be implemented by this element.

If the a element has an href attribute, then it represents a hyperlink (a hypertext anchor).

If the a element has no href attribute, then the element represents a placeholder for where a link might otherwise have been placed, if it had been relevant.

The target, ping, rel, media, hreflang, and type attributes must be omitted if the href attribute is not present.

If a site uses a consistent navigation tool bar on every page, then the link that would normally link to the page itself could be marked up using an a element:

```
<nav>
 <ul>
  <li> <a href="/">Home</a> </li>
  <li> <a href="/news">News</a> </li>
  <li> <a>Examples</a> </li>
  <li> <a href="/legal">Legal</a> </li>
 </ul>
</nav>
```

Interactive user agents should allow users to follow hyperlinks created using the a element. The href, target and ping attributes decide how the link is followed. The rel, media, hreflang, and type attributes may be used to indicate to the user the likely nature of the target resource before the user follows the link.

The activation behavior of a elements that represent hyperlinks is to run the following steps:

1. If the DOMActivate event in question is not trusted (i.e. a click() method call was the

reason for the event being dispatched), and the a element's target attribute is ... then raise an INVALID_ACCESS_ERR exception and abort these steps.

2. If the target of the click event is an img element with an ismap attribute specified, then server-side image map processing must be performed, as follows:

    1. If the DOMActivate event was dispatched as the result of a real pointing-device-triggered click event on the img element, then let *x* be the distance in CSS pixels from the left edge of the image's left border, if it has one, or the left edge of the image otherwise, to the location of the click, and let *y* be the distance in CSS pixels from the top edge of the image's top border, if it has one, or the top edge of the image otherwise, to the location of the click. Otherwise, let *x* and *y* be zero.

    2. Let the ***hyperlink suffix*** be a U+003F QUESTION MARK character, the value of *x* expressed as a base-ten integer using ASCII digits (U+0030 DIGIT ZERO to U+0039 DIGIT NINE), a U+002C COMMA character, and the value of *y* expressed as a base-ten integer using ASCII digits.

3. Finally, the user agent must follow the hyperlink defined by the a element. If the steps above defined a *hyperlink suffix*, then take that into account when following the hyperlink.

The DOM attributes **href**, **ping**, **target**, **rel**, **media**, **hreflang**, and **type**, must reflect the respective content attributes of the same name.

The DOM attribute **relList** must reflect the rel content attribute.

> The a element may be wrapped around entire paragraphs, lists, tables, and so forth, even entire sections, so long as there is no interactive content within (e.g. buttons or other links). This example shows how this can be used to make an entire advertising block into a link:
>
> ```
> <aside class="advertising">
>  <h1>Advertising</h1>
>  <a href="http://ad.example.com/?adid=1929&pubid=1422">
>   <section>
>    <h1>Mellblomatic 9000!</h1>
>    Turn all your widgets into mellbloms!
>    Only $9.99 plus shipping and handling.
>   </section>
>  </a>
>  <a href="http://ad.example.com/?adid=375&pubid=1422">
>   <section>
>    <h1>The Mellblom Browser</h1>
>    Web browsing at the speed of light.
>    No other browser goes faster!
>   </section>
>  </a>
> </aside>
> ```

4.6.2 The q element

Categories

Flow content.
Phrasing content.

Contexts in which this element may be used:
Where phrasing content is expected.

Content model:
Phrasing content.

Content attributes:
Global attributes
`cite`

DOM interface:
The `q` element uses the `HTMLQuoteElement` interface.

The `q` element represents some phrasing content quoted from another source.

Quotation punctuation (such as quotation marks) must not appear immediately before, after, or inside `q` elements; they will be inserted into the rendering by the user agent.

Content inside a `q` element must be quoted from another source, whose address, if it has one, should be cited in the **`cite`** attribute. The source may be fictional, as when quoting characters in a novel or screenplay.

If the `cite` attribute is present, it must be a valid URL. To obtain the corresponding citation link, the value of the attribute must be resolved relative to the element. User agents should allow users to follow such citation links.

The `q` element must not be used in place of quotation marks that do not represent quotes; for example, it is inappropriate to use the `q` element for marking up sarcastic statements.

The use of `q` elements to mark up quotations is entirely optional; using explicit quotation punctuation without `q` elements is just as correct.

Here is a simple example of the use of the `q` element:

```
<p>The man said <q>Things that are impossible just take
longer</q>. I disagreed with him.</p>
```

Here is an example with both an explicit citation link in the `q` element, and an explicit citation outside:

```
<p>The W3C page <cite>About W3C</cite> says the W3C's
mission is <q cite="http://www.w3.org/Consortium/">To lead the
World Wide Web to its full potential by developing protocols and
guidelines that ensure long-term growth for the Web</q>. I
disagree with this mission.</p>
```

In the following example, the quotation itself contains a quotation:

```
<p>In <cite>Example One</cite>, he writes <q>The man
said <q>Things that are impossible just take longer</q>. I
disagreed with him</q>. Well, I disagree even more!</p>
```

In the following example, quotation marks are used instead of the `q` element:

```
<p>His best argument was “I disagree”, which
I thought was laughable.</p>
```

In the following example, there is no quote — the quotation marks are used to name a

word. Use of the `q` element in this case would be inappropriate.

```
<p>The word "ineffable" could have been used to describe the disaster
resulting from the campaign's mismanagement.</p>
```

4.6.3 The `cite` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `cite` element represents the title of a work (e.g. a book, a paper, an essay, a poem, a score, a song, a script, a film, a TV show, a game, a sculpture, a painting, a theatre production, a play, an opera, a musical, an exhibition, etc). This can be a work that is being quoted or referenced in detail (i.e. a citation), or it can just be a work that is mentioned in passing.

A person's name is not the title of a work — even if people call that person a piece of work — and the element must therefore not be used to mark up people's names. (In some cases, the `b` element might be appropriate for names; e.g. in a gossip article where the names of famous people are keywords rendered with a different style to draw attention to them. In other cases, if an element is *really* needed, the `span` element can be used.)

A ship is similarly not a work, and the element must not be used to mark up ship names (the `i` element can be used for that purpose).

This next example shows a typical use of the `cite` element:

```
<p>My favourite book is <cite>The Reality Dysfunction</cite> by
Peter F. Hamilton. My favourite comic is <cite>Pearls Before
Swine</cite> by Stephan Pastis. My favourite track is <cite>Jive
Samba</cite> by the Cannonball Adderley Sextet.</p>
```

This is correct usage:

```
<p>According to the Wikipedia article <cite>HTML</cite>, as it
stood in mid-February 2008, leaving attribute values unquoted is
unsafe. This is obviously an over-simplification.</p>
```

The following, however, is incorrect usage, as the `cite` element here is containing far more than the title of the work:

```
<!-- do not copy this example, it is an example of bad usage! -->
<p>According to <cite>the Wikipedia article on HTML</cite>, as it
stood in mid-February 2008, leaving attribute values unquoted is
unsafe. This is obviously an over-simplification.</p>
```

The cite element is obviously a key part of any citation in a bibliography, but it is only used to mark the title:

```
<p><cite>Universal Declaration of Human Rights</cite>, United Nations,
December 1948.  Adopted by General Assembly resolution 217 A (III).</p>
```

Note: A citation is not a quote (for which the q element is appropriate).

This is incorrect usage, because cite is not for quotes:

```
<p><cite>This is wrong!</cite>, said Ian.</p>
```

This is also incorrect usage, because a person is not a work:

```
<p><q>This is still wrong!</q>, said <cite>Ian</cite>.</p>
```

The correct usage does not use a cite element:

```
<p><q>This is correct</q>, said Ian.</p>
```

As mentioned above, the b element might be relevant for marking names as being keywords in certain kinds of documents:

```
<p>And then <b>Ian</b> said <q>this might be right, in a
gossip column, maybe!</q>.</p>
```

4.6.4 The em element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses HTMLElement.

The em element represents stress emphasis of its contents.

The level of emphasis that a particular piece of content has is given by its number of ancestor em elements.

The placement of emphasis changes the meaning of the sentence. The element thus forms an integral part of the content. The precise way in which emphasis is used in this way depends on the language.

These examples show how changing the emphasis changes the meaning. First, a general statement of fact, with no emphasis:

```
<p>Cats are cute animals.</p>
```

By emphasizing the first word, the statement implies that the kind of animal under discussion is in question (maybe someone is asserting that dogs are cute):

```
<p><em>Cats</em> are cute animals.</p>
```

Moving the emphasis to the verb, one highlights that the truth of the entire sentence is in question (maybe someone is saying cats are not cute):

```
<p>Cats <em>are</em> cute animals.</p>
```

By moving it to the adjective, the exact nature of the cats is reasserted (maybe someone suggested cats were *mean* animals):

```
<p>Cats are <em>cute</em> animals.</p>
```

Similarly, if someone asserted that cats were vegetables, someone correcting this might emphasize the last word:

```
<p>Cats are cute <em>animals</em>.</p>
```

By emphasizing the entire sentence, it becomes clear that the speaker is fighting hard to get the point across. This kind of emphasis also typically affects the punctuation, hence the exclamation mark here.

```
<p><em>Cats are cute animals!</em></p>
```

Anger mixed with emphasizing the cuteness could lead to markup such as:

```
<p><em>Cats are <em>cute</em> animals!</em></p>
```

4.6.5 The `strong` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `strong` element represents strong importance for its contents.

The relative level of importance of a piece of content is given by its number of ancestor `strong` elements; each `strong` element increases the importance of its contents.

Changing the importance of a piece of text with the `strong` element does not change the meaning of the sentence.

Here is an example of a warning notice in a game, with the various parts marked up according to how important they are:

```
<p><strong>Warning.</strong> This dungeon is dangerous.
<strong>Avoid the ducks.</strong> Take any gold you find.
<strong><strong>Do not take any of the diamonds</strong>,
they are explosive and <strong>will destroy anything within
ten meters.</strong></strong> You have been warned.</p>
```

4.6.6 The `small` element

Categories

Flow content.
Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Content attributes:

Global attributes

DOM interface:

Uses `HTMLElement`.

The `small` element represents small print or other side comments.

Note: Small print is typically legalese describing disclaimers, caveats, legal restrictions, or copyrights. Small print is also sometimes used for attribution.

Note: The `small` element does not "de-emphasize" or lower the importance of text emphasised by the `em` element or marked as important with the `strong` element.

In this example the footer contains contact information and a copyright.

```
<footer>
 <address>
  For more details, contact
  <a href="mailto:js@example.com">John Smith</a>.
 </address>
 <p><small>© copyright 2038 Example Corp.</small></p>
</footer>
```

In this second example, the `small` element is used for a side comment in an article.

```
<p>Example Corp today announced record profits for the
second quarter <small>(Full Disclosure: Foo News is a subsidiary of
Example Corp)</small>, leading to speculation about a third quarter
merger with Demo Group.</p>
```

This is distinct from a sidebar, which might be multiple paragraphs long and is removed from the main flow of text. In the following example, we see a sidebar from the same article. This sidebar also has small print, indicating the source of the information in the sidebar.

```
<aside>
 <h1>Example Corp</h1>
 <p>This company mostly creates small software and Web
 sites.</p>
```

```
 <p>The Example Corp company mission is "To provide entertainment
 and news on a sample basis".</p>
 <p><small>Information obtained from <a
 href="http://example.com/about.html">example.com</a> home
 page.</small></p>
</aside>
```

In this last example, the `small` element is marked as being *important* small print.

```
<p><strong><small>Continued use of this service will result in a kiss.</small
```

4.6.7 The `mark` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `mark` element represents a run of text in one document marked or highlighted for reference purposes, due to its relevance in another context. When used in a quotation or other block of text referred to from the prose, it indicates a highlight that was not originally present but which has been added to bring the reader's attention to a part of the text that might not have been considered important by the original author when the block was originally written, but which is now under previously unexpected scrutiny. When used in the main prose of a document, it indicates a part of the document that has been highlighted due to its likely relevance to the user's current activity.

This example shows how the `mark` example can be used to bring attention to a particular part of a quotation:

```
<p lang="en-US">Consider the following quote:</p>
<blockquote lang="en-GB">
 <p>Look around and you will find, no-one's really
 <mark>colour</mark> blind.</p>
</blockquote>
<p lang="en-US">As we can tell from the <em>spelling</em> of the word,
the person writing this quote is clearly not American.</p>
```

Another example of the `mark` element is highlighting parts of a document that are matching some search string. If someone looked at a document, and the server knew that the user was searching for the word "kitten", then the server might return the document with one paragraph modified as follows:

```
<p>I also have some <mark>kitten</mark>s who are visiting me
these days. They're really cute. I think they like my garden! Maybe I
should adopt a <mark>kitten</mark>.</p>
```

In the following snippet, a paragraph of text refers to a specific part of a code fragment.

```
<p>The highlighted part below is where the error lies:</p>
<pre><code>var i: Integer;
begin
   i := <mark>1.1</mark>;
end.</code></pre>
```

This is another example showing the use of `mark` to highlight a part of quoted text that was originally not emphasised. In this example, common typographic conventions have led the author to explicitly style `mark` elements in quotes to render in italics.

```
<article>
 <style>
  blockquote mark, q mark {
    font: inherit; font-style: italic;
    text-decoration: none;
    background: transparent; color: inherit;
  }
  .bubble em {
    font: inherit; font-size: larger;
    text-decoration: underline;
  }
 </style>
 <h1>She knew</h1>
 <p>Did you notice the subtle joke in the joke on panel 4?</p>
 <blockquote>
  <p class="bubble">I didn't <em>want</em> to believe. <mark>Of course
  on some level I realized it was a known-plaintext attack.</mark> But I
  couldn't admit it until I saw for myself.</p>
 </blockquote>
 <p>(Emphasis mine.) I thought that was great. It's so pedantic, yet it
 explains everything neatly.</p>
</article>
```

Note, incidentally, the distinction between the `em` element in this example, which is part of the original text being quoted, and the `mark` element, which is highlighting a part for comment.

The following example shows the difference between denoting the *importance* of a span of text (`strong`) as opposed to denoting the *relevance* of a span of text (`mark`). It is an extract from a textbook, where the extract has had the parts relevant to the exam highlighted. The safety warnings, important though they may be, are apparently not relevant to the exam.

```
<h3>Wormhole Physics Introduction</h3>

<p><mark>A wormhole in normal conditions can be held open for a
maximum of just under 39 minutes.</mark> Conditions that can increase
the time include a powerful energy source coupled to one or both of
the gates connecting the wormhole, and a large gravity well (such as a
black hole).</p>

<p><mark>Momentum is preserved across the wormhole. Electromagnetic
radiation can travel in both directions through a wormhole,
but matter cannot.</mark></p>

<p>When a wormhole is created, a vortex normally forms.
<strong>Warning: The vortex caused by the wormhole opening will
annihilate anything in its path.</strong> Vortexes can be avoided when
using sufficiently advanced dialing technology.</p>

<p><mark>An obstruction in a gate will prevent it from accepting a
wormhole connection.</mark></p>
```

4.6.8 The `dfn` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content, but there must be no descendant `dfn` elements.
Content attributes:
Global attributes
Also, the `title` attribute has special semantics on this element.
DOM interface:
Uses `HTMLElement`.

The `dfn` element represents the defining instance of a term. The paragraph, description list group, or section that is the nearest ancestor of the `dfn` element must also contain the definition(s) for the term given by the `dfn` element.

Defining term: If the `dfn` element has a `title` attribute, then the exact value of that attribute is the term being defined. Otherwise, if it contains exactly one element child node and no child text nodes, and that child element is an `abbr` element with a `title` attribute, then the exact value of *that* attribute is the term being defined. Otherwise, it is the exact `textContent` of the `dfn` element that gives the term being defined.

If the `title` attribute of the `dfn` element is present, then it must contain only the term being defined.

Note: The `title` attribute of ancestor elements does not affect `dfn` elements.

An `a` element that links to a `dfn` element represents an instance of the term defined by the `dfn` element.

In the following fragment, the term "GDO" is first defined in the first paragraph, then used in the second.

```
<p>The <dfn><abbr title="Garage Door Opener">GDO</abbr></dfn>
is a device that allows off-world teams to open the iris.</p>
<!-- ... later in the document: -->
<p>Teal'c activated his <abbr title="Garage Door Opener">GDO</abbr>
and so Hammond ordered the iris to be opened.</p>
```

With the addition of an `a` element, the reference can be made explicit:

```
<p>The <dfn id=gdo><abbr title="Garage Door Opener">GDO</abbr></dfn>
is a device that allows off-world teams to open the iris.</p>
<!-- ... later in the document: -->
<p>Teal'c activated his <a href=#gdo><abbr title="Garage Door Opener">GDO</abl
and so Hammond ordered the iris to be opened.</p>
```

4.6.9 The `abbr` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
Also, the `title` attribute has special semantics on this element.
DOM interface:
Uses `HTMLElement`.

The `abbr` element represents an abbreviation or acronym, optionally with its expansion. The **`title`** attribute may be used to provide an expansion of the abbreviation. The attribute, if specified, must contain an expansion of the abbreviation, and nothing else.

> The paragraph below contains an abbreviation marked up with the `abbr` element. This paragraph defines the term "Web Hypertext Application Technology Working Group".
>
> ```
> The <dfn id=whatwg><abbr title="Web Hypertext Application
> Technology Working Group">WHATWG</abbr></dfn> is a loose
> unofficial collaboration of Web browser manufacturers and interested
> parties who wish to develop new technologies designed to allow authors
> to write and deploy Applications over the World Wide Web.
> ```
>
> An alternative way to write this would be:
>
> ```
> The <dfn id=whatwg>Web Hypertext Application Technology
> Working Group</dfn> (<abbr title="Web Hypertext Application
> Technology Working Group">WHATWG</abbr>) is a loose unofficial
> collaboration of Web browser manufacturers and interested parties who
> wish to develop new technologies designed to allow authors to write
> and deploy Applications over the World Wide Web.
> ```

> This paragraph has two abbreviations. Notice how only one is defined; the other, with no expansion associated with it, does not use the `abbr` element.
>
> ```
> The <abbr title="Web Hypertext Application Technology Working
> Group">WHATWG</abbr> started working on HTML5 in 2004.
> ```

> This paragraph links an abbreviation to its definition.
>
> ```
> The <a href="#whatwg"><abbr title="Web Hypertext Application
> Technology Working Group">WHATWG</abbr></a> community does not
> have much representation from Asia.
> ```

> This paragraph marks up an abbreviation without giving an expansion, possibly as a hook to apply styles for abbreviations (e.g. smallcaps).
>
> ```
> Philip` and Dashiva both denied that they were going to
> get the issue counts from past revisions of the specification to
> backfill the <abbr>WHATWG</abbr> issue graph.
> ```

If an abbreviation is pluralized, the expansion's grammatical number (plural vs singular) must match the grammatical number of the contents of the element.

> Here the plural is outside the element, so the expansion is in the singular:

```
<p>Two <abbr title="Working Group">WG</abbr>s worked on
this specification: the <abbr>WHATWG</abbr> and the
<abbr>HTMLWG</abbr>.</p>
```

Here the plural is inside the element, so the expansion is in the plural:

```
<p>Two <abbr title="Working Groups">WGs</abbr> worked on
this specification: the <abbr>WHATWG</abbr> and the
<abbr>HTMLWG</abbr>.</p>
```

Abbreviations do not have to be marked up using this element. It is expected to be useful in the following cases:

- Abbreviations for which the author wants to give expansions, where using the `abbr` element with a `title` attribute is an alternative to including the expansion inline (e.g. in parentheses).
- Abbreviations that are likely to be unfamiliar to the document's readers, for which authors are encouraged to either mark up the abbreviation using a `abbr` element with a `title` attribute or include the expansion inline in the text the first time the abbreviation is used.
- Abbreviations whose presence needs to be semantically annotated, e.g. so that they can be identified from a style sheet and given specific styles, for which the `abbr` element can be used without a `title` attribute.

Providing an expansion in a `title` attribute once will not necessarily cause other `abbr` elements in the same document with the same contents but without a `title` attribute to behave as if they had the same expansion. Every `abbr` element is independent.

4.6.10 The `time` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
`datetime`
DOM interface:

```
interface HTMLTimeElement : HTMLElement {
           attribute DOMString dateTime;
  readonly attribute Date date;
  readonly attribute Date time;
  readonly attribute Date timezone;
};
```

The `time` element represents a precise date and/or a time in the proleptic Gregorian calendar. [GREGORIAN]

This element is intended as a way to encode modern dates and times in a machine-readable way so that user agents can offer to add them to the user's calendar. For example, adding birthday reminders or scheduling events.

> ***The `time` element is not intended for encoding times for which a precise date or time cannot be established. For example, it would be inappropriate for encoding times like "one millisecond after the big bang", "the early part of the Jurassic period", or "a winter around 250 BCE".***
>
> ***For dates before the introduction of the Gregorian calendar, authors are encouraged to not use the `time` element, or else to be very careful about converting dates and times from the period to the Gregorian calendar. This is complicated by the way the Gregorian calendar was phased in at different times in different countries, ranging from part way through the 16th century all the way to early in the 20th.***

The **`datetime`** attribute, if present, must contain a valid date or time string that identifies the date or time being specified.

If the `datetime` attribute is not present, then the date or time must be specified in the content of the element, such that the element's `textContent` is a valid date or time string in content, and the date, if any, must be expressed using the Gregorian calendar.

> The `time` element can be used to encode dates, for example in Microformats. The following shows a hypothetical way of encoding an event using a variant on hCalendar that uses the `time` element:
>
> ```
> 
>  <a class="url" href="http://www.web2con.com/">http://www.web2con.com/</a>
>   Web 2.0 Conference:
>   <time class="dtstart" datetime="2007-10-05">October 5</time> -
>   <time class="dtend" datetime="2007-10-20">19</time>,
>   at the Argent Hotel, San Francisco, CA
>  
> ```

> The `time` element is not necessary for encoding dates or times. In the following snippet, the time is encoded using `time`, so that it can be restyled (e.g. using XBL2) to match local conventions, while the year is not marked up at all, since marking it up would not be particularly useful.
>
> ```
> I usually have a snack at <time>16:00</time>.
> I've liked model trains since at least 1983.
> ```
>
> Using a styling technology that supports restyling times, the first paragraph from the above snipet could be rendered as follows:
>
> *I usually have a snack at 4pm.*
>
> Or it could be rendered as follows:
>
> *I usually have a snack at 16h00.*

The **`dateTime`** DOM attribute must reflect the `datetime` content attribute.

User agents, to obtain the **date**, **time**, and **timezone** represented by a `time` element, must follow these steps:

1. If the `datetime` attribute is present, then use the rules to parse a date or time string with the flag *in attribute* from the value of that attribute, and let the result be *result*.

2. Otherwise, use the rules to parse a date or time string with the flag *in content* from the element's `textContent`, and let the result be *result*.

3. If *result* is empty (because the parsing failed), then the date is unknown, the time is unknown, and the timezone is unknown.

4. Otherwise: if *result* contains a date, then that is the date; if *result* contains a time, then that is the time; and if *result* contains a timezone, then the timezone is the element's timezone. (A timezone can only be present if both a date and a time are also present.)

time **`.date`**

Returns a `Date` object representing the date component of the element's value, at midnight in the UTC timezone.

Returns null if there is no date.

time **`.time`**

Returns a `Date` object representing the time component of the element's value, on 1970-01-01 in the UTC timezone.

Returns null if there is no time.

time **`.timezone`**

Returns a `Date` object representing the time corresponding to 1970-01-01 00:00 UTC in the timezone given by the element's value.

Returns null if there is no timezone.

The **`date`** DOM attribute must return null if the date is unknown, and otherwise must return the time corresponding to midnight UTC (i.e. the first second) of the given date.

The **`time`** DOM attribute must return null if the time is unknown, and otherwise must return the time corresponding to the given time of 1970-01-01, with the timezone UTC.

The **`timezone`** DOM attribute must return null if the timezone is unknown, and otherwise must return the time corresponding to 1970-01-01 00:00 UTC in the given timezone, with the timezone set to UTC (i.e. the time corresponding to 1970-01-01 at 00:00 UTC plus the offset corresponding to the timezone).

> In the following snippet:
>
> ```
> Our first date was <time datetime="2006-09-23">a Saturday</time>.
> ```
>
> ...the `time` element's `date` attribute would have the value 1,158,969,600,000ms, and the `time` and `timezone` attributes would return null.

In the following snippet:

```
<p>We stopped talking at <time datetime="2006-09-24T05:00-07:00">5am the next
```

...the `time` element's `date` attribute would have the value 1,159,056,000,000ms, the `time` attribute would have the value 18,000,000ms, and the `timezone` attribute would return −25,200,000ms. To obtain the actual time, the three attributes can be added together, obtaining 1,159,048,800,000, which is the specified date and time in UTC.

Finally, in the following snippet:

```
<p>Many people get up at <time>08:00</time>.</p>
```

...the `time` element's `date` attribute would have the value null, the `time` attribute would have the value 28,800,000ms, and the `timezone` attribute would return null.

4.6.11 The `progress` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
`value`
`max`
DOM interface:

```
interface HTMLProgressElement : HTMLElement {
           attribute float value;
           attribute float max;
  readonly attribute float position;
};
```

The `progress` element represents the completion progress of a task. The progress is either indeterminate, indicating that progress is being made but that it is not clear how much more work remains to be done before the task is complete (e.g. because the task is waiting for a remote host to respond), or the progress is a number in the range zero to a maximum, giving the fraction of work that has so far been completed.

There are two attributes that determine the current task completion represented by the element.

The **`value`** attribute specifies how much of the task has been completed, and the **`max`** attribute specifies how much work the task requires in total. The units are arbitrary and not specified.

Instead of using the attributes, authors are recommended to include the current value and the maximum value inline as text inside the element.

Here is a snippet of a Web application that shows the progress of some automated task:

```
<section>
 <h2>Task Progress</h2>
 <p>Progress: <progress><span id="p">0</span>%</progress></p>
 <script>
  var progressBar = document.getElementById('p');
  function updateProgress(newValue) {
    progressBar.textContent = newValue;
  }
 </script>
</section>
```

(The `updateProgress()` method in this example would be called by some other code on the page to update the actual progress bar as the task progressed.)

Author requirements: The `max` and `value` attributes, when present, must have values that are valid floating point numbers. The `max` attribute, if present, must have a value greater than zero. The `value` attribute, if present, must have a value equal to or greater than zero, and less than or equal to the value of the `max` attribute, if present, or 1, otherwise.

Note: The `progress` element is the wrong element to use for something that is just a gauge, as opposed to task progress. For instance, indicating disk space usage using `progress` would be inappropriate. Instead, the `meter` element is available for such use cases.

User agent requirements: User agents must parse the `max` and `value` attributes' values according to the rules for parsing floating point number values.

If the `value` attribute is omitted, then user agents must also parse the `textContent` of the `progress` element in question using the steps for finding one or two numbers of a ratio in a string. These steps will return nothing, one number, one number with a denominator punctuation character, or two numbers.

Using the results of this processing, user agents must determine whether the progress bar is an indeterminate progress bar, or whether it is a determinate progress bar, and in the latter case, what its current and maximum values are, all as follows:

1. If the `max` attribute is omitted, and the `value` is omitted, and the results of parsing the `textContent` was nothing, then the progress bar is an indeterminate progress bar. Abort these steps.

2. Otherwise, it is a determinate progress bar.

3. If the `max` attribute is included, then, if a value could be parsed out of it, then the maximum value is that value.

4. Otherwise, if the `max` attribute is absent but the `value` attribute is present, or, if the `max` attribute is present but no value could be parsed from it, then the maximum is 1.

5. Otherwise, if neither attribute is included, then, if the `textContent` contained one number with an associated denominator punctuation character, then the maximum value is the value associated with that denominator punctuation character; otherwise, if the

`textContent` contained two numbers, the maximum value is the higher of the two values; otherwise, the maximum value is 1.

6. If the `value` attribute is present on the element and a value could be parsed out of it, that value is the current value of the progress bar. Otherwise, if the attribute is present but no value could be parsed from it, the current value is zero.

7. Otherwise if the `value` attribute is absent and the `max` attribute is present, then, if the `textContent` was parsed and found to contain just one number, with no associated denominator punctuation character, then the current value is that number. Otherwise, if the `value` attribute is absent and the `max` attribute is present then the current value is zero.

8. Otherwise, if neither attribute is present, then the current value is the lower of the one or two numbers that were found in the `textContent` of the element.

9. If the maximum value is less than or equal to zero, then it is reset to 1.

10. If the current value is less than zero, then it is reset to zero.

11. Finally, if the current value is greater than the maximum value, then the current value is reset to the maximum value.

UA requirements for showing the progress bar: When representing a `progress` element to the user, the UA should indicate whether it is a determinate or indeterminate progress bar, and in the former case, should indicate the relative position of the current value relative to the maximum value.

The **`max`** and **`value`** DOM attributes must reflect the respective content attributes of the same name. When the relevant content attributes are absent, the DOM attributes must return zero. The value parsed from the `textContent` never affects the DOM values.

progress **`.position`**

For a determinate progress bar (one with known current and maximum values), returns the result of dividing the current value by the maximum value.

For an indeterminate progress bar, returns −1.

If the progress bar is an indeterminate progress bar, then the **`position`** DOM attribute must return −1. Otherwise, it must return the result of dividing the current value by the maximum value.

4.6.12 The `meter` element

Categories

Flow content.
Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Content attributes:

Global attributes
value
min
low
high
max
optimum

DOM interface:

```
interface HTMLMeterElement : HTMLElement {
           attribute float value;
           attribute float min;
           attribute float max;
           attribute float low;
           attribute float high;
           attribute float optimum;
};
```

The meter element represents a scalar measurement within a known range, or a fractional value; for example disk usage, the relevance of a query result, or the fraction of a voting population to have selected a particular candidate.

This is also known as a gauge.

Note: The meter element should not be used to indicate progress (as in a progress bar). For that role, HTML provides a separate progress element.

Note: The meter element also does not represent a scalar value of arbitrary range — for example, it would be wrong to use this to report a weight, or height, unless there is a known maximum value.

There are six attributes that determine the semantics of the gauge represented by the element.

The **min** attribute specifies the lower bound of the range, and the **max** attribute specifies the upper bound. The **value** attribute specifies the value to have the gauge indicate as the "measured" value.

The other three attributes can be used to segment the gauge's range into "low", "medium", and "high" parts, and to indicate which part of the gauge is the "optimum" part. The **low** attribute specifies the range that is considered to be the "low" part, and the **high** attribute specifies the range that is considered to be the "high" part. The **optimum** attribute gives the position that is "optimum"; if that is higher than the "high" value then this indicates that the higher the value, the better; if it's lower than the "low" mark then it indicates that lower values are better, and naturally if it is in between then it indicates that neither high nor low values are good.

Authoring requirements: The recommended way of giving the value is to include it as

contents of the element, either as two numbers (the higher number represents the maximum, the other number the current value, and the minimum is assumed to be zero), or as a percentage or similar (using one of the characters such as "%"), or as a fraction. However, it is also possible to use the attributes to specify these values.

One of the following conditions, along with all the requirements that are listed with that condition, must be met:

There are exactly two numbers in the contents of the element, and the `value`, `min`, and `max` attributes are all omitted

If specified, the `low`, `high`, and `optimum` attributes must have values greater than or equal to zero and less than or equal to the bigger of the two numbers in the contents of the element.

If both the `low` and `high` attributes are specified, then the `low` attribute's value must be less than or equal to the value of the `high` attribute.

There is exactly one number followed by zero or more White_Space characters and a valid denominator punctuation character in the contents of the element, and the `value`, `min`, and `max` attributes are all omitted

If specified, the `low`, `high`, and `optimum` attributes must have values greater than or equal to zero and less than or equal to the value associated with the denominator punctuation character.

If both the `low` and `high` attributes are specified, then the `low` attribute's value must be less than or equal to the value of the `high` attribute.

There is exactly one number in the contents of the element, and the `value` attribute is omitted

There are no numbers in the contents of the element, and the `value` attribute is specified

If the `min` attribute attribute is specified, then the *minimum* is that attribue's value; otherwise, it is 0.

If the `max` attribute attribute is specified, then the *maximum* is that attribue's value; otherwise, it is 1.

If there is exactly one number in the contents of the element, then *value* is that number; otherwise, *value* is the value of the `value` attribute.

The following inequalities must hold, as applicable:

- *minimum* ≤ *value* ≤ *maximum*
- *minimum* ≤ `low` ≤ *maximum* (if `low` is specified)
- *minimum* ≤ `high` ≤ *maximum* (if `high` is specified)
- *minimum* ≤ `optimum` ≤ *maximum* (if `optimum` is specified)

If both the `low` and `high` attributes are specified, then the `low` attribute's value must be less than or equal to the value of the `high` attribute.

For the purposes of these requirements, a number is a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), optionally including with a single U+002E FULL STOP character (.), and separated from other numbers by at least one character that isn't any of those; interpreted as a base ten number.

The `value`, `min`, `low`, `high`, `max`, and `optimum` attributes, when present, must have values that are valid floating point numbers.

Note: If no minimum or maximum is specified, then the range is assumed to be 0..1, and the value thus has to be within that range.

The following examples all represent a measurement of three quarters (of the maximum of whatever is being measured):

```
<meter>75%</meter>
<meter>750‰</meter>
<meter>3/4</meter>
<meter>6 blocks used (out of 8 total)</meter>
<meter>max: 100; current: 75</meter>
<meter><object data="graph75.png">0.75</object></meter>
<meter min="0" max="100" value="75"></meter>
```

The following example is incorrect use of the element, because it doesn't give a range (and since the default maximum is 1, both of the gauges would end up looking maxed out):

```
<p>The grapefruit pie had a radius of <meter>12cm</meter>
and a height of <meter>2cm</meter>.</p> <!-- BAD! -->
```

Instead, one would either not include the meter element, or use the meter element with a defined range to give the dimensions in context compared to other pies:

```
<p>The grapefruit pie had a radius of 12cm and a height of
2cm.</p>
<dl>
 <dt>Radius: <dd> <meter min=0 max=20 value=12>12cm</meter>
 <dt>Height: <dd> <meter min=0 max=10 value=2>2cm</meter>
</dl>
```

There is no explicit way to specify units in the `meter` element, but the units may be specified in the `title` attribute in free-form text.

The example above could be extended to mention the units:

```
<dl>
 <dt>Radius: <dd> <meter min=0 max=20 value=12 title="centimeters">12cm</mete
 <dt>Height: <dd> <meter min=0 max=10 value=2 title="centimeters">2cm</meter>
</dl>
```

User agent requirements: User agents must parse the `min`, `max`, `value`, `low`, `high`, and `optimum` attributes using the rules for parsing floating point number values.

If the `value` attribute has been omitted, the user agent must also process the `textContent` of the element according to the steps for finding one or two numbers of a ratio in a string. These steps will return nothing, one number, one number with a denominator punctuation character, or two numbers.

User agents must then use all these numbers to obtain values for six points on the gauge, as follows. (The order in which these are evaluated is important, as some of the values refer to earlier ones.)

The minimum value

If the `min` attribute is specified and a value could be parsed out of it, then the minimum value is that value. Otherwise, the minimum value is zero.

The maximum value

If the `max` attribute is specified and a value could be parsed out of it, the maximum value is that value.

Otherwise, if the `max` attribute is specified but no value could be parsed out of it, or if it was not specified, but either or both of the `min` or `value` attributes *were* specified, then the maximum value is 1.

Otherwise, none of the `max`, `min`, and `value` attributes were specified. If the result of processing the `textContent` of the element was either nothing or just one number with no denominator punctuation character, then the maximum value is 1; if the result was one number but it had an associated denominator punctuation character, then the maximum value is the value associated with that denominator punctuation character; and finally, if there were two numbers parsed out of the `textContent`, then the maximum is the higher of those two numbers.

If the above machinations result in a maximum value less than the minimum value, then the maximum value is actually the same as the minimum value.

The actual value

If the `value` attribute is specified and a value could be parsed out of it, then that value is the actual value.

If the `value` attribute is not specified but the `max` attribute *is* specified and the result of processing the `textContent` of the element was one number with no associated denominator punctuation character, then that number is the actual value.

If neither of the `value` and `max` attributes are specified, then, if the result of processing the `textContent` of the element was one number (with or without an associated denominator punctuation character), then that is the actual value, and if the result of processing the `textContent` of the element was two numbers, then the actual value is the lower of the two numbers found.

Otherwise, if none of the above apply, the actual value is zero.

If the above procedure results in an actual value less than the minimum value, then the actual value is actually the same as the minimum value.

If, on the other hand, the result is an actual value greater than the maximum value, then the actual value is the maximum value.

The low boundary

If the `low` attribute is specified and a value could be parsed out of it, then the low boundary is that value. Otherwise, the low boundary is the same as the minimum value.

If the low boundary is then less than the minimum value, then the low boundary is actually the same as the minimum value. Similarly, if the low boundary is greater than the maximum value, then it is actually the maximum value instead.

The high boundary

If the `high` attribute is specified and a value could be parsed out of it, then the high

boundary is that value. Otherwise, the high boundary is the same as the maximum value.

If the high boundary is then less than the low boundary, then the high boundary is actually the same as the low boundary. Similarly, if the high boundary is greater than the maximum value, then it is actually the maximum value instead.

The optimum point

If the `optimum` attribute is specified and a value could be parsed out of it, then the optimum point is that value. Otherwise, the optimum point is the midpoint between the minimum value and the maximum value.

If the optimum point is then less than the minimum value, then the optimum point is actually the same as the minimum value. Similarly, if the optimum point is greater than the maximum value, then it is actually the maximum value instead.

All of which will result in the following inequalities all being true:

- minimum value ≤ actual value ≤ maximum value
- minimum value ≤ low boundary ≤ high boundary ≤ maximum value
- minimum value ≤ optimum point ≤ maximum value

UA requirements for regions of the gauge: If the optimum point is equal to the low boundary or the high boundary, or anywhere in between them, then the region between the low and high boundaries of the gauge must be treated as the optimum region, and the low and high parts, if any, must be treated as suboptimal. Otherwise, if the optimum point is less than the low boundary, then the region between the minimum value and the low boundary must be treated as the optimum region, the region between the low boundary and the high boundary must be treated as a suboptimal region, and the region between the high boundary and the maximum value must be treated as an even less good region. Finally, if the optimum point is higher than the high boundary, then the situation is reversed; the region between the high boundary and the maximum value must be treated as the optimum region, the region between the high boundary and the low boundary must be treated as a suboptimal region, and the remaining region between the low boundary and the minimum value must be treated as an even less good region.

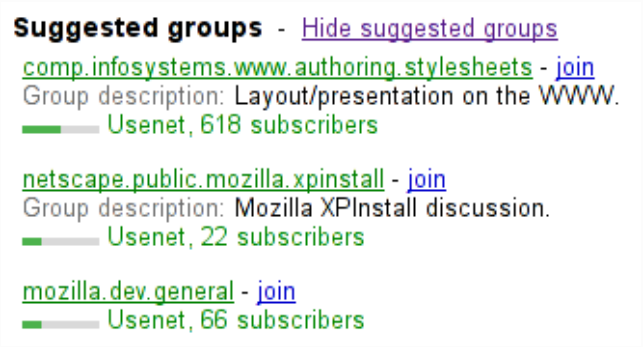
UA requirements for showing the gauge: When representing a `meter` element to the user, the UA should indicate the relative position of the actual value to the minimum and maximum values, and the relationship between the actual value and the three regions of the gauge.

> The following markup:

```
<h3>Suggested groups</h3>
<menu type="toolbar">
 <a href="?cmd=hsg" onclick="hideSuggestedGroups()">Hide suggested groups</a>
</menu>
<ul>
 <li>
  <p><a href="/group/comp.infosystems.www.authoring.stylesheets/view">comp.in
     <a href="/group/comp.infosystems.www.authoring.stylesheets/subscribe">jo
  <p>Group description: <strong>Layout/presentation on the WWW.</strong></p>
  <p><meter value="0.5">Moderate activity,</meter> Usenet, 618 subscribers</p
 </li>
 <li>
  <p><a href="/group/netscape.public.mozilla.xpinstall/view">netscape.public.
     <a href="/group/netscape.public.mozilla.xpinstall/subscribe">join</a></p
  <p>Group description: <strong>Mozilla XPInstall discussion.</strong></p>
```

```
      <p><meter value="0.25">Low activity,</meter> Usenet, 22 subscribers</p>
     </li>
     <li>
      <p><a href="/group/mozilla.dev.general/view">mozilla.dev.general</a> -
         <a href="/group/mozilla.dev.general/subscribe">join</a></p>
      <p><meter value="0.25">Low activity,</meter> Usenet, 66 subscribers</p>
     </li>
    </ul>
```

Might be rendered as follows:



User agents may combine the value of the `title` attribute and the other attributes to provide context-sensitive help or inline text detailing the actual values.

For example, the following snippet:

```
<meter min=0 max=60 value=23.2 title=seconds></meter>
```

...might cause the user agent to display a gauge with a tooltip saying "Value: 23.2 out of 60." on one line and "seconds" on a second line.

The **`min`**, **`max`**, **`value`**, **`low`**, **`high`**, and **`optimum`** DOM attributes must reflect the respective content attributes of the same name. When the relevant content attributes are absent, the DOM attributes must return zero. The value parsed from the `textContent` never affects the DOM values.

4.6.13 The `code` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `code` element represents a fragment of computer code. This could be an XML element name, a filename, a computer program, or any other string that a computer would recognize.

Although there is no formal way to indicate the language of computer code being marked up, authors who wish to mark `code` elements with the language used, e.g. so that syntax highlighting scripts can use the right rules, may do so by adding a class prefixed with "`language-`" to the element.

> The following example shows how the element can be used in a paragraph to mark up element names and computer code, including punctuation.
>
> ```
> The <code>code</code> element represents a fragment of computer
> code.
>
> When you call the <code>activate()</code> method on the
> <code>robotSnowman</code> object, the eyes glow.
>
> The example below uses the <code>begin</code> keyword to indicate
> the start of a statement block. It is paired with an <code>end</code>
> keyword, which is followed by the <code>.</code> punctuation character
> (full stop) to indicate the end of the program.
> ```

> The following example shows how a block of code could be marked up using the `pre` and `code` elements.
>
> ```
> <pre><code class="language-pascal">var i: Integer;
> begin
>    i := 1;
> end.</code></pre>
> ```
>
> A class is used in that example to indicate the language used.

Note: See the `pre` element for more details.

4.6.14 The `var` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `var` element represents a variable. This could be an actual variable in a mathematical expression or programming context, or it could just be a term used as a placeholder in prose.

> In the paragraph below, the letter "n" is being used as a variable in prose:
>
> ```
> If there are <var>n</var> pipes leading to the ice
> cream factory then I expect at least <var>n</var>
> flavours of ice cream to be available for purchase!
> ```

For mathematics, in particular for anything beyond the simplest of expressions, MathML is more appropriate. However, the `var` element can still be used to refer to specific variables that are then mentioned in MathML expressions.

In this example, an equation is shown, with a legend that references the variables in the equation. The expression itself is marked up with MathML, but the variables are mentioned in the figure's legend using `var`.

```
<figure>
 <math>
  <mi>a</mi>
  <mo>=</mo>
  <msqrt>
   <msup><mi>b</mi><mn>2</mn></msup>
   <mi>+</mi>
   <msup><mi>c</mi><mn>2</mn></msup>
  </msqrt>
 </math>
 <legend>
  Using Pythagoras' theorem to solve for the hypotenuse <var>a</var> of
  a triangle with sides <var>b</var> and <var>c</var>
 </legend>
</figure>
```

4.6.15 The `samp` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `samp` element represents (sample) output from a program or computing system.

Note: See the `pre` and `kbd` elements for more details.

This example shows the `samp` element being used inline:

```
<p>The computer said <samp>Too much cheese in tray
two</samp> but I didn't know what that meant.</p>
```

This second example shows a block of sample output. Nested `samp` and `kbd` elements allow for the styling of specific elements of the sample output using a style sheet.

```
<pre><samp><samp class="prompt">jdoe@mowmow:~$</samp> <kbd>ssh demo.example.c
Last login: Tue Apr 12 09:10:17 2005 from mowmow.example.com on pts/1
Linux demo 2.6.10-grsec+gg3+e+fhs6b+nfs+gr0501+++p3+c4a+gr2b-reslog-v6.189 #1

<samp class="prompt">jdoe@demo:~$</samp> <samp class="cursor">_</samp></samp>
```

4.6.16 The `kbd` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `kbd` element represents user input (typically keyboard input, although it may also be used to represent other input, such as voice commands).

When the `kbd` element is nested inside a `samp` element, it represents the input as it was echoed by the system.

When the `kbd` element *contains* a `samp` element, it represents input based on system output, for example invoking a menu item.

When the `kbd` element is nested inside another `kbd` element, it represents an actual key or other single unit of input as appropriate for the input mechanism.

> Here the `kbd` element is used to indicate keys to press:
>
> ```
> To make George eat an apple, press <kbd><kbd>Shift</kbd>+<kbd>F3</kbd></kb
> ```
>
> In this second example, the user is told to pick a particular menu item. The outer `kbd` element marks up a block of input, with the inner `kbd` elements representing each individual step of the input, and the `samp` elements inside them indicating that the steps are input based on something being displayed by the system, in this case menu labels:
>
> ```
> To make George eat an apple, select
>     <kbd><kbd><samp>File</samp></kbd>|<kbd><samp>Eat Apple...</samp></kbd></kb
> 
> ```

4.6.17 The `sub` and `sup` elements

Categories
Flow content.
Phrasing content.
Contexts in which these elements may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes

DOM interface:
Uses `HTMLElement`.

The `sup` element represents a superscript and the `sub` element represents a subscript.

These elements must be used only to mark up typographical conventions with specific meanings, not for typographical presentation for presentation's sake. For example, it would be inappropriate for the `sub` and `sup` elements to be used in the name of the LaTeX document preparation system. In general, authors should use these elements only if the *absence* of those elements would change the meaning of the content.

When the `sub` element is used inside a `var` element, it represents the subscript that identifies the variable in a family of variables.

```
<p>The coordinate of the <var>i</var>th point is
(<var>x<sub><var>i</var></sub></var>, <var>y<sub><var>i</var></sub></var>).
For example, the 10th point has coordinate
(<var>x<sub>10</sub></var>, <var>y<sub>10</sub></var>).</p>
```

In certain languages, superscripts are part of the typographical conventions for some abbreviations.

```
<p>The most beautiful women are
<span lang="fr"><abbr>M<sup>lle</sup></abbr> Gwendoline</span> and
<span lang="fr"><abbr>M<sup>me</sup></abbr> Denise</span>.</p>
```

Mathematical expressions often use subscripts and superscripts. Authors are encouraged to use MathML for marking up mathematics, but authors may opt to use `sub` and `sup` if detailed mathematical markup is not desired. [MathML]

```
<var>E</var>=<var>m</var><var>c</var><sup>2</sup>

f(<var>x</var>, <var>n</var>) = log<sub>4</sub><var>x</var><sup><var>n</var><
```

4.6.18 The `span` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `span` element doesn't mean anything on its own, but can be useful when used together with other attributes, e.g. `class`, `lang`, or `dir`. It represents its children.

4.6.19 The `i` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `i` element represents a span of text in an alternate voice or mood, or otherwise offset from the normal prose, such as a taxonomic designation, a technical term, an idiomatic phrase from another language, a thought, a ship name, or some other prose whose typical typographic presentation is italicized.

Terms in languages different from the main text should be annotated with `lang` attributes (`xml:lang` in XML).

The examples below show uses of the `i` element:

```
<p>The <i class="taxonomy">Felis silvestris catus</i> is cute.</p>
<p>The term <i>prose content</i> is defined above.</p>
<p>There is a certain <i lang="fr">je ne sais quoi</i> in the air.</p>
```

In the following example, a dream sequence is marked up using `i` elements.

```
<p>Raymond tried to sleep.</p>
<p><i>The ship sailed away on Thursday</i>, he
dreamt. <i>The ship had many people aboard, including a beautiful
princess called Carey. He watched her, day-in, day-out, hoping she
would notice him, but she never did.</i></p>
<p><i>Finally one night he picked up the courage to speak with
her—</i></p>
<p>Raymond woke with a start as the fire alarm rang out.</p>
```

Authors are encouraged to use the `class` attribute on the `i` element to identify why the element is being used, so that if the style of a particular use (e.g. dream sequences as opposed to taxonomic terms) is to be changed at a later date, the author doesn't have to go through the entire document (or series of related documents) annotating each use. Similarly, authors are encouraged to consider whether other elements might be more applicable than the `i` element, for instance the `em` element for marking up stress emphasis, or the `dfn` element to mark up the defining instance of a term.

Note: Style sheets can be used to format `i` elements, just like any other element can be restyled. Thus, it is not the case that content in `i` elements will necessarily be italicized.

4.6.20 The `b` element

Categories

Flow content.
Phrasing content.

Contexts in which this element may be used:
Where phrasing content is expected.

Content model:
Phrasing content.

Content attributes:
Global attributes

DOM interface:
Uses `HTMLElement`.

The `b` element represents a span of text to be stylistically offset from the normal prose without conveying any extra importance, such as key words in a document abstract, product names in a review, or other spans of text whose typical typographic presentation is boldened.

> The following example shows a use of the `b` element to highlight key words without marking them up as important:
>
> ```
> The frobonitor and barbinator components are fried.
> ```

> In the following example, objects in a text adventure are highlighted as being special by use of the `b` element.
>
> ```
> You enter a small room. Your sword glows
> brighter. A rat scurries past the corner wall.
> ```

> Another case where the `b` element is appropriate is in marking up the lede (or lead) sentence or paragraph. The following example shows how a BBC article about kittens adopting a rabbit as their own could be marked up using HTML5 elements:
>
> ```
> <article>
>  <h2>Kittens 'adopted' by pet rabbit</h2>
>  Six abandoned kittens have found an unexpected new
>  mother figure – a pet rabbit.
>  Veterinary nurse Melanie Humble took the three-week-old
>  kittens to her Aberdeen home.
> [...]
> ```

The `b` element should be used as a last resort when no other element is more appropriate. In particular, headers should use the `h1` to `h6` elements, stress emphasis should use the `em` element, importance should be denoted with the `strong` element, and text marked or highlighted should use the `mark` element.

> The following would be *incorrect* usage:
>
> ```
> WARNING! Do not frob the barbinator!
> ```
>
> In the previous example, the correct element to use would have been `strong`, not `b`.

Note: Style sheets can be used to format `b` elements, just like any other element can be restyled. Thus, it is not the case that content in `b` elements will necessarily be boldened.

4.6.21 The `bdo` element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content.
Content attributes:
Global attributes
Also, the `dir` global attribute has special semantics on this element.
DOM interface:
Uses `HTMLElement`.

The `bdo` element represents explicit text directionality formatting control for its children. It allows authors to override the Unicode bidi algorithm by explicitly specifying a direction override. [BIDI]

Authors must specify the `dir` attribute on this element, with the value `ltr` to specify a left-to-right override and with the value `rtl` to specify a right-to-left override.

If the element has the `dir` attribute set to the exact value `ltr`, then for the purposes of the bidi algorithm, the user agent must act as if there was a U+202D LEFT-TO-RIGHT OVERRIDE character at the start of the element, and a U+202C POP DIRECTIONAL FORMATTING at the end of the element.

If the element has the `dir` attribute set to the exact value `rtl`, then for the purposes of the bidi algorithm, the user agent must act as if there was a U+202E RIGHT-TO-LEFT OVERRIDE character at the start of the element, and a U+202C POP DIRECTIONAL FORMATTING at the end of the element.

The requirements on handling the `bdo` element for the bidi algorithm may be implemented indirectly through the style layer. For example, an HTML+CSS user agent should implement these requirements by implementing the CSS `unicode-bidi` property. [CSS21]

4.6.22 The `ruby` element

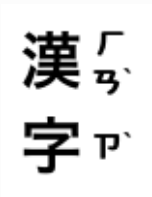
Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
One or more groups of: phrasing content followed either by a single `rt` element, or an `rp` element, an `rt` element, and another `rp` element.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `ruby` element allows one or more spans of phrasing content to be marked with ruby

annotations. Ruby annotations are short runs of text presented alongside base text, primarily used in East Asian typography as a guide for pronounciation or to include other annotations. In Japanese, this form of typography is also known as *furigana*.

A `ruby` element represents the spans of phrasing content it contains, ignoring all the child `rt` and `rp` elements and their descendants. Those spans of phrasing content have associated annotations created using the `rt` element.

> In this example, each ideograph in the Japanese text 漢字 is annotated with its kanji reading.
>
> ```
> ...
> <ruby>
>  漢 <rt> かん </rt>
>  字 <rt> じ   </rt>
> </ruby>
> ...
> ```
>
> This might be rendered as:
>
> The two main ideographs, each with its kanji annotation rendered in a smaller font above it.

> In this example, each ideograph in the traditional Chinese text 漢字 is annotated with its bopomofo reading.
>
> ```
> <ruby>
>  漢 <rt> ㄏㄢˋ </rt>
>  字 <rt> ㄗˋ   </rt>
> </ruby>
> ```
>
> This might be rendered as:
>
> 
>

> In this example, each ideograph in the simplified Chinese text 汉字 is annotated with its pinyin reading.
>
> ```
> ...
> <ruby>
>  汉 <rt> hàn </rt>
>  字 <rt> zì  </rt>
> </ruby>
> ...
> ```
>
> This might be rendered as:
>
>
>

4.6.23 The `rt` element

Categories
None.
Contexts in which this element may be used:
As a child of a `ruby` element.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `rt` element marks the ruby text component of a ruby annotation.

An `rt` element that is a child of a `ruby` element represents an annotation (given by its children) for the zero or more nodes of phrasing content that immediately precedes it in the `ruby` element, ignoring `rp` elements.

An `rt` element that is not a child of a `ruby` element represents the same thing as its children.

4.6.24 The `rp` element

Categories
None.
Contexts in which this element may be used:
As a child of a `ruby` element, either immediately before or immediately after an `rt` element.
Content model:
If the `rp` element is immediately after an `rt` element that is immediately preceded by another `rp` element: a single character from Unicode character class Pe.
Otherwise: a single character from Unicode character class Ps.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `rp` element can be used to provide parentheses around a ruby text component of a ruby annotation, to be shown by user agents that don't support ruby annotations.

An `rp` element that is a child of a `ruby` element represents nothing and its contents must be ignored. An `rp` element whose parent element is not a `ruby` element represents its children.

> The example above, in which each ideograph in the text 漢字 is annotated with its kanji reading, could be expanded to use `rp` so that in legacy user agents the readings are in parentheses:
>
> ```
> ...
> <ruby>
> ```

```
 漢 <rp>(</rp><rt>かん</rt><rp>)</rp>
 字 <rp>(</rp><rt>じ</rt><rp>)</rp>
</ruby>
...
```

In conforming user agents the rendering would be as above, but in user agents that do not support ruby, the rendering would be:

```
... 漢（かん）字（じ）...
```

4.6.25 Usage summary

We need to summarize the various elements, in particular to distinguish b/i/em/strong/var/q/mark/cite.

4.6.26 Footnotes

HTML does not have a dedicated mechanism for marking up footnotes. Here are the recommended alternatives.

For short inline annotations, the `title` attribute should be used.

In this example, two parts of a dialog are annotated.

```
<dialog>
 <dt>Customer
 <dd>Hello! I wish to register a complaint. Hello. Miss?
 <dt>Shopkeeper
 <dd><span title="Colloquial pronunciation of 'What do you'"
 >Watcha</span> mean, miss?
 <dt>Customer
 <dd>Uh, I'm sorry, I have a cold. I wish to make a complaint.
 <dt>Shopkeeper
 <dd>Sorry, <span title="This is, of course, a lie.">we're
 closing for lunch</span>.
</dialog>
```

For longer annotations, the `a` element should be used, pointing to an element later in the document. The convention is that the contents of the link be a number in square brackets.

In this example, a footnote in the dialog links to a paragraph below the dialog. The paragraph then reciprocally links back to the dialog, allowing the user to return to the location of the footnote.

```
<dialog>
 <dt>Announcer
 <dd>Number 16: The <i>hand</i>.
 <dt>Interviewer
 <dd>Good evening. I have with me in the studio tonight Mr
 Norman St John Polevaulter, who for the past few years has
 been contradicting people. Mr Polevaulter, why <em>do</em>
 you contradict people?
 <dt>Norman
 <dd>I don't. <a href="#fn1" id="r1">[1]</a>
 <dt>Interviewer
 <dd>You told me you did!
</dialog>
```

```
<section>
 <p id="fn1"><a href="#r1">[1]</a> This is, naturally, a lie,
 but paradoxically if it were true he could not say so without
 contradicting the interviewer and thus making it false.</p>
</section>
```

For side notes, longer annotations that apply to entire sections of the text rather than just specific words or sentences, the `aside` element should be used.

In this example, a sidebar is given after a dialog, giving some context to the dialog.

```
<dialog>
 <dt>Customer
 <dd>I will not buy this record, it is scratched.
 <dt>Shopkeeper
 <dd>I'm sorry?
 <dt>Customer
 <dd>I will not buy this record, it is scratched.
 <dt>Shopkeeper
 <dd>No no no, this's'a tobacconist's.
</dialog>
<aside>
 <p>In 1970, the British Empire lay in ruins, and foreign
 nationalists frequented the streets — many of them Hungarians
 (not the streets — the foreign nationals). Sadly, Alexander
 Yalt has been publishing incompetently-written phrase books.
</aside>
```

4.7 Edits

The `ins` and `del` elements represent edits to the document.

4.7.1 The `ins` element

Categories
Flow content.
When the element only contains phrasing content: phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Transparent.
Content attributes:
Global attributes
`cite`
`datetime`
DOM interface:
Uses the `HTMLModElement` interface.

The `ins` element represents an addition to the document.

The following represents the addition of a single paragraph:

```
<aside>
 <ins>
  <p> I like fruit. </p>
 </ins>
```

```
</aside>
```

As does this, because everything in the `aside` element here counts as phrasing content and therefore there is just one paragraph:

```
<aside>
 <ins>
  Apples are <em>tasty</em>.
 </ins>
 <ins>
  So are pears.
 </ins>
</aside>
```

`ins` elements should not cross implied paragraph boundaries.

The following example represents the addition of two paragraphs, the second of which was inserted in two parts. The first `ins` element in this example thus crosses a paragraph boundary, which is considered poor form.

```
<aside>
 <ins datetime="2005-03-16T00:00Z">
  <p> I like fruit. </p>
  Apples are <em>tasty</em>.
 </ins>
 <ins datetime="2007-12-19T00:00Z">
  So are pears.
 </ins>
</aside>
```

Here is a better way of marking this up. It uses more elements, but none of the elements cross implied paragraph boundaries.

```
<aside>
 <ins datetime="2005-03-16T00:00Z">
  <p> I like fruit. </p>
 </ins>
 <ins datetime="2005-03-16T00:00Z">
  Apples are <em>tasty</em>.
 </ins>
 <ins datetime="2007-12-19T00:00Z">
  So are pears.
 </ins>
</aside>
```

4.7.2 The `del` element

Categories

Flow content.

When the element only contains phrasing content: phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Transparent.

Content attributes:

Global attributes

`cite`

`datetime`

DOM interface:

Uses the `HTMLModElement` interface.

The `del` element represents a removal from the document.

`del` elements should not cross implied paragraph boundaries.

4.7.3 Attributes common to `ins` and `del` elements

The **`cite`** attribute may be used to specify the address of a document that explains the change. When that document is long, for instance the minutes of a meeting, authors are encouraged to include a fragment identifier pointing to the specific part of that document that discusses the change.

If the `cite` attribute is present, it must be a valid URL that explains the change. To obtain the corresponding citation link, the value of the attribute must be resolved relative to the element. User agents should allow users to follow such citation links.

The **`datetime`** attribute may be used to specify the time and date of the change.

If present, the `datetime` attribute must be a valid global date and time string value.

User agents must parse the `datetime` attribute according to the parse a global date and time string algorithm. If that doesn't return a time, then the modification has no associated timestamp (the value is non-conforming; it is not a valid global date and time string). Otherwise, the modification is marked as having been made at the given datetime. User agents should use the associated timezone information to determine which timezone to present the given datetime in.

The `ins` and `del` elements must implement the `HTMLModElement` interface:

```
interface HTMLModElement : HTMLElement {
           attribute DOMString cite;
           attribute DOMString dateTime;
};
```

The **`cite`** DOM attribute must reflect the element's `cite` content attribute. The **`dateTime`** DOM attribute must reflect the element's `datetime` content attribute.

4.7.4 Edits and paragraphs

Since the `ins` and `del` elements do not affect paragraphing, it is possible, in some cases where paragraphs are implied (without explicit `p` elements), for an `ins` or `del` element to span both an entire paragraph or other non-phrasing content elements and part of another paragraph.

For example:

```
<section>
 <ins>
  <p>
   This is a paragraph that was inserted.
  </p>
  This is another paragraph whose first sentence was inserted
```

```
  at the same time as the paragraph above.
 </ins>
 This is a second sentence, which was there all along.
</section>
```

By only wrapping some paragraphs in `p` elements, one can even get the end of one paragraph, a whole second paragraph, and the start of a third paragraph to be covered by the same `ins` or `del` element (though this is very confusing, and not considered good practice):

```
<section>
 This is the first paragraph. <ins>This sentence was
 inserted.
 <p>This second paragraph was inserted.</p>
 This sentence was inserted too.</ins> This is the
 third paragraph in this example.
</section>
```

However, due to the way implied paragraphs are defined, it is not possible to mark up the end of one paragraph and the start of the very next one using the same `ins` or `del` element. You instead have to use one (or two) `p` element(s) and two `ins` or `del` elements:

For example:

```
<section>
 <p>This is the first paragraph. <del>This sentence was
 deleted.</del></p>
 <p><del>This sentence was deleted too.</del> That
 sentence needed a separate &lt;del&gt; element.</p>
</section>
```

Partly because of the confusion described above, authors are strongly recommended to always mark up all paragraphs with the `p` element, and to not have any `ins` or `del` elements that cross across any implied paragraphs.

4.7.5 Edits and lists

The content models of the `ol` and `ul` elements do not allow `ins` and `del` elements as children. Lists always represent all their items, including items that would otherwise have been marked as deleted.

To indicate that an item is inserted or deleted, an `ins` or `del` element can be wrapped around the contents of the `li` element. To indicate that an item has been replaced by another, a single `li` element can have one or more `del` elements followed by one or more `ins` elements.

> In the following example, a list that started empty had items added and removed from it over time. The bits in the example that have been emphasised show the parts that are the "current" state of the list. The list item numbers don't take into account the edits, though.
>
> ```
> <h1>Stop-ship bugs</h1>
> <ol>
>  <li><ins datetime="2008-02-12T15:20Z">Bug 225:
>  Rain detector doesn't work in snow</ins></li>
>  <li><del datetime="2008-03-01T20:22Z"><ins datetime="2008-02-14T12:02Z">Bug 2
>  Water buffer overflows in April</ins></del></li>
>  <li><ins datetime="2008-02-16T13:50Z">Bug 230:
>  Water heater doesn't use renewable fuels</ins></li>
>  <li><del datetime="2008-02-20T21:15Z"><ins datetime="2008-02-16T14:25Z">Bug 2
>  Carbon dioxide emissions detected after startup</ins></del></li>
> ```

```
    </ol>
```

In the following example, a list that started with just fruit was replaced by a list with just colors.

```
<h1>List of <del>fruits</del><ins>colors</ins></h1>
<ul>
 <li><del>Lime</del><ins>Green</ins></li>
 <li><del>Apple</del></li>
 <li>Orange</li>
 <li><del>Pear</del></li>
 <li><ins>Teal</ins></li>
 <li><del>Lemon</del><ins>Yellow</ins></li>
 <li>Olive</li>
 <li><ins>Purple</ins>
</ul>
```

4.8 Embedded content

4.8.1 The `figure` element

Categories
Flow content.
Sectioning root.
Contexts in which this element may be used:
Where flow content is expected.
Content model:
Either: one `legend` element followed by flow content.
Or: Flow content followed by one `legend` element.
Or: Flow content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `figure` element represents some flow content, optionally with a caption, which can be moved away from the main flow of the document without affecting the document's meaning.

The element can thus be used to annotate illustrations, diagrams, photos, code listings, etc, that are referred to from the main content of the document, but that could, without affecting the flow of the document, be moved away from that primary content, e.g. to the side of the page, to dedicated pages, or to an appendix.

The first `legend` element child of the element, if any, represents the caption of the `figure` element's contents. If there is no child `legend` element, then there is no caption.

The remainder of the element's contents, if any, represents the content.

This example shows the `figure` element to mark up a code listing.

```
<p>In <a href="#l4">listing 4</a> we see the primary core interface
API declaration.</p>
<figure id="l4">
 <legend>Listing 4. The primary core interface API declaration.</legend>
 <pre><code>interface PrimaryCore {
```

```
  boolean verifyDataLine();
  void sendData(in sequence&lt;byte> data);
  void initSelfDestruct();
}</code></pre>
</figure>
<p>The API is designed to use UTF-8.</p>
```

Here we see a `figure` element to mark up a photo.

```
<figure>
 <img src="bubbles-work.jpeg"
      alt="Bubbles, sitting in his office chair, works on his
           latest project intently.">
 <legend>Bubbles at work</legend>
</figure>
```

In this example, we see an image that is *not* a figure, as well as an image and a video that are.

```
<h2>Malinko's comics</h2>

<p>This case centered on some sort of "intellectual property"
infringement related to a comic (see Exhibit A). The suit started
after a trailer ending with these words:</p>

<img src="promblem-packed-action.png" alt="ROUGH COPY! Promblem-Packed Action

<p>...was aired. A lawyer, armed with a Bigger Notebook, launched a
preemptive strike using snowballs. A complete copy of the trailer is
included with Exhibit B.</p>

<figure>
 <img src="ex-a.png" alt="Two squiggles on a dirty piece of paper.">
 <legend>Exhibit A. The alleged <cite>rough copy</cite> comic.</legend>
</figure>

<figure>
 <video src="ex-b.mov"></video>
 <legend>Exhibit B. The <code>Rough Copy</cite> trailer.</legend>
</figure>

<p>The case was resolved out of court.</p>
```

Here, a part of a poem is marked up using `figure`.

```
<figure>
 <p>'Twas brillig, and the slithy toves<br>
Did gyre and gimble in the wabe;<br>
All mimsy were the borogoves,<br>
And the mome raths outgrabe.</p>
 <legend><cite>Jabberwocky</cite> (first verse). Lewis Carroll, 1832-98</lege
</figure>
```

In this example, which could be part of a much larger work discussing a castle, the figure has three images in it.

```
<figure>
 <img src="castle1423.jpeg" title="Etching. Anonymous, ca. 1423."
      alt="The castle has one tower, and a tall wall around it.">
 <img src="castle1858.jpeg" title="Oil-based paint on canvas. Maria Towle, 18
      alt="The castle now has two towers and two walls.">
 <img src="castle1999.jpeg" title="Film photograph. Peter Jankle, 1999."
      alt="The castle lies in ruins, the original tower all that remains in o
 <legend>The castle through the ages: 1423, 1858, and 1999 respectively.</leg
</figure>
```

4.8.2 The `img` element

Categories

Flow content.
Phrasing content.
Embedded content.
If the element has an `usemap` attribute: Interactive content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Empty.

Content attributes:

Global attributes
`alt`
`src`
`usemap`
`ismap`
`width`
`height`

DOM interface:

```
[NamedConstructor=Image(),
 NamedConstructor=Image(in unsigned long width),
 NamedConstructor=Image(in unsigned long width, in unsigned long height)]
interface HTMLImageElement : HTMLElement {
           attribute DOMString alt;
           attribute DOMString src;
           attribute DOMString useMap;
           attribute boolean isMap;
           attribute unsigned long width;
           attribute unsigned long height;
  readonly attribute boolean complete;
};
```

An `img` element represents an image.

The image given by the **`src`** attribute is the embedded content, and the value of the **`alt`** attribute is the `img` element's fallback content.

The `src` attribute must be present, and must contain a valid URL referencing a non-interactive, optionally animated, image resource that is neither paged nor scripted. If the *base URI of the element* is the same as the document's address, then the `src` attribute's value must not be the empty string.

> ***Note: Images can thus be static bitmaps (e.g. PNGs, GIFs, JPEGs), single-page vector documents (single-page PDFs, XML files with an SVG root element), animated bitmaps (APNGs, animated GIFs), animated vector graphics (XML files with an SVG root element that use declarative SMIL animation), and so forth. However, this also precludes SVG files with script, multipage PDF files, interactive MNG files, HTML documents, plain text documents, and so forth.***

The requirements on the `alt` attribute's value are described in the next section.

There has been some suggestion that the `longdesc` attribute from HTML4 should be included. This has been considered and rejected in the past, but if new evidence is found showing the attribute to actually help users rather than harm them, it may be reconsidered.

The `img` must not be used as a layout tool. In particular, `img` elements should not be used to display fully transparent images, as they rarely convey meaning and rarely add anything useful to the document.

Unless the user agent cannot support images, or its support for images has been disabled, or the user agent only fetches elements on demand, or the element's `src` attribute has a value that is an *ignored self-reference*, then, when an `img` is created with a `src` attribute, and whenever the `src` attribute is set subsequently, the user agent must resolve the value of that attribute, relative to the element, and if that is successful must then fetch that resource.

The `src` attribute's value is an *ignored self-reference* if its value is the empty string, and the *base URI of the element* is the same as the document's address.

Fetching the image must delay the `load` event.

?Warning! ***This, unfortunately, can be used to perform a rudimentary port scan of the user's local network (especially in conjunction with scripting, though scripting isn't actually necessary to carry out such an attack). User agents may implement cross-origin access control policies that mitigate this attack.***

If the image's type is a supported image type, and the image is a valid image of that type, then the image is said to be *available* (this affects exactly what the element represents, as defined below). This can be true even before the image is completely downloaded, if the user agent supports incremental rendering of images; in such cases, each task that is queued by the networking task source while the image is being fetched must update the presentation of the image appropriately.

If the image was not fetched (e.g. because the UA's image support is disabled, or because the `src` attribute's value is an *ignored self-reference*), or if the conditions in the previous paragraph are not met, then the image is *not available*.

> ***Note: An image might be available in one view but not another. For instance, a `Document` could be rendered by a screen reader providing a speech synthesis view of the output of a Web browser using the screen media. In this case, the image would be available in the Web browser's screen view, but not available in the screen reader's view.***

Whether the image is fetched successfully or not (e.g. whether the response code was a 2xx code or equivalent) must be ignored when determining the image's type and whether it is a valid image.

> ***Note: This allows servers to return images with error responses, and have them displayed.***

The user agents should apply the image sniffing rules to determine the type of the image, with the image's associated Content-Type headers giving the *official type*. If these rules are not applied, then the type of the image must be the type given by the image's associated Content-Type headers.

User agents must not support non-image resources with the `img` element (e.g. XML files whose root element is an HTML element). User agents must not run executable code (e.g. scripts) embedded in the image resource. User agents must only display the first page of a multipage resource (e.g. a PDF file). User agents must not allow the resource to act in an interactive fashion, but should honour any animation in the resource.

This specification does not specify which image types are to be supported.

The task that is queued by the networking task source once the resource has been fetched, must, if the download was successful and the image is *available*, queue a task to fire a `load` event on the `img` element (this happens after `complete` starts returning true); and otherwise, if the fetching process fails without a response from the remote server, or completes but the image is not a valid or supported image, queue a task to fire an `error` event on the `img` element.

What an `img` element represents depends on the `src` attribute and the `alt` attribute.

- **If the `src` attribute is set and the `alt` attribute is set to the empty string**

  The image is either decorative or supplemental to the rest of the content, redundant with some other information in the document.

  If the image is *available* and the user agent is configured to display that image, then the element represents the image specified by the `src` attribute.

  Otherwise, the element represents nothing, and may be omitted completely from the rendering. User agents may provide the user with a notification that an image is present but has been omitted from the rendering.

- **If the `src` attribute is set and the `alt` attribute is set to a value that isn't empty**

  The image is a key part of the content; the `alt` attribute gives a textual equivalent or replacement for the image.

  If the image is *available* and the user agent is configured to display that image, then the element represents the image specified by the `src` attribute.

  Otherwise, the element represents the text given by the `alt` attribute. User agents may provide the user with a notification that an image is present but has been omitted from the rendering.

- **If the `src` attribute is set and the `alt` attribute is not**

  The image might be a key part of the content, and there is no textual equivalent of the image available.

  Note: In a conforming document, the absence of the `alt` attribute indicates that the image is a key part of the content but that a textual replacement for the image was not available when the image was generated.

If the image is *available*, the element represents the image specified by the `src` attribute.

If the image is not *available* or if the user agent is not configured to display the image, then the user agent should display some sort of indicator that there is an image that is not being rendered, and may, if requested by the user, or if so configured, or when required to provide contextual information in response to navigation, provide caption information for the image, derived as follows:

1. If the image has a `title` attribute whose value is not the empty string, then the value of that attribute is the caption information; abort these steps.

2. If the image is the child of a `figure` element that has a child `legend` element, then the contents of the first such `legend` element are the caption information; abort these steps.

3. Run the algorithm to create the outline for the document.

4. If the `img` element did not end up associated with a heading in the outline, or if there are any other images that are lacking an `alt` attribute and that are associated with the same heading in the outline as the `img` element in question, then there is no caption information; abort these steps.

5. The caption information is the heading with which the image is associated according to the outline.

↪ **If the `src` attribute is not set and either the `alt` attribute is set to the empty string or the `alt` attribute is not set at all**

The element represents nothing.

↪ **Otherwise**

The element represents the text given by the `alt` attribute.

The `alt` attribute does not represent advisory information. User agents must not present the contents of the `alt` attribute in the same way as content of the `title` attribute.

User agents may always provide the user with the option to display any image, or to prevent any image from being displayed. User agents may also apply image analysis heuristics to help the user make sense of the image when the user is unable to make direct use of the image, e.g. due to a visual disability or because they are using a text terminal with no graphics capabilities.

The *contents* of `img` elements, if any, are ignored for the purposes of rendering.

The `usemap` attribute, if present, can indicate that the image has an associated image map.

The **`ismap`** attribute, when used on an element that is a descendant of an `a` element with an `href` attribute, indicates by its presence that the element provides access to a server-side image map. This affects how events are handled on the corresponding `a` element.

The `ismap` attribute is a boolean attribute. The attribute must not be specified on an element that does not have an ancestor `a` element with an `href` attribute.

The `img` element supports dimension attributes.

The DOM attributes **`alt`**, **`src`**, **`useMap`**, and **`isMap`** each must reflect the respective content attributes of the same name.

image **`.width`** **[= *value*]**
image **`.height`** **[= *value*]**

These attributes return the actual rendered dimensions of the image, or zero if the dimensions are not known.

They can be set, to change the corresponding content attributes.

image **`.complete`**

Returns true if the image has been downloaded, decoded, and found to be valid; otherwise, returns false.

image **= new `Image`([*width* [, *height*]])**

Returns a new `img` element, with the `width` and `height` attributes set to the values passed in the relevant arguments, if applicable.

The DOM attributes **`width`** and **`height`** must return the rendered width and height of the image, in CSS pixels, if the image is being rendered, and is being rendered to a visual medium; or else the intrinsic width and height of the image, in CSS pixels, if the image is *available* but not being rendered to a visual medium; or else 0, if the image is not *available* or its dimensions are not known. [CSS21]

On setting, they must act as if they reflected the respective content attributes of the same name.

The DOM attribute **`complete`** must return true if the user agent has fetched the image specified in the `src` attribute, and it is a valid image, even if the final task queued by the networking task source for the fetching of the image resource has not yet been processed. Otherwise, the attribute must return false.

Note: The value of `complete` can thus change while a script is executing.

Three constructors are provided for creating `HTMLImageElement` objects (in addition to the factory methods from DOM Core such as `createElement()`): **`Image()`**, **`Image(`*`width`*`)`**, and **`Image(`*`width`*`, `*`height`*`)`**. When invoked as constructors, these must return a new `HTMLImageElement` object (a new `img` element). If the *width* argument is present, the new object's `width` content attribute must be set to *width*. If the *height* argument is also present, the new object's `height` content attribute must be set to *height*.

A single image can have different appropriate alternative text depending on the context.

In each of the following cases, the same image is used, yet the `alt` text is different

each time. The image is the coat of arms of the Canton Geneva in Switzerland.

Here it is used as a supplementary icon:

```
<p>I lived in <img src="carouge.svg" alt=""> Carouge.</p>
```

Here it is used as an icon representing the town:

```
<p>Home town: <img src="carouge.svg" alt="Carouge"></p>
```

Here it is used as part of a text on the town:

```
<p>Carouge has a coat of arms.</p>
<p><img src="carouge.svg" alt="The coat of arms depicts a lion, sitting in fr
<p>It is used as decoration all over the town.</p>
```

Here it is used as a way to support a similar text where the description is given as well as, instead of as an alternative to, the image:

```
<p>Carouge has a coat of arms.</p>
<p><img src="carouge.svg" alt=""></p>
<p>The coat of arms depicts a lion, sitting in front of a tree.
It is used as decoration all over the town.</p>
```

Here it is used as part of a story:

```
<p>He picked up the folder and a piece of paper fell out.</p>
<p><img src="carouge.svg" alt="Shaped like a shield, the paper had a
red background, a green tree, and a yellow lion with its tongue
hanging out and whose tail was shaped like an S."></p>
<p>He stared at the folder. S! The answer he had been looking for all
this time was simply the letter S! How had he not seen that before? It all
came together now. The phone call where Hector had referred to a lion's tail,
the time Marco had stuck his tongue out...</p>
```

Here it is not known at the time of publication what the image will be, only that it will be a coat of arms of some kind, and thus no replacement text can be provided, and instead only a brief caption for the image is provided, in the `title` attribute:

```
<p>The last user to have uploaded a coat of arms uploaded this one:</p>
<p><img src="last-uploaded-coat-of-arms.cgi" title="User-uploaded coat of arm
```

Ideally, the author would find a way to provide real replacement text even in this case, e.g. by asking the previous user. Not providing replacement text makes the document more difficult to use for people who are unable to view images, e.g. blind users, or users or very low-bandwidth connections or who pay by the byte, or users who are forced to use a text-only Web browser.

Here are some more examples showing the same picture used in different contexts, with different appropriate alternate texts each time.

```
<article>
 <h1>My cats</h1>
 <h2>Fluffy</h2>
 <p>Fluffy is my favourite.</p>
 <img src="fluffy.jpg" alt="She likes playing with a ball of yarn.">
 <p>She's just too cute.</p>
 <h2>Miles</h2>
 <p>My other cat, Miles just eats and sleeps.</p>
</article>

<article>
```

```
 <h1>Photography</h1>
 <h2>Shooting moving targets indoors</h2>
 <p>The trick here is to know how to anticipate; to know at what speed and
 what distance the subject will pass by.</p>
 <img src="fluffy.jpg" alt="A cat flying by, chasing a ball of yarn, can be
 photographed quite nicely using this technique.">
 <h2>Nature by night</h2>
 <p>To achieve this, you'll need either an extremely sensitive film, or
 immense flash lights.</p>
</article>

<article>
 <h1>About me</h1>
 <h2>My pets</h2>
 <p>I've got a cat named Fluffy and a dog named Miles.</p>
 <img src="fluffy.jpg" alt="Fluffy, my cat, tends to keep itself busy.">
 <p>My dog Miles and I like go on long walks together.</p>
 <h2>music</h2>
 <p>After our walks, having emptied my mind, I like listening to Bach.</p>
</article>

<article>
 <h1>Fluffy and the Yarn</h1>
 <p>Fluffy was a cat who liked to play with yarn. He also liked to jump.</p>
 <aside><img src="fluffy.jpg" alt="" title="Fluffy"></aside>
 <p>He would play in the morning, he would play in the evening.</p>
</article>
```

4.8.2.1 Requirements for providing text to act as an alternative for images

The requirements for the `alt` attribute depend on what the image is intended to represent, as described in the following sections.

4.8.2.1.1 A LINK OR BUTTON CONTAINING NOTHING BUT THE IMAGE

When an a element that is a hyperlink, or a `button` element, has no textual content but contains one or more images, the `alt` attributes must contain text that together convey the purpose of the link or button.

> In this example, a user is asked to pick his preferred color from a list of three. Each color is given by an image, but for users who have configured their user agent not to display images, the color names are used instead:
>
> ```
> <h1>Pick your color</h1>
> <ul>
>  <li><a href="green.html"><img src="green.jpeg" alt="Green"></a></li>
>  <li><a href="blue.html"><img src="blue.jpeg" alt="Blue"></a></li>
>  <li><a href="red.html"><img src="red.jpeg" alt="Red"></a></li>
> </ul>
> ```

> In this example, each button has a set of images to indicate the kind of color output desired by the user. The first image is used in each case to give the alternative text.
>
> ```
> <button name="rgb"><img src="red" alt="RGB"><img src="green" alt=""><img src=
> <button name="cmyk"><img src="cyan" alt="CMYK"><img src="magenta" alt=""><img
> ```
>
> Since each image represents one part of the text, it could also be written like this:
>
> ```
> <button name="rgb"><img src="red" alt="R"><img src="green" alt="G"><img src="
> <button name="cmyk"><img src="cyan" alt="C"><img src="magenta" alt="M"><img s
> ```

However, with other alternative text, this might not work, and putting all the alternative text into one image in each case might make more sense:

```
<button name="rgb"><img src="red" alt="sRGB profile"><img src="green" alt="">
<button name="cmyk"><img src="cyan" alt="CMYK profile"><img src="magenta" alt
```

4.8.2.1.2 A PHRASE OR PARAGRAPH WITH AN ALTERNATIVE GRAPHICAL REPRESENTATION: CHARTS, DIAGRAMS, GRAPHS, MAPS, ILLUSTRATIONS

Sometimes something can be more clearly stated in graphical form, for example as a flowchart, a diagram, a graph, or a simple map showing directions. In such cases, an image can be given using the `img` element, but the lesser textual version must still be given, so that users who are unable to view the image (e.g. because they have a very slow connection, or because they are using a text-only browser, or because they are listening to the page being read out by a hands-free automobile voice Web browser, or simply because they are blind) are still able to understand the message being conveyed.

The text must be given in the `alt` attribute, and must convey the same message as the image specified in the `src` attribute.

It is important to realize that the alternative text is a *replacement* for the image, not a description of the image.

In the following example we have a flowchart in image form, with text in the `alt` attribute rephrasing the flowchart in prose form:

```
<p>In the common case, the data handled by the tokenization stage
comes from the network, but it can also come from script.</p>
<p><img src="images/parsing-model-overview.png" alt="The network
passes data to the Tokeniser stage, which passes data to the Tree
Construction stage. From there, data goes to both the DOM and to
Script Execution. Script Execution is linked to the DOM, and, using
document.write(), passes data to the Tokeniser."></p>
```

Here's another example, showing a good solution and a bad solution to the problem of including an image in a description.

First, here's the good solution. This sample shows how the alternative text should just be what you would have put in the prose if the image had never existed.

```
<!-- This is the correct way to do things. -->
<p>
 You are standing in an open field west of a house.
 <img src="house.jpeg" alt="The house is white, with a boarded front door.">
 There is a small mailbox here.
</p>
```

Second, here's the bad solution. In this incorrect way of doing things, the alternative text is simply a description of the image, instead of a textual replacement for the image. It's bad because when the image isn't shown, the text doesn't flow as well as in the first example.

```
<!-- This is the wrong way to do things. -->
<p>
 You are standing in an open field west of a house.
 <img src="house.jpeg" alt="A white house, with a boarded front door.">
 There is a small mailbox here.
```

```
    </p>
```

4.8.2.1.3 A SHORT PHRASE OR LABEL WITH AN ALTERNATIVE GRAPHICAL REPRESENTATION: ICONS, LOGOS

A document can contain information in iconic form. The icon is intended to help users of visual browsers to recognize features at a glance.

In some cases, the icon is supplemental to a text label conveying the same meaning. In those cases, the `alt` attribute must be present but must be empty.

> Here the icons are next to text that conveys the same meaning, so they have an empty `alt` attribute:
>
> ```
> <nav>
>  <a href="/help/"><img src="/icons/help.png" alt=""> Help</a>
>  <a href="/configure/"><img src="/icons/configuration.png" alt="">
>  Configuration Tools</a>
> </nav>
> ```

In other cases, the icon has no text next to it describing what it means; the icon is supposed to be self-explanatory. In those cases, an equivalent textual label must be given in the `alt` attribute.

> Here, posts on a news site are labeled with an icon indicating their topic.
>
> ```
> <body>
>  <article>
>   <header>
>    <h1>Ratatouille wins Best Movie of the Year award</h1>
>    <img src="movies.png" alt="Movies">
>   </header>
>   Pixar has won yet another Best Movie of the Year award,
>   making this its 8th win in the last 12 years.
>  </article>
>  <article>
>   <header>
>    <h1>Latest TWiT episode is online</h1>
>    <img src="podcasts.png" alt="Podcasts">
>   </header>
>   The latest TWiT episode has been posted, in which we hear
>   several tech news stories as well as learning much more about the
>   iPhone. This week, the panelists compare how reflective their
>   iPhones' Apple logos are.
>  </article>
> </body>
> ```

Many pages include logos, insignia, flags, or emblems, which stand for a particular entity such as a company, organization, project, band, software package, country, or some such.

If the logo is being used to represent the entity, e.g. as a page header, the `alt` attribute must contain the name of the entity being represented by the logo. The `alt` attribute must *not* contain text like the word "logo", as it is not the fact that it is a logo that is being conveyed, it's the entity itself.

If the logo is being used next to the name of the entity that it represents, then the logo is supplemental, and its `alt` attribute must instead be empty.

If the logo is merely used as decorative material (as branding, or, for example, as a side

image in an article that mentions the entity to which the logo belongs), then the entry below on purely decorative images applies. If the logo is actually being discussed, then it is being used as a phrase or paragraph (the description of the logo) with an alternative graphical representation (the logo itself), and the first entry above applies.

> In the following snippets, all four of the above cases are present. First, we see a logo used to represent a company:
>
> ```
> <h1><img src="XYZ.gif" alt="The XYZ company"></h1>
> ```
>
> Next, we see a paragraph which uses a logo right next to the company name, and so doesn't have any alternative text:
>
> ```
> <article>
>  <h2>News</h2>
>  We have recently been looking at buying the <img src="alpha.gif"
>  alt=""> ABG company, a small Greek company
>  specializing in our type of product.
> ```
>
> In this third snippet, we have a logo being used in an aside, as part of the larger article discussing the acquisition:
>
> ```
>  <aside> <img src="alpha-large.gif" alt=""> </aside>
>  The ABG company has had a good quarter, and our
>  pie chart studies of their accounts suggest a much bigger blue slice
>  than its green and orange slices, which is always a good sign.
> </article>
> ```
>
> Finally, we have an opinion piece talking about a logo, and the logo is therefore described in detail in the alternative text.
>
> ```
> Consider for a moment their logo:
>
> <img src="/images/logo" alt="It consists of a green circle with a
> green question mark centered inside it.">
>
> How unoriginal can you get? I mean, oooooh, a question mark, how
> revolutionary, how utterly ground-breaking, I'm
> sure everyone will rush to adopt those specifications now! They could
> at least have tried for some sort of, I don't know, sequence of
> rounded squares with varying shades of green and bold white outlines,
> at least that would look good on the cover of a blue book.
> ```
>
> This example shows how the alternative text should be written such that if the image isn't *available*, and the text is used instead, the text flows seamlessly into the surrounding text, as if the image had never been there in the first place.

4.8.2.1.4 Text that has been rendered to a graphic for typographical effect

Sometimes, an image just consists of text, and the purpose of the image is not to highlight the actual typographic effects used to render the text, but just to convey the text itself.

In such cases, the `alt` attribute must be present but must consist of the same text as written in the image itself.

> Consider a graphic containing the text "Earth Day", but with the letters all decorated with flowers and plants. If the text is merely being used as a header, to spice up the page for graphical users, then the correct alternative text is just the same text "Earth

Day", and no mention need be made of the decorations:

```
<h1><img src="earthdayheader.png" alt="Earth Day"></h1>
```

4.8.2.1.5 A GRAPHICAL REPRESENTATION OF SOME OF THE SURROUNDING TEXT

In many cases, the image is actually just supplementary, and its presence merely reinforces the surrounding text. In these cases, the `alt` attribute must be present but its value must be the empty string.

In general, an image falls into this category if removing the image doesn't make the page any less useful, but including the image makes it a lot easier for users of visual browsers to understand the concept.

A flowchart that repeats the previous paragraph in graphical form:

```
<p>The network passes data to the Tokeniser stage, which
passes data to the Tree Construction stage. From there, data goes
to both the DOM and to Script Execution. Script Execution is
linked to the DOM, and, using document.write(), passes data to
the Tokeniser.</p>
<p><img src="images/parsing-model-overview.png" alt=""></p>
```

In these cases, it would be wrong to include alternative text that consists of just a caption. If a caption is to be included, then either the `title` attribute can be used, or the `figure` and `legend` elements can be used. In the latter case, the image would in fact be a phrase or paragraph with an alternative graphical representation, and would thus require alternative text.

```
<!-- Using the title="" attribute -->
<p>The network passes data to the Tokeniser stage, which
passes data to the Tree Construction stage. From there, data goes
to both the DOM and to Script Execution. Script Execution is
linked to the DOM, and, using document.write(), passes data to
the Tokeniser.</p>
<p><img src="images/parsing-model-overview.png" alt=""
        title="Flowchart representation of the parsing model."></p>

<!-- Using <figure> and <legend> -->
<p>The network passes data to the Tokeniser stage, which
passes data to the Tree Construction stage. From there, data goes
to both the DOM and to Script Execution. Script Execution is
linked to the DOM, and, using document.write(), passes data to
the Tokeniser.</p>
<figure>
 <img src="images/parsing-model-overview.png" alt="The Network leads
 to the Tokeniser, which leads to the Tree Construction. The Tree
 Construction leads to two items. The first is Script Execution, which
 leads via document.write() back to the Tokeniser. The second item
 from which Tree Construction leads is the DOM. The DOM is related to
 the Script Execution.">
 <legend>Flowchart representation of the parsing model.</legend>
</figure>

<!-- This is WRONG. Do not do this. Instead, do what the above examples do. -
<p>The network passes data to the Tokeniser stage, which
passes data to the Tree Construction stage. From there, data goes
to both the DOM and to Script Execution. Script Execution is
linked to the DOM, and, using document.write(), passes data to
the Tokeniser.</p>
<p><img src="images/parsing-model-overview.png"
        alt="Flowchart representation of the parsing model."></p>
```

```
<!-- Never put the image's caption in the alt="" attribute! -->
```

A graph that repeats the previous paragraph in graphical form:

```
<p>According to a study covering several billion pages,
about 62% of documents on the Web in 2007 triggered the Quirks
rendering mode of Web browsers, about 30% triggered the Almost
Standards mode, and about 9% triggered the Standards mode.</p>
<p><img src="rendering-mode-pie-chart.png" alt=""></p>
```

4.8.2.1.6 A PURELY DECORATIVE IMAGE THAT DOESN'T ADD ANY INFORMATION

In general, if an image is decorative but isn't especially page-specific, for example an image that forms part of a site-wide design scheme, the image should be specified in the site's CSS, not in the markup of the document.

However, a decorative image that isn't discussed by the surrounding text still has some relevance can be included in a page using the `img` element. Such images are decorative, but still form part of the content. In these cases, the `alt` attribute must be present but its value must be the empty string.

Examples where the image is purely decorative despite being relevant would include things like a photo of the Black Rock City landscape in a blog post about an event at Burning Man, or an image of a painting inspired by a poem, on a page reciting that poem. The following snippet shows an example of the latter case (only the first verse is included in this snippet):

```
<h1>The Lady of Shalott</h1>
<p><img src="shalott.jpeg" alt=""></p>
<p>On either side the river lie<br>
Long fields of barley and of rye,<br>
That clothe the wold and meet the sky;<br>
And through the field the road run by<br>
To many-tower'd Camelot;<br>
And up and down the people go,<br>
Gazing where the lilies blow<br>
Round an island there below,<br>
The island of Shalott.</p>
```

4.8.2.1.7 A GROUP OF IMAGES THAT FORM A SINGLE LARGER PICTURE WITH NO LINKS

When a picture has been sliced into smaller image files that are then displayed together to form the complete picture again, one of the images must have its `alt` attribute set as per the relevant rules that would be appropriate for the picture as a whole, and then all the remaining images must have their `alt` attribute set to the empty string.

In the following example, a picture representing a company logo for XYZ Corp has been split into two pieces, the first containing the letters "XYZ" and the second with the word "Corp". The alternative text ("XYZ Corp") is all in the first image.

```
<h1><img src="logo1.png" alt="XYZ Corp"><img src="logo2.png" alt=""></h1>
```

In the following example, a rating is shown as three filled stars and two empty stars. While the alternative text could have been "★★★☆☆", the author has instead decided to more helpfully give the rating in the form "3 out of 5". That is the alternative text of

the first image, and the rest have blank alternative text.

```
<p>Rating: <meter max=5 value=3><img src="1" alt="3 out of 5"
  ><img src="1" alt=""><img src="1" alt=""><img src="0" alt=""
  ><img src="0" alt=""></meter></p>
```

4.8.2.1.8 A GROUP OF IMAGES THAT FORM A SINGLE LARGER PICTURE WITH LINKS

Generally, image maps should be used instead of slicing an image for links.

However, if an image is indeed sliced and any of the components of the sliced picture are the sole contents of links, then one image per link must have alternative text in its `alt` attribute representing the purpose of the link.

In the following example, a picture representing the flying spaghetti monster emblem, with each of the left noodly appendages and the right noodly appendages in different images, so that the user can pick the left side or the right side in an adventure.

```
<h1>The Church</h1>
<p>You come across a flying spaghetti monster. Which side of His
Noodliness do you wish to reach out for?</p>
<p><a href="?go=left" ><img src="fsm-left.png"  alt="Left side. "></a
  ><img src="fsm-middle.png" alt=""
  ><a href="?go=right"><img src="fsm-right.png" alt="Right side."></a></p>
```

4.8.2.1.9 A KEY PART OF THE CONTENT

In some cases, the image is a critical part of the content. This could be the case, for instance, on a page that is part of a photo gallery. The image is the whole *point* of the page containing it.

How to provide alternative text for an image that is a key part of the content depends on the image's provenance.

The general case

When it is possible for detailed alternative text to be provided, for example if the image is part of a series of screenshots in a magazine review, or part of a comic strip, or is a photograph in a blog entry about that photograph, text that can serve as a substitute for the image must be given as the contents of the `alt` attribute.

A screenshot in a gallery of screenshots for a new OS, with some alternative text:

```
<figure>
 <img src="KDE%20Light%20desktop.png"
      alt="The desktop is blue, with icons along the left hand side in
           two columns, reading System, Home, K-Mail, etc. A window is
           open showing that menus wrap to a second line if they
           cannot fit in the window. The window has a list of icons
           along the top, with an address bar below it, a list of
           icons for tabs along the left edge, a status bar on the
           bottom, and two panes in the middle. The desktop has a bar
           at the bottom of the screen with a few buttons, a pager, a
           list of open applications, and a clock.">
 <legend>Screenshot of a KDE desktop.</legend>
</figure>
```

A graph in a financial report:

```
<img src="sales.gif"
     title="Sales graph"
     alt="From 1998 to 2005, sales increased by the following percentage
     with each year: 624%, 75%, 138%, 40%, 35%, 9%, 21%">
```

Note that "sales graph" would be inadequate alternative text for a sales graph. Text that would be a good *caption* is not generally suitable as replacement text.

Images that defy a complete description

In certain cases, the nature of the image might be such that providing thorough alternative text is impractical. For example, the image could be indistinct, or could be a complex fractal, or could be a detailed topographical map.

In these cases, the `alt` attribute must contain some suitable alternative text, but it may be somewhat brief.

Sometimes there simply is no text that can do justice to an image. For example, there is little that can be said to usefully describe a Rorschach inkblot test. However, a description, even if brief, is still better than nothing:

```
<figure>
 <img src="/commons/a/a7/Rorschach1.jpg" alt="A shape with left-right
 symmetry with indistinct edges, with a small gap in the center, two
 larger gaps offset slightly from the center, with two similar gaps
 under them. The outline is wider in the top half than the bottom
 half, with the sides extending upwards higher than the center, and
 the center extending below the sides.">
 <legend>A black outline of the first of the ten cards
 in the Rorschach inkblot test.</legend>
</figure>
```

Note that the following would be a very bad use of alternative text:

```
<!-- This example is wrong. Do not copy it. -->
<figure>
 <img src="/commons/a/a7/Rorschach1.jpg" alt="A black outline
 of the first of the ten cards in the Rorschach inkblot test.">
 <legend>A black outline of the first of the ten cards
 in the Rorschach inkblot test.</legend>
</figure>
```

Including the caption in the alternative text like this isn't useful because it effectively duplicates the caption for users who don't have images, taunting them twice yet not helping them any more than if they had only read or heard the caption once.

Another example of an image that defies full description is a fractal, which, by definition, is infinite in complexity.

The following example shows one possible way of providing alternative text for the full view of an image of the Mandelbrot set.

```
<img src="ms1.jpeg" alt="The Mandelbrot set appears as a cardioid with
its cusp on the real axis in the positive direction, with a smaller
bulb aligned along the same center line, touching it in the negative
direction, and with these two shapes being surrounded by smaller bulbs
of various sizes.">
```

Images whose contents are not known

In some unfortunate cases, there might be no alternative text available at all, either because the image is obtained in some automated fashion without any associated alternative text (e.g. a Webcam), or because the page is being generated by a script using user-provided images where the user did not provide suitable or usable alternative text (e.g. photograph sharing sites), or because the author does not himself know what the images represent (e.g. a blind photographer sharing an image on his blog).

In such cases, the `alt` attribute's value may be omitted, but one of the following conditions must be met as well:

- The `title` attribute is present and has a non-empty value.
- The `img` element is in a `figure` element that contains a `legend` element that contains content other than inter-element whitespace.
- The `img` element is part of the only paragraph directly in its section, and is the only `img` element without an `alt` attribute in its section, and its section has an associated heading.

Note: Such cases are to be kept to an absolute minimum. If there is even the slightest possibility of the author having the ability to provide real alternative text, then it would not be acceptable to omit the `alt` attribute.

> A photo on a photo-sharing site, if the site received the image with no metadata other than the caption:
>
> ```
> <figure>
>  <img src="1100670787_6a7c664aef.jpg">
>  <legend>Bubbles traveled everywhere with us.</legend>
> </figure>
> ```
>
> It could also be marked up like this:
>
> ```
> <article>
>  <h1>Bubbles traveled everywhere with us.</h1>
>  <img src="1100670787_6a7c664aef.jpg">
> </article>
> ```
>
> In either case, though, it would be better if a detailed description of the important parts of the image obtained from the user and included on the page.

> A blind user's blog in which a photo taken by the user is shown. Initially, the user might not have any idea what the photo he took shows:
>
> ```
> <article>
>  <h1>I took a photo</h1>
>  I went out today and took a photo!
>  <figure>
>   <img src="photo2.jpeg">
>   <legend>A photograph taken blindly from my front porch.</legend>
>  </figure>
> </article>
> ```
>
> Eventually though, the user might obtain a description of the image from his friends and could then include alternative text:
>
> ```
> <article>
> ```

```
 <h1>I took a photo</h1>
 <p>I went out today and took a photo!</p>
 <figure>
  <img src="photo2.jpeg" alt="The photograph shows my hummingbird
  feeder hanging from the edge of my roof. It is half full, but there
  are no birds around. In the background, out-of-focus trees fill the
  shot. The feeder is made of wood with a metal grate, and it contains
  peanuts. The edge of the roof is wooden too, and is painted white
  with light blue streaks.">
  <legend>A photograph taken blindly from my front porch.</legend>
 </figure>
</article>
```

Sometimes the entire point of the image is that a textual description is not available, and the user is to provide the description. For instance, the point of a CAPTCHA image is to see if the user can literally read the graphic. Here is one way to mark up a CAPTCHA (note the `title` attribute):

```
<p><label>What does this image say?
<img src="captcha.cgi?id=8934" title="CAPTCHA">
<input type=text name=captcha></label>
(If you cannot see the image, you can use an <a
href="?audio">audio</a> test instead.)</p>
```

Another example would be software that displays images and asks for alternative text precisely for the purpose of then writing a page with correct alternative text. Such a page could have a table of images, like this:

```
<table>
 <thead>
  <tr> <th> Image <th> Description
 <tbody>
  <tr>
   <td> <img src="2421.png" title="Image 640 by 100, filename 'banner.gi
   <td> <input name="alt2421">
  <tr>
   <td> <img src="2422.png" title="Image 200 by 480, filename 'ad3.gif'"
   <td> <input name="alt2422">
</table>
```

Notice that even in this example, as much useful information as possible is still included in the `title` attribute.

Note: Since some users cannot use images at all (e.g. because they have a very slow connection, or because they are using a text-only browser, or because they are listening to the page being read out by a hands-free automobile voice Web browser, or simply because they are blind), the `alt` attribute is only allowed to be omitted rather than being provided with replacement text when no alternative text is available and none can be made available, as in the above examples. Lack of effort from the part of the author is not an acceptable reason for omitting the `alt` attribute.

4.8.2.1.10 An image not intended for the user

Generally authors should avoid using `img` elements for purposes other than showing images.

If an `img` element is being used for purposes other than showing an image, e.g. as part of a service to count page views, then the `alt` attribute must be the empty string.

4.8.2.1.11 An image in an e-mail or document intended for a specific person who is known to be able to view images

When an image is included in a communication (such as an HTML e-mail) aimed at someone who is known to be able to view images, the `alt` attribute may be omitted. However, even in such cases it is strongly recommended that alternative text be included (as appropriate according to the kind of image involved, as described in the above entries), so that the e-mail is still usable should the user use a mail client that does not support images, or should the e-mail be forwarded on to other users whose abilities might not include easily seeing images.

4.8.2.1.12 General guidelines

The most general rule for writing alternative text is that the intent is that replacing every image with the text of its `alt` attribute not change the meaning of the page.

So, in general, alternative text can be written by considering what one would have written had one not been able to include the image.

A corollary to this is that the `alt` attribute's value should never contain text that could be considered the image's *caption*, *title*, or *legend*. It is supposed to contain replacement text that could be used by users *instead* of the image; it is not meant to supplement the image. The `title` attribute can be used for supplemental information.

> ***Note: One way to think of alternative text is to think about how you would read the page containing the image to someone over the phone, without mentioning that there is an image present. Whatever you say instead of the image is typically a good start for writing the alternative text.***

4.8.2.1.13 Guidance for markup generators

Markup generators (such as WYSIWYG authoring tools) should, wherever possible, obtain alternative text from their users. However, it is recognised that in many cases, this will not be possible.

For images that are the sole contents of links, markup generators should examine the link target to determine the title of the target, or the URL of the target, and use information obtained in this manner as the alternative text.

As a last resort, implementors should either set the `alt` attribute to the empty string, under the assumption that the image is a purely decorative image that doesn't add any information but is still specific to the surrounding content, or omit the `alt` attribute altogther, under the assumption that the image is a key part of the content.

Markup generators should generally avoid using the image's own file name as the alternative text.

4.8.3 The `iframe` element

Categories

Flow content.
Phrasing content.
Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Text that conforms to the requirements given in the prose.

Content attributes:

Global attributes
`src`
`name`
`sandbox`
`seamless`
`width`
`height`

DOM interface:

```
interface HTMLIFrameElement : HTMLElement {
           attribute DOMString src;
           attribute DOMString name;
           attribute DOMString sandbox;
           attribute boolean seamless;
           attribute DOMString width;
           attribute DOMString height;
};
```

Objects implementing the `HTMLIFrameElement` interface must also implement the `EmbeddingElement` interface defined in the Window Object specification. [WINDOW]

The `iframe` element represents a nested browsing context.

The **`src`** attribute gives the address of a page that the nested browsing context is to contain. The attribute, if present, must be a valid URL. When the browsing context is created, if the attribute is present, the user agent must resolve the value of that attribute, relative to the element, and if that is successful, must then navigate the element's browsing context to the resulting absolute URL, with replacement enabled, and with the `iframe` element's document's browsing context as the source browsing context. If the user navigates away from this page, the `iframe`'s corresponding `WindowProxy` object will proxy new `Window` objects for new `Document` objects, but the `src` attribute will not change.

Whenever the `src` attribute is set, the user agent must resolve the value of that attribute, relative to the element, and if that is successful, the nested browsing context must be navigated to the resulting absolute URL, with the `iframe` element's document's browsing context as the source browsing context.

If the `src` attribute is not set when the element is created, or if its value cannot be resolved, the browsing context will remain at the initial `about:blank` page.

The **`name`** attribute, if present, must be a valid browsing context name. The given value is used to name the nested browsing context. When the browsing context is created, if the attribute is present, the browsing context name must be set to the value of this attribute; otherwise, the browsing context name must be set to the empty string.

Whenever the `name` attribute is set, the nested browsing context's name must be changed to the new value. If the attribute is removed, the browsing context name must be set to the empty string.

When content loads in an `iframe`, after any `load` events are fired within the content itself, the user agent must fire a `load` event at the `iframe` element. When content fails to load (e.g. due to a network error), then the user agent must fire an `error` event at the element instead.

When there is an active parser in the `iframe`, and when anything in the `iframe` that is delaying the `load` event in the `iframe`'s browsing context, the `iframe` must delay the `load` event.

> ***Note: If, during the handling of the `load` event, the browsing context in the `iframe` is again navigated, that will further delay the `load` event.***

The **`sandbox`** attribute, when specified, enables a set of extra restrictions on any content hosted by the `iframe`. Its value must be an unordered set of unique space-separated tokens. The allowed values are `allow-same-origin`, `allow-forms`, and `allow-scripts`. When the attribute is set, the content is treated as being from a unique origin, forms and scripts are disabled, links are prevented from targetting other browsing contexts, and plugins are disabled. The `allow-same-origin` token allows the content to be treated as being from the same origin instead of forcing it into a unique origin, and the `allow-forms` and `allow-scripts` tokens re-enable forms and scripts respectively (though scripts are still prevented from creating popups).

While the `sandbox` attribute is specified, the `iframe` element's nested browsing context, and all the browsing contexts nested within it (either directly or indirectly through other nested browsing contexts) must have the following flags set:

The *sandboxed navigation browsing context flag*

This flag prevents content from navigating browsing contexts other than the sandboxed browsing context itself (or browsing contexts further nested inside it).

This flag also prevents content from creating new auxiliary browsing contexts, e.g. using the `target` attribute or the `window.open()` method.

The *sandboxed plugins browsing context flag*

This flag prevents content from instantiating plugins, whether using the `embed` element, the `object` element, the `applet` element, or through navigation of a nested browsing context.

The *sandboxed origin browsing context flag*, unless the `sandbox` attribute's value, when split on spaces, is found to have the `allow-same-origin` keyword set

This flag forces content into a unique origin for the purposes of the same-origin policy.

This flag also prevents script from reading the `document.cookie` DOM attribute.

> ***The `allow-same-origin` attribute is intended for two cases.***
>
> ***First, it can be used to allow content from the same site to be sandboxed to disable scripting, while still allowing access to the DOM of the***

sandboxed content.

Second, it can be used to embed content from a third-party site, sandboxed to prevent that site from opening popup windows, etc, without preventing the embedded page from communicating back to its originating site, using the database APIs to store data, etc.

?Warning! ***This flag only takes effect when the nested browsing context of the `iframe` is navigated.***

The *sandboxed forms browsing context flag*, unless the `sandbox` attribute's value, when split on spaces, is found to have the `allow-forms` keyword set

This flag blocks form submission.

The *sandboxed scripts browsing context flag*, unless the `sandbox` attribute's value, when split on spaces, is found to have the `allow-scripts` keyword set

This flag blocks script execution.

?Warning! ***If the `sandbox` attribute is dynamically added after the `iframe` has loaded a page, scripts already compiled by that page (whether in `script` elements, or in event handler attributes, or elsewhere) will continue to run. Only new scripts will be prevented from executing by this flag.***

These flags must not be set unless the conditions listed above define them as being set.

> In this example, some completely-unknown, potentially hostile, user-provided HTML content is embedded in a page. Because it is sandboxed, it is treated by the user agent as being from a unique origin, despite the content being served from the same site. Thus it is affected by all the normal cross-site restrictions. In addition, the embedded page has scripting disabled, plugins disabled, forms disabled, and it cannot navigate any frames or windows other than itself (or any frames or windows it itself embeds).
>
> ```
> We're not scared of you! Here is your content, unedited:
> <iframe sandbox src="getusercontent.cgi?id=12193"></iframe>
> ```
>
> Note that cookies are still send to the server in the `getusercontent.cgi` request, though they are not visible in the `document.cookie` DOM attribute.

> In this example, a gadget from another site is embedded. The gadget has scripting and forms enabled, and the origin sandbox restrictions are lifted, allowing the gadget to communicate with its originating server. The sandbox is still useful, however, as it disables plugins and popups, thus reducing the risk of the user being exposed to malware and other annoyances.
>
> ```
> <iframe sandbox="allow-same-origin allow-forms allow-scripts"
>         src="http://maps.example.com/embedded.html"></iframe>
> ```

The **`seamless`** attribute is a boolean attribute. When specified, it indicates that the `iframe` element's browsing context is to be rendered in a manner that makes it appear to be part of the containing document (seamlessly included in the parent document). Specifically, when the attribute is set on an element and while the browsing context's active document has the same origin as the `iframe` element's document, or the browsing context's active document's *address* has the same origin as the `iframe` element's document, the following requirements apply:

- The user agent must set the **seamless browsing context flag** to true for that browsing context. This will cause links to open in the parent browsing context.

- In a CSS-supporting user agent: the user agent must add all the style sheets that apply to the `iframe` element to the cascade of the active document of the `iframe` element's nested browsing context, at the appropriate cascade levels, before any style sheets specified by the document itself.

- In a CSS-supporting user agent: the user agent must, for the purpose of CSS property inheritance only, treat the root element of the active document of the `iframe` element's nested browsing context as being a child of the `iframe` element. (Thus inherited properties on the root element of the document in the `iframe` will inherit the computed values of those properties on the `iframe` element instead of taking their initial values.)

- In visual media, in a CSS-supporting user agent: the user agent should set the intrinsic width of the `iframe` to the width that the element would have if it was a non-replaced block-level element with 'width: auto'.

- In visual media, in a CSS-supporting user agent: the user agent should set the intrinsic height of the `iframe` to the height of the bounding box around the content rendered in the `iframe` at its current width (as given in the previous bullet point), as it would be if the scrolling position was such that the top of the viewport for the content rendered in the `iframe` was aligned with the origin of that content's canvas.

- In visual media, in a CSS-supporting user agent: the user agent must force the height of the initial containing block of the active document of the nested browsing context of the `iframe` to zero.

  Note: This is intended to get around the otherwise circular dependency of percentage dimensions that depend on the height of the containing block, thus affecting the height of the document's bounding box, thus affecting the height of the viewport, thus affecting the size of the initial containing block.

- In speech media, the user agent should render the nested browsing context without announcing that it is a separate document.

- User agents should, in general, act as if the active document of the `iframe`'s nested browsing context was part of the document that the `iframe` is in.

  > For example if the user agent supports listing all the links in a document, links in "seamlessly" nested documents would be included in that list without being significantly distinguished from links in the document itself.

If the attribute is not specified, or if the origin conditions listed above are not met, then the user agent should render the nested browsing context in a manner that is clearly distinguishable as a separate browsing context, and the seamless browsing context flag must be set to false for that browsing context.

?Warning! ***It is important that user agents recheck the above conditions whenever the active document of the nested browsing context of the `iframe` changes, such that the seamless browsing context flag gets unset if the nested browsing context is navigated***

to another origin.

Note: The attribute can be set or removed dynamically, with the rendering updating in tandem.

> In this example, the site's navigation is embedded using a client-side include using an `iframe`. Any links in the `iframe` will, in new user agents, be automatically opened in the `iframe`'s parent browsing context; for legacy user agents, the site could also include a `base` element with a `target` attribute with the value `_parent`. Similarly, in new user agents the styles of the parent page will be automatically applied to the contents of the frame, but to support legacy user agents authors might wish to include the styles explicitly.
>
> ```
> <nav><iframe seamless src="nav.include.html"></iframe></nav>
> ```

The `iframe` element supports dimension attributes for cases where the embedded content has specific dimensions (e.g. ad units have well-defined dimensions).

An `iframe` element never has fallback content, as it will always create a nested browsing context, regardless of whether the specified initial contents are successfully used.

Descendants of `iframe` elements represent nothing. (In legacy user agents that do not support `iframe` elements, the contents would be parsed as markup that could act as fallback content.)

The content model of `iframe` elements is text, except that the text must be such that ...

anyone have any bright ideas?

Note: The HTML parser treats markup inside `iframe` elements as text.

The DOM attributes **`src`**, **`name`**, **`sandbox`**, and **`seamless`** must reflect the respective content attributes of the same name.

4.8.4 The `embed` element

Categories
Flow content.
Phrasing content.
Embedded content.

Contexts in which this element may be used:
Where embedded content is expected.

Content model:
Empty.

Content attributes:
Global attributes
`src`

`type`
`width`
`height`
Any other attribute that has no namespace (see prose).

DOM interface:

```
interface HTMLEmbedElement : HTMLElement {
           attribute DOMString src;
           attribute DOMString type;
           attribute DOMString width;
           attribute DOMString height;
};
```

Depending on the type of content instantiated by the `embed` element, the node may also support other interfaces.

The `embed` element represents an integration point for an external (typically non-HTML) application or interactive content.

The **`src`** attribute gives the address of the resource being embedded. The attribute, if present, must contain a valid URL.

The **`type`** attribute, if present, gives the MIME type of the plugin to instantiate. The value must be a valid MIME type, optionally with parameters. If both the `type` attribute and the `src` attribute are present, then the `type` attribute must specify the same type as the explicit Content-Type metadata of the resource given by the `src` attribute. [RFC2046]

When the element is created with neither a `src` attribute nor a `type` attribute, and when attributes are removed such that neither attribute is present on the element anymore, any plugins instantiated for the element must be removed, and the `embed` element represents nothing.

When the sandboxed plugins browsing context flag is set on the browsing context for which the `embed` element's document is the active document, then the user agent must render the `embed` element in a manner that conveys that the plugin was disabled. The user agent may offer the user the option to override the sandbox and instantiate the plugin anyway; if the user invokes such an option, the user agent must act as if the sandboxed plugins browsing context flag was not set for the purposes of this element.

?Warning! Plugins are disabled in sandboxed browsing contexts because they might not honor the restrictions imposed by the sandbox (e.g. they might allow scripting even when scripting in the sandbox is disabled). User agents should convey the danger of overriding the sandbox to the user if an option to do so is provided.

When the element is created with a `src` attribute, and whenever the `src` attribute is subsequently set, and whenever the `type` attribute is set or removed while the element has a `src` attribute, if the element is not in a sandboxed browsing context, user agents must resolve the value of the attribute, relative to the element, and if that is successful, should fetch the resulting absolute URL, find and instantiate an appropriate plugin based on the content's type, and hand that plugin the content of the resource, replacing any previously instantiated plugin for the element.

Fetching the resource must delay the `load` event.

The **type of the content** being embedded is defined as follows:

1. If the element has a `type` attribute, and that attribute's value is a type that a plugin supports, then the value of the `type` attribute is the content's type.

2. Otherwise, if the <path> component of the URL of the specified resource matches a pattern that a plugin supports, then the content's type is the type that that plugin can handle.

   > For example, a plugin might say that it can handle resources with <path> components that end with the four character string ".swf".

   It would be better if browsers didn't do extension sniffing like this, and only based their decision on the actual contents of the resource. Couldn't we just apply the sniffed type of a resource steps?

3. Otherwise, if the specified resource has explicit Content-Type metadata, then that is the content's type.

4. Otherwise, the content has no type and there can be no appropriate plugin for it.

Whether the resource is fetched successfully or not (e.g. whether the response code was a 2xx code or equivalent) must be ignored when determining the resource's type and when handing the resource to the plugin.

> ***Note: This allows servers to return data for plugins even with error responses (e.g. HTTP 500 Internal Server Error codes can still contain plugin data).***

When the element is created with a `type` attribute and no `src` attribute, and whenever the `type` attribute is subsequently set, so long as no `src` attribute is set, and whenever the `src` attribute is removed when the element has a `type` attribute, if the element is not in a sandboxed browsing context, user agents should find and instantiate an appropriate plugin based on the value of the `type` attribute.

Any (namespace-less) attribute may be specified on the `embed` element, so long as its name is XML-compatible and contains no characters in the range U+0041 .. U+005A (LATIN CAPITAL LETTER A LATIN CAPITAL LETTER Z).

> ***Note: All attributes in HTML documents get lowercased automatically, so the restriction on uppercase letters doesn't affect such documents.***

The user agent should pass the names and values of all the attributes of the `embed` element that have no namespace to the plugin used, when it is instantiated.

If the plugin instantiated for the `embed` element supports a scriptable interface, the `HTMLEmbedElement` object representing the element should expose that interface while the element is instantiated.

The `embed` element has no fallback content. If the user agent can't find a suitable plugin, then the user agent must use a default plugin. (This default could be as simple as saying "Unsupported Format".)

The `embed` element supports dimension attributes.

The DOM attributes **`src`** and **`type`** each must reflect the respective content attributes of the same name.

4.8.5 The `object` element

Categories

Flow content.
Phrasing content.
Embedded content.
Listed, submittable, form-associated element.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Zero or more `param` elements, then, transparent.

Content attributes:

Global attributes
`data`
`type`
`name`
`usemap`
`form`
`width`
`height`

DOM interface:

```
interface HTMLObjectElement : HTMLElement {
           attribute DOMString data;
           attribute DOMString type;
           attribute DOMString name;
           attribute DOMString useMap;
  readonly attribute HTMLFormElement form;
           attribute DOMString width;
           attribute DOMString height;
};
```

Objects implementing the `HTMLObjectElement` interface must also implement the `EmbeddingElement` interface defined in the Window Object specification. [WINDOW]

Depending on the type of content instantiated by the `object` element, the node may also support other interfaces.

The `object` element can represent an external resource, which, depending on the type of the resource, will either be treated as an image, as a nested browsing context, or as an external resource to be processed by a plugin.

The **`data`** attribute, if present, specifies the address of the resource. If present, the attribute must be a valid URL.

The **`type`** attribute, if present, specifies the type of the resource. If present, the attribute must

be a valid MIME type, optionally with parameters. [RFC2046]

One or both of the `data` and `type` attributes must be present.

The **`name`** attribute, if present, must be a valid browsing context name. The given value is used to name the nested browsing context, if applicable.

When the element is created, and subsequently whenever the `classid` attribute changes or is removed, or, if the `classid` attribute is not present, whenever the `data` attribute changes or is removed, or, if neither `classid` attribute nor the `data` attribute are present, whenever the `type` attribute changes or is removed, the user agent must run the following steps to determine what the `object` element represents:

1. If the element has an ancestor `object` element that is *not* showing its fallback content, then jump to the last step in the overall set of steps (fallback).

2. If the `classid` attribute is present, and has a value that isn't the empty string, then: if the user agent can find a plugin suitable according to the value of the `classid` attribute, and plugins aren't being sandboxed, then that plugin should be used, and the value of the `data` attribute, if any, should be passed to the plugin. If no suitable plugin can be found, or if the plugin reports an error, jump to the last step in the overall set of steps (fallback).

3. If the `data` attribute is present, then:

    1. If the `type` attribute is present and its value is not a type that the user agent supports, and is not a type that the user agent can find a plugin for, then the user agent may jump to the last step in the overall set of steps (fallback) without fetching the content to examine its real type.

    2. Resolve the URL specified by the `data` attribute, relative to the element.

        If that is successful, fetch the resulting absolute URL.

        The fetching of the resource must delay the `load` event.

    3. If the resource is not yet available (e.g. because the resource was not available in the cache, so that loading the resource required making a request over the network), then jump to the last step in the overall set of steps (fallback). When the resource becomes available, or if the load fails, restart this algorithm from this step. Resources can load incrementally; user agents may opt to consider a resource "available" whenever enough data has been obtained to begin processing the resource.

    4. If the load failed (e.g. the URL could not be resolved, there was an HTTP 404 error, there was a DNS error), fire an `error` event at the element, then jump to the last step in the overall set of steps (fallback).

    5. Determine the *resource type*, as follows:

        1. Let the *resource type* be unknown.

        2. If the resource has associated Content-Type metadata, then let the *resource type* be the type specified in the resource's Content-Type metadata.

3. If the *resource type* is unknown or "`application/octet-stream`" and there is a `type` attribute present on the `object` element, then change the *resource type* to instead be the type specified in that `type` attribute.

   Otherwise, if the *resource type* is "`application/octet-stream`" but there is no `type` attribute on the `object` element, then change the *resource type* to be unknown, so that the sniffing rules in the next step are invoked.

4. If the *resource type* is still unknown, then change the *resource type* to instead be the sniffed type of the resource.

6. Handle the content as given by the first of the following cases that matches:

↪ **If the *resource type* can be handled by a plugin and plugins aren't being sandboxed**

The user agent should use that plugin and pass the content of the resource to that plugin. If the plugin reports an error, then jump to the last step in the overall set of steps (fallback).

↪ **If the *resource type* is an XML MIME type**

↪ **If the *resource type* is HTML**

↪ **If the *resource type* does not start with "`image/`"**

The `object` element must be associated with a nested browsing context, if it does not already have one. The element's nested browsing context must then be navigated to the given resource, with replacement enabled, and with the `object` element's document's browsing context as the source browsing context. (The `data` attribute of the `object` element doesn't get updated if the browsing context gets further navigated to other locations.)

The `object` element represents the nested browsing context.

If the `name` attribute is present, the browsing context name must be set to the value of this attribute; otherwise, the browsing context name must be set to the empty string.

navigation might end up treating it as something else, because it can do sniffing. how should we handle that? it could also refetch the resource entirely, maybe from another application cache.

↪ **If the *resource type* starts with "`image/`", and support for images has not been disabled**

Apply the image sniffing rules to determine the type of the image.

The `object` element represents the specified image. The image is not a nested browsing context.

If the image cannot be rendered, e.g. because it is malformed or in an unsupported format, jump to the last step in the overall set of steps (fallback).

↪ **Otherwise**

The given *resource type* is not supported. Jump to the last step in the

overall set of steps (fallback).

7. The element's contents are not part of what the `object` element represents.

8. Once the resource is completely loaded, fire a `load` event at the element.

4. If the `data` attribute is absent but the `type` attribute is present, plugins aren't being sandboxed, and the user agent can find a plugin suitable according to the value of the `type` attribute, then that plugin should be used. If no suitable plugin can be found, or if the plugin reports an error, jump to the next step (fallback).

5. (Fallback.) The `object` element represents the element's children, ignoring any leading `param` element children. This is the element's fallback content.

When the algorithm above instantiates a plugin, the user agent should pass the names and values of all the attributes on the element, and all the names and values of parameters given by `param` elements that are children of the `object` element, in tree order, to the plugin used. If the plugin supports a scriptable interface, the `HTMLObjectElement` object representing the element should expose that interface. The `object` element represents the plugin. The plugin is not a nested browsing context.

If the sandboxed plugins browsing context flag is set on the browsing context for which the `object` element's document is the active document, then the steps above must always act as if they had failed to find a plugin, even if one would otherwise have been used.

Due to the algorithm above, the contents of `object` elements act as fallback content, used only when referenced resources can't be shown (e.g. because it returned a 404 error). This allows multiple `object` elements to be nested inside each other, targeting multiple user agents with different capabilities, with the user agent picking the first one it supports.

Whenever the `name` attribute is set, if the `object` element has a nested browsing context, its name must be changed to the new value. If the attribute is removed, if the `object` element has a browsing context, the browsing context name must be set to the empty string.

The `usemap` attribute, if present while the `object` element represents an image, can indicate that the object has an associated image map. The attribute must be ignored if the `object` element doesn't represent an image.

The `form` attribute is used to explicitly associate the `object` element with its form owner.

Constraint validation: `object` elements are always barred from constraint validation.

The `object` element supports dimension attributes.

The DOM attributes **`data`**, **`type`**, **`name`**, and **`useMap`** each must reflect the respective content attributes of the same name.

> In the following example, a Java applet is embedded in a page using the `object` element. (Generally speaking, it is better to avoid using applets like these and instead use native JavaScript and HTML to provide the functionality, since that way the application will work on all Web browsers without requiring a third-party plugin. Many devices, especially embedded devices, do not support third-party technologies like Java.)

```
<figure>
 <object type="application/x-java-applet">
  <param name="code" value="MyJavaClass">
  <p>You do not have Java available, or it is disabled.</p>
 </object>
 <legend>My Java Clock</legend>
</figure>
```

In this example, an HTML page is embedded in another using the `object` element.

```
<figure>
 <object data="clock.html"></object>
 <legend>My HTML Clock</legend>
</figure>
```

4.8.6 The `param` element

Categories

None.

Contexts in which this element may be used:

As a child of an `object` element, before any flow content.

Content model:

Empty.

Content attributes:

Global attributes
`name`
`value`

DOM interface:

```
interface HTMLParamElement : HTMLElement {
           attribute DOMString name;
           attribute DOMString value;
};
```

The `param` element defines parameters for plugins invoked by `object` elements. It does not represent anything on its own.

The **`name`** attribute gives the name of the parameter.

The **`value`** attribute gives the value of the parameter.

Both attributes must be present. They may have any value.

If both attributes are present, and if the parent element of the `param` is an `object` element, then the element defines a **parameter** with the given name/value pair.

The DOM attributes **`name`** and **`value`** must both reflect the respective content attributes of the same name.

4.8.7 The `video` element

Categories

Flow content.
Phrasing content.
Embedded content.
If the element has a `controls` attribute: Interactive content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

If the element has a `src` attribute: transparent.
If the element does not have a `src` attribute: one or more `source` elements, then, transparent.

Content attributes:

Global attributes
`src`
`poster`
`autobuffer`
`autoplay`
`loop`
`controls`
`width`
`height`

DOM interface:

```
interface HTMLVideoElement : HTMLMediaElement {
           attribute DOMString width;
           attribute DOMString height;
  readonly attribute unsigned long videoWidth;
  readonly attribute unsigned long videoHeight;
           attribute DOMString poster;
};
```

A `video` element represents a video or movie.

Content may be provided inside the `video` element. User agents should not show this content to the user; it is intended for older Web browsers which do not support `video`, so that legacy video plugins can be tried, or to show text to the users of these older browser informing them of how to access the video contents.

Note: In particular, this content is not fallback content intended to address accessibility concerns. To make video content accessible to the blind, deaf, and those with other physical or cognitive disabilities, authors are expected to provide alternative media streams and/or to embed accessibility aids (such as caption or subtitle tracks) into their media streams.

The `video` element is a media element whose media data is ostensibly video data, possibly with associated audio data.

The `src`, `autobuffer`, `autoplay`, `loop`, and `controls` attributes are the attributes common to all media elements.

The **`poster`** attribute gives the address of an image file that the user agent can show while no video data is available. The attribute, if present, must contain a valid URL. If the specified resource is to be used, then, when the element is created or when the `poster` attribute is set,

its value must be resolved relative to the element, and if that is successful, the resulting absolute URL must be fetched; this must delay the load event. The **poster frame** is then the image obtained from that resource, if any.

> ***Note: The image given by the `poster` attribute, the poster frame, is intended to be a representative frame of the video (typically one of the first non-blank frames) that gives the user an idea of what the video is like.***

The **`poster`** DOM attribute must reflect the `poster` content attribute.

When no video data is available (the element's `readyState` attribute is either `HAVE_NOTHING`, or `HAVE_METADATA` but no video data has yet been obtained at all), the `video` element represents either the poster frame, or nothing.

When a `video` element is paused and the current playback position is the first frame of video, the element represents either the frame of video corresponding to the current playback position or the poster frame, at the discretion of the user agent.

Notwithstanding the above, the poster frame should be preferred over nothing, but the poster frame should not be shown again after a frame of video has been shown.

When a `video` element is paused at any other position, the element represents the frame of video corresponding to the current playback position, or, if that is not yet available (e.g. because the video is seeking or buffering), the last frame of the video to have been rendered.

When a `video` element is potentially playing, it represents the frame of video at the continuously increasing "current" position. When the current playback position changes such that the last frame rendered is no longer the frame corresponding to the current playback position in the video, the new frame must be rendered. Similarly, any audio associated with the video must, if played, be played synchronized with the current playback position, at the specified volume with the specified mute state.

When a `video` element is neither potentially playing nor paused (e.g. when seeking or stalled), the element represents the last frame of the video to have been rendered.

> ***Note: Which frame in a video stream corresponds to a particular playback position is defined by the video stream's format.***

In addition to the above, the user agent may provide messages to the user (such as "buffering", "no video loaded", "error", or more detailed information) by overlaying text or icons on the video or other areas of the element's playback area, or in another appropriate manner.

User agents that cannot render the video may instead make the element represent a link to an external video playback utility or to the video data itself.

video **`.videoWidth`**
video **`.videoHeight`**

These attributes return the intrinsic dimensions of the video, or zero if the

dimensions are not known.

The **intrinsic width** and **intrinsic height** of the media resource are the dimensions of the resource in CSS pixels after taking into account the resource's dimensions, aspect ratio, clean aperture, resolution, and so forth, as defined for the format used by the resource.

The **`videoWidth`** DOM attribute must return the intrinsic width of the video in CSS pixels. The **`videoHeight`** DOM attribute must return the intrinsic height of the video in CSS pixels. If the element's `readyState` attribute is `HAVE_NOTHING`, then the attributes must return 0.

The `video` element supports dimension attributes.

Video content should be rendered inside the element's playback area such that the video content is shown centered in the playback area at the largest possible size that fits completely within it, with the video content's aspect ratio being preserved. Thus, if the aspect ratio of the playback area does not match the aspect ratio of the video, the video will be shown letterboxed or pillarboxed. Areas of the element's playback area that do not contain the video represent nothing.

The intrinsic width of a `video` element's playback area is the intrinsic width of the video resource, if that is available; otherwise it is the intrinsic width of the poster frame, if that is available; otherwise it is 300 CSS pixels.

The intrinsic height of a `video` element's playback area is the intrinsic height of the video resource, if that is available; otherwise it is the intrinsic height of the poster frame, if that is available; otherwise it is 150 CSS pixels.

User agents should provide controls to enable or disable the display of closed captions associated with the video stream, though such features should, again, not interfere with the page's normal rendering.

User agents may allow users to view the video content in manners more suitable to the user (e.g. full-screen or in an independent resizable window). As for the other user interface features, controls to enable this should not interfere with the page's normal rendering unless the user agent is exposing a user interface. In such an independent context, however, user agents may make full user interfaces visible, with, e.g., play, pause, seeking, and volume controls, even if the `controls` attribute is absent.

User agents may allow video playback to affect system features that could interfere with the user's experience; for example, user agents could disable screensavers while video playback is in progress.

?Warning! User agents should not provide a public API to cause videos to be shown full-screen. A script, combined with a carefully crafted video file, could trick the user into thinking a system-modal dialog had been shown, and prompt the user for a password. There is also the danger of "mere" annoyance, with pages launching full-screen videos when links are clicked or pages navigated. Instead, user-agent specific interface features may be provided to easily allow the user to obtain a full-screen playback mode.

4.8.7.1 Video and audio codecs for `video` elements

User agents may support any video and audio codecs and container formats.

It would be helpful for interoperability if all browsers could support the same codecs. However, there are no known codecs that satisfy all the current players: we need a codec that is known to not require per-unit or per-distributor licensing, that is compatible with the open source development model, that is of sufficient quality as to be usable, and that is not an additional submarine patent risk for large companies. This is an ongoing issue and this section will be updated once more information is available.

Note: Certain user agents might support no codecs at all, e.g. text browsers running over SSH connections.

4.8.8 The `audio` element

Categories

Flow content.
Phrasing content.
Embedded content.
If the element has a `controls` attribute: Interactive content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

If the element has a `src` attribute: transparent.
If the element does not have a `src` attribute: one or more `source` elements, then, transparent.

Content attributes:

Global attributes
`src`
`autobuffer`
`autoplay`
`loop`
`controls`

DOM interface:

```
[NamedConstructor=Audio(),
 NamedConstructor=Audio(in DOMString src)]
interface HTMLAudioElement : HTMLMediaElement {
  // no members
};
```

An `audio` element represents a sound or audio stream.

Content may be provided inside the `audio` element. User agents should not show this content to the user; it is intended for older Web browsers which do not support `audio`, so that legacy audio plugins can be tried, or to show text to the users of these older browser informing them

of how to access the audio contents.

> ***Note: In particular, this content is not fallback content intended to address accessibility concerns. To make audio content accessible to the deaf or to those with other physical or cognitive disabilities, authors are expected to provide alternative media streams and/or to embed accessibility aids (such as transcriptions) into their media streams.***

The `audio` element is a media element whose media data is ostensibly audio data.

The `src`, `autobuffer`, `autoplay`, `loop`, and `controls` attributes are the attributes common to all media elements.

When an `audio` element is potentially playing, it must have its audio data played synchronized with the current playback position, at the specified volume with the specified mute state.

When an `audio` element is not potentially playing, audio must not play for the element.

> ***audio*** **= new `Audio`([*url*])**
>
> Returns a new `audio` element, with the `src` attribute set to the value passed in the argument, if applicable.

Two constructors are provided for creating `HTMLAudioElement` objects (in addition to the factory methods from DOM Core such as `createElement()`): **`Audio()`** and **`Audio(`*`src`*`)`**. When invoked as constructors, these must return a new `HTMLAudioElement` object (a new `audio` element). If the *src* argument is present, the object created must have its `src` content attribute set to the provided value, and the user agent must invoke the object's resource selection algorithm before returning.

4.8.8.1 Audio codecs for `audio` elements

User agents may support any audio codecs and container formats.

User agents must support the WAVE container format with audio encoded using the PCM format.

4.8.9 The `source` element

Categories
None.
Contexts in which this element may be used:
As a child of a media element, before any flow content.
Content model:
Empty.
Content attributes:

Global attributes
src
type
media

DOM interface:

```
interface HTMLSourceElement : HTMLElement {
           attribute DOMString src;
           attribute DOMString type;
           attribute DOMString media;
};
```

The source element allows authors to specify multiple media resources for media elements. It does not represent anything on its own.

The **src** attribute gives the address of the media resource. The value must be a valid URL. This attribute must be present.

The **type** attribute gives the type of the media resource, to help the user agent determine if it can play this media resource before fetching it. If specified, its value must be a MIME type. The codecs parameter may be specified and might be necessary to specify exactly how the resource is encoded. [RFC2046] [RFC4281]

> The following list shows some examples of how to use the codecs= MIME parameter in the type attribute.
>
> **H.264 Simple baseline profile video (main and extended video compatible) level 3 and Low-Complexity AAC audio in MP4 container**
>
> ```
> <source src="video.mp4" type="video/mp4; codecs="avc1.42E01E, mp4a.
> ```
>
> **H.264 Extended profile video (baseline-compatible) level 3 and Low-Complexity AAC audio in MP4 container**
>
> ```
> <source src="video.mp4" type="video/mp4; codecs="avc1.58A01E, mp4a.
> ```
>
> **H.264 Main profile video level 3 and Low-Complexity AAC audio in MP4 container**
>
> ```
> <source src="video.mp4" type="video/mp4; codecs="avc1.4D401E, mp4a.
> ```
>
> **H.264 "High" profile video (incompatible with main, baseline, or extended profiles) level 3 and Low-Complexity AAC audio in MP4 container**
>
> ```
> <source src="video.mp4" type="video/mp4; codecs="avc1.64001E, mp4a.
> ```
>
> **MPEG-4 Visual Simple Profile Level 0 video and Low-Complexity AAC audio in MP4 container**
>
> ```
> <source src="video.mp4" type="video/mp4; codecs="mp4v.20.8, mp4a.40
> ```
>
> **MPEG-4 Advanced Simple Profile Level 0 video and Low-Complexity AAC audio in MP4 container**
>
> ```
> <source src="video.mp4" type="video/mp4; codecs="mp4v.20.240, mp4a.
> ```
>
> **MPEG-4 Visual Simple Profile Level 0 video and AMR audio in 3GPP container**

```
        <source src="video.3gp" type="video/3gpp; codecs=&quot;mp4v.20.8, samr&q
```

Theora video and Vorbis audio in Ogg container

```
        <source src="video.ogv" type="video/ogg; codecs=&quot;theora, vorbis&quo
```

Theora video and Speex audio in Ogg container

```
        <source src="video.ogv" type="video/ogg; codecs=&quot;theora, speex&quot
```

Vorbis audio alone in Ogg container

```
        <source src="audio.ogg" type="audio/ogg; codecs=vorbis">
```

Speex audio alone in Ogg container

```
        <source src="audio.spx" type="audio/ogg; codecs=speex">
```

FLAC audio alone in Ogg container

```
        <source src="audio.oga" type="audio/ogg; codecs=flac">
```

Dirac video and Vorbis audio in Ogg container

```
        <source src="video.ogv" type="video/ogg; codecs=&quot;dirac, vorbis&quot
```

Theora video and Vorbis audio in Matroska container

```
        <source src="video.mkv" type="video/x-matroska; codecs=&quot;theora, vor
```

The **`media`** attribute gives the intended media type of the media resource, to help the user agent determine if this media resource is useful to the user before fetching it. Its value must be a valid media query. [MQ]

The DOM attributes **`src`**, **`type`**, and **`media`** must reflect the respective content attributes of the same name.

4.8.10 Media elements

Media elements implement the following interface:

```
interface HTMLMediaElement : HTMLElement {

  // error state
  readonly attribute MediaError error;

  // network state
           attribute DOMString src;
  readonly attribute DOMString currentSrc;
  const unsigned short NETWORK_EMPTY = 0;
  const unsigned short NETWORK_IDLE = 1;
  const unsigned short NETWORK_LOADING = 2;
  const unsigned short NETWORK_LOADED = 3;
  const unsigned short NETWORK_NO_SOURCE = 4;
  readonly attribute unsigned short networkState;
           attribute boolean autobuffer;

  readonly attribute TimeRanges buffered;
  void load();
  DOMString canPlayType(in DOMString type);

  // ready state
```

```
  const unsigned short HAVE_NOTHING = 0;
  const unsigned short HAVE_METADATA = 1;
  const unsigned short HAVE_CURRENT_DATA = 2;
  const unsigned short HAVE_FUTURE_DATA = 3;
  const unsigned short HAVE_ENOUGH_DATA = 4;
  readonly attribute unsigned short readyState;
  readonly attribute boolean seeking;

  // playback state
           attribute float currentTime;
  readonly attribute float duration;
  readonly attribute boolean paused;
           attribute float defaultPlaybackRate;
           attribute float playbackRate;
  readonly attribute TimeRanges played;
  readonly attribute TimeRanges seekable;
  readonly attribute boolean ended;
           attribute boolean autoplay;
           attribute boolean loop;
  void play();
  void pause();

  // cue ranges
  void addCueRange(in DOMString className, in DOMString id, in float start, in flo
  void removeCueRanges(in DOMString className);

  // controls
           attribute boolean controls;
           attribute float volume;
           attribute boolean muted;
};

[Callback=FunctionOnly, NoInterfaceObject]
interface CueRangeCallback {
  void handleEvent(in DOMString id);
};
```

The **media element attributes**, `src`, `autobuffer`, `autoplay`, `loop`, and `controls`, apply to all media elements. They are defined in this section.

Media elements are used to present audio data, or video and audio data, to the user. This is referred to as **media data** in this section, since this section applies equally to media elements for audio or for video. The term **media resource** is used to refer to the complete set of media data, e.g. the complete video file, or complete audio file.

Unless otherwise specified, the task source for all the tasks queued in this section and its subsections is the **media element event task source**.

4.8.10.1 Error codes

media **`.error`**

Returns a `MediaError` object representing the current error state of the element.

Returns null if there is no error.

All media elements have an associated error status, which records the last error the element encountered since its resource selection algorithm was last invoked. The **`error`** attribute, on

getting, must return the `MediaError` object created for this last error, or null if there has not been an error.

```
interface MediaError {
  const unsigned short MEDIA_ERR_ABORTED = 1;
  const unsigned short MEDIA_ERR_NETWORK = 2;
  const unsigned short MEDIA_ERR_DECODE = 3;
  const unsigned short MEDIA_ERR_NONE_SUPPORTED = 4;
  readonly attribute unsigned short code;
};
```

media **. error . code**

Returns the current error's error code, from the list below.

The **code** attribute of a `MediaError` object must return the code for the error, which must be one of the following:

MEDIA_ERR_ABORTED **(numeric value 1)**
: The fetching process for the media resource was aborted by the user agent at the user's request.

MEDIA_ERR_NETWORK **(numeric value 2)**
: A network error of some description caused the user agent to stop fetching the media resource, after the resource was established to be usable.

MEDIA_ERR_DECODE **(numeric value 3)**
: An error of some description occurred while decoding the media resource, after the resource was established to be usable.

MEDIA_ERR_NONE_SUPPORTED **(numeric value 4)**
: No suitable media resource could be found.

4.8.10.2 Location of the media resource

The **src** content attribute on media elements gives the address of the media resource (video, audio) to show. The attribute, if present, must contain a valid URL.

The **src** DOM attribute on media elements must reflect the respective content attribute of the same name.

media **. currentSrc**

Returns the address of the current media resource.

Returns the empty string when there is no media resource.

The **currentSrc** DOM attribute is initially the empty string. Its value is changed by the resource selection algorithm defined below.

Note: There are two ways to specify a media resource, the `src` attribute, or `source` elements. The attribute overrides the elements.

4.8.10.3 Media types

A media resource can be described in terms of its *type*, specifically a MIME type, optionally with a `codecs` parameter. [RFC2046] [RFC4281].

Types are usually somewhat incomplete descriptions; for example "`video/mpeg`" doesn't say anything except what the container type is, and even a type like "`video/mp4; codecs="avc1.42E01E, mp4a.40.2"`" doesn't include information like the actual bitrate (only the maximum bitrate). Thus, given a type, a user agent can often only know whether it *might* be able to play media of that type (with varying levels of confidence), or whether it definitely *cannot* play media of that type.

A type that the user agent knows it cannot render is one that describes a resource that the user agent definitely does not support, for example because it doesn't recognise the container type, or it doesn't support the listed codecs.

***media*.`canPlayType`(*type*)**

Returns "no", "maybe", or "probably" based on how confident the user agent is that it can play media resources of the given type.

The **`canPlayType(`*type*`)`** method must return the string "`no`" if *type* is a type that the user agent knows it cannot render; it must return "`probably`" if the user agent is confident that the type represents a media resource that it can render if used in with this `audio` or `video` element; and it must return "`maybe`" otherwise. Implementors are encouraged to return "`maybe`" unless the type can be confidently established as being supported or not. Generally, a user agent should never return "`probably`" if the type doesn't have a `codecs` parameter.

This script tests to see if the user agent supports a (fictional) new format to dynamically decide whether to use a `video` element or a plugin:

```
<section id="video">
 <p><a href="playing-cats.nfv">Download video</a></p>
</section>
<script>
 var videoSection = document.getElementById('video');
 var videoElement = document.createElement('video');
 var support = videoElement.canPlayType('video/x-new-fictional-format;codecs=
 if (support != "probably" && "New Fictional Video Plug-in" in navigator.plug
   // not confident of browser support
   // but we have a plugin
   // so use plugin instead
   videoElement = document.createElement("embed");
 } else if (support == "no") {
   // no support from browser and no plugin
   // do nothing
   videoElement = null;
 }
 if (videoElement) {
```

```
        while (videoSection.hasChildNodes())
          videoSection.removeChild(videoSection.firstChild);
        videoElement.setAttribute("src", "playing-cats.nfv");
        videoSection.appendChild(videoElement);
      }
    </script>
```

Note: To express the type of a media resource to allow the user agent to avoid downloading resources it can't render, authors can use the `source` element's `type` attribute.

4.8.10.4 Network states

media **. `networkState`**

Returns the current state of network activity for the element, from the codes in the list below.

As media elements interact with the network, their current network activity is represented by the **`networkState`** attribute. On getting, it must return the current network state of the element, which must be one of the following values:

`NETWORK_EMPTY` **(numeric value 0)**
: The element has not yet been initialized. All attributes are in their initial states.

`NETWORK_IDLE` **(numeric value 1)**
: The element's resource selection algorithm is active and has selected a resource, but it is not actually using the network at this time.

`NETWORK_LOADING` **(numeric value 2)**
: The user agent is actively trying to download data.

`NETWORK_LOADED` **(numeric value 3)**
: The entire media resource has been obtained and is available to the user agent locally. Network connectivity could be lost without affecting the media playback.

`NETWORK_NO_SOURCE` **(numeric value 4)**
: The element's resource selection algorithm is active, but it has failed to find a resource to use.

The resource selection algorithm defined below describes exactly when the `networkState` attribute changes value and what events fire to indicate changes in this state.

Note: Some resources, e.g. streaming Web radio, can never reach the `NETWORK_LOADED` state.

4.8.10.5 Loading the media resource

media **. `load`()**

Causes the element to reset and start selecting and loading a new media resource from scratch.

All media elements have an **autoplaying flag**, which must begin in the true state, and a **delaying-the-load-event flag**, which must begin in the false state. While the delaying-the-load-event flag is true, the element must delay the load event.

When the **`load()`** method on a media element is invoked, the user agent must run the following steps. Note that this algorithm might get aborted, e.g. if the `load()` method itself is invoked again.

1. If the `load()` method for this element is already being invoked, then abort these steps.
2. Abort any already-running instance of the resource selection algorithm for this element.
3. If there are any tasks from the media element's media element event task source in one of the task queues, then remove those tasks.

   Note: Basically, pending events and callbacks for the media element are discarded when the media element starts loading a new resource.

4. If the media element's `networkState` is set to `NETWORK_LOADING` or `NETWORK_IDLE`, set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_ABORTED`, and fire a progress event called `abort` at the media element.
5. Set the `error` attribute to null and the autoplaying flag to true.
6. Set the `playbackRate` attribute to the value of the `defaultPlaybackRate` attribute.
7. If the media element's `networkState` is not set to `NETWORK_EMPTY`, then run these substeps:
   1. If a fetching process is in progress for the media element, the user agent should stop it.
   2. Set the `networkState` attribute to NETWORK_EMPTY.
   3. If `readyState` is not set to `HAVE_NOTHING`, then set it to that state.
   4. If the `paused` attribute is false, then set to true.
   5. If `seeking` is true, set it to false.
   6. Set the current playback position to 0.
   7. Fire a simple event called `emptied` at the media element.
8. Invoke the media element's resource selection algorithm.
9. ***Note: Playback of any previously playing media resource for this element***

stops.

The **resource selection algorithm** for a media element is as follows. This algorithm is always invoked synchronously, but one of the first steps in the algorithm is to return and continue running the remaining steps asynchronously, meaning that it runs in the background with scripts and other tasks running in parallel.

1. If the media element has neither a `src` attribute nor any `source` element children, run these substeps:

    1. Set the `networkState` to `NETWORK_NO_SOURCE`.

    2. Run the remainder of the resource selection algorithm steps asynchronously, allowing the task that invoked this algorithm to continue.

    3. While the media element has neither a `src` attribute nor any `source` element children, wait. (This step might wait forever.)

    4. Before the task that set the `src` attribute or inserted the `source` element has a chance to complete, set the media element's delaying-the-load-event flag to true (this delays the load event), and set its `networkState` to `NETWORK_LOADING`.

    Otherwise, run these substeps:

    1. Set the media element's delaying-the-load-event flag to true (this delays the load event), and set its `networkState` to `NETWORK_LOADING`.

    2. Run the remainder of the resource selection algorithm steps asynchronously, allowing the task that invoked this algorithm to continue.

2. ***Note: By this point, the algorithm is running asynchronously.***

3. Queue a task to fire a progress event called `loadstart` at the media element.

4. If the media element has a `src` attribute, then run these substeps:

    1. Resolve the URL given in that attribute, relative to the media element.

    2. If that is successful, then run the resource fetch algorithm with the resulting absolute URL. If that algorithm returns without aborting *this* one, then the load failed.

    3. Reaching this step indicates that either the URL failed to resolve, or the media resource failed to load. Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_NONE_SUPPORTED`.

    4. Set the element's `networkState` attribute to the NETWORK_NO_SOURCE value.

    5. Queue a task to fire a progress event called `error` at the media element.

    6. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.

7. Abort these steps. Until the `load()` method is invoked, the element won't attempt to load another resource.

Otherwise, the `source` elements will be used; run these substeps:

1. Let *pointer* be a position defined by two adjacent nodes in the media element's child list, treating the start of the list (before the first child in the list, if any) and end of the list (after the last child in the list, if any) as nodes in their own right. One node is the node before *pointer*, and the other node is the node after *pointer*. Initially, let *pointer* be the position between the start of the list and the next node (either the first child node of the media element, if there are any, or the end of the list, if it is empty).

   As elements are inserted and removed into the media element, *pointer* must be updated as follows:

   If a new element is inserted between the two nodes that define *pointer*
   Let *pointer* be the point between the node before *pointer* and the new node. In other words, insertions at *pointer* go after *pointer*.
   If the node before *pointer* is removed
   Let *pointer* be the point between the node after *pointer* and the node before the node after *pointer*. In other words, *pointer* doesn't move relative to the remaining nodes.
   If the node after *pointer* is removed
   Let *pointer* be the point between the node before *pointer* and the node after the node before *pointer*. Just as with the previous case, *pointer* doesn't move relative to the remaining nodes.

   Other changes don't affect *pointer*.

2. *Search loop:* Run these substeps atomically (so that the DOM cannot change while they are running):

   1. If the node after *pointer* is the end of the list, then jump to the step below labeled *waiting*.

   2. If the node after *pointer* is a `source` element, let *candidate* be that element; otherwise, let *candidate* be null.

   3. Advance *pointer* so that the node before *pointer* is now the node that was after *pointer*, and the node after *pointer* is the node after the node that used to be after *pointer*.

   4. If *candidate* is null, restart the step labelled *search loop* from the first substep.

3. If any of the following conditions are true, then jump back to the step labelled *search loop*:

   - The *candidate* element does not have a `src` attribute.

   - Resolving the URL given by the *candidate* element's `src` attribute relative to *candidate* fails.

- The *candidate* element has a `type` attribute whose value, when parsed as a MIME type (including any codecs described by the `codec` parameter), represents a type that the user agent knows it cannot render.

- The *candidate* element has a `media` attribute whose value, when processed according to the rules for media queries, does not match the current environment. [MQ]

4. Set the `networkState` to `NETWORK_LOADING` again, in case it was set to `NETWORK_NO_SOURCE` above.

5. Run the resource fetch algorithm with the absolute URL that resulted from resolving the URL given by the *candidate* element's `src` attribute relative to *candidate*. If that algorithm returns without aborting *this* one, then the load failed.

6. Return to the step labeled *search loop*.

7. *Waiting:* Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_NONE_SUPPORTED`.

8. Set the element's `networkState` attribute to the NETWORK_NO_SOURCE value

9. Queue a task to fire a progress event called `error` at the media element.

10. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.

11. Wait until the node after *pointer* is a node other than the end of the list. (This step might wait forever.)

12. Before the task that inserted the `source` element has a chance to complete, set the element's delaying-the-load-event flag back to true. This delays the load event again, in case it hasn't been fired yet.

13. Jump back to the step labeled *search loop*.

The **resource fetch algorithm** for a media element and a given absolute URL is as follows:

1. Let the *current media resource* be the resource given by the absolute URL passed to this algorithm. This is now the element's media resource.

2. Set the `currentSrc` attribute to the absolute URL of the *current media resource*.

3. Begin to fetch the *current media resource*.

   Every 350ms (±200ms) or for every byte received, whichever is *least* frequent, queue a task to fire a progress event called `progress` at the element.

   If at any point the user agent has received no data for more than about three seconds, then queue a task to fire a progress event called `stalled` at the element.

   User agents may allow users to selectively block or slow media data downloads. When a media element's download has been blocked altogether, the user agent must act as if it was stalled (as opposed to acting as if the connection was closed). The rate of the

download may also be throttled automatically by the user agent, e.g. to balance the download with other connections sharing the same bandwidth.

User agents may decide to not download more content at any time, e.g. after buffering five minutes of a one hour media resource, while waiting for the user to decide whether to play the resource or not, or while waiting for user input in an interactive resource. When a media element's download has been suspended, the user agent must set the `networkState` to `NETWORK_IDLE` and queue a task to fire a progress event called `suspend` at the element. If and when downloading of the resource resumes, the user agent must set the `networkState` to `NETWORK_LOADING`.

The `autobuffer` attribute provides a hint that the author expects that downloading the entire resource optimistically will be worth it, even in the absence of the `autoplay` attribute. In the absence of either attribute, the user agent is likely to find that waiting until the user starts playback before downloading any further content leads to a more efficient use of the network resources.

When a user agent decides to completely stall a download, e.g. if it is waiting until the user starts playback before downloading any further content, the element's delaying-the-load-event flag must be set to false. This stops delaying the load event.

The user agent may use whatever means necessary to fetch the resource (within the constraints put forward by this and other specifications); for example, reconnecting to the server in the face of network errors, using HTTP partial range requests, or switching to a streaming protocol. The user agent must consider a resource erroneous only if it has given up trying to fetch it.

The networking task source tasks to process the data as it is being fetched must, when appropriate, include the relevant substeps from the following list:

↪ **If the media data cannot be fetched at all, due to network errors, causing the user agent to give up trying to fetch the resource**

↪ **If the media data can be fetched but is in an unsupported format, or can otherwise not be rendered at all**

DNS errors, HTTP 4xx and 5xx errors (and equivalents in other protocols), and other fatal network errors that occur before the user agent has established whether the *current media resource* is usable, as well as the file using an unsupported container format, or using unsupported codecs for all the data, must cause the user agent to execute the following steps:

1. The user agent should cancel the fetching process.
2. Abort this subalgorithm, returning to the resource selection algorithm.

↪ **Once enough of the media data has been fetched to determine the duration of the media resource, its dimensions, and other metadata**

This indicates that the resource is usable. The user agent must follow these substeps:

1. Set the current playback position to the *earliest possible position*.
2. Set the `readyState` attribute to `HAVE_METADATA`.
3. For `video` elements, set the `videoWidth` and `videoHeight` attributes.

4. Set the `duration` attribute to the duration of the resource.

Note: The user agent will queue a task to fire a simple event called `durationchange` at the element at this point.

5. Queue a task to fire a simple event called `loadedmetadata` at the element.

6. If either the media resource or the address of the *current media resource* indicate a particular start time, then seek to that time. Ignore any resulting exceptions (if the position is out of range, it is effectively ignored).

> For example, a fragment identifier could be used to indicate a start position.

7. Once the `readyState` attribute reaches `HAVE_CURRENT_DATA`, set the element's delaying-the-load-event flag to false. This stops delaying the load event.

Note: A user agent that is attempting to reduce network usage while still fetching the metadata for each media resource would also stop buffering at this point, causing the `networkState` attribute to switch to the `NETWORK_IDLE` value, if the media element did not have an `autobuffer` or `autoplay` attribute.

Note: The user agent is required to determine the duration of the media resource and go through this step before playing.

↪ **If the connection is interrupted, causing the user agent to give up trying to fetch the resource**

Fatal network errors that occur after the user agent has established whether the *current media resource* is usable must cause the user agent to execute the following steps:

1. The user agent should cancel the fetching process.

2. Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_NETWORK`.

3. Queue a task to fire a progress event called `error` at the media element.

4. Set the element's `networkState` attribute to the NETWORK_EMPTY value and queue a task to fire a simple event called `emptied` at the element.

5. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.

6. Abort the overall resource selection algorithm.

↪ **If the media data is corrupted**

Fatal errors in decoding the media data that occur after the user agent has established whether the *current media resource* is usable must cause the user agent to execute the following steps:

1. The user agent should cancel the fetching process.
2. Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_DECODE`.
3. Queue a task to fire a progress event called `error` at the media element.
4. Set the element's `networkState` attribute to the NETWORK_EMPTY value and queue a task to fire a simple event called `emptied` at the element.
5. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.
6. Abort the overall resource selection algorithm.

↪ **If the media data fetching process is aborted by the user**

The fetching process is aborted by the user, e.g. because the user navigated the browsing context to another page, the user agent must execute the following steps. These steps are not followed if the `load()` method itself is invoked while these steps are running, as the steps above handle that particular kind of abort.

1. The user agent should cancel the fetching process.
2. Set the `error` attribute to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_ABORT`.
3. Queue a task to fire a progress event called `abort` at the media element.
4. If the media element's `readyState` attribute has a value equal to `HAVE_NOTHING`, set the element's `networkState` attribute to the NETWORK_EMPTY value and queue a task to fire a simple event called `emptied` at the element. Otherwise, set set the element's `networkState` attribute to the NETWORK_IDLE value.
5. Set the element's delaying-the-load-event flag to false. This stops delaying the load event.
6. Abort the overall resource selection algorithm.

↪ **If the media data can be fetched but has non-fatal errors or uses, in part, codecs that are unsupported, preventing the user agent from rendering the content completely correctly but not preventing playback altogether**

The server returning data that is partially usable but cannot be optimally rendered must cause the user agent to execute the following steps.

1. Should we fire a 'warning' event? Set the 'error' flag to

'MEDIA_ERR_SUBOPTIMAL' or something?

When the user agent has completely fetched of the entire media resource, it must move on to the next step. This might never happen, e.g. when streaming an infinite resource such as Web radio.

4. If the fetching process completes without errors, then set the `networkState` attribute to `NETWORK_LOADED`, and queue a task to fire a progress event called `load` at the element.

5. Then, abort the overall resource selection algorithm.

If a media element whose `networkState` has the value `NETWORK_EMPTY` is inserted into a document, the user agent must invoke the media element's resource selection algorithm.

The **`autobuffer`** attribute is a boolean attribute. Its presence hints to the user agent that the author believes that the media element will likely be used, even though the element does not have an `autoplay` attribute. (The attribute has no effect if used in conjunction with the `autoplay` attribute, though including both is not an error.) This attribute may be ignored altogether. The attribute must be ignored if the `autoplay` attribute is present.

The **`autobuffer`** DOM attribute must reflect the content attribute of the same name.

media **`.buffered`**

Returns a `TimeRanges` object that represents the ranges of the media resource that the user agent has buffered.

The **`buffered`** attribute must return a new static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent has buffered, at the time the attribute is evaluated. Users agents must accurately determine the ranges available, even for media streams where this can only be determined by tedious inspection.

Note: Typically this will be a single range anchored at the zero point, but if, e.g. the user agent uses HTTP range requests in response to seeking, then there could be multiple ranges.

User agents may discard previously buffered data.

Note: Thus, a time position included within a range of the objects return by the `buffered` attribute at one time can end up being not included in the range(s) of objects returned by the same attribute at later times.

4.8.10.6 Offsets into the media resource

media **`.duration`**

Returns the length of the media resource, in seconds.

Returns NaN if the duration isn't available.

Returns Infinity for unbounded streams.

media **.** **`currentTime`** **[= *value*]**

Returns the current playback position, in seconds.

Can be set, to seek to the given time.

Will throw an `INVALID_STATE_ERR` exception if there is no selected media resources. Will throw an `INDEX_SIZE_ERR` exception if the given time is not within the ranges to which the user agent can seek.

The **`duration`** attribute must return the length of the media resource, in seconds. If no media data is available, then the attributes must return the Not-a-Number (NaN) value. If the media resource is known to be unbounded (e.g. a streaming radio), then the attribute must return the positive Infinity value.

The user agent must determine the duration of the media resource before playing any part of the media data and before setting `readyState` to a value equal to or greater than `HAVE_METADATA`, even if doing so requires seeking to multiple parts of the resource.

When the length of the media resource changes (e.g. from being unknown to known, or from a previously established length to a new length) the user agent must queue a task to fire a simple event called `durationchange` at the media element.

> If an "infinite" stream ends for some reason, then the duration would change from positive Infinity to the time of the last frame or sample in the stream, and the `durationchange` event would be fired. Similarly, if the user agent initially estimated the media resource's duration instead of determining it precisely, and later revises the estimate based on new information, then the duration would change and the `durationchange` event would be fired.

Media elements have a **current playback position**, which must initially be zero. The current position is a time.

The **`currentTime`** attribute must, on getting, return the current playback position, expressed in seconds. On setting, the user agent must seek to the new value (which might raise an exception).

If the media resource is a streaming resource, then the user agent might be unable to obtain certain parts of the resource after it has expired from its buffer. The ***earliest possible position*** is the earliest position in the stream that the user agent can ever obtain again.

When the *earliest possible position* changes, if the the current playback position is before the *earliest possible position*, the user agent must seek to the *earliest possible position*.

The **`loop`** attribute is a boolean attribute that, if specified, indicates that the media element is to seek back to the start of the media resource upon reaching the end.

The **`loop`** DOM attribute must reflect the content attribute of the same name.

4.8.10.7 The ready states

media **. readyState**

Returns a value that expresses the current state of the element with respect to rendering the current playback position, from the codes in the list below.

Media elements have a *ready state*, which describes to what degree they are ready to be rendered at the current playback position. The possible values are as follows; the ready state of a media element at any particular time is the greatest value describing the state of the element:

HAVE_NOTHING **(numeric value 0)**
No information regarding the media resource is available. No data for the current playback position is available. Media elements whose `networkState` attribute is `NETWORK_EMPTY` are always in the `HAVE_NOTHING` state.

HAVE_METADATA **(numeric value 1)**
Enough of the resource has been obtained that the duration of the resource is available. In the case of a `video` element, the dimensions of the video are also available. The API will no longer raise an exception when seeking. No media data is available for the immediate current playback position.

HAVE_CURRENT_DATA **(numeric value 2)**
Data for the immediate current playback position is available, but not enough data is available that the user agent could successfully advance the current playback position at all without immediately reverting to the `HAVE_METADATA` state. For example, in video this corresponds to the user agent having data from the current frame, but not the next frame.

HAVE_FUTURE_DATA **(numeric value 3)**
Data for the immediate current playback position is available, as well as enough data for the user agent to advance the current playback position at least a little without immediately reverting to the `HAVE_METADATA` state. For example, In video this corresponds to the user agent having data for at least the current frame and the next frame.

HAVE_ENOUGH_DATA **(numeric value 4)**
Data for the immediate current playback position is available, as well as enough data for the user agent to advance the current playback position at least a little without immediately reverting to the `HAVE_METADATA` state, and, in addition, the user agent estimates that data is being fetched at a rate where the current playback position, if it were to advance at the rate given by the `defaultPlaybackRate` attribute, would not overtake the available data before playback reaches the end of the media resource.

When the ready state of a media element whose `networkState` is not `NETWORK_EMPTY` changes, the user agent must follow the steps given below:

↪ **If the previous ready state was `HAVE_NOTHING`, and the new ready state is `HAVE_METADATA`**

Note: A `loadedmetadata` DOM event will be fired as part of the `load()`

algorithm.

↪ **If the previous ready state was HAVE_METADATA and the new ready state is HAVE_CURRENT_DATA**

If this is the first time this occurs for this media element since the `load()` algorithm was last invoked, the user agent must queue a task to fire a simple event called `loadeddata` at the element.

↪ **If the previous ready state was HAVE_FUTURE_DATA or more, and the new ready state is HAVE_CURRENT_DATA or less**

Note: A `waiting` DOM event can be fired, depending on the current state of playback.

↪ **If the previous ready state was HAVE_CURRENT_DATA or less, and the new ready state is HAVE_FUTURE_DATA**

The user agent must queue a task to fire a simple event called `canplay`.

If the element is potentially playing, the user agent must queue a task to fire a simple event called `playing`.

↪ **If the new ready state is HAVE_ENOUGH_DATA**

If the previous ready state was HAVE_CURRENT_DATA or less, the user agent must queue a task to fire a simple event called `canplay`, and, if the element is also potentially playing, queue a task to fire a simple event called `playing`.

If the autoplaying flag is true, and the `paused` attribute is true, and the media element has an `autoplay` attribute specified, then the user agent may also set the `paused` attribute to false, queue a task to fire a simple event called `play`, and queue a task to fire a simple event called `playing`.

Note: User agents are not required to autoplay, and it is suggested that user agents honor user preferences on the matter. Authors are urged to use the `autoplay` attribute rather than using script to force the video to play, so as to allow the user to override the behavior if so desired.

In any case, the user agent must finally queue a task to fire a simple event called `canplaythrough`.

Note: It is possible for the ready state of a media element to jump between these states discontinuously. For example, the state of a media element can jump straight from HAVE_METADATA to HAVE_ENOUGH_DATA without passing through the HAVE_CURRENT_DATA and HAVE_FUTURE_DATA states.

The **`readyState`** DOM attribute must, on getting, return the value described above that describes the current ready state of the media element.

The **`autoplay`** attribute is a boolean attribute. When present, the user agent (as described in the algorithm described herein) will automatically begin playback of the media resource as soon as it can do so without stopping.

The **autoplay** DOM attribute must reflect the content attribute of the same name.

4.8.10.8 Cue ranges

media **.** **addCueRange(*className*, *id*, *start*, *end*, *pauseOnExit*, *enterCallback*, *exitCallback*)**

Registers a range of time, given in seconds, and a pair of callbacks, the first of which will be invoked when the current playback position enters the range, and the second of which will be invoked when it exits the range. The callbacks are invoked with the given ID as their argument.

In addition, if the *pauseOnExit* argument is true, then playback will pause when it reaches the end of the range.

media **.** **removeCueRange(*className*)**

Removes all the ranges that were registered with the given class name.

Media elements have a set of **cue ranges**. Each cue range is made up of the following information:

A class name
: A group of related ranges can be given the same class name so that they can all be removed at the same time.

An identifier
: A string can be assigned to each cue range for identification by script. The string need not be unique and can contain any value.

A start time

An end time
: The actual time range, using the same timeline as the media resource itself.

A "pause" boolean
: A flag indicating whether to pause playback on exit.

An "enter" callback
: A callback that is called when the current playback position enters the range.

An "exit" callback
: A callback that is called when the current playback position exits the range.

An "active" boolean
: A flag indicating whether the range is active or not.

The **addCueRange(*className*, *id*, *start*, *end*, *pauseOnExit*, *enterCallback*, *exitCallback*)** method must, when called, add a cue range to the media element, that cue range having the class name *className*, the identifier *id*, the start time *start* (in seconds), the end time *end* (in seconds), the "pause" boolean with the same value as *pauseOnExit*, the "enter" callback *enterCallback*, the "exit" callback *exitCallback*, and an "active" boolean that is true if the current playback position is equal to or greater than the start time and less than the end time, and false otherwise.

The **removeCueRanges(*className*)** method must, when called, remove all the cue ranges of the

media element which have the class name *className*.

4.8.10.9 Playing the media resource

media **. `paused`**

Returns true if playback is paused; false otherwise.

media **. `ended`**

Returns true if playback has reached the end of the media resource.

media **. `defaultPlaybackRate` [= *value*]**

Returns the default rate of playback, for when the user is not fast-forwarding or reversing through the media resource.

Can be set, to change the default rate of playback.

The default rate has no direct effect on playback, but if the user switches to a fast-forward mode, when they return to the normal playback mode, it is expected that the rate of playback will be returned to the default rate of playback.

media **. `playbackRate` [= *value*]**

Returns the current rate playback, where 1.0 is normal speed.

Can be set, to change the rate of playback.

media **. `played`**

Returns a `TimeRanges` object that represents the ranges of the media resource that the user agent has played.

media **. `play`()**

Sets the `paused` attribute to false, loading the media resource and beginning playback if necessary. If the playback had ended, will restart it from the start.

media **. `pause`()**

Sets the `paused` attribute to true, loading the media resource if necessary.

The **`paused`** attribute represents whether the media element is paused or not. The attribute must initially be true.

A media element is said to be **potentially playing** when its `paused` attribute is false, the `readyState` attribute is either `HAVE_FUTURE_DATA` or `HAVE_ENOUGH_DATA`, the element has not ended playback, playback has not stopped due to errors, and the element has not paused for user interaction.

A media element is said to have **ended playback** when the element's `readyState` attribute is `HAVE_METADATA` or greater, and the current playback position is the end of the media resource,

and the media element does not have a `loop` attribute specified.

The **`ended`** attribute must return true if the media element has ended playback, and false otherwise.

A media element is said to have **stopped due to errors** when the element's `readyState` attribute is `HAVE_METADATA` or greater, and the user agent encounters a non-fatal error during the processing of the media data, and due to that error, is not able to play the content at the current playback position.

A media element is said to have **paused for user interaction** when its `paused` attribute is false, the `readyState` attribute is either `HAVE_FUTURE_DATA` or `HAVE_ENOUGH_DATA` and the user agent has reached a point in the media resource where the user has to make a selection for the resource to continue.

It is possible for a media element to have both ended playback and paused for user interaction at the same time.

When a media element that is potentially playing stops playing because it has paused for user interaction, the user agent must queue a task to fire a simple event called `timeupdate` at the element.

When a media element that is potentially playing stops playing because its `readyState` attribute changes to a value lower than `HAVE_FUTURE_DATA`, without the element having ended playback, or playback having stopped due to errors, or playback having paused for user interaction, or the seeking algorithm being invoked, the user agent must queue a task to fire a simple event called `timeupdate` at the element, and queue a task to fire a simple event called `waiting` at the element.

When the current playback position reaches the end of the media resource, then the user agent must follow these steps:

1. If the media element has a `loop` attribute specified, then seek to the *earliest possible position* of the media resource and abort these steps.

2. Stop playback.

   Note: The `ended` attribute becomes true.

3. The user agent must queue a task to fire a simple event called `timeupdate` at the element.

4. The user agent must queue a task to fire a simple event called `ended` at the element.

The **`defaultPlaybackRate`** attribute gives the desired speed at which the media resource is to play, as a multiple of its intrinsic speed. The attribute is mutable: on getting it must return the last value it was set to, or 1.0 if it hasn't yet been set; on setting the attribute must be set to the new value.

The **`playbackRate`** attribute gives the speed at which the media resource plays, as a multiple of its intrinsic speed. If it is not equal to the `defaultPlaybackRate`, then the implication is that the user is using a feature such as fast forward or slow motion playback. The attribute is mutable: on getting it must return the last value it was set to, or 1.0 if it hasn't yet been set; on

setting the attribute must be set to the new value, and the playback must change speed (if the element is potentially playing).

The "play" function in a user agent's interface must set the `playbackRate` attribute to the value of the `defaultPlaybackRate` attribute before invoking the `play()` method's steps. Features such as fast-forward or rewind must be implemented by only changing the `playbackRate` attribute.

When the `defaultPlaybackRate` or `playbackRate` attributes change value (either by being set by script or by being changed directly by the user agent, e.g. in response to user control) the user agent must queue a task to fire a simple event called `ratechange` at the media element.

The **`played`** attribute must return a new static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent has so far rendered, at the time the attribute is evaluated.

When the **`play()`** method on a media element is invoked, the user agent must run the following steps.

1. If the media element's `networkState` attribute has the value `NETWORK_EMPTY`, then the user agent must invoke the media element's resource selection algorithm.

2. If the playback has ended, then the user agent must seek to the *earliest possible position* of the media resource.

   Note: This will cause the user agent to queue a task to fire a simple event called `timeupdate` at the media element.

3. If the media element's `paused` attribute is true, it must be set to false.

   If this changed the value of `paused`, the user agent must run the following substeps:

   1. Queue a task to fire a simple event called `play` at the element.

   2. If the media element's `readyState` attribute has the value `HAVE_NOTHING`, `HAVE_METADATA`, or `HAVE_CURRENT_DATA`, queue a task to fire a simple event called `waiting` at the element.

   3. Otherwise, the media element's `readyState` attribute has the value `HAVE_FUTURE_DATA` or `HAVE_ENOUGH_DATA`; queue a task to fire a simple event called `playing` at the element.

4. The media element's autoplaying flag must be set to false.

5. The method must then return.

When the **`pause()`** method is invoked, the user agent must run the following steps:

1. If the media element's `networkState` attribute has the value `NETWORK_EMPTY`, then the user agent must invoke the media element's resource selection algorithm.

2. If the media element's `paused` attribute is false, it must be set to true.

3. The media element's autoplaying flag must be set to false.

4. If the second step above changed the value of `paused`, then the user agent must queue a task to fire a simple event called `timeupdate` at the element, and queue a task to fire a simple event called `pause` at the element.

When a media element is potentially playing and its `Document` is an active document, its current playback position must increase monotonically at `playbackRate` units of media time per unit time of wall clock time.

> ***Note: This specification doesn't define how the user agent achieves the appropriate playback rate — depending on the protocol and media available, it is plausible that the user agent could negotiate with the server to have the server provide the media data at the appropriate rate, so that (except for the period between when the rate is changed and when the server updates the stream's playback rate) the client doesn't actually have to drop or interpolate any frames.***

When the `playbackRate` is negative (playback is backwards), any corresponding audio must be muted. When the `playbackRate` is so low or so high that the user agent cannot play audio usefully, the corresponding audio must also be muted. If the `playbackRate` is not 1.0, the user agent may apply pitch adjustments to the audio as necessary to render it faithfully.

The `playbackRate` can be 0.0, in which case the current playback position doesn't move, despite playback not being paused (`paused` doesn't become true, and the `pause` event doesn't fire).

Media elements that are potentially playing while not in a `Document` must not play any video, but should play any audio component. Media elements must not stop playing just because all references to them have been removed; only once a media element to which no references exist has reached a point where no further audio remains to be played for that element (e.g. because the element is paused, or because the end of the clip has been reached, or because its `playbackRate` is 0.0) may the element be garbage collected.

When the current playback position of a media element changes (e.g. due to playback or seeking), the user agent must run the following steps. If the current playback position changes while the steps are running, then the user agent must wait for the steps to complete, and then must immediately rerun the steps. (These steps are thus run as often as possible or needed — if one iteration takes a long time, this can cause certain ranges to be skipped over as the user agent rushes ahead to "catch up".)

1. Let *current ranges* be an ordered list of cue ranges, initialized to contain all the cue ranges of the media element whose start times are less than or equal to the current playback position and whose end times are greater than the current playback position, in the order they were added to the element.

2. Let *other ranges* be an ordered list of cue ranges, initialized to contain all the cue ranges of the media element that are not present in *current ranges*, in the order they were added to the element.

3. If the time was reached through the usual monotonic increase of the current playback

position during normal playback, and if the user agent has not fired a `timeupdate` event at the element in the past 15 to 250ms, then the user agent must queue a task to fire a simple event called `timeupdate` at the element. (In the other cases, such as explicit seeks, relevant events get fired as part of the overall process of changing the current playback position.)

> ***Note: The event thus is not to be fired faster than about 66Hz or slower than 4Hz. User agents are encouraged to vary the frequency of the event based on the system load and the average cost of processing the event each time, so that the UI updates are not any more frequent than the user agent can comfortably handle while decoding the video.***

4. If none of the cue ranges in *current ranges* have their "active" boolean set to "false" (inactive) and none of the cue ranges in *other ranges* have their "active" boolean set to "true" (active), then abort these steps.

5. If the time was reached through the usual monotonic increase of the current playback position during normal playback, and there are cue ranges in *other ranges* that have both their "active" boolean and their "pause" boolean set to "true", then immediately act as if the element's `pause()` method had been invoked. (In the other cases, such as explicit seeks, playback is not paused by exiting a cue range, even if that cue range has its "pause" boolean set to "true".)

6. For each non-null "exit" callback of the cue ranges in *other ranges* that have their "active" boolean set to "true" (active), in list order, queue a task that invokes the callback, passing the cue range's identifier as the callback's only argument.

7. For each non-null "enter" callback of the cue ranges in *current ranges* that have their "active" boolean set to "false" (inactive), in list order, queue a task that invokes the callback, passing the cue range's identifier as the callback's only argument.

8. Set the "active" boolean of all the cue ranges in the *current ranges* list to "true" (active), and the "active" boolean of all the cue ranges in the *other ranges* list to "false" (inactive).

When a media element is removed from a `Document`, if the media element's `networkState` attribute has a value other than `NETWORK_EMPTY` then the user agent must act as if the `pause()` method had been invoked.

> ***Note: If the media element's `Document` stops being a fully active document, then the playback will stop until the document is active again.***

4.8.10.10 Seeking

media **`.seeking`**

Returns true if the user agent is currently seeking.

media **`.seekable`**

Returns a `TimeRanges` object that represents the ranges of the media resource to

which it is possible for the user agent to seek.

The **seeking** attribute must initially have the value false.

When the user agent is required to **seek** to a particular *new playback position* in the media resource, it means that the user agent must run the following steps:

1. If the media element's `readyState` is `HAVE_NOTHING`, then the user agent must raise an `INVALID_STATE_ERR` exception (if the seek was in response to a DOM method call or setting of a DOM attribute), and abort these steps.
2. If the *new playback position* is later than the end of the media resource, then let it be the end of the media resource instead.
3. If the *new playback position* is less than the *earliest possible position*, let it be that position instead.
4. If the (possibly now changed) *new playback position* is not in one of the ranges given in the `seekable` attribute, then the user agent must raise an `INDEX_SIZE_ERR` exception (if the seek was in response to a DOM method call or setting of a DOM attribute), and abort these steps.
5. The current playback position must be set to the given *new playback position*.
6. The `seeking` DOM attribute must be set to true.
7. The user agent must queue a task to fire a simple event called `timeupdate` at the element.
8. If the media element was potentially playing immediately before it started seeking, but seeking caused its `readyState` attribute to change to a value lower than `HAVE_FUTURE_DATA`, the user agent must queue a task to fire a simple event called `waiting` at the element.
9. If, when it reaches this step, the user agent has still not established whether or not the media data for the *new playback position* is available, and, if it is, decoded enough data to play back that position, the user agent must queue a task to fire a simple event called `seeking` at the element.
10. If the seek was in response to a DOM method call or setting of a DOM attribute, then continue the script. The remainder of these steps must be run asynchronously.
11. The user agent must wait until it has established whether or not the media data for the *new playback position* is available, and, if it is, until it has decoded enough data to play back that position.
12. The `seeking` DOM attribute must be set to false.
13. The user agent must queue a task to fire a simple event called `seeked` at the element.

The **seekable** attribute must return a new static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent is able to seek to, at the time the

attribute is evaluated.

Note: If the user agent can seek to anywhere in the media resource, e.g. because it a simple movie file and the user agent and the server support HTTP Range requests, then the attribute would return an object with one range, whose start is the time of the first frame (typically zero), and whose end is the same as the time of the first frame plus the `duration` attribute's value (which would equal the time of the last frame).

Note: The range might be continuously changing, e.g. if the user agent is buffering a sliding window on an infinite stream. This is the behavior seen with DVRs viewing live TV, for instance.

Media resources might be internally scripted or interactive. Thus, a media element could play in a non-linear fashion. If this happens, the user agent must act as if the algorithm for seeking was used whenever the current playback position changes in a discontinuous fashion (so that the relevant events fire).

4.8.10.11 User interface

The **`controls`** attribute is a boolean attribute. If present, it indicates that the author has not provided a scripted controller and would like the user agent to provide its own set of controls.

If the attribute is present, or if scripting is disabled for the media element, then the user agent should **expose a user interface to the user**. This user interface should include features to begin playback, pause playback, seek to an arbitrary position in the content (if the content supports arbitrary seeking), change the volume, and show the media content in manners more suitable to the user (e.g. full-screen video or in an independent resizable window). Other controls may also be made available.

If the attribute is absent, then the user agent should avoid making a user interface available that could conflict with an author-provided user interface. User agents may make the following features available, however, even when the attribute is absent:

User agents may provide controls to affect playback of the media resource (e.g. play, pause, seeking, and volume controls), but such features should not interfere with the page's normal rendering. For example, such features could be exposed in the media element's context menu.

Where possible (specifically, for starting, stopping, pausing, and unpausing playback, for muting or changing the volume of the audio, and for seeking), user interface features exposed by the user agent must be implemented in terms of the DOM API described above, so that, e.g., all the same events fire.

The **`controls`** DOM attribute must reflect the content attribute of the same name.

media **.** **`volume`** **[=** ***value*** **]**

Returns the current playback volume, as a number in the range 0.0 to 1.0,

where 0.0 is the quietest and 1.0 the loudest.

Can be set, to change the volume.

Throws an `INDEX_SIZE_ERR` if the new value is not in the range 0.0 .. 1.0.

media **.** `muted` **[=** ***value*** **]**

Returns true if audio is muted, overriding the `volume` attribute, and false if the `volume` attribute is being honored.

Can be set, to change whether the audio is muted or not.

The **`volume`** attribute must return the playback volume of any audio portions of the media element, in the range 0.0 (silent) to 1.0 (loudest). Initially, the volume must be 1.0, but user agents may remember the last set value across sessions, on a per-site basis or otherwise, so the volume may start at other values. On setting, if the new value is in the range 0.0 to 1.0 inclusive, the attribute must be set to the new value and the playback volume must be correspondingly adjusted as soon as possible after setting the attribute, with 0.0 being silent, and 1.0 being the loudest setting, values in between increasing in loudness. The range need not be linear. The loudest setting may be lower than the system's loudest possible setting; for example the user could have set a maximum volume. If the new value is outside the range 0.0 to 1.0 inclusive, then, on setting, an `INDEX_SIZE_ERR` exception must be raised instead.

The **`muted`** attribute must return true if the audio channels are muted and false otherwise. Initially, the audio channels should not be muted (false), but user agents may remember the last set value across sessions, on a per-site basis or otherwise, so the muted state may start as muted (true). On setting, the attribute must be set to the new value; if the new value is true, audio playback for this media resource must then be muted, and if false, audio playback must then be enabled.

Whenever either the `muted` or `volume` attributes are changed, the user agent must queue a task to fire a simple event called `volumechange` at the media element.

4.8.10.12 Time ranges

Objects implementing the `TimeRanges` interface represent a list of ranges (periods) of time.

```
interface TimeRanges {
  readonly attribute unsigned long length;
  float start(in unsigned long index);
  float end(in unsigned long index);
};
```

media **.** **`length`**

Returns the number of ranges in the object.

time **=** ***media*** **.** **`start`****(*****index*****)**

Returns the time for the start of the range with the given index.

Throws an `INDEX_SIZE_ERR` if the index is out of range.

time **=** ***media*** **.** **`end`****(*****index*****)**

Returns the time for the end of the range with the given index.

Throws an `INDEX_SIZE_ERR` if the index is out of range.

The **`length`** DOM attribute must return the number of ranges represented by the object.

The **`start(`*`index`*`)`** method must return the position of the start of the *index*th range represented by the object, in seconds measured from the start of the timeline that the object covers.

The **`end(`*`index`*`)`** method must return the position of the end of the *index*th range represented by the object, in seconds measured from the start of the timeline that the object covers.

These methods must raise `INDEX_SIZE_ERR` exceptions if called with an *index* argument greater than or equal to the number of ranges represented by the object.

When a `TimeRanges` object is said to be a **normalized `TimeRanges` object**, the ranges it represents must obey the following criteria:

- The start of a range must be greater than the end of all earlier ranges.
- The start of a range must be less than the end of that same range.

In other words, the ranges in such an object are ordered, don't overlap, aren't empty, and don't touch (adjacent ranges are folded into one bigger range).

The timelines used by the objects returned by the `buffered`, `seekable` and `played` DOM attributes of media elements must be the same as that element's media resource's timeline.

4.8.10.13 Event summary

The following events fire on media elements as part of the processing model described above:

Event name	Interface	Dispatched when...	Preconditions
`loadstart`	`ProgressEvent` [PROGRESS]	The user agent begins looking for media data, as part of the resource selection algorithm.	`networkState` equals `NETWORK_LOADING`
`progress`	`ProgressEvent` [PROGRESS]	The user agent is fetching media data.	`networkState` equals `NETWORK_LOADING`
`suspend`	`ProgressEvent` [PROGRESS]	The user agent is intentionally not currently fetching media data, but does not have the entire media resource downloaded.	`networkState` equals `NETWORK_IDLE`
`load`	`ProgressEvent` [PROGRESS]	The user agent finishes fetching the entire media resource.	`networkState` equals `NETWORK_LOADED`
`abort`	`ProgressEvent` [PROGRESS]	The user agent stops fetching the media data before it is	`error` is an object with the code `MEDIA_ERR_ABORTED`.

Event name	Interface	Dispatched when...	Preconditions
		completely downloaded.	networkState equals either NETWORK_EMPTY or NETWORK_LOADED, depending on when the download was aborted.
error	ProgressEvent [PROGRESS]	An error occurs while fetching the media data.	error is an object with the code MEDIA_ERR_NETWORK or higher. networkState equals either NETWORK_EMPTY or NETWORK_LOADED, depending on when the download was aborted.
emptied	Event	A media element whose networkState was previously not in the NETWORK_EMPTY state has just switched to that state (either because of a fatal error during load that's about to be reported, or because the load() method was invoked while the resource selection algorithm was already running, in which case it is fired synchronously during the load() method call).	networkState is NETWORK_EMPTY; all the DOM attributes are in their initial states.
stalled	ProgressEvent	The user agent is trying to fetch media data, but data is unexpectedly not forthcoming.	networkState is NETWORK_LOADING.
play	Event	Playback has begun. Fired after the play() method has returned.	paused is newly false.
pause	Event	Playback has been paused. Fired after the pause method has returned.	paused is newly true.
loadedmetadata	Event	The user agent has just determined the the duration and dimensions of the media resource.	readyState is newly equal to HAVE_METADATA or greater for the first time.
loadeddata	Event	The user agent can render the media data at the current playback position for the first time.	readyState newly increased to HAVE_CURRENT_DATA or greater for the first time.
waiting	Event	Playback has stopped because the next frame is not available, but the user agent expects that frame to become available in due course.	readyState is newly equal to or less than HAVE_CURRENT_DATA, and paused is false. Either seeking is true, or the current playback position is not

Event name	Interface	Dispatched when...	Preconditions
			contained in any of the ranges in `buffered`. It is possible for playback to stop for two other reasons without `paused` being false, but those two reasons do not fire this event: maybe playback ended, or playback stopped due to errors.
playing	Event	Playback has started.	`readyState` is newly equal to or greater than `HAVE_FUTURE_DATA`, `paused` is false, `seeking` is false, or the current playback position is contained in one of the ranges in `buffered`.
canplay	Event	The user agent can resume playback of the media data, but estimates that if playback were to be started now, the media resource could not be rendered at the current playback rate up to its end without having to stop for further buffering of content.	`readyState` newly increased to `HAVE_FUTURE_DATA` or greater.
canplaythrough	Event	The user agent estimates that if playback were to be started now, the media resource could be rendered at the current playback rate all the way to its end without having to stop for further buffering.	`readyState` is newly equal to `HAVE_ENOUGH_DATA`.
seeking	Event	The `seeking` DOM attribute changed to true and the seek operation is taking long enough that the user agent has time to fire the event.	
seeked	Event	The `seeking` DOM attribute changed to false.	
timeupdate	Event	The current playback position changed as part of normal playback or in an especially interesting way, for example discontinuously.	
ended	Event	Playback has stopped because the end of the media resource was reached.	`currentTime` equals the end of the media resource; `ended` is true.
ratechange	Event	Either the	

Event name	Interface	Dispatched when...	Preconditions
		`defaultPlaybackRate` or the `playbackRate` attribute has just been updated.	
durationchange	`Event`	The `duration` attribute has just been updated.	
volumechange	`Event`	Either the `volume` attribute or the `muted` attribute has changed. Fired after the relevant attribute's setter has returned.	

4.8.10.14 Security and privacy considerations

The main security and privacy implications of the `video` and `audio` elements come from the ability to embed media cross-origin. There are two directions that threats can flow: from hostile content to a victim page, and from a hostile page to victim content.

If a victim page embeds hostile content, the threat is that the content might contain scripted code that attempts to interact with the `Document` that embeds the content. To avoid this, user agents must ensure that there is no access from the content to the embedding page. In the case of media content that uses DOM concepts, the embedded content must be treated as if it was in its own unrelated top-level browsing context.

> For instance, if an SVG animation was embedded in a `video` element, the user agent would not give it access to the DOM of the outer page. From the perspective of scripts in the SVG resource, the SVG file would appear to be in a lone top-level browsing context with no parent.

If a hostile page embeds victom content, the threat is that the embedding page could obtain information from the content that it would not otherwise have access to. The API does expose some information: the existence of the media, its type, its duration, its size, and the performance characteristics of its host. Such information is already potentially problematic, but in practice the same information can more or less be obtained using the `img` element, and so it has been deemed acceptable.

However, significantly more sensitive information could be obtained if the user agent further exposes metadata within the content such as subtitles or chapter titles. This version of the API does not expose such information. Future extensions to this API will likely reuse a mechanism such as CORS to check that the embedded content's site has opted in to exposing such information. [CORS]

> An attacker could trick a user running within a corporate network into visiting a site that attempts to load a video from a previously leaked location on the corporation's intranet. If such a video included confidential plans for a new product, then being able to read the subtitles would present a confidentiality breach.

4.8.11 The `canvas` element

Categories

Flow content.
Phrasing content.
Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Transparent.

Content attributes:

Global attributes
`width`
`height`

DOM interface:

```
interface HTMLCanvasElement : HTMLElement {
           attribute unsigned long width;
           attribute unsigned long height;

  DOMString toDataURL([Optional] in DOMString type, [Variadic] in any args

  Object getContext(in DOMString contextId);
};
```

The `canvas` element represents a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, or other visual images on the fly.

Authors should not use the `canvas` element in a document when a more suitable element is available. For example, it is inappropriate to use a `canvas` element to render a page heading: if the desired presentation of the heading is graphically intense, it should be marked up using appropriate elements (typically `h1`) and then styled using CSS and supporting technologies such as XBL.

When authors use the `canvas` element, they must also provide content that, when presented to the user, conveys essentially the same function or purpose as the bitmap canvas. This content may be placed as content of the `canvas` element. The contents of the `canvas` element, if any, are the element's fallback content.

In interactive visual media, if scripting is enabled for the `canvas` element, the `canvas` element represents an embedded element with a dynamically created image.

In non-interactive, static, visual media, if the `canvas` element has been previously painted on (e.g. if the page was viewed in an interactive visual medium and is now being printed, or if some script that ran during the page layout process painted on the element), then the `canvas` element represents embedded content with the current image and size. Otherwise, the element represents its fallback content instead.

In non-visual media, and in visual media if scripting is disabled for the `canvas` element, the `canvas` element represents its fallback content instead.

The `canvas` element has two attributes to control the size of the coordinate space: **`width`** and **`height`**. These attributes, when specified, must have values that are valid non-negative integers. The rules for parsing non-negative integers must be used to obtain their numeric values. If an attribute is missing, or if parsing its value returns an error, then the default value must be used instead. The `width` attribute defaults to 300, and the `height` attribute defaults to 150.

The intrinsic dimensions of the `canvas` element equal the size of the coordinate space, with the numbers interpreted in CSS pixels. However, the element can be sized arbitrarily by a style sheet. During rendering, the image is scaled to fit this layout size.

The size of the coordinate space does not necessarily represent the size of the actual bitmap that the user agent will use internally or during rendering. On high-definition displays, for instance, the user agent may internally use a bitmap with two device pixels per unit in the coordinate space, so that the rendering remains at high quality throughout.

Whenever the `width` and `height` attributes are set (whether to a new value or to the previous value), the bitmap and any associated contexts must be cleared back to their initial state and reinitialized with the newly specified coordinate space dimensions.

The **`width`** and **`height`** DOM attributes must reflect the respective content attributes of the same name.

> Only one square appears to be drawn in the following example:
>
> ```
> // canvas is a reference to a <canvas> element
> var context = canvas.getContext('2d');
> context.fillRect(0,0,50,50);
> canvas.setAttribute('width', '300'); // clears the canvas
> context.fillRect(0,100,50,50);
> canvas.width = canvas.width; // clears the canvas
> context.fillRect(100,0,50,50); // only this square remains
> ```

When the canvas is initialized it must be set to fully transparent black.

To draw on the canvas, authors must first obtain a reference to a **context** using the **`getContext(`*`contextId`*`)`** method of the `canvas` element.

> ***context* = *canvas* . `getContext`(*contextId*)**
>
> Returns an object that exposes an API for drawing on the canvas.
>
> Returns null if the given context ID is not supported.

This specification only defines one context, with the name "`2d`". If `getContext()` is called with that exact string for its *contextId* argument, then the UA must return a reference to an object implementing `CanvasRenderingContext2D`. Other specifications may define their own contexts, which would return different objects.

Vendors may also define experimental contexts using the syntax *`vendorname-context`*, for example, `moz-3d`.

When the UA is passed an empty string or a string specifying a context that it does not support, then it must return null. String comparisons must be case-sensitive.

Note: A future version of this specification will probably define a `3d` context (probably based on the OpenGL ES API).

***url* = *canvas* . `toDataURL`([*type*, ...])**

Returns a `data:` URL for the image in the canvas.

The first argument, if provided, controls the type of the image to be returned (e.g. PNG or JPEG). The default is `image/png`; that type is also used if the given type isn't supported. The other arguments are specific to the type, and control the way that the image is generated, as given in the table below.

The **`toDataURL()`** method must, when called with no arguments, return a `data:` URL containing a representation of the image as a PNG file. [PNG].

If the canvas has no pixels (i.e. either its horizontal dimension or its vertical dimension is zero) then the method must return the string "`data:,`". (This is the shortest `data:` URL; it represents the empty string in a `text/plain` resource.)

When the `toDataURL(type)` method, when called with one *or more* arguments, must return a `data:` URL containing a representation of the image in the format given by *type*. The possible values are MIME types with no parameters, for example `image/png`, `image/jpeg`, or even maybe `image/svg+xml` if the implementation actually keeps enough information to reliably render an SVG image from the canvas.

For image types that do not support an alpha channel, the image must be composited onto a solid black background using the source-over operator, and the resulting image must be the one used to create the `data:` URL.

Only support for `image/png` is required. User agents may support other types. If the user agent does not support the requested type, it must return the image using the PNG format.

User agents must convert the provided type to lower case before establishing if they support that type and before creating the `data:` URL.

Note: When trying to use types other than `image/png`, authors can check if the image was really returned in the requested format by checking to see if the returned string starts with one the exact strings "`data:image/png,`" or "`data:image/png;`". If it does, the image is PNG, and thus the requested type was not supported. (The one exception to this is if the canvas has either no height or no width, in which case the result might simply be "`data:,`".)

If the method is invoked with the first argument giving a type corresponding to one of the types given in the first column of the following table, and the user agent supports that type, then the subsequent arguments, if any, must be treated as described in the second cell of that row.

Type	Other arguments
image/jpeg	The second argument, if it is a number between 0.0 and 1.0, must be treated as the desired quality level. If it is not a number or is outside that range, the user agent must use its default value, as if the argument had been omitted.

Type	Other arguments

Other arguments must be ignored and must not cause the user agent to raise an exception. A future version of this specification will probably define other parameters to be passed to `toDataURL()` to allow authors to more carefully control compression settings, image metadata, etc.

4.8.11.1 The 2D context

When the `getContext()` method of a `canvas` element is invoked with **`2d`** as the argument, a `CanvasRenderingContext2D` object is returned.

There is only one `CanvasRenderingContext2D` object per canvas, so calling the `getContext()` method with the `2d` argument a second time must return the same object.

The 2D context represents a flat Cartesian surface whose origin (0,0) is at the top left corner, with the coordinate space having *x* values increasing when going right, and *y* values increasing when going down.

```
interface CanvasRenderingContext2D {

  // back-reference to the canvas
  readonly attribute HTMLCanvasElement canvas;

  // state
  void save(); // push state on state stack
  void restore(); // pop state stack and restore state

  // transformations (default transform is the identity matrix)
  void scale(in float x, in float y);
  void rotate(in float angle);
  void translate(in float x, in float y);
  void transform(in float m11, in float m12, in float m21, in float m22, in float
  void setTransform(in float m11, in float m12, in float m21, in float m22, in flo

  // compositing
           attribute float globalAlpha; // (default 1.0)
           attribute DOMString globalCompositeOperation; // (default source-over)

  // colors and styles
           attribute any strokeStyle; // (default black)
           attribute any fillStyle; // (default black)
  CanvasGradient createLinearGradient(in float x0, in float y0, in float x1, in fl
  CanvasGradient createRadialGradient(in float x0, in float y0, in float r0, in fl
  CanvasPattern createPattern(in HTMLImageElement image, in DOMString repetition);
  CanvasPattern createPattern(in HTMLCanvasElement image, in DOMString repetition)

  // line caps/joins
           attribute float lineWidth; // (default 1)
           attribute DOMString lineCap; // "butt", "round", "square" (default "but
           attribute DOMString lineJoin; // "round", "bevel", "miter" (default "mi
           attribute float miterLimit; // (default 10)

  // shadows
           attribute float shadowOffsetX; // (default 0)
           attribute float shadowOffsetY; // (default 0)
           attribute float shadowBlur; // (default 0)
           attribute DOMString shadowColor; // (default transparent black)

  // rects
  void clearRect(in float x, in float y, in float w, in float h);
```

```
  void fillRect(in float x, in float y, in float w, in float h);
  void strokeRect(in float x, in float y, in float w, in float h);

  // path API
  void beginPath();
  void closePath();
  void moveTo(in float x, in float y);
  void lineTo(in float x, in float y);
  void quadraticCurveTo(in float cpx, in float cpy, in float x, in float y);
  void bezierCurveTo(in float cp1x, in float cp1y, in float cp2x, in float cp2y, i
  void arcTo(in float x1, in float y1, in float x2, in float y2, in float radius);
  void rect(in float x, in float y, in float w, in float h);
  void arc(in float x, in float y, in float radius, in float startAngle, in float
  void fill();
  void stroke();
  void clip();
  boolean isPointInPath(in float x, in float y);

  // text
           attribute DOMString font; // (default 10px sans-serif)
           attribute DOMString textAlign; // "start", "end", "left", "right", "cen
           attribute DOMString textBaseline; // "top", "hanging", "middle", "alpha
  void fillText(in DOMString text, in float x, in float y, [Optional] in float max
  void strokeText(in DOMString text, in float x, in float y, [Optional] in float m
  TextMetrics measureText(in DOMString text);

  // drawing images
  void drawImage(in HTMLImageElement image, in float dx, in float dy, [Optional] i
  void drawImage(in HTMLImageElement image, in float sx, in float sy, in float sw,
  void drawImage(in HTMLCanvasElement image, in float dx, in float dy, [Optional]
  void drawImage(in HTMLCanvasElement image, in float sx, in float sy, in float sw
  void drawImage(in HTMLVideoElement image, in float dx, in float dy, [Optional] i
  void drawImage(in HTMLVideoElement image, in float sx, in float sy, in float sw,

  // pixel manipulation
  ImageData createImageData(in float sw, in float sh);
  ImageData getImageData(in float sx, in float sy, in float sw, in float sh);
  void putImageData(in ImageData imagedata, in float dx, in float dy, [Optional] i
};

interface CanvasGradient {
  // opaque object
  void addColorStop(in float offset, in DOMString color);
};

interface CanvasPattern {
  // opaque object
};

interface TextMetrics {
  readonly attribute float width;
};

interface ImageData {
  readonly attribute unsigned long width;
  readonly attribute unsigned long height;
  readonly attribute CanvasPixelArray data;
};

[IndexGetter, IndexSetter]
interface CanvasPixelArray {
  readonly attribute unsigned long length;
};
```

context . canvas

Returns the canvas element.

The **canvas** attribute must return the canvas element that the context paints on.

Unless otherwise stated, for the 2D context interface, any method call with a numeric argument whose value is infinite or a NaN value must be ignored.

Whenever the CSS value currentColor is used as a color in this API, the "computed value of the 'color' property" for the purposes of determining the computed value of the currentColor keyword is the computed value of the 'color' property on the element in question at the time that the color is specified (e.g. when the appropriate attribute is set, or when the method is called; not when the color is rendered or otherwise used). If the computed value of the 'color' property is undefined for a particular case (e.g. because the element is not in a document), then the "computed value of the 'color' property" for the purposes of determining the computed value of the currentColor keyword is fully opaque black. [CSS3COLOR]

4.8.11.1.1 The canvas state

Each context maintains a stack of drawing states. **Drawing states** consist of:

- The current transformation matrix.
- The current clipping region.
- The current values of the following attributes: strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, globalCompositeOperation, font, textAlign, textBaseline.

Note: The current path and the current bitmap are not part of the drawing state. The current path is persistent, and can only be reset using the beginPath() method. The current bitmap is a property of the canvas, not the context.

context **. save()**

Pushes the current state onto the stack.

context **. restore()**

Pops the top state on the stack, restoring the context to that state.

The **save()** method must push a copy of the current drawing state onto the drawing state stack.

The **restore()** method must pop the top entry in the drawing state stack, and reset the drawing state it describes. If there is no saved state, the method must do nothing.

4.8.11.1.2 TRANSFORMATIONS

The transformation matrix is applied to coordinates when creating shapes and paths.

When the context is created, the transformation matrix must initially be the identity transform. It may then be adjusted using the transformation methods.

The transformations must be performed in reverse order. For instance, if a scale transformation that doubles the width is applied, followed by a rotation transformation that rotates drawing operations by a quarter turn, and a rectangle twice as wide as it is tall is then drawn on the canvas, the actual result will be a square.

context **.** **`scale`**(***x***, ***y***)

Changes the transformation matrix to apply a scaling transformation with the given characteristics.

context **.** **`rotate`**(***angle***)

Changes the transformation matrix to apply a rotation transformation with the given characteristics.

context **.** **`translate`**(***x***, ***y***)

Changes the transformation matrix to apply a translation transformation with the given characteristics.

context **.** **`transform`**(***m11***, ***m12***, ***m21***, ***m22***, ***dx***, ***dy***)

Changes the transformation matrix to apply the matrix given by the arguments as described below.

context **.** **`setTransform`**(***m11***, ***m12***, ***m21***, ***m22***, ***dx***, ***dy***)

Changes the transformation matrix *to* the matrix given by the arguments as described below.

The **`scale(`*`x`*`, `*`y`*`)`** method must add the scaling transformation described by the arguments to the transformation matrix. The *x* argument represents the scale factor in the horizontal direction and the *y* argument represents the scale factor in the vertical direction. The factors are multiples.

The **`rotate(`*`angle`*`)`** method must add the rotation transformation described by the argument to the transformation matrix. The *angle* argument represents a clockwise rotation angle expressed in radians.

The **`translate(`*`x`*`, `*`y`*`)`** method must add the translation transformation described by the arguments to the transformation matrix. The *x* argument represents the translation distance in the horizontal direction and the *y* argument represents the translation distance in the vertical direction. The arguments are in coordinate space units.

The **`transform(`*`m11`*`, `*`m12`*`, `*`m21`*`, `*`m22`*`, `*`dx`*`, `*`dy`*`)`** method must multiply the current transformation matrix with the matrix described by:

$$\begin{matrix} m11 & m21 & dx \\ m12 & m22 & dy \\ 0 & 0 & 1 \end{matrix}$$

The **`setTransform(`*`m11`*`, `*`m12`*`, `*`m21`*`, `*`m22`*`, `*`dx`*`, `*`dy`*`)`** method must reset the current transform to the identity matrix, and then invoke the `transform(`*`m11`*`, `*`m12`*`, `*`m21`*`, `*`m22`*`, `*`dx`*`, `*`dy`*`)` method with the same arguments.

4.8.11.1.3 Compositing

***context*`.globalAlpha` [= *value*]**

Returns the current alpha value applied to rendering operations.

Can be set, to change the alpha value. Values outside of the range 0.0 .. 1.0 are ignored.

***context*`.globalCompositeOperation` [= *value*]**

Returns the current composition operation, from the list below.

Can be set, to change the composition operation. Unknown values are ignored.

All drawing operations are affected by the global compositing attributes, `globalAlpha` and `globalCompositeOperation`.

The **`globalAlpha`** attribute gives an alpha value that is applied to shapes and images before they are composited onto the canvas. The value must be in the range from 0.0 (fully transparent) to 1.0 (no additional transparency). If an attempt is made to set the attribute to a value outside this range, the attribute must retain its previous value. When the context is created, the `globalAlpha` attribute must initially have the value 1.0.

The **`globalCompositeOperation`** attribute sets how shapes and images are drawn onto the existing bitmap, once they have had `globalAlpha` and the current transformation matrix applied. It must be set to a value from the following list. In the descriptions below, the source image, *A*, is the shape or image being rendered, and the destination image, *B*, is the current state of the bitmap.

`source-atop`
: *A* atop *B*. Display the source image wherever both images are opaque. Display the destination image wherever the destination image is opaque but the source image is transparent. Display transparency elsewhere.

`source-in`
: *A* in *B*. Display the source image wherever both the source image and destination image are opaque. Display transparency elsewhere.

`source-out`

A out *B*. Display the source image wherever the source image is opaque and the destination image is transparent. Display transparency elsewhere.

***source-over* (default)**

A over *B*. Display the source image wherever the source image is opaque. Display the destination image elsewhere.

destination-atop

B atop *A*. Same as `source-atop` but using the destination image instead of the source image and vice versa.

destination-in

B in *A*. Same as `source-in` but using the destination image instead of the source image and vice versa.

destination-out

B out *A*. Same as `source-out` but using the destination image instead of the source image and vice versa.

destination-over

B over *A*. Same as `source-over` but using the destination image instead of the source image and vice versa.

lighter

A plus *B*. Display the sum of the source image and destination image, with color values approaching 1 as a limit.

copy

A (*B* is ignored). Display the source image instead of the destination image.

xor

A xor *B*. Exclusive OR of the source image and destination image.

vendorName-operationName

Vendor-specific extensions to the list of composition operators should use this syntax.

These values are all case-sensitive — they must be used exactly as shown. User agents must not recognize values that are not a case-sensitive match for one of the values given above.

The operators in the above list must be treated as described by the Porter-Duff operator given at the start of their description (e.g. *A* over *B*). [PORTERDUFF]

On setting, if the user agent does not recognize the specified value, it must be ignored, leaving the value of `globalCompositeOperation` unaffected.

When the context is created, the `globalCompositeOperation` attribute must initially have the value `source-over`.

4.8.11.1.4 Colors and styles

***context* . `strokeStyle` [= *value*]**

Returns the current style used for stroking shapes.

Can be set, to change the stroke style.

The style can be either a string containing a CSS color, or a `CanvasGradient` or `CanvasPattern` object. Invalid values are ignored.

***context* . `fillStyle` [= *value*]**

Returns the current style used for filling shapes.

Can be set, to change the fill style.

The style can be either a string containing a CSS color, or a `CanvasGradient` or `CanvasPattern` object. Invalid values are ignored.

The **`strokeStyle`** attribute represents the color or style to use for the lines around shapes, and the **`fillStyle`** attribute represents the color or style to use inside the shapes.

Both attributes can be either strings, `CanvasGradient`s, or `CanvasPattern`s. On setting, strings must be parsed as CSS <color> values and the color assigned, and `CanvasGradient` and `CanvasPattern` objects must be assigned themselves. [CSS3COLOR] If the value is a string but is not a valid color, or is neither a string, a `CanvasGradient`, nor a `CanvasPattern`, then it must be ignored, and the attribute must retain its previous value.

On getting, if the value is a color, then the serialization of the color must be returned. Otherwise, if it is not a color but a `CanvasGradient` or `CanvasPattern`, then the respective object must be returned. (Such objects are opaque and therefore only useful for assigning to other attributes or for comparison to other gradients or patterns.)

The **serialization of a color** for a color value is a string, computed as follows: if it has alpha equal to 1.0, then the string is a lowercase six-digit hex value, prefixed with a "#" character (U+0023 NUMBER SIGN), with the first two digits representing the red component, the next two digits representing the green component, and the last two digits representing the blue component, the digits being in the range 0-9 a-f (U+0030 to U+0039 and U+0061 to U+0066). Otherwise, the color value has alpha less than 1.0, and the string is the color value in the CSS `rgba()` functional-notation format: the literal string `rgba` (U+0072 U+0067 U+0062 U+0061) followed by a U+0028 LEFT PARENTHESIS, a base-ten integer in the range 0-255 representing the red component (using digits 0-9, U+0030 to U+0039, in the shortest form possible), a literal U+002C COMMA and U+0020 SPACE, an integer for the green component, a comma and a space, an integer for the blue component, another comma and space, a U+0030 DIGIT ZERO, a U+002E FULL STOP (representing the decimal point), one or more digits in the range 0-9 (U+0030 to U+0039) representing the fractional part of the alpha value, and finally a U+0029 RIGHT PARENTHESIS.

When the context is created, the `strokeStyle` and `fillStyle` attributes must initially have the string value `#000000`.

There are two types of gradients, linear gradients and radial gradients, both represented by objects implementing the opaque `CanvasGradient` interface.

Once a gradient has been created (see below), stops are placed along it to define how the colors are distributed along the gradient. The color of the gradient at each stop is the color specified for that stop. Between each such stop, the colors and the alpha component must be linearly interpolated over the RGBA space without premultiplying the alpha value to find the color to use at that offset. Before the first stop, the color must be the color of the first stop. After the last stop, the color must be the color of the last stop. When there are no stops, the gradient is transparent black.

***gradient* . `addColorStop`(*offset*, *color*)**

Adds a color stop with the given color to the gradient at the given offset. 0.0 is the offset at one end of the gradient, 1.0 is the offset at the other end.

Throws an `INDEX_SIZE_ERR` exception if the offset it out of range. Throws a `SYNTAX_ERR` exception if the color cannot be parsed.

***gradient* = *context* . `createLinearGradient`(*x0*, *y0*, *x1*, *y1*)**

Returns a `CanvasGradient` object that represents a linear gradient that paints along the line given by the coordinates represented by the arguments.

If any of the arguments are not finite numbers, throws a `NOT_SUPPORTED_ERR` exception.

***gradient* = *context* . `createRadialGradient`(*x0*, *y0*, *r0*, *x1*, *y1*, *r1*)**

Returns a `CanvasGradient` object that represents a radial gradient that paints along the cone given by the circles represented by the arguments.

If any of the arguments are not finite numbers, throws a `NOT_SUPPORTED_ERR` exception. If either of the radii are negative throws an `INDEX_SIZE_ERR` exception.

The **`addColorStop(`*`offset, color`*`)`** method on the `CanvasGradient` interface adds a new stop to a gradient. If the *offset* is less than 0, greater than 1, infinite, or NaN, then an `INDEX_SIZE_ERR` exception must be raised. If the *color* cannot be parsed as a CSS color, then a `SYNTAX_ERR` exception must be raised. Otherwise, the gradient must have a new stop placed, at offset *offset* relative to the whole gradient, and with the color obtained by parsing *color* as a CSS <color> value. If multiple stops are added at the same offset on a gradient, they must be placed in the order added, with the first one closest to the start of the gradient, and each subsequent one infinitesimally further along towards the end point (in effect causing all but the first and last stop added at each point to be ignored).

The **`createLinearGradient(`*`x0, y0, x1, y1`*`)`** method takes four arguments that represent the start point (*x0*, *y0*) and end point (*x1*, *y1*) of the gradient. If any of the arguments to `createLinearGradient()` are infinite or NaN, the method must raise a `NOT_SUPPORTED_ERR` exception. Otherwise, the method must return a linear `CanvasGradient` initialized with the specified line.

Linear gradients must be rendered such that all points on a line perpendicular to the line that crosses the start and end points have the color at the point where those two lines cross (with the colors coming from the interpolation and extrapolation described above). The points in the linear gradient must be transformed as described by the current transformation matrix when rendering.

If *x0* = *x1* and *y0* = *y1*, then the linear gradient must paint nothing.

The **`createRadialGradient(`*`x0, y0, r0, x1, y1, r1`*`)`** method takes six arguments, the first three representing the start circle with origin (*x0*, *y0*) and radius *r0*, and the last three representing the end circle with origin (*x1*, *y1*) and radius *r1*. The values are in coordinate space units. If any of the arguments are infinite or NaN, a `NOT_SUPPORTED_ERR` exception must

be raised. If either of *r0* or *r1* are negative, an `INDEX_SIZE_ERR` exception must be raised. Otherwise, the method must return a radial `CanvasGradient` initialized with the two specified circles.

Radial gradients must be rendered by following these steps:

1. If $x_0 = x_1$ and $y_0 = y_1$ and $r_0 = r_1$, then the radial gradient must paint nothing. Abort these steps.

2. Let $x(\omega) = (x_1-x_0)\omega + x_0$

   Let $y(\omega) = (y_1-y_0)\omega + y_0$

   Let $r(\omega) = (r_1-r_0)\omega + r_0$

   Let the color at ω be the color at that position on the gradient (with the colors coming from the interpolation and extrapolation described above).

3. For all values of ω where $r(\omega) > 0$, starting with the value of ω nearest to positive infinity and ending with the value of ω nearest to negative infinity, draw the circumference of the circle with radius $r(\omega)$ at position $(x(\omega), y(\omega))$, with the color at ω, but only painting on the parts of the canvas that have not yet been painted on by earlier circles in this step for this rendering of the gradient.

Note: This effectively creates a cone, touched by the two circles defined in the creation of the gradient, with the part of the cone before the start circle (0.0) using the color of the first offset, the part of the cone after the end circle (1.0) using the color of the last offset, and areas outside the cone untouched by the gradient (transparent black).

Gradients must be painted only where the relevant stroking or filling effects requires that they be drawn.

The points in the radial gradient must be transformed as described by the current transformation matrix when rendering.

Patterns are represented by objects implementing the opaque `CanvasPattern` interface.

***pattern* = *context* . `createPattern`(*image*, *repetition*)**

Returns a `CanvasPattern` object that uses the given image and repeats in the direction(s) given by the *repetition* argument.

The allowed values for `repeat` are `repeat` (both directions), `repeat-x` (horizontal only), `repeat-y` (vertical only), and `no-repeat` (neither). If the *repetition* argument is empty or null, the value `repeat` is used.

If the first argument isn't an image, throws a `TYPE_MISMATCH_ERR` exception. If the image is not fully decoded yet, or has no image data, throws an `INVALID_STATE_ERR` exception. If the second argument isn't one of the allowed values, throws a `SYNTAX_ERR` exception.

To create objects of this type, the **`createPattern(`*`image, repetition`*`)`** method is used. The first argument gives the image to use as the pattern (either an `HTMLImageElement` or an `HTMLCanvasElement`). Modifying this image after calling the `createPattern()` method must not affect the pattern. The second argument must be a string with one of the following values: `repeat`, `repeat-x`, `repeat-y`, `no-repeat`. If the empty string or null is specified, `repeat` must be assumed. If an unrecognized value is given, then the user agent must raise a `SYNTAX_ERR` exception. User agents must recognize the four values described above exactly (e.g. they must not do case folding). The method must return a `CanvasPattern` object suitably initialized.

The *image* argument must be an instance of an `HTMLImageElement` or `HTMLCanvasElement`. If the *image* is of the wrong type or null, the implementation must raise a `TYPE_MISMATCH_ERR` exception.

If the *image* argument is an `HTMLImageElement` object whose `complete` attribute is false, then the implementation must raise an `INVALID_STATE_ERR` exception.

If the *image* argument is an `HTMLCanvasElement` object with either a horizontal dimension or a vertical dimension equal to zero, then the implementation must raise an `INVALID_STATE_ERR` exception.

Patterns must be painted so that the top left of the first image is anchored at the origin of the coordinate space, and images are then repeated horizontally to the left and right (if the `repeat-x` string was specified) or vertically up and down (if the `repeat-y` string was specified) or in all four directions all over the canvas (if the `repeat` string was specified). The images are not scaled by this process; one CSS pixel of the image must be painted on one coordinate space unit. Of course, patterns must actually be painted only where the stroking or filling effect requires that they be drawn, and are affected by the current transformation matrix.

When the `createPattern()` method is passed, as its *image* argument, an animated image, the poster frame of the animation, or the first frame of the animation if there is no poster frame, must be used.

4.8.11.1.5 Line styles

context **. `lineWidth` [= *value*]**

Returns the current line width.

Can be set, to change the line width. Values that are not finite values greater than zero are ignored.

context **. `lineCap` [= *value*]**

Returns the current line cap style.

Can be set, to change the line cap style.

The possible line cap styles are `butt`, `round`, and `square`. Other values are ignored.

context **. `lineJoin` [= *value*]**

Returns the current line join style.

Can be set, to change the line join style.

The possible line join styles are `bevel`, `round`, and `miter`. Other values are ignored.

context **.** **`miterLimit`** **[= *value*]**

Returns the current miter limit ratio.

Can be set, to change the miter limit ratio. Values that are not finite values greater than zero are ignored.

The **`lineWidth`** attribute gives the width of lines, in coordinate space units. On setting, zero, negative, infinite, and NaN values must be ignored, leaving the value unchanged.

When the context is created, the `lineWidth` attribute must initially have the value `1.0`.

The **`lineCap`** attribute defines the type of endings that UAs will place on the end of lines. The three valid values are `butt`, `round`, and `square`. The `butt` value means that the end of each line has a flat edge perpendicular to the direction of the line (and that no additional line cap is added). The `round` value means that a semi-circle with the diameter equal to the width of the line must then be added on to the end of the line. The `square` value means that a rectangle with the length of the line width and the width of half the line width, placed flat against the edge perpendicular to the direction of the line, must be added at the end of each line. On setting, any other value than the literal strings `butt`, `round`, and `square` must be ignored, leaving the value unchanged.

When the context is created, the `lineCap` attribute must initially have the value `butt`.

The **`lineJoin`** attribute defines the type of corners that UAs will place where two lines meet. The three valid values are `bevel`, `round`, and `miter`.

On setting, any other value than the literal strings `bevel`, `round`, and `miter` must be ignored, leaving the value unchanged.

When the context is created, the `lineJoin` attribute must initially have the value `miter`.

A join exists at any point in a subpath shared by two consecutive lines. When a subpath is closed, then a join also exists at its first point (equivalent to its last point) connecting the first and last lines in the subpath.

In addition to the point where the join occurs, two additional points are relevant to each join, one for each line: the two corners found half the line width away from the join point, one perpendicular to each line, each on the side furthest from the other line.

A filled triangle connecting these two opposite corners with a straight line, with the third point of the triangle being the join point, must be rendered at all joins. The `lineJoin` attribute controls whether anything else is rendered. The three aforementioned values have the following meanings:

The `bevel` value means that this is all that is rendered at joins.

The `round` value means that a filled arc connecting the two aforementioned corners of the join,

abutting (and not overlapping) the aforementioned triangle, with the diameter equal to the line width and the origin at the point of the join, must be rendered at joins.

The `miter` value means that a second filled triangle must (if it can given the miter length) be rendered at the join, with one line being the line between the two aforementioned corners, abutting the first triangle, and the other two being continuations of the outside edges of the two joining lines, as long as required to intersect without going over the miter length.

The miter length is the distance from the point where the lines touch on the inside of the join to the intersection of the line edges on the outside of the join. The miter limit ratio is the maximum allowed ratio of the miter length to half the line width. If the miter length would cause the miter limit ratio to be exceeded, this second triangle must not be rendered.

The miter limit ratio can be explicitly set using the **`miterLimit`** attribute. On setting, zero, negative, infinite, and NaN values must be ignored, leaving the value unchanged.

When the context is created, the `miterLimit` attribute must initially have the value `10.0`.

4.8.11.1.6 SHADOWS

All drawing operations are affected by the four global shadow attributes.

context **.** **`shadowColor`** **[=** ***value*** **]**

Returns the current shadow color.

Can be set, to change the shadow color. Values that cannot be parsed as CSS colors are ignored.

context **.** **`shadowOffsetX`** **[=** ***value*** **]**
context **.** **`shadowOffsetY`** **[=** ***value*** **]**

Returns the current shadow offset.

Can be set, to change the shadow offset. Values that are not finite numbers are ignored.

context **.** **`shadowBlur`** **[=** ***value*** **]**

Returns the current level of blur applied to shadows.

Can be set, to change the blur level. Values that are not finite numbers greater than or equal to zero are ignored.

The **`shadowColor`** attribute sets the color of the shadow.

When the context is created, the `shadowColor` attribute initially must be fully-transparent black.

On getting, the serialization of the color must be returned.

On setting, the new value must be parsed as a CSS <color> value and the color assigned. If the value is not a valid color, then it must be ignored, and the attribute must retain its previous

value. [CSS3COLOR]

The **`shadowOffsetX`** and **`shadowOffsetY`** attributes specify the distance that the shadow will be offset in the positive horizontal and positive vertical distance respectively. Their values are in coordinate space units. They are not affected by the current transformation matrix.

When the context is created, the shadow offset attributes must initially have the value `0`.

On getting, they must return their current value. On setting, the attribute being set must be set to the new value, except if the value is infinite or NaN, in which case the new value must be ignored.

The **`shadowBlur`** attribute specifies the size of the blurring effect. (The units do not map to coordinate space units, and are not affected by the current transformation matrix.)

When the context is created, the `shadowBlur` attribute must initially have the value `0`.

On getting, the attribute must return its current value. On setting the attribute must be set to the new value, except if the value is negative, infinite or NaN, in which case the new value must be ignored.

When shadows are drawn, they must be rendered as follows:

1. Let *A* be the source image for which a shadow is being created.
2. Let *B* be an infinite transparent black bitmap, with a coordinate space and an origin identical to *A*.
3. Copy the alpha channel of *A* to *B*, offset by `shadowOffsetX` in the positive *x* direction, and `shadowOffsetY` in the positive *y* direction.
4. If `shadowBlur` is greater than 0:
    1. If `shadowBlur` is less than 8, let σ be half the value of `shadowBlur`; otherwise, let σ be the square root of multiplying the value of `shadowBlur` by 2.
    2. Perform a 2D Gaussian Blur on *B*, using σ as the standard deviation.

    User agents may limit values of σ to an implementation-specific maximum value to avoid exceeding hardware limitations during the Gaussian blur operation.
5. Set the red, green, and blue components of every pixel in *B* to the red, green, and blue components (respectively) of the color of `shadowColor`.
6. Multiply the alpha component of every pixel in *B* by the alpha component of the color of `shadowColor`.
7. The shadow is in the bitmap *B*, and is rendered as part of the drawing model described below.

4.8.11.1.7 Simple shapes (rectangles)

There are three methods that immediately draw rectangles to the bitmap. They each take four

arguments; the first two give the *x* and *y* coordinates of the top left of the rectangle, and the second two give the width *w* and height *h* of the rectangle, respectively.

The current transformation matrix must be applied to the following four coordinates, which form the path that must then be closed to get the specified rectangle: (*x*, *y*), (*x*+*w*, *y*), (*x*+*w*, *y*+*h*), (*x*, *y*+*h*).

Shapes are painted without affecting the current path, and are subject to the clipping region, and, with the exception of `clearRect()`, also shadow effects, global alpha, and global composition operators.

context **. `clearRect`(*x*, *y*, *w*, *h*)**

Clears all pixels on the canvas in the given rectangle to transparent black.

context **. `fillRect`(*x*, *y*, *w*, *h*)**

Paints the given rectangle onto the canvas, using the current fill style.

context **. `strokeRect`(*x*, *y*, *w*, *h*)**

Paints the box that outlines the given rectangle onto the canvas, using the current stroke style.

The **`clearRect(x, y, w, h)`** method must clear the pixels in the specified rectangle that also intersect the current clipping region to a fully transparent black, erasing any previous image. If either height or width are zero, this method has no effect.

The **`fillRect(x, y, w, h)`** method must paint the specified rectangular area using the `fillStyle`. If either height or width are zero, this method has no effect.

The **`strokeRect(x, y, w, h)`** method must stroke the specified rectangle's path using the `strokeStyle`, `lineWidth`, `lineJoin`, and (if appropriate) `miterLimit` attributes. If both height and width are zero, this method has no effect, since there is no path to stroke (it's a point). If only one of the two is zero, then the method will draw a line instead (the path for the outline is just a straight line along the non-zero dimension).

4.8.11.1.8 Complex shapes (paths)

The context always has a current path. There is only one current path, it is not part of the drawing state.

A **path** has a list of zero or more subpaths. Each subpath consists of a list of one or more points, connected by straight or curved lines, and a flag indicating whether the subpath is closed or not. A closed subpath is one where the last point of the subpath is connected to the first point of the subpath by a straight line. Subpaths with fewer than two points are ignored when painting the path.

context **.** **`beginPath`****()**

Resets the current path.

context **.** **`moveTo`****(*x*, *y*)**

Creates a new subpath with the given point.

context **.** **`closePath`****()**

Marks the current subpath as closed, and starts a new subpath with a point the same as the start and end of the newly closed subpath.

context **.** **`lineTo`****(*x*, *y*)**

Adds the given point to the current subpath, connected to the previous one by a straight line.

context **.** **`quadraticCurveTo`****(*cpx*, *cpy*, *x*, *y*)**

Adds the given point to the current path, connected to the previous one by a quadratic Bézier curve with the given control point.

context **.** **`bezierCurveTo`****(*cpx*, *cpy*, *x*, *y*)**

Adds the given point to the current path, connected to the previous one by a cubic Bézier curve with the given control points.

context **.** **`arcTo`****(*x1*, *y1*, *x2*, *y2*, *radius*)**

Adds a point to the current path, connected to the previous one by a straight line, then adds a second point to the current path, connected to the previous one by an arc whose properties are described by the arguments.

Throws an `INDEX_SIZE_ERR` exception if the given radius is negative.

context **.** **`arc`****(*x*, *y*, *radius*, *startAngle*, *endAngle*, *anticlockwise*)**

Adds points to the subpath such that the arc described by the circumference of the circle described by the arguments, starting at the given start angle and ending at the given end angle, going in the given direction, is added the the path, connected to the previous point by a straight line.

Throws an `INDEX_SIZE_ERR` exception if the given radius is negative.

context **.** **`rect`****(*x*, *y*, *w*, *h*)**

Adds a new closed subpath to the path, representing the given rectangle.

context **.** **`fill`****()**

Fills the subpaths with the current fill style.

context **.** **`stroke`****()**

Strokes the subpaths with the current stroke style.

context **.** **`clip`****()**

Further constrains the clipping region to the given path.

context **`.isPointInPath(`*x*, *y*`)`**

Returns true if the given point is in the current path.

Initially, the context's path must have zero subpaths.

The points and lines added to the path by these methods must be transformed according to the current transformation matrix as they are added.

The **`beginPath()`** method must empty the list of subpaths so that the context once again has zero subpaths.

The **`moveTo(`*x*`, `*y*`)`** method must create a new subpath with the specified point as its first (and only) point.

The **`closePath()`** method must do nothing if the context has no subpaths. Otherwise, it must mark the last subpath as closed, create a new subpath whose first point is the same as the previous subpath's first point, and finally add this new subpath to the path. (If the last subpath had more than one point in its list of points, then this is equivalent to adding a straight line connecting the last point back to the first point, thus "closing" the shape, and then repeating the last `moveTo()` call.)

New points and the lines connecting them are added to subpaths using the methods described below. In all cases, the methods only modify the last subpath in the context's paths.

The **`lineTo(`*x*`, `*y*`)`** method must do nothing if the context has no subpaths. Otherwise, it must connect the last point in the subpath to the given point (*x*, *y*) using a straight line, and must then add the given point (*x*, *y*) to the subpath.

The **`quadraticCurveTo(`*cpx*`, `*cpy*`, `*x*`, `*y*`)`** method must do nothing if the context has no subpaths. Otherwise it must connect the last point in the subpath to the given point (*x*, *y*) using a quadratic Bézier curve with control point (*cpx*, *cpy*), and must then add the given point (*x*, *y*) to the subpath. [BEZIER]

The **`bezierCurveTo(`*cp1x*`, `*cp1y*`, `*cp2x*`, `*cp2y*`, `*x*`, `*y*`)`** method must do nothing if the context has no subpaths. Otherwise, it must connect the last point in the subpath to the given point (*x*, *y*) using a cubic Bézier curve with control points (*cp1x*, *cp1y*) and (*cp2x*, *cp2y*). Then, it must add the point (*x*, *y*) to the subpath. [BEZIER]

The **`arcTo(`*x1*`, `*y1*`, `*x2*`, `*y2*`, `*radius*`)`** method must do nothing if the context has no subpaths. If the context *does* have a subpath, then the behavior depends on the arguments and the last point in the subpath.

Negative values for *radius* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

Let the point (*x0*, *y0*) be the last point in the subpath.

If the point (*x0*, *y0*) is equal to the point (*x1*, *y1*), or if the point (*x1*, *y1*) is equal to the point (*x2*, *y2*), or if the radius *radius* is zero, then the method must add the point (*x1*, *y1*) to the

subpath, and connect that point to the previous point (*x0*, *y0*) by a straight line.

Otherwise, if the points (*x0*, *y0*), (*x1*, *y1*), and (*x2*, *y2*) all lie on a single straight line, then: if the direction from (*x0*, *y0*) to (*x1*, *y1*) is the same as the direction from (*x1*, *y1*) to (*x2*, *y2*), then the method must add the point (*x1*, *y1*) to the subpath, and connect that point to the previous point (*x0*, *y0*) by a straight line; otherwise, the direction from (*x0*, *y0*) to (*x1*, *y1*) is the opposite of the direction from (*x1*, *y1*) to (*x2*, *y2*), and the method must add a point (x_∞, y_∞) to the subpath, and connect that point to the previous point (*x0*, *y0*) by a straight line, where (x_∞, y_∞) is the point that is infinitely far away from (*x1*, *y1*), that lies on the same line as (*x0*, *y0*), (*x1*, *y1*), and (*x2*, *y2*), and that is on the same side of (*x1*, *y1*) on that line as (*x2*, *y2*).

Otherwise, let *The Arc* be the shortest arc given by circumference of the circle that has radius *radius*, and that has one point tangent to the half-infinite line that crosses the point (*x0*, *y0*) and ends at the point (*x1*, *y1*), and that has a different point tangent to the half-infinite line that ends at the point (*x1*, *y1*) and crosses the point (*x2*, *y2*). The points at which this circle touches these two lines are called the start and end tangent points respectively.

The method must connect the point (*x0*, *y0*) to the start tangent point by a straight line, adding the start tangent point to the subpath, and then must connect the start tangent point to the end tangent point by *The Arc*, adding the end tangent point to the subpath.

The **`arc(x, y, radius, startAngle, endAngle, anticlockwise)`** method draws an arc. If the context has any subpaths, then the method must add a straight line from the last point in the subpath to the start point of the arc. In any case, it must draw the arc between the start point of the arc and the end point of the arc, and add the start and end points of the arc to the subpath. The arc and its start and end points are defined as follows:

Consider a circle that has its origin at (*x*, *y*) and that has radius *radius*. The points at *startAngle* and *endAngle* along this circle's circumference, measured in radians clockwise from the positive x-axis, are the start and end points respectively.

If the *anticlockwise* argument is false and *endAngle*-*startAngle* is equal to or greater than 2π, or, if the *anticlockwise* argument is *true* and *startAngle*-*endAngle* is equal to or greater than 2π, then the arc is the whole circumference of this circle.

Otherwise, the arc is the path along the circumference of this circle from the start point to the end point, going anti-clockwise if the *anticlockwise* argument is true, and clockwise otherwise. Since the points are on the circle, as opposed to being simply angles from zero, the arc can never cover an angle greater than 2π radians. If the two points are the same, or if the radius is zero, then the arc is defined as being of zero length in both directions.

Negative values for *radius* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`rect(x, y, w, h)`** method must create a new subpath containing just the four points (*x*, *y*), (*x*+*w*, *y*), (*x*+*w*, *y*+*h*), (*x*, *y*+*h*), with those four points connected by straight lines, and must then mark the subpath as closed. It must then create a new subpath with the point (*x*, *y*) as the only point in the subpath.

The **`fill()`** method must fill all the subpaths of the current path, using `fillStyle`, and using the non-zero winding number rule. Open subpaths must be implicitly closed when being filled (without affecting the actual subpaths).

Note: Thus, if two overlapping but otherwise independent subpaths have opposite windings, they cancel out and result in no fill. If they have the same winding, that area just gets painted once.

The **`stroke()`** method must calculate the strokes of all the subpaths of the current path, using the `lineWidth`, `lineCap`, `lineJoin`, and (if appropriate) `miterLimit` attributes, and then fill the combined stroke area using the `strokeStyle` attribute.

Note: Since the subpaths are all stroked as one, overlapping parts of the paths in one stroke operation are treated as if their union was what was painted.

Paths, when filled or stroked, must be painted without affecting the current path, and must be subject to shadow effects, global alpha, the clipping region, and global composition operators. (Transformations affect the path when the path is created, not when it is painted, though the stroke *style* is still affected by the transformation during painting.)

Zero-length line segments must be pruned before stroking a path. Empty subpaths must be ignored.

The **`clip()`** method must create a new **clipping region** by calculating the intersection of the current clipping region and the area described by the current path, using the non-zero winding number rule. Open subpaths must be implicitly closed when computing the clipping region, without affecting the actual subpaths. The new clipping region replaces the current clipping region.

When the context is initialized, the clipping region must be set to the rectangle with the top left corner at (0,0) and the width and height of the coordinate space.

The **`isPointInPath(`*`x`*`, `*`y`*`)`** method must return true if the point given by the *x* and *y* coordinates passed to the method, when treated as coordinates in the canvas coordinate space unaffected by the current transformation, is inside the current path as determined by the non-zero winding number rule; and must return false otherwise. Points on the path itself are considered to be inside the path. If either of the arguments is infinite or NaN, then the method must return false.

4.8.11.1.9 TEXT

context **`.font`** **[= *value*]**

Returns the current font settings.

Can be set, to change the font. The syntax is the same as for the CSS 'font' property; values that cannot be parsed as CSS font values are ignored.

Relative keywords and lengths are computed relative to the default font, 10px sans-serif.

context **`.textAlign`** **[= *value*]**

Returns the current text alignment settings.

Can be set, to change the alignment. The possible values are `start`, `end`, `left`, `right`, and `center`. The default is `start`. Other values are ignored.

context **.** **`textBaseline`** **[=** ***value*** **]**

Returns the current baseline alignment settings.

Can be set, to change the baseline alignment. The possible values and their meanings are given below. The default is `alphabetic`. Other values are ignored.

context **.** **`fillText`****(*****text*****,** ***x*****,** ***y*** **[,** ***maxWidth*** **])**
context **.** **`strokeText`****(*****text*****,** ***x*****,** ***y*** **[,** ***maxWidth*** **])**

Fills or strokes (respectively) the given text at the given position. If a maximum width is provided, the text will be scaled to fit that width if necessary.

metrics **=** ***context*** **.** **`measureText`****(*****text*****)**

Returns a `TextMetrics` object with the metrics of the given text in the current font.

metrics **.** **`width`**

Returns the advance width of the text that was passed to the `measureText()` method.

The **`font`** DOM attribute, on setting, must be parsed the same way as the 'font' property of CSS (but without supporting property-independent stylesheet syntax like 'inherit'), and the resulting font must be assigned to the context, with the 'line-height' component forced to 'normal'. If the new value is syntactically incorrect, then it must be ignored, without assigning a new font value. [CSS]

Font names must be interpreted in the context of the `canvas` element's stylesheets; any fonts embedded using `@font-face` must therefore be available. [CSSWEBFONTS]

Only vector fonts should be used by the user agent; if a user agent were to use bitmap fonts then transformations would likely make the font look very ugly.

On getting, the `font` attribute must return the serialized form of the current font of the context. [CSSOM]

When the context is created, the font of the context must be set to 10px sans-serif. When the 'font-size' component is set to lengths using percentages, 'em' or 'ex' units, or the 'larger' or 'smaller' keywords, these must be interpreted relative to the computed value of the 'font-size' property of the corresponding `canvas` element at the time that the attribute is set. When the 'font-weight' component is set to the relative values 'bolder' and 'lighter', these must be interpreted relative to the computed value of the 'font-weight' property of the corresponding `canvas` element at the time that the attribute is set. If the computed values are undefined for a particular case (e.g. because the `canvas` element is not in a document), then the relative keywords must be interpreted relative to the normal-weight 10px sans-serif default.

The **`textAlign`** DOM attribute, on getting, must return the current value. On setting, if the value is one of `start`, `end`, `left`, `right`, or `center`, then the value must be changed to the new value. Otherwise, the new value must be ignored. When the context is created, the `textAlign`

attribute must initially have the value `start`.

The **`textBaseline`** DOM attribute, on getting, must return the current value. On setting, if the value is one of `top`, `hanging`, `middle`, `alphabetic`, `ideographic`, or `bottom`, then the value must be changed to the new value. Otherwise, the new value must be ignored. When the context is created, the `textBaseline` attribute must initially have the value `alphabetic`.

The `textBaseline` attribute's allowed keywords correspond to alignment points in the font:

The keywords map to these alignment points as follows:

`top`
: The top of the em square

`hanging`
: The hanging baseline

`middle`
: The middle of the em square

`alphabetic`
: The alphabetic baseline

`ideographic`
: The ideographic baseline

`bottom`
: The bottom of the em square

The **`fillText()`** and **`strokeText()`** methods take three or four arguments, *text*, *x*, *y*, and optionally *maxWidth*, and render the given *text* at the given (*x*, *y*) coordinates ensuring that the text isn't wider than *maxWidth* if specified, using the current `font`, `textAlign`, and `textBaseline` values. Specifically, when the methods are called, the user agent must run the following steps:

1. Let *font* be the current font of the context, as given by the `font` attribute.

2. Replace all the space characters in *text* with U+0020 SPACE characters.

3. Form a hypothetical infinitely wide CSS line box containing a single inline box containing

the text *text*, with all the properties at their initial values except the 'font' property of the inline box set to *font* and the 'direction' property of the inline box set to the directionality of the `canvas` element. [CSS]

4. If the *maxWidth* argument was specified and the hypothetical width of the inline box in the hypothetical line box is greater than *maxWidth* CSS pixels, then change *font* to have a more condensed font (if one is available or if a reasonably readable one can be synthesized by applying a horizontal scale factor to the font) or a smaller font, and return to the previous step.

5. Let the *anchor point* be a point on the inline box, determined by the `textAlign` and `textBaseline` values, as follows:

   Horizontal position:

   If `textAlign` is `left`
   If `textAlign` is `start` and the directionality of the `canvas` element is 'ltr'
   If `textAlign` is `end` and the directionality of the `canvas` element is 'rtl'
   Let the *anchor point*'s horizontal position be the left edge of the inline box.
   If `textAlign` is `right`
   If `textAlign` is `end` and the directionality of the `canvas` element is 'ltr'
   If `textAlign` is `start` and the directionality of the `canvas` element is 'rtl'
   Let the *anchor point*'s horizontal position be the right edge of the inline box.
   If `textAlign` is `center`
   Let the *anchor point*'s horizontal position be half way between the left and right edges of the inline box.

   Vertical position:

   If `textBaseline` is `top`
   Let the *anchor point*'s vertical position be the top of the em box of the first available font of the inline box.
   If `textBaseline` is `hanging`
   Let the *anchor point*'s vertical position be the hanging baseline of the first available font of the inline box.
   If `textBaseline` is `middle`
   Let the *anchor point*'s vertical position be half way between the bottom and the top of the em box of the first available font of the inline box.
   If `textBaseline` is `alphabetic`
   Let the *anchor point*'s vertical position be the alphabetic baseline of the first available font of the inline box.
   If `textBaseline` is `ideographic`
   Let the *anchor point*'s vertical position be the ideographic baseline of the first available font of the inline box.
   If `textBaseline` is `bottom`
   Let the *anchor point*'s vertical position be the bottom of the em box of the first available font of the inline box.

6. Paint the hypothetical inline box as the shape given by the text's glyphs, as transformed by the current transformation matrix, and anchored and sized so that before applying the current transformation matrix, the *anchor point* is at (*x*, *y*) and each CSS pixel is mapped to one coordinate space unit.

For `fillText()` `fillStyle` must be applied to the glyphs and `strokeStyle` must be ignored. For `strokeText()` the reverse holds and `strokeStyle` must be applied to the glyph outlines and `fillStyle` must be ignored.

Text is painted without affecting the current path, and is subject to shadow effects, global alpha, the clipping region, and global composition operators.

The **`measureText()`** method takes one argument, *text*. When the method is invoked, the user agent must replace all the space characters in *text* with U+0020 SPACE characters, and then must form a hypothetical infinitely wide CSS line box containing a single inline box containing the text *text*, with all the properties at their initial values except the 'font' property of the inline element set to the current font of the context, as given by the `font` attribute, and must then return a new `TextMetrics` object with its `width` attribute set to the width of that inline box, in CSS pixels. [CSS]

The `TextMetrics` interface is used for the objects returned from `measureText()`. It has one attribute, **`width`**, which is set by the `measureText()` method.

Note: Glyphs rendered using `fillText()` and `strokeText()` can spill out of the box given by the font size (the em square size) and the width returned by `measureText()` (the text width). This version of the specification does not provide a way to obtain the bounding box dimensions of the text. If the text is to be rendered and removed, care needs to be taken to replace the entire area of the canvas that the clipping region covers, not just the box given by the em square height and measured text width.

Note: A future version of the 2D context API may provide a way to render fragments of documents, rendered using CSS, straight to the canvas. This would be provided in preference to a dedicated way of doing multiline layout.

4.8.11.1.10 IMAGES

To draw images onto the canvas, the **`drawImage`** method can be used.

This method can be invoked with three different sets of argments:

- `drawImage(`*`image`*`, `*`dx`*`, `*`dy`*`)`
- `drawImage(`*`image`*`, `*`dx`*`, `*`dy`*`, `*`dw`*`, `*`dh`*`)`
- `drawImage(`*`image`*`, `*`sx`*`, `*`sy`*`, `*`sw`*`, `*`sh`*`, `*`dx`*`, `*`dy`*`, `*`dw`*`, `*`dh`*`)`

Each of those three can take either an `HTMLImageElement`, an `HTMLCanvasElement`, or an `HTMLVideoElement` for the *image* argument.

***context* . `drawImage`(*image*, *dx*, *dy*)**
***context* . `drawImage`(*image*, *dx*, *dy*, *dw*, *dh*)**
***context* . `drawImage`(*image*, *sx*, *sy*, *sw*, *sh*, *dx*, *dy*, *dw*, *dh*)**

Draws the given image onto the canvas. The arguments are interpreted as per the diagram below.

If not specified, the *dw* and *dh* arguments must default to the values of *sw* and *sh*, interpreted such that one CSS pixel in the image is treated as one unit in the canvas coordinate space. If the *sx*, *sy*, *sw*, and *sh* arguments are omitted, they must default to 0, 0, the image's intrinsic width in image pixels, and the image's intrinsic height in image pixels, respectively.

The *image* argument must be an instance of an `HTMLImageElement`, `HTMLCanvasElement`, or `HTMLVideoElement`. If the *image* is of the wrong type or null, the implementation must raise a `TYPE_MISMATCH_ERR` exception.

If the *image* argument is an `HTMLImageElement` object whose `complete` attribute is false, then the implementation must raise an `INVALID_STATE_ERR` exception.

If the *image* argument is an `HTMLVideoElement` object whose `readyState` attribute is either `HAVE_NOTHING` or `HAVE_METADATA`, then the implementation must raise an `INVALID_STATE_ERR` exception.

The source rectangle is the rectangle whose corners are the four points (*sx*, *sy*), (*sx*+*sw*, *sy*), (*sx*+*sw*, *sy*+*sh*), (*sx*, *sy*+*sh*).

If the source rectangle is not entirely within the source image, or if one of the *sw* or *sh* arguments is zero, the implementation must raise an `INDEX_SIZE_ERR` exception.

The destination rectangle is the rectangle whose corners are the four points (*dx*, *dy*), (*dx*+*dw*, *dy*), (*dx*+*dw*, *dy*+*dh*), (*dx*, *dy*+*dh*).

When `drawImage()` is invoked, the region of the image specified by the source rectangle must be painted on the region of the canvas specified by the destination rectangle, after applying the current transformation matrix to the points of the destination rectangle.

Note: When a canvas is drawn onto itself, the drawing model requires the source to be copied before the image is drawn back onto the canvas, so it is possible to copy parts of a canvas onto overlapping parts of itself.

When the `drawImage()` method is passed, as its *image* argument, an `HTMLImageElement` representing an animated image, the poster frame of the animation, or the first frame of the animation if there is no poster frame, must be used.

When the *image* argument is an `HTMLVideoElement`, then the frame at the current playback position must be used as the source image.

Images are painted without affecting the current path, and are subject to shadow effects, global alpha, the clipping region, and global composition operators.

4.8.11.1.11 Pixel manipulation

***imagedata* = *context* . `createImageData`(*sw*, *sh*)**

Returns an `ImageData` object with the given dimensions in CSS pixels (which might map to a different number of actual device pixels exposed by the object itself). All the pixels in the returned object are transparent black.

***imagedata* = *context* . `getImageData`(*sx*, *sy*, *sw*, *sh*)**

Returns an `ImageData` object containing the image data for the given rectangle of the canvas.

Throws a `NOT_SUPPORTED_ERR` exception if any of the arguments are not finite. Throws an `INDEX_SIZE_ERR` exception if the either of the width or height arguments are zero.

***imagedata* . `width`**
***imagedata* . `height`**

Returns the actual dimensions of the data in the `ImageData` object, in device pixels.

***imagedata* . `data`**

Returns the one-dimensional array containing the data.

***context* . `putImageData`(*imagedata*, *dx*, *dy* [, *dirtyX*, *dirtyY*, *dirtyWidth*, *dirtyHeight*])**

Paints the data from the given `ImageData` object onto the canvas. If a dirty rectangle is provided, only the pixels from that rectangle are painted.

If the first argument isn't an `ImageData` object, throws a `TYPE_MISMATCH_ERR` exception. Throws a `NOT_SUPPORTED_ERR` exception if any of the other arguments are not finite.

The **`createImageData(sw, sh)`** method must return an `ImageData` object representing a rectangle with a width in CSS pixels equal to the absolute magnitude of *sw* and a height in CSS pixels equal to the absolute magnitude of *sh*, filled with transparent black.

The **`getImageData(sx, sy, sw, sh)`** method must return an `ImageData` object representing the underlying pixel data for the area of the canvas denoted by the rectangle whose corners are the four points (*sx*, *sy*), (*sx*+*sw*, *sy*), (*sx*+*sw*, *sy*+*sh*), (*sx*, *sy*+*sh*), in canvas coordinate space units. Pixels outside the canvas must be returned as transparent black. Pixels must be returned as non-premultiplied alpha values.

If any of the arguments to `createImageData()` or `getImageData()` are infinite or NaN, the method must instead raise a `NOT_SUPPORTED_ERR` exception. If either the *sw* or *sh* arguments are zero, the method must instead raise an `INDEX_SIZE_ERR` exception.

`ImageData` objects must be initialized so that their **`width`** attribute is set to *w*, the number of physical device pixels per row in the image data, their **`height`** attribute is set to *h*, the number of rows in the image data, and their **`data`** attribute is initialized to a `CanvasPixelArray` object holding the image data. At least one pixel's worth of image data must be returned.

The `CanvasPixelArray` object provides ordered, indexed access to the color components of each pixel of the image data. The data must be represented in left-to-right order, row by row top to bottom, starting with the top left, with each pixel's red, green, blue, and alpha components being given in that order for each pixel. Each component of each device pixel represented in this array must be in the range 0..255, representing the 8 bit value for that component. The components must be assigned consecutive indices starting with 0 for the top left pixel's red component.

The `CanvasPixelArray` object thus represents *h*×*w*×4 integers. The **`length`** attribute of a `CanvasPixelArray` object must return this number.

The object's indices of the supported indexed properties are the numbers in the range 0 .. *h*×*w*×4-1.

When a `CanvasPixelArray` object is **indexed to retrieve an indexed property** *index*, the value returned must be the value of the *index*th component in the array.

When a `CanvasPixelArray` object is **indexed to modify an indexed property** *index* with value *value*, the value of the *index*th component in the array must be set to *value*. JS `undefined` values must be converted to zero. Other values must first be converted to numbers using JavaScript's ToNumber algorithm, and if the result is a NaN value, then the value must be converted to zero. If the result is less than 0, it must be clamped to zero. If the result is more than 255, it must be clamped to 255. If the number is not an integer, it should be rounded to the nearest integer using the IEEE 754r *convertToIntegerTiesToEven* rounding mode. [ECMA262] [IEEE754R]

> ***Note: The width and height (w and h) might be different from the sw and sh arguments to the above methods, e.g. if the canvas is backed by a high-resolution bitmap, or if the sw and sh arguments are negative.***

The **`putImageData(`*`imagedata`*`, `*`dx`*`, `*`dy`*`, `*`dirtyX`*`, `*`dirtyY`*`, `*`dirtyWidth`*`, `*`dirtyHeight`*`)`** method writes data from `ImageData` structures back to the canvas.

If any of the arguments to the method are infinite or NaN, the method must raise a `NOT_SUPPORTED_ERR` exception.

If the first argument to the method is null or not an `ImageData` object then the `putImageData()` method must raise a `TYPE_MISMATCH_ERR` exception.

When the last four arguments are omitted, they must be assumed to have the values 0, 0, the `width` member of the *imagedata* structure, and the `height` member of the *imagedata* structure, respectively.

When invoked with arguments that do not, per the last few paragraphs, cause an exception to be raised, the `putImageData()` method must act as follows:

1. Let dx_{device} be the x-coordinate of the device pixel in the underlying pixel data of the canvas corresponding to the *dx* coordinate in the canvas coordinate space.

Let dy_{device} be the y-coordinate of the device pixel in the underlying pixel data of the canvas corresponding to the *dy* coordinate in the canvas coordinate space.

2. If *dirtyWidth* is negative, let *dirtyX* be *dirtyX*+*dirtyWidth*, and let *dirtyWidth* be equal to the absolute magnitude of *dirtyWidth*.

   If *dirtyHeight* is negative, let *dirtyY* be *dirtyY*+*dirtyHeight*, and let *dirtyHeight* be equal to the absolute magnitude of *dirtyHeight*.

3. If *dirtyX* is negative, let *dirtyWidth* be *dirtyWidth*+*dirtyX*, and let *dirtyX* be zero.

   If *dirtyY* is negative, let *dirtyHeight* be *dirtyHeight*+*dirtyY*, and let *dirtyY* be zero.

4. If *dirtyX*+*dirtyWidth* is greater than the `width` attribute of the *imagedata* argument, let *dirtyWidth* be the value of that `width` attribute, minus the value of *dirtyX*.

   If *dirtyY*+*dirtyHeight* is greater than the `height` attribute of the *imagedata* argument, let *dirtyHeight* be the value of that `height` attribute, minus the value of *dirtyY*.

5. If, after those changes, either *dirtyWidth* or *dirtyHeight* is negative or zero, stop these steps without affecting the canvas.

6. Otherwise, for all integer values of *x* and *y* where *dirtyX* ≤ *x* < *dirtyX*+*dirtyWidth* and *dirtyY* ≤ *y* < *dirtyY*+*dirtyHeight*, copy the four channels of the pixel with coordinate (*x*, *y*) in the *imagedata* data structure to the pixel with coordinate (dx_{device}+*x*, dy_{device}+*y*) in the underlying pixel data of the canvas.

The handling of pixel rounding when the specified coordinates do not exactly map to the device coordinate space is not defined by this specification, except that the following must result in no visible changes to the rendering:

```
context.putImageData(context.getImageData(x, y, w, h), x, y);
```

...for any value of *x*, *y*, *w*, and *h*, and the following two calls:

```
context.createImageData(w, h);
context.getImageData(0, 0, w, h);
```

...must return `ImageData` objects with the same dimensions, for any value of *w* and *h*. In other words, while user agents may round the arguments of these methods so that they map to device pixel boundaries, any rounding performed must be performed consistently for all of the `createImageData()`, `getImageData()` and `putImageData()` operations.

The current path, transformation matrix, shadow attributes, global alpha, the clipping region, and global composition operator must not affect the `getImageData()` and `putImageData()` methods.

> The data returned by `getImageData()` is at the resolution of the canvas backing store, which is likely to not be one device pixel to each CSS pixel if the display used is a high resolution display.
>
> In the following example, the script generates an `ImageData` object so that it can draw onto it.
>
> ```
> // canvas is a reference to a <canvas> element
> ```

```
var context = canvas.getContext('2d');

// create a blank slate
var data = context.createImageData(canvas.width, canvas.height);

// create some plasma
FillPlasma(data, 'green'); // green plasma

// add a cloud to the plasma
AddCloud(data, data.width/2, data.height/2); // put a cloud in the middle

// paint the plasma+cloud on the canvas
context.putImageData(data, 0, 0);

// support methods
function FillPlasma(data, color) { ... }
function AddCloud(data, x, y) { ... }
```

Here is an example of using `getImageData()` and `putImageData()` to implement an edge detection filter.

```
<!DOCTYPE HTML>
<html>
 <head>
  <title>Edge detection demo</title>
  <script>
   var image = new Image();
   function init() {
     image.onload = demo;
     image.src = "image.jpeg";
   }
   function demo() {
     var canvas = document.getElementsByTagName('canvas')[0];
     var context = canvas.getContext('2d');

     // draw the image onto the canvas
     context.drawImage(image, 0, 0);

     // get the image data to manipulate
     var input = context.getImageData(0, 0, canvas.width, canvas.height);

     // get an empty slate to put the data into
     var output = context.createImageData(canvas.width, canvas.height);

     // alias some variables for convenience
     // notice that we are using input.width and input.height here
     // as they might not be the same as canvas.width and canvas.height
     // (in particular, they might be different on high-res displays)
     var w = input.width, h = input.height;
     var inputData = input.data;
     var outputData = output.data;

     // edge detection
     for (var y = 1; y < h-1; y += 1) {
       for (var x = 1; x < w-1; x += 1) {
         for (var c = 0; c < 3; c += 1) {
           var i = (y*w + x)*4 + c;
           outputData[i] = 127 + -inputData[i - w*4 - 4] -   inputData[i - w*
                                 -inputData[i - 4]       + 8*inputData[i]
                                 -inputData[i + w*4 - 4] -   inputData[i + w*
         }
         outputData[(y*w + x)*4 + 3] = 255; // alpha
       }
     }

     // put the image data back after manipulation
     context.putImageData(output, 0, 0);
   }
  </script>
```

```
 </head>
 <body onload="init()">
  <canvas></canvas>
 </body>
</html>
```

4.8.11.1.12 Drawing model

When a shape or image is painted, user agents must follow these steps, in the order given (or act as if they do):

1. Render the shape or image, creating image *A*, as described in the previous sections. For shapes, the current fill, stroke, and line styles must be honored, and the stroke must itself also be subjected to the current transformation matrix.
2. Render the shadow from image *A*, using the current shadow styles, creating image *B*.
3. Multiply the alpha component of every pixel in *B* by `globalAlpha`.
4. Within the clipping region, composite *B* over the current canvas bitmap using the current composition operator.
5. Multiply the alpha component of every pixel in *A* by `globalAlpha`.
6. Within the clipping region, composite *A* over the current canvas bitmap using the current composition operator.

4.8.11.2 Color spaces and color correction

The `canvas` APIs must perform color correction at only two points: when rendering images with their own gamma correction and color space information onto the canvas, to convert the image to the color space used by the canvas (e.g. using the `drawImage()` method with an `HTMLImageElement` object), and when rendering the actual canvas bitmap to the output device.

> ***Note: Thus, in the 2D context, colors used to draw shapes onto the canvas will exactly match colors obtained through the `getImageData()` method.***

The `toDataURL()` method must not include color space information in the resource returned. Where the output format allows it, the color of pixels in resources created by `toDataURL()` must match those returned by the `getImageData()` method.

In user agents that support CSS, the color space used by a `canvas` element must match the color space used for processing any colors for that element in CSS.

The gamma correction and color space information of images must be handled in such a way that an image rendered directly using an `img` element would use the same colors as one painted on a `canvas` element that is then itself rendered. Furthermore, the rendering of images that have no color correction information (such as those returned by the `toDataURL()` method) must be rendered with no color correction.

> ***Note: Thus, in the 2D context, calling the `drawImage()` method to render the***

output of the `toDataURL()` method to the canvas, given the appropriate dimensions, has no visible effect.

4.8.11.3 Security with `canvas` elements

Information leakage can occur if scripts from one origin can access information (e.g. read pixels) from images from another origin (one that isn't the same).

To mitigate this, `canvas` elements are defined to have a flag indicating whether they are *origin-clean*. All `canvas` elements must start with their *origin-clean* set to true. The flag must be set to false if any of the following actions occur:

- The element's 2D context's `drawImage()` method is called with an `HTMLImageElement` whose origin is not the same as that of the `Document` object that owns the `canvas` element.
- The element's 2D context's `drawImage()` method is called with an `HTMLCanvasElement` whose *origin-clean* flag is false.
- The element's 2D context's `fillStyle` attribute is set to a `CanvasPattern` object that was created from an `HTMLImageElement` whose origin was not the same as that of the `Document` object that owns the `canvas` element when the pattern was created.
- The element's 2D context's `fillStyle` attribute is set to a `CanvasPattern` object that was created from an `HTMLCanvasElement` whose *origin-clean* flag was false when the pattern was created.
- The element's 2D context's `strokeStyle` attribute is set to a `CanvasPattern` object that was created from an `HTMLImageElement` whose origin was not the same as that of the `Document` object that owns the `canvas` element when the pattern was created.
- The element's 2D context's `strokeStyle` attribute is set to a `CanvasPattern` object that was created from an `HTMLCanvasElement` whose *origin-clean* flag was false when the pattern was created.

Whenever the `toDataURL()` method of a `canvas` element whose *origin-clean* flag is set to false is called, the method must raise a `SECURITY_ERR` exception.

Whenever the `getImageData()` method of the 2D context of a `canvas` element whose *origin-clean* flag is set to false is called with otherwise correct arguments, the method must raise a `SECURITY_ERR` exception.

Note: Even resetting the canvas state by changing its `width` or `height` attributes doesn't reset the origin-clean flag.

4.8.12 The `map` element

Categories

Flow content.

Contexts in which this element may be used:
Where flow content is expected.
Content model:
Flow content.
Content attributes:
Global attributes
name
DOM interface:

```
interface HTMLMapElement : HTMLElement {
           attribute DOMString name;
  readonly attribute HTMLCollection areas;
  readonly attribute HTMLCollection images;
};
```

The map element, in conjunction with any area element descendants, defines an image map. The element represents its children.

The **name** attribute gives the map a name so that it can be referenced. The attribute must be present and must have a non-empty value with no space characters. If the id attribute is also specified, both attributes must have the same value.

map **. areas**

Returns an HTMLCollection of the area elements in the map.

map **. images**

Returns an HTMLCollection of the img and object elements that use the map.

The **areas** attribute must return an HTMLCollection rooted at the map element, whose filter matches only area elements.

The **images** attribute must return an HTMLCollection rooted at the Document node, whose filter matches only img and object elements that are associated with this map element according to the image map processing model.

The DOM attribute **name** must reflect the content attribute of the same name.

4.8.13 The area element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected, but only if there is a map element ancestor.
Content model:
Empty.
Content attributes:

Global attributes
alt
coords
shape
href
target
ping
rel
media
hreflang
type

DOM interface:

```
interface HTMLAreaElement : HTMLElement {
           attribute DOMString alt;
           attribute DOMString coords;
           attribute DOMString shape;
           attribute DOMString href;
           attribute DOMString target;
           attribute DOMString ping;
           attribute DOMString rel;
  readonly attribute DOMTokenList relList;
           attribute DOMString media;
           attribute DOMString hreflang;
           attribute DOMString type;
};
```

The area element represents either a hyperlink with some text and a corresponding area on an image map, or a dead area on an image map.

If the area element has an href attribute, then the area element represents a hyperlink. In this case, the **alt** attribute must be present. It specifies the text of the hyperlink. Its value must be text that, when presented with the texts specified for the other hyperlinks of the image map, and with the alternative text of the image, but without the image itself, provides the user with the same kind of choice as the hyperlink would when used without its text but with its shape applied to the image. The alt attribute may be left blank if there is another area element in the same image map that points to the same resource and has a non-blank alt attribute.

If the area element has no href attribute, then the area represented by the element cannot be selected, and the alt attribute must be omitted.

In both cases, the shape and coords attributes specify the area.

The **shape** attribute is an enumerated attribute. The following table lists the keywords defined for this attribute. The states given in the first cell of the rows with keywords give the states to which those keywords map. Some of the keywords are non-conforming, as noted in the last column.

State	Keywords	Notes

State	Keywords	Notes
	`polygon`	Non-conforming
Rectangle state	`rect`	
	`rectangle`	Non-conforming

The attribute may be omitted. The *missing value default* is the rectangle state.

The **`coords`** attribute must, if specified, contain a valid list of integers. This attribute gives the coordinates for the shape described by the `shape` attribute. The processing for this attribute is described as part of the image map processing model.

In the **circle state**, `area` elements must have a `coords` attribute present, with three integers, the last of which must be non-negative. The first integer must be the distance in CSS pixels from the left edge of the image to the center of the circle, the second integer must be the distance in CSS pixels from the top edge of the image to the center of the circle, and the third integer must be the radius of the circle, again in CSS pixels.

In the **default state** state, `area` elements must not have a `coords` attribute. (The area is the whole image.)

In the **polygon state**, `area` elements must have a `coords` attribute with at least six integers, and the number of integers must be even. Each pair of integers must represent a coordinate given as the distances from the left and the top of the image in CSS pixels respectively, and all the coordinates together must represent the points of the polygon, in order.

In the **rectangle state**, `area` elements must have a `coords` attribute with exactly four integers, the first of which must be less than the third, and the second of which must be less than the fourth. The four points must represent, respectively, the distance from the left edge of the image to the left side of the rectangle, the distance from the top edge to the top side, the distance from the left edge to the right side, and the distance from the top edge to the bottom side, all in CSS pixels.

When user agents allow users to follow hyperlinks created using the `area` element, as described in the next section, the `href`, `target` and `ping` attributes decide how the link is followed. The `rel`, `media`, `hreflang`, and `type` attributes may be used to indicate to the user the likely nature of the target resource before the user follows the link.

The `target`, `ping`, `rel`, `media`, `hreflang`, and `type` attributes must be omitted if the `href` attribute is not present.

The activation behavior of `area` elements is to run the following steps:

1. If the `DOMActivate` event in question is not trusted (i.e. a `click()` method call was the reason for the event being dispatched), and the `area` element's `target` attribute is ... then raise an `INVALID_ACCESS_ERR` exception.

2. Otherwise, the user agent must follow the hyperlink defined by the `area` element, if any.

The DOM attributes **`alt`**, **`coords`**, **`href`**, **`target`**, **`ping`**, **`rel`**, **`media`**, **`hreflang`**, and **`type`**, each must reflect the respective content attributes of the same name.

The DOM attribute **`shape`** must reflect the `shape` content attribute, limited to only known values.

The DOM attribute **`relList`** must reflect the `rel` content attribute.

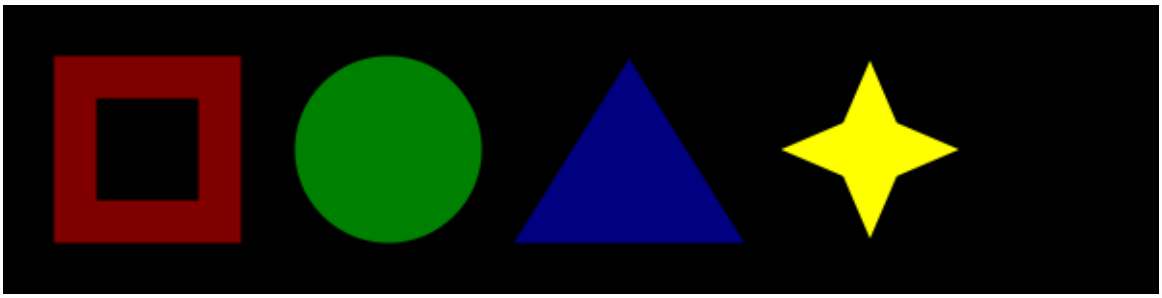
4.8.14 Image maps

4.8.14.1 Authoring

An **image map** allows geometric areas on an image to be associated with hyperlinks.

An image, in the form of an `img` element or an `object` element representing an image, may be associated with an image map (in the form of a `map` element) by specifying a **`usemap`** attribute on the `img` or `object` element. The `usemap` attribute, if specified, must be a valid hash-name reference to a `map` element.

Consider an image that looks as follows:



If we wanted just the coloured areas to be clickable, we could do it as follows:

```
<p>
 Please select a shape:
 <img src="shapes.png" usemap="#shapes"
      alt="Four shapes are available: a red hollow box, a green circle, a blu
 <map name="shapes">
  <area shape=rect coords="50,50,100,100"> <!-- the hole in the red box -->
  <area shape=rect coords="25,25,125,125" href="red.html" alt="Red box.">
  <area shape=circle coords="200,75,50" href="green.html" alt="Green circle.">
  <area shape=poly coords="325,25,262,125,388,125" href="blue.html" alt="Blue
  <area shape=poly coords="450,25,435,60,400,75,435,90,450,125,465,90,500,75,
        href="yellow.html" alt="Yellow star.">
 </map>
</p>
```

4.8.14.2 Processing model

If an `img` element or an `object` element representing an image has a `usemap` attribute specified, user agents must process it as follows:

1. First, rules for parsing a hash-name reference to a `map` element must be followed. This will return either an element (the *map*) or null.

2. If that returned null, then abort these steps. The image is not associated with an image map after all.

3. Otherwise, the user agent must collect all the `area` elements that are descendants of the *map*. Let those be the *areas*.

Having obtained the list of area elements that form the image map (the *areas*), interactive user agents must process the list in one of two ways.

If the user agent intends to show the text that the img element represents, then it must use the following steps.

> ***Note: In user agents that do not support images, or that have images disabled, object elements cannot represent images, and thus this section never applies (the fallback content is shown instead). The following steps therefore only apply to img elements.***

1. Remove all the area elements in *areas* that have no href attribute.

2. Remove all the area elements in *areas* that have no alt attribute, or whose alt attribute's value is the empty string, *if* there is another area element in *areas* with the same value in the href attribute and with a non-empty alt attribute.

3. Each remaining area element in *areas* represents a hyperlink. Those hyperlinks should all be made available to the user in a manner associated with the text of the img.

   In this context, user agents may represent area and img elements with no specified alt attributes, or whose alt attributes are the empty string or some other non-visible text, in a user-agent-defined fashion intended to indicate the lack of suitable author-provided text.

If the user agent intends to show the image and allow interaction with the image to select hyperlinks, then the image must be associated with a set of layered shapes, taken from the area elements in *areas*, in reverse tree order (so the last specified area element in the *map* is the bottom-most shape, and the first element in the *map*, in tree order, is the top-most shape).

Each area element in *areas* must be processed as follows to obtain a shape to layer onto the image:

1. Find the state that the element's shape attribute represents.

2. Use the rules for parsing a list of integers to parse the element's coords attribute, if it is present, and let the result be the *coords* list. If the attribute is absent, let the *coords* list be the empty list.

3. If the number of items in the *coords* list is less than the minimum number given for the area element's current state, as per the following table, then the shape is empty; abort these steps.

State	Minimum number of items
Circle state	3
Default state	0
Polygon state	6
Rectangle state	4

4. Check for excess items in the *coords* list as per the entry in the following list

corresponding to the shape attribute's state:

- **Circle state**
  Drop any items in the list beyond the third.
- **Default state**
  Drop all items in the list.
- **Polygon state**
  Drop the last item if there's an odd number of items.
- **Rectangle state**
  Drop any items in the list beyond the fourth.

5. If the shape attribute represents the rectangle state, and the first number in the list is numerically less than the third number in the list, then swap those two numbers around.

6. If the shape attribute represents the rectangle state, and the second number in the list is numerically less than the fourth number in the list, then swap those two numbers around.

7. If the shape attribute represents the circle state, and the third number in the list is less than or equal to zero, then the shape is empty; abort these steps.

8. Now, the shape represented by the element is the one described for the entry in the list below corresponding to the state of the shape attribute:

   - **Circle state**
     Let *x* be the first number in *coords*, *y* be the second number, and *r* be the third number.

     The shape is a circle whose center is *x* CSS pixels from the left edge of the image and *x* CSS pixels from the top edge of the image, and whose radius is *r* pixels.

   - **Default state**
     The shape is a rectangle that exactly covers the entire image.

   - **Polygon state**
     Let x_i be the $(2i)$th entry in *coords*, and y_i be the $(2i+1)$th entry in *coords* (the first entry in *coords* being the one with index 0).

     Let *the coordinates* be (x_i, y_i), interpreted in CSS pixels measured from the top left of the image, for all integer values of *i* from 0 to $(N/2)-1$, where *N* is the number of items in *coords*.

     The shape is a polygon whose vertices are given by *the coordinates*, and whose interior is established using the even-odd rule. [GRAPHICS]

   - **Rectangle state**
     Let *x1* be the first number in *coords*, *y1* be the second number, *x2* be the third number, and *y2* be the fourth number.

     The shape is a rectangle whose top-left corner is given by the coordinate (*x1*, *y1*) and whose bottom right corner is given by the coordinate (*x2*, *y2*), those coordinates being interpreted as CSS pixels from the top left corner of the image.

For historical reasons, the coordinates must be interpreted relative to the *displayed* image, even if it stretched using CSS or the image element's `width` and `height` attributes.

Mouse clicks on an image associated with a set of layered shapes per the above algorithm must be dispatched to the top-most shape covering the point that the pointing device indicated (if any), and then, must be dispatched again (with a new `Event` object) to the image element itself. User agents may also allow individual `area` elements representing hyperlinks to be selected and activated (e.g. using a keyboard); events from this are not also propagated to the image.

Note: Because a `map` element (and its `area` elements) can be associated with multiple `img` and `object` elements, it is possible for an `area` element to correspond to multiple focusable areas of the document.

Image maps are *live*; if the DOM is mutated, then the user agent must act as if it had rerun the algorithms for image maps.

4.8.15 MathML

The `math` element from the MathML namespace falls into the embedded content category for the purposes of the content models in this specification.

User agents must handle text other than inter-element whitespace found in MathML elements whose content models do not allow raw text by pretending for the purposes of MathML content models, layout, and rendering that that text is actually wrapped in an `mtext` element in the MathML namespace. (Such text is not, however, conforming.)

User agents must act as if any MathML element whose contents does not match the element's content model was replaced, for the purposes of MathML layout and rendering, by an `merror` element in the MathML namespace containing some appropriate error message.

To enable authors to use MathML tools that only accept MathML in its XML form, interactive HTML user agents are encouraged to provide a way to export any MathML fragment as a namespace-well-formed XML fragment.

4.8.16 SVG

The `svg` element from the SVG namespace falls into the embedded content category for the purposes of the content models in this specification.

To enable authors to use SVG tools that only accept SVG in its XML form, interactive HTML user agents are encouraged to provide a way to export any SVG fragment as a namespace-well-formed XML fragment.

When the SVG `foreignObject` element contains elements from the HTML namespace, such elements must all be flow content. [SVG]

The content model for `title` elements in the SVG namespace inside HTML documents is phrasing content. (This further constrains the requirements given in the SVG specification.)

4.8.17 Dimension attributes

Author requirements: The **`width`** and **`height`** attributes on `img`, `iframe`, `embed`, `object`, `video`, and, when their `type` attribute is in the Image Button state, `input` elements may be specified to give the dimensions of the visual content of the element (the width and height respectively, relative to the nominal direction of the output medium), in CSS pixels. The attributes, if specified, must have values that are valid non-negative integers greater than zero.

The specified dimensions given may differ from the dimensions specified in the resource itself, since the resource may have a resolution that differs from the CSS pixel resolution. (On screens, CSS pixels have a resolution of 96ppi, but in general the CSS pixel resolution depends on the reading distance.) If both attributes are specified, then one of the following statements must be true:

- *specified width* - 0.5 ≤ *specified height* * *target ratio* ≤ *specified width* + 0.5
- *specified height* - 0.5 ≤ *specified width* / *target ratio* ≤ *specified height* + 0.5

The *target ratio* is the ratio of the intrinsic width to the intrinsic height in the resource. The *specified width* and *specified height* are the values of the `width` and `height` attributes respectively.

The two attributes must be omitted if the resource in question does not have both an intrinsic width and an intrinsic height.

Note: Basically, the dimension attributes can't be used to stretch the image.

User agent requirements: User agents are expected to use these attributes as hints for the rendering.

The **`width`** and **`height`** DOM attributes on the `iframe`, `embed`, `object`, and `video` elements must reflect the respective content attributes of the same name.

4.9 Tabular data

4.9.1 Introduction

This section is non-normative.

...examples, how to write tables accessibly, a brief mention of the table model, etc...

4.9.2 The `table` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

In this order: optionally a `caption` element, followed by either zero or more `colgroup` elements, followed optionally by a `thead` element, followed optionally by a `tfoot` element, followed by either zero or more `tbody` elements *or* one or more `tr` elements, followed optionally by a `tfoot` element (but there can only be one `tfoot` element child in total).

Content attributes:

Global attributes

DOM interface:

```
interface HTMLTableElement : HTMLElement {
           attribute HTMLTableCaptionElement caption;
  HTMLElement createCaption();
  void deleteCaption();
           attribute HTMLTableSectionElement tHead;
  HTMLElement createTHead();
  void deleteTHead();
           attribute HTMLTableSectionElement tFoot;
  HTMLElement createTFoot();
  void deleteTFoot();
  readonly attribute HTMLCollection tBodies;
  HTMLElement createTBody();
  readonly attribute HTMLCollection rows;
  HTMLElement insertRow([Optional] in long index);
  void deleteRow(in long index);
};
```

The `table` element represents data with more than one dimension, in the form of a table.

The `table` element takes part in the table model.

Tables must not be used as layout aids. Historically, some Web authors have misused tables in HTML as a way to control their page layout. This usage is non-conforming, because tools attempting to extract tabular data from such documents would obtain very confusing results. In particular, users of accessibility tools like screen readers are likely to find it very difficult to navigate pages with tables used for layout.

Note: There are a variety of alternatives to using HTML tables for layout, primarily using CSS positioning and CSS tables.

User agents that do table analysis on arbitrary content are encouraged to find heuristics to determine which tables actually contain data and which are merely being used for layout. This specification does not define a precise heuristic.

Tables have rows and columns given by their descendants. A table must not have an empty row or column, as described in the description of the table model.

If a `table` element has a `summary` attribute, the user agent may report the contents of that attribute to the user.

Note: Authors are encouraged to use the `caption` element instead of the `summary` attribute.

table **.** **`caption`** **[=** ***value*** **]**

Returns the table's `caption` element.

Can be set, to replace the `caption` element. If the new value is not a `caption` element, throws a `HIERARCHY_REQUEST_ERR` exception.

caption **=** ***table*** **.** **`createCaption`****()**

Ensures the table has a `caption` element, and returns it.

table **.** **`deleteCaption`****()**

Ensures the table does not have a `caption` element.

table **.** **`tHead`** **[=** ***value*** **]**

Returns the table's `thead` element.

Can be set, to replace the `thead` element. If the new value is not a `thead` element, throws a `HIERARCHY_REQUEST_ERR` exception.

thead **=** ***table*** **.** **`createTHead`****()**

Ensures the table has a `thead` element, and returns it.

table **.** **`deleteTHead`****()**

Ensures the table does not have a `thead` element.

table **.** **`tFoot`** **[=** ***value*** **]**

Returns the table's `tfoot` element.

Can be set, to replace the `tfoot` element. If the new value is not a `tfoot` element, throws a `HIERARCHY_REQUEST_ERR` exception.

tfoot **=** ***table*** **.** **`createTFoot`****()**

Ensures the table has a `tfoot` element, and returns it.

table **.** **`deleteTFoot`****()**

Ensures the table does not have a `tfoot` element.

table **.** **`tBodies`**

Returns an `HTMLCollection` of the `tbody` elements of the table.

tbody **=** ***table*** **.** **`createTBody`****()**

Creates a `tbody` element, inserts it into the table, and returns it.

table **.** **`rows`**

Returns an `HTMLCollection` of the `tr` elements of the table.

tr **=** ***table*** **.** **`insertRow`****(*****index*****)**

Creates a `tr` element, along with a `tbody` if required, inserts them into the table at the position given by the argument, and returns the `tr`.

The position is relative to the rows in the table. The index −1 is equivalent to inserting at the end of the table.

If the given position is less than −1 or greater than the number of rows, throws an `INDEX_SIZE_ERR` exception.

table **`.deleteRow(`*index*`)`**

Removes the `tr` element with the given position in the table.

The position is relative to the rows in the table. The index −1 is equivalent to deleting the last row of the table.

If the given position is less than −1 or greater than the index of the last row, or if there are no rows, throws an `INDEX_SIZE_ERR` exception.

The **`caption`** DOM attribute must return, on getting, the first `caption` element child of the `table` element, if any, or null otherwise. On setting, if the new value is a `caption` element, the first `caption` element child of the `table` element, if any, must be removed, and the new value must be inserted as the first node of the `table` element. If the new value is not a `caption` element, then a `HIERARCHY_REQUEST_ERR` DOM exception must be raised instead.

The **`createCaption()`** method must return the first `caption` element child of the `table` element, if any; otherwise a new `caption` element must be created, inserted as the first node of the `table` element, and then returned.

The **`deleteCaption()`** method must remove the first `caption` element child of the `table` element, if any.

The **`tHead`** DOM attribute must return, on getting, the first `thead` element child of the `table` element, if any, or null otherwise. On setting, if the new value is a `thead` element, the first `thead` element child of the `table` element, if any, must be removed, and the new value must be inserted immediately before the first element in the `table` element that is neither a `caption` element nor a `colgroup` element, if any, or at the end of the table otherwise. If the new value is not a `thead` element, then a `HIERARCHY_REQUEST_ERR` DOM exception must be raised instead.

The **`createTHead()`** method must return the first `thead` element child of the `table` element, if any; otherwise a new `thead` element must be created and inserted immediately before the first element in the `table` element that is neither a `caption` element nor a `colgroup` element, if any, or at the end of the table otherwise, and then that new element must be returned.

The **`deleteTHead()`** method must remove the first `thead` element child of the `table` element, if any.

The **`tFoot`** DOM attribute must return, on getting, the first `tfoot` element child of the `table` element, if any, or null otherwise. On setting, if the new value is a `tfoot` element, the first `tfoot` element child of the `table` element, if any, must be removed, and the new value must be inserted immediately before the first element in the `table` element that is neither a `caption` element, a `colgroup` element, nor a `thead` element, if any, or at the end of the table if there are no such elements. If the new value is not a `tfoot` element, then a `HIERARCHY_REQUEST_ERR` DOM exception must be raised instead.

The **`createTFoot()`** method must return the first `tfoot` element child of the `table` element, if any; otherwise a new `tfoot` element must be created and inserted immediately before the first element in the `table` element that is neither a `caption` element, a `colgroup` element, nor a `thead` element, if any, or at the end of the table if there are no such elements, and then that new element must be returned.

The **`deleteTFoot()`** method must remove the first `tfoot` element child of the `table` element, if any.

The **`tBodies`** attribute must return an `HTMLCollection` rooted at the `table` node, whose filter matches only `tbody` elements that are children of the `table` element.

The **`createTBody()`** method must create a new `tbody` element, insert it immediately after the last `tbody` element in the `table` element, if any, or at the end of the `table` element if the `table` element has no `tbody` element children, and then must return the new `tbody` element.

The **`rows`** attribute must return an `HTMLCollection` rooted at the `table` node, whose filter matches only `tr` elements that are either children of the `table` element, or children of `thead`, `tbody`, or `tfoot` elements that are themselves children of the `table` element. The elements in the collection must be ordered such that those elements whose parent is a `thead` are included first, in tree order, followed by those elements whose parent is either a `table` or `tbody` element, again in tree order, followed finally by those elements whose parent is a `tfoot` element, still in tree order.

The behavior of the **`insertRow(`*`index`*`)`** method depends on the state of the table. When it is called, the method must act as required by the first item in the following list of conditions that describes the state of the table and the *index* argument:

- **If *index* is less than −1 or greater than the number of elements in `rows` collection:**
  The method must raise an `INDEX_SIZE_ERR` exception.
- **If the `rows` collection has zero elements in it, and the `table` has no `tbody` elements in it:**
  The method must create a `tbody` element, then create a `tr` element, then append the `tr` element to the `tbody` element, then append the `tbody` element to the `table` element, and finally return the `tr` element.
- **If the `rows` collection has zero elements in it:**
  The method must create a `tr` element, append it to the last `tbody` element in the table, and return the `tr` element.
- **If *index* is missing, equal to −1, or equal to the number of items in `rows` collection:**
  The method must create a `tr` element, and append it to the parent of the last `tr` element in the `rows` collection. Then, the newly created `tr` element must be returned.
- **Otherwise:**
  The method must create a `tr` element, insert it immediately before the *index*th `tr` element in the `rows` collection, in the same parent, and finally must return the newly created `tr` element.

When the **`deleteRow(`*`index`*`)`** method is called, the user agent must run the following steps:

1. If *index* is equal to −1, then *index* must be set to the number if items in the `rows` collection, minus one.

2. Now, if *index* is less than zero, or greater than or equal to the number of elements in the

`rows` collection, the method must instead raise an `INDEX_SIZE_ERR` exception, and these steps must be aborted.

3. Otherwise, the method must remove the *index*th element in the `rows` collection from its parent.

4.9.3 The `caption` element

Categories
None.
Contexts in which this element may be used:
As the first element child of a `table` element.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:
Uses `HTMLElement`.

The `caption` element represents the title of the `table` that is its parent, if it has a parent and that is a `table` element.

The `caption` element takes part in the table model.

The `caption` element should be included for any table where the reader might have difficulty understanding the content or where the table's structure would not be obvious to the user of a screen reader. The element's contents should describe what the purpose of the table is, along with any information that could be useful for understanding and using the table.

When a `table` element is in a `figure` element alone but for the `figure`'s `legend`, the `caption` element should be omitted in favor of the `legend`.

Consider, for instance, the following table:

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

In the abstract, this table is not clear. However, with a caption giving the table's number (for reference in the main prose) and explaining its use, it makes more sense:

```
<caption>
<strong>Table 1.</strong>
This table shows the total score obtained from rolling two six-sided
dice. The first row represents the value of the first die, the first
column the value of the second die. The total is given in the cell
that corresponds to the values of the two dice.
</caption>
```

This provides the user with more context:

Table 1. *This table shows the total score obtained from rolling two six-sided dice. The first row represents the value of the first die, the first column the value of the second die. The total is given in the cell that corresponds to the values of the two dice.*

	1	**2**	**3**	**4**	**5**	**6**
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

4.9.4 The `colgroup` element

Categories
None.
Contexts in which this element may be used:
As a child of a `table` element, after any `caption` elements and before any `thead`, `tbody`, `tfoot`, and `tr` elements.
Content model:
Zero or more `col` elements.
Content attributes:
Global attributes
`span`
DOM interface:

```
interface HTMLTableColElement : HTMLElement {
           attribute unsigned long span;
};
```

The `colgroup` element represents a group of one or more columns in the `table` that is its parent, if it has a parent and that is a `table` element.

If the `colgroup` element contains no `col` elements, then the element may have a **`span`** content attribute specified, whose value must be a valid non-negative integer greater than zero.

The `colgroup` element and its `span` attribute take part in the table model.

The **`span`** DOM attribute must reflect the respective content attribute of the same name. The value must be limited to only positive non-zero numbers.

4.9.5 The `col` element

Categories
None.
Contexts in which this element may be used:

As a child of a `colgroup` element that doesn't have a `span` attribute.

Content model:

Empty.

Content attributes:

Global attributes

`span`

DOM interface:

`HTMLTableColElement`, same as for `colgroup` elements. This interface defines one member, `span`.

If a `col` element has a parent and that is a `colgroup` element that itself has a parent that is a `table` element, then the `col` element represents one or more columns in the column group represented by that `colgroup`.

The element may have a **`span`** content attribute specified, whose value must be a valid non-negative integer greater than zero.

The `col` element and its `span` attribute take part in the table model.

The **`span`** DOM attribute must reflect the content attribute of the same name. The value must be limited to only positive non-zero numbers.

4.9.6 The `tbody` element

Categories

None.

Contexts in which this element may be used:

As a child of a `table` element, after any `caption`, `colgroup`, and `thead` elements, but only if there are no `tr` elements that are children of the `table` element.

Content model:

Zero or more `tr` elements

Content attributes:

Global attributes

DOM interface:

```
interface HTMLTableSectionElement : HTMLElement {
  readonly attribute HTMLCollection rows;
  HTMLElement insertRow([Optional] in long index);
  void deleteRow(in long index);
};
```

The `HTMLTableSectionElement` interface is also used for `thead` and `tfoot` elements.

The `tbody` element represents a block of rows that consist of a body of data for the parent `table` element, if the `tbody` element has a parent and it is a `table`.

The `tbody` element takes part in the table model.

***tbody* . `rows`**

Returns an `HTMLCollection` of the `tr` elements of the table section.

***tr* = *tbody* . `insertRow`([*index*])**

Creates a `tr` element, inserts it into the table section at the position given by the argument, and returns the `tr`.

The position is relative to the rows in the table section. The index −1, which is the default if the argument is omitted, is equivalent to inserting at the end of the table section.

If the given position is less than −1 or greater than the number of rows, throws an `INDEX_SIZE_ERR` exception.

***tbody* . `deleteRow`(*index*)**

Removes the `tr` element with the given position in the table section.

The position is relative to the rows in the table section. The index −1 is equivalent to deleting the last row of the table section.

If the given position is less than −1 or greater than the index of the last row, or if there are no rows, throws an `INDEX_SIZE_ERR` exception.

The **`rows`** attribute must return an `HTMLCollection` rooted at the element, whose filter matches only `tr` elements that are children of the element.

The **`insertRow(`*`index`*`)`** method must, when invoked on an element *table section*, act as follows:

If *index* is less than −1 or greater than the number of elements in the `rows` collection, the method must raise an `INDEX_SIZE_ERR` exception.

If *index* is missing, equal to −1, or equal to the number of items in the `rows` collection, the method must create a `tr` element, append it to the element *table section*, and return the newly created `tr` element.

Otherwise, the method must create a `tr` element, insert it as a child of the *table section* element, immediately before the *index*th `tr` element in the `rows` collection, and finally must return the newly created `tr` element.

The **`deleteRow(`*`index`*`)`** method must remove the *index*th element in the `rows` collection from its parent. If *index* is less than zero or greater than or equal to the number of elements in the `rows` collection, the method must instead raise an `INDEX_SIZE_ERR` exception.

4.9.7 The `thead` element

Categories
None.

Contexts in which this element may be used:
As a child of a `table` element, after any `caption`, and `colgroup` elements and before any `tbody`, `tfoot`, and `tr` elements, but only if there are no other `thead` elements that are children of the `table` element.
Content model:
Zero or more `tr` elements
Content attributes:
Global attributes
DOM interface:
`HTMLTableSectionElement`, as defined for `tbody` elements.

The `thead` element represents the block of rows that consist of the column labels (headers) for the parent `table` element, if the `thead` element has a parent and it is a `table`.

The `thead` element takes part in the table model.

4.9.8 The `tfoot` element

Categories
None.
Contexts in which this element may be used:
As a child of a `table` element, after any `caption`, `colgroup`, and `thead` elements and before any `tbody` and `tr` elements, but only if there are no other `tfoot` elements that are children of the `table` element.
As a child of a `table` element, after any `caption`, `colgroup`, `thead`, `tbody`, and `tr` elements, but only if there are no other `tfoot` elements that are children of the `table` element.
Content model:
Zero or more `tr` elements
Content attributes:
Global attributes
DOM interface:
`HTMLTableSectionElement`, as defined for `tbody` elements.

The `tfoot` element represents the block of rows that consist of the column summaries (footers) for the parent `table` element, if the `tfoot` element has a parent and it is a `table`.

The `tfoot` element takes part in the table model.

4.9.9 The `tr` element

Categories
None.
Contexts in which this element may be used:
As a child of a `thead` element.
As a child of a `tbody` element.
As a child of a `tfoot` element.
As a child of a `table` element, after any `caption`, `colgroup`, and `thead` elements, but only if there are no `tbody` elements that are children of the `table` element.

Content model:
Zero or more td or th elements
Content attributes:
Global attributes
DOM interface:

```
interface HTMLTableRowElement : HTMLElement {
  readonly attribute long rowIndex;
  readonly attribute long sectionRowIndex;
  readonly attribute HTMLCollection cells;
  HTMLElement insertCell([Optional] in long index);
  void deleteCell(in long index);
};
```

The tr element represents a row of cells in a table.

The tr element takes part in the table model.

***tr* . rowIndex**
Returns the position of the row in the table's rows list.

Returns −1 if the element isn't in a table.

***tr* . sectionRowIndex**
Returns the position of the row in the table section's rows list.

Returns −1 if the element isn't in a table section.

***tr* . cells**
Returns an HTMLCollection of the td and th elements of the row.

***cell* = *tr* . insertCell([*index*])**
Creates a td element, inserts it into the table row at the position given by the argument, and returns the td.

The position is relative to the cells in the row. The index −1, which is the default if the argument is omitted, is equivalent to inserting at the end of the row.

If the given position is less than −1 or greater than the number of cells, throws an INDEX_SIZE_ERR exception.

***tr* . deleteCell(*index*)**
Removes the td or th element with the given position in the row.

The position is relative to the cells in the row. The index −1 is equivalent to deleting the last cell of the row.

If the given position is less than −1 or greater than the index of the last cell, or if there are no cells, throws an INDEX_SIZE_ERR exception.

The **rowIndex** attribute must, if the element has a parent table element, or a parent tbody, thead, or tfoot element and a *grandparent* table element, return the index of the tr element in that table element's rows collection. If there is no such table element, then the attribute must return −1.

The **`sectionRowIndex`** attribute must, if the element has a parent `table`, `tbody`, `thead`, or `tfoot` element, return the index of the `tr` element in the parent element's `rows` collection (for tables, that's the `rows` collection; for table sections, that's the `rows` collection). If there is no such parent element, then the attribute must return −1.

The **`cells`** attribute must return an `HTMLCollection` rooted at the `tr` element, whose filter matches only `td` and `th` elements that are children of the `tr` element.

The **`insertCell(`*`index`*`)`** method must act as follows:

If *index* is less than −1 or greater than the number of elements in the `cells` collection, the method must raise an `INDEX_SIZE_ERR` exception.

If *index* is missing, equal to −1, or equal to the number of items in `cells` collection, the method must create a `td` element, append it to the `tr` element, and return the newly created `td` element.

Otherwise, the method must create a `td` element, insert it as a child of the `tr` element, immediately before the *index*th `td` or `th` element in the `cells` collection, and finally must return the newly created `td` element.

The **`deleteCell(`*`index`*`)`** method must remove the *index*th element in the `cells` collection from its parent. If *index* is less than zero or greater than or equal to the number of elements in the `cells` collection, the method must instead raise an `INDEX_SIZE_ERR` exception.

4.9.10 The `td` element

Categories
Sectioning root.
Contexts in which this element may be used:
As a child of a `tr` element.
Content model:
Flow content.
Content attributes:
Global attributes
`colspan`
`rowspan`
`headers`
DOM interface:

```
interface HTMLTableDataCellElement : HTMLTableCellElement {};
```

The `td` element represents a data cell in a table.

The `td` element and its `colspan`, `rowspan`, and `headers` attributes take part in the table model.

4.9.11 The `th` element

Categories

None.

Contexts in which this element may be used:

As a child of a `tr` element.

Content model:

Phrasing content.

Content attributes:

Global attributes
`colspan`
`rowspan`
`headers`
`scope`

DOM interface:

```
interface HTMLTableHeaderCellElement : HTMLTableCellElement {
           attribute DOMString scope;
};
```

The `th` element represents a header cell in a table.

The `th` element may have a **`scope`** content attribute specified. The `scope` attribute is an enumerated attribute with five states, four of which have explicit keywords:

The *`row`* keyword, which maps to the *row* state

The *row* state means the header cell applies to some of the subsequent cells in the same row(s).

The *`col`* keyword, which maps to the *column* state

The *column* state means the header cell applies to some of the subsequent cells in the same column(s).

The *`rowgroup`* keyword, which maps to the *row group* state

The *row group* state means the header cell applies to all the remaining cells in the row group. A `th` element's `scope` attribute must not be in the row group state if the element is not anchored in a row group.

The *`colgroup`* keyword, which maps to the *column group* state

The *column group* state means the header cell applies to all the remaining cells in the column group. A `th` element's `scope` attribute must not be in the column group state if the element is not anchored in a column group.

The *auto* state

The *auto* state makes the header cell apply to a set of cells selected based on context.

The `scope` attribute's *missing value default* is the *auto* state.

The `th` element and its `colspan`, `rowspan`, `headers`, and `scope` attributes take part in the table model.

The **`scope`** DOM attribute must reflect the content attribute of the same name.

4.9.12 Attributes common to `td` and `th` elements

The `td` and `th` elements may have a **`colspan`** content attribute specified, whose value must be a valid non-negative integer greater than zero.

The td and th elements may also have a **rowspan** content attribute specified, whose value must be a valid non-negative integer.

These attributes give the number of columns and rows respectively that the cell is to span. These attributes must not be used to overlap cells, as described in the description of the table model.

The td and th element may have a **headers** content attribute specified. The headers attribute, if specified, must contain a string consisting of an unordered set of unique space-separated tokens, each of which must have the value of an ID of a th element taking part in the same table as the td or th element (as defined by the table model).

A th element with ID *id* is said to be *directly targetted* by all td and th elements in the same table that have headers attributes whose values include as one of their tokens the ID *id*. A th element *A* is said to be *targetted* by a th or td element *B* if either *A* is *directly targetted* by *B* or if there exists an element *C* that is itself *targetted* by the element *B* and *A* is *directly targetted* by *C*.

A th element must not be *targetted* by itself.

The colspan, rowspan, and headers attributes take part in the table model.

The td and th elements implement interfaces that inherit from the HTMLTableCellElement interface:

```
interface HTMLTableCellElement : HTMLElement {
           attribute long colSpan;
           attribute long rowSpan;
           attribute DOMString headers;
  readonly attribute long cellIndex;
};
```

***cell*.cellIndex**

Returns the position of the cell in the row's cells list.

Returns 0 if the element isn't in a row.

The **colSpan** DOM attribute must reflect the content attribute of the same name. The value must be limited to only positive non-zero numbers.

The **rowSpan** DOM attribute must reflect the content attribute of the same name. Its default value, which must be used if parsing the attribute as a non-negative integer returns an error, is also 1.

The **headers** DOM attribute must reflect the content attribute of the same name.

The **cellIndex** DOM attribute must, if the element has a parent tr element, return the index of the cell's element in the parent element's cells collection. If there is no such parent element,

then the attribute must return 0.

4.9.13 Processing model

The various table elements and their content attributes together define the **table model**.

A **table** consists of cells aligned on a two-dimensional grid of **slots** with coordinates (x, y). The grid is finite, and is either empty or has one or more slots. If the grid has one or more slots, then the x coordinates are always in the range $0 \le x < x_{width}$, and the y coordinates are always in the range $0 \le y < y_{height}$. If one or both of x_{width} and y_{height} are zero, then the table is empty (has no slots). Tables correspond to `table` elements.

A **cell** is a set of slots anchored at a slot ($cell_x$, $cell_y$), and with a particular *width* and *height* such that the cell covers all the slots with coordinates (x, y) where $cell_x \le x < cell_x + width$ and $cell_y \le y < cell_y + height$. Cells can either be *data cells* or *header cells*. Data cells correspond to `td` elements, and header cells correspond to `th` elements. Cells of both types can have zero or more associated header cells.

It is possible, in certain error cases, for two cells to occupy the same slot.

A **row** is a complete set of slots from $x=0$ to $x=x_{width}-1$, for a particular value of y. Rows correspond to `tr` elements.

A **column** is a complete set of slots from $y=0$ to $y=y_{height}-1$, for a particular value of x. Columns can correspond to `col` elements, but in the absence of `col` elements are implied.

A **row group** is a set of rows anchored at a slot (0, $group_y$) with a particular *height* such that the row group covers all the slots with coordinates (x, y) where $0 \le x < x_{width}$ and $group_y \le y < group_y + height$. Row groups correspond to `tbody`, `thead`, and `tfoot` elements. Not every row is necessarily in a row group.

A **column group** is a set of columns anchored at a slot ($group_x$, 0) with a particular *width* such that the column group covers all the slots with coordinates (x, y) where $group_x \le x < group_x + width$ and $0 \le y < y_{height}$. Column groups correspond to `colgroup` elements. Not every column is necessarily in a column group.

Row groups cannot overlap each other. Similarly, column groups cannot overlap each other.

A cell cannot cover slots that are from two or more row groups. It is, however, possible for a cell to be in multiple column groups. All the slots that form part of one cell are part of zero or one row groups and zero or more column groups.

In addition to cells, columns, rows, row groups, and column groups, tables can have a `caption` element associated with them. This gives the table a heading, or legend.

A **table model error** is an error with the data represented by `table` elements and their descendants. Documents must not have table model errors.

4.9.13.1 Forming a table

To determine which elements correspond to which slots in a table associated with a `table`

element, to determine the dimensions of the table (x_{width} and y_{height}), and to determine if there are any table model errors, user agents must use the following algorithm:

1. Let x_{width} be zero.
2. Let y_{height} be zero.
3. Let *pending `tfoot` elements* be a list of `tfoot` elements, initially empty.
4. Let *the table* be the table represented by the `table` element. The x_{width} and y_{height} variables give *the table*'s dimensions. *The table* is initially empty.
5. If the `table` element has no children elements, then return *the table* (which will be empty), and abort these steps.
6. Associate the first `caption` element child of the `table` element with *the table*. If there are no such children, then it has no associated `caption` element.
7. Let the *current element* be the first element child of the `table` element.

   If a step in this algorithm ever requires the *current element* to be **advanced to the next child of the `table`** when there is no such next child, then the user agent must jump to the step labeled *end*, near the end of this algorithm.
8. While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:

   - `colgroup`
   - `thead`
   - `tbody`
   - `tfoot`
   - `tr`
9. If the *current element* is a `colgroup`, follow these substeps:

   1. *Column groups*: Process the *current element* according to the appropriate case below:

      ↪ **If the *current element* has any `col` element children**

      Follow these steps:

      1. Let x_{start} have the value of x_{width}.
      2. Let the *current column* be the first `col` element child of the `colgroup` element.
      3. *Columns*: If the *current column* `col` element has a `span` attribute, then parse its value using the rules for parsing non-negative integers.

         If the result of parsing the value is not an error or zero, then let *span* be that value.

         Otherwise, if the `col` element has no `span` attribute, or if trying to

parse the attribute's value resulted in an error, then let *span* be 1.

4. Increase x_{width} by *span*.

5. Let the last *span* columns in *the table* correspond to the *current column* `col` element.

6. If *current column* is not the last `col` element child of the `colgroup` element, then let the *current column* be the next `col` element child of the `colgroup` element, and return to the step labeled *columns*.

7. Let all the last columns in *the table* from x=x_{start} to x=x_{width}-1 form a new column group, anchored at the slot (x_{start}, 0), with width x_{width}-x_{start}, corresponding to the `colgroup` element.

↪ **If the *current element* has no `col` element children**

1. If the `colgroup` element has a `span` attribute, then parse its value using the rules for parsing non-negative integers.

   If the result of parsing the value is not an error or zero, then let *span* be that value.

   Otherwise, if the `colgroup` element has no `span` attribute, or if trying to parse the attribute's value resulted in an error, then let *span* be 1.

2. Increase x_{width} by *span*.

3. Let the last *span* columns in *the table* form a new column group, anchored at the slot (x_{width}-*span*, 0), with width *span*, corresponding to the `colgroup` element.

2. Advance the *current element* to the next child of the `table`.

3. While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:

   - `colgroup`
   - `thead`
   - `tbody`
   - `tfoot`
   - `tr`

4. If the *current element* is a `colgroup` element, jump to the step labeled *column groups* above.

10. Let $y_{current}$ be zero.

11. Let the *list of downward-growing cells* be an empty list.

12. *Rows*: While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:

- `thead`
- `tbody`
- `tfoot`
- `tr`

13. If the *current element* is a `tr`, then run the algorithm for processing rows, advance the *current element* to the next child of the `table`, and return to the step labeled *rows*.

14. Run the algorithm for ending a row group.

15. If the *current element* is a `tfoot`, then add that element to the list of *pending* `tfoot` *elements*, advance the *current element* to the next child of the `table`, and return to the step labeled *rows*.

16. The *current element* is either a `thead` or a `tbody`.

    Run the algorithm for processing row groups.

17. Advance the *current element* to the next child of the `table`.

18. Return to the step labeled *rows*.

19. *End*: For each `tfoot` element in the list of *pending* `tfoot` *elements*, in tree order, run the algorithm for processing row groups.

20. If there exists a row or column in the table *the table* containing only slots that do not have a cell anchored to them, then this is a table model error.

21. Return *the table*.

The **algorithm for processing row groups**, which is invoked by the set of steps above for processing `thead`, `tbody`, and `tfoot` elements, is:

1. Let y_{start} have the value of y_{height}.

2. For each `tr` element that is a child of the element being processed, in tree order, run the algorithm for processing rows.

3. If $y_{height} > y_{start}$, then let all the last rows in *the table* from $y=y_{start}$ to $y=y_{height}-1$ form a new row group, anchored at the slot with coordinate $(0, y_{start})$, with height $y_{height}-y_{start}$, corresponding to the element being processed.

4. Run the algorithm for ending a row group.

The **algorithm for ending a row group**, which is invoked by the set of steps above when starting and ending a block of rows, is:

1. While $y_{current}$ is less than y_{height}, follow these steps:

    1. Run the algorithm for growing downward-growing cells.

    2. Increase $y_{current}$ by 1.

2. Empty the *list of downward-growing cells*.

The **algorithm for processing rows**, which is invoked by the set of steps above for processing `tr` elements, is:

1. If y_{height} is equal to $y_{current}$, then increase y_{height} by 1. ($y_{current}$ is never *greater* than y_{height}.)

2. Let $x_{current}$ be 0.

3. Run the algorithm for growing downward-growing cells.

4. If the `tr` element being processed has no `td` or `th` element children, then increase $y_{current}$ by 1, abort this set of steps, and return to the algorithm above.

5. Let *current cell* be the first `td` or `th` element in the `tr` element being processed.

6. *Cells*: While $x_{current}$ is less than x_{width} and the slot with coordinate ($x_{current}$, $y_{current}$) already has a cell assigned to it, increase $x_{current}$ by 1.

7. If $x_{current}$ is equal to x_{width}, increase x_{width} by 1. ($x_{current}$ is never *greater* than x_{width}.)

8. If the *current cell* has a `colspan` attribute, then parse that attribute's value, and let *colspan* be the result.

   If parsing that value failed, or returned zero, or if the attribute is absent, then let *colspan* be 1, instead.

9. If the *current cell* has a `rowspan` attribute, then parse that attribute's value, and let *rowspan* be the result.

   If parsing that value failed or if the attribute is absent, then let *rowspan* be 1, instead.

10. If *rowspan* is zero, then let *cell grows downward* be true, and set *rowspan* to 1. Otherwise, let *cell grows downward* be false.

11. If $x_{width} < x_{current}+colspan$, then let x_{width} be $x_{current}+colspan$.

12. If $y_{height} < y_{current}+rowspan$, then let y_{height} be $y_{current}+rowspan$.

13. Let the slots with coordinates (x, y) such that $x_{current} \le x < x_{current}+colspan$ and $y_{current} \le y < y_{current}+rowspan$ be covered by a new cell c, anchored at ($x_{current}$, $y_{current}$), which has width *colspan* and height *rowspan*, corresponding to the *current cell* element.

    If the *current cell* element is a `th` element, let this new cell c be a header cell; otherwise, let it be a data cell.

    To establish which header cells apply to the *current cell* element, use the algorithm for assigning header cells described in the next section.

    If any of the slots involved already had a cell covering them, then this is a table model error. Those slots now have two cells overlapping.

14. If *cell grows downward* is true, then add the tuple {c, $x_{current}$, *colspan*} to the *list of downward-growing cells*.

15. Increase $x_{current}$ by *colspan*.

16. If *current cell* is the last `td` or `th` element in the `tr` element being processed, then increase $y_{current}$ by 1, abort this set of steps, and return to the algorithm above.

17. Let *current cell* be the next `td` or `th` element in the `tr` element being processed.

18. Return to the step labelled *cells*.

When the algorithms above require the user agent to run the **algorithm for growing downward-growing cells**, the user agent must, for each {*cell*, $cell_x$, *width*} tuple in the *list of downward-growing cells*, if any, extend the cell *cell* so that it also covers the slots with coordinates (x, $y_{current}$), where $cell_x \leq x < cell_x+width$.

4.9.13.2 Forming relationships between data cells and header cells

Each cell can be assigned zero or more header cells. The **algorithm for assigning header cells** to a cell *principal cell* is as follows.

1. Let *header list* be an empty list of cells.

2. Let ($principal_x$, $principal_y$) be the coordinate of the slot to which the *principal cell* is anchored.

3. ↪ **If the *principal cell* has a `headers` attribute specified**

    1. Take the value of the *principal cell*'s `headers` attribute and split it on spaces, letting *id list* be the list of tokens obtained.

    2. For each token in the *id list*, if the first element in the `Document` with an ID equal to the token is a cell in the same table, and that cell is not the *principal cell*, then add that cell to *header list*.

    ↪ **If *principal cell* does not have a `headers` attribute specified**

    1. Let $principal_{width}$ be the width of the *principal cell*.

    2. Let $principal_{height}$ be the height of the *principal cell*.

    3. For each value of y from $principal_y$ to $principal_y+principal_{height}-1$, run the internal algorithm for scanning and assigning header cells, with the *principal cell*, the *header list*, the initial coordinate ($principal_x$,y), and the increments $\Delta x=-1$ and $\Delta y=0$.

    4. For each value of x from $principal_x$ to $principal_x+principal_{width}-1$, run the internal algorithm for scanning and assigning header cells, with the *principal cell*, the *header list*, the initial coordinate (x,$principal_y$), and the increments $\Delta x=0$ and $\Delta y=-1$.

    5. If the *principal cell* is anchored in a row group, then add all header cells that are row group headers and are anchored in the same row group with an x-coordinate less than or equal to $principal_x+principal_{width}-1$ and a y-coordinate less than or equal to $principal_y+principal_{height}-1$ to

header list.

6. If the *principal cell* is anchored in a column group, then add all header cells that are column group headers and are anchored in the same column group with an *x*-coordinate less than or equal to $principal_x+principal_{width}-1$ and a *y*-coordinate less than or equal to $principal_y+principal_{height}-1$ to *header list*.

4. Remove all the empty cells from the *header list*.

5. Remove any duplicates from the *header list*.

6. Assign the headers in the *header list* to the *principal cell*.

The **internal algorithm for scanning and assigning header cells**, given a *principal cell*, a *header list*, an initial coordinate ($initial_x$, $initial_y$), and Δx and Δy increments, is as follows:

1. Let *x* equal $initial_x$.

2. Let *y* equal $initial_y$.

3. Let *opaque headers* be an empty list of cells.

4. ↪ **If *principal cell* is a header cell**
   Let *in header block* be true, and let *headers from current header block* be a list of cells containing just the *principal cell*.

   ↪ **Otherwise**
   Let *in header block* be false and let *headers from current header block* be an empty list of cells.

5. *Loop*: Increment *x* by Δx; increment *y* by Δy.

   Note: For each invocation of this algorithm, one of Δx and Δy will be −1, and the other will be 0.

6. If either *x* or *y* is less than 0, then abort this internal algorithm.

7. If there is no cell covering slot (*x*, *y*), or if there is more than one cell covering slot (*x*, *y*), return to the substep marked *loop*.

8. Let *current cell* be the cell covering slot (*x*, *y*).

9. ↪ **If *current cell* is a header cell**

   1. Set *in header block* to true.

   2. Add *current cell* to *headers from current header block*.

   3. Let *blocked* be false.

   4. ↪ **If Δ*x* is 0**
      If there are any cells in the *opaque headers* list anchored with

the same *x*-coordinate as the *current cell*, and with the same width as *current cell*, then let *blocked* be true.

If the *current cell* is not a column header, then let *blocked* be true.

↪ **If Δ*y* is 0**

If there is are any cells in the *opaque headers* list anchored with the same *y*-coordinate as the *current cell*, and with the same height as *current cell*, then let *blocked* be true.

If the *current cell* is not a row header, then let *blocked* be true.

5. If *blocked* is false, then add the *current cell* to the *headers list*.

↪ **If *current cell* is a data cell and *in header block* is true**

Set *in header block* to false. Add all the cells in *headers from current header block* to the *opaque headers* list, and empty the *headers from current header block* list.

10. Return to the step marked *loop*.

A header cell anchored at the slot with coordinate (*x*, *y*) with width *width* and height *height* is said to be a **column header** if any of the following conditions are true:

- The cell's `scope` attribute is in the column state, or
- The cell's `scope` attribute is in the auto state, and there are no data cells in any of the cells covering slots with *y*-coordinates *y* .. *y*+*height*-1.

A header cell anchored at the slot with coordinate (*x*, *y*) with width *width* and height *height* is said to be a **row header** if any of the following conditions are true:

- The cell's `scope` attribute is in the row state, or
- The cell's `scope` attribute is in the auto state, the cell is not a column header, and there are no data cells in any of the cells covering slots with *x*-coordinates *x* .. *x*+*width*-1.

A header cell is said to be a **column group header** if its `scope` attribute is in the column group state.

A header cell is said to be a **row group header** if its `scope` attribute is in the row group state.

A cell is said to be an **empty cell** if it contains no elements and its text content, if any, consists only of White_Space characters.

4.10 Forms

Forms allow unscripted client-server interaction: given a form, a user can provide data, submit it to the server, and have the server act on it accordingly (e.g. returning the results of a search or calculation). The elements used in forms can also be used for user interaction with no associated submission mechanism, in conjunction with scripts.

Mostly for historical reasons, elements in this section fall into several overlapping (but subtly different) categories in addition to the usual ones like flow content, phrasing content, and interactive content.

A number of the elements are **form-associated elements**, which means they can have a form owner and, to expose this, have a `form` content attribute with a matching `form` DOM attribute.

The form-associated elements fall into several subcategories:

Submittable elements
: Denotes elements that can be used for constructing the form data set when a `form` element is submitted.

Resettable elements
: Denotes elements that can be affected when a `form` element is reset.

Listed
: Denotes elements that are listed in the *`form`*`.elements` and *`fieldset`*`.elements` APIs.

Labelable
: Denotes elements that can be associated with `label` elements.

In addition, some submittable elements can be, depending on their attributes, **buttons**. The prose below defines when an element is a button. Some buttons are specifically **submit buttons**.

> ***Note: The `object` element is also a form-associated element and can, with the use of a suitable plugin, partake in form submission.***

4.10.1 The `form` element

Categories
: Flow content.

Contexts in which this element may be used:
: Where flow content is expected.

Content model:
: Flow content, but with no `form` element descendants.

Content attributes:
: Global attributes
`accept-charset`
`action`
`autocomplete`
`enctype`
`method`
`name`
`novalidate`
`target`

DOM interface:

```
[Callable=namedItem]
interface HTMLFormElement : HTMLElement {
```

```
            attribute DOMString acceptCharset;
            attribute DOMString action;
            attribute boolean autocomplete;
            attribute DOMString enctype;
            attribute DOMString method;
            attribute DOMString name;
            attribute boolean novalidate;
            attribute DOMString target;

  readonly attribute HTMLFormControlsCollection elements;
  readonly attribute long length;
  [IndexGetter] any item(in DOMString name);
  [NameGetter=OverrideBuiltins] any namedItem(in DOMString name);

  void submit();
  void reset();
  boolean checkValidity();

  void dispatchFormInput();
  void dispatchFormChange();
};
```

The `form` element represents a collection of form-associated elements, some of which can represent editable values that can be submitted to a server for processing.

The **`accept-charset`** attribute gives the character encodings that are to be used for the submission. If specified, the value must be an ordered set of unique space-separated tokens, and each token must be the preferred name of an ASCII-compatible character encoding. [IANACHARSET]

The **`name`** attribute represents the `form`'s name within the `forms` collection. The value must not be the empty string, and the value must be unique amongst the `form` elements in the `forms` collection that it is in, if any.

The **`autocomplete`** attribute is an enumerated attribute. The attribute has two states. The `on` keyword maps to the **on** state, and the `off` keyword maps to the **off** state. The attribute may also be omitted. The *missing value default* is the on state. The off state indicates that by default, `input` elements in the form will have their resulting autocompletion state set to *off*; the on state indicates that by default, `input` elements in the form will have their resulting autocompletion state set to *on*.

The `action`, `enctype`, `method`, `novalidate`, and `target` attributes are attributes for form submission.

***form*`.elements`**

Returns an `HTMLCollection` of the form controls in the form (excluding image buttons for historical reasons).

***form*`.length`**

Returns the number of form controls in the form (excluding image buttons for historical reasons).

***element* = *form*`.item(`*index*`)`**
***form*`[`*index*`]`**

Returns the *index*th element in the form (excluding image buttons for historical reasons).

***element* = *form* . `namedItem`(*name*)**
***form*[*name*]**

Returns the form control in the form with the given ID or `name` (excluding image buttons for historical reasons).

Once an element has been referenced using a particular name, that name will continue being available as a way to reference that element in this method, even if the element's actual ID or `name` changes, for as long as the element remains in the `Document`.

If there are multiple matching items, then a `NodeList` object containing all those elements is returned.

Returns null if no element with that ID or `name` could be found.

***form* . `submit`()**

Submits the form.

***form* . `reset`()**

Resets the form.

***form* . `checkValidity`()**

Returns true if the form's controls are all valid; otherwise, returns false.

***form* . `dispatchFormInput`()**

Dispatches a `forminput` event at all the form controls.

***form* . `dispatchFormChange`()**

Dispatches a `formchange` event at all the form controls.

The **`autocomplete`** and **`name`** DOM attributes must reflect the respective content attributes of the same name.

The **`acceptCharset`** DOM attribute must reflect the `accept-charset` content attribute.

The **`elements`** DOM attribute must return an `HTMLFormControlsCollection` rooted at the `Document` node, whose filter matches listed elements whose form owner is the `form` element, with the exception of `input` elements whose `type` attribute is in the Image Button state, which must, for historical reasons, be excluded from this particular collection.

The **`length`** DOM attribute must return the number of nodes represented by the `elements` collection.

The indices of the supported indexed properties at any instant are the indicies supported by the object returned by the `elements` attribute at that instant.

The **`item(index)`** method must return the value returned by the method of the same name on the `elements` collection, when invoked with the same argument.

Each `form` element has a mapping of names to elements called the **past names map**. It is used to persist names of controls even when they change names.

The names of the supported named properties are the union of the names currently supported by the object returned by the `elements` attribute, and the names currently in the past names map.

The **`namedItem(name)`** method, when called, must run the following steps:

1. If *name* is one of the names of the supported named properties of the object returned by the `elements` attribute, then run these substeps:

    1. Let *candidate* be the object returned by the `namedItem()` method on the object returned by the `elements` attribute when passed the *name* argument.

    2. If *candidate* is an element, then add a mapping from *name* to *candidate* in the `form` element's past names map, replacing the previous entry with the same name, if any.

    3. Return *candidate* and abort these steps.

2. Otherwise, *name* is the name of one of the entries in the `form` element's past names map: return the object associated with *name* in that map.

If an element listed in the `form` element's past names map is removed from the `Document`, then its entries must be removed from the map.

The **`submit()`** method, when invoked, must submit the `form` element from the `form` element itself.

The **`reset()`** method, when invoked, must reset the `form` element.

If the **`checkValidity()`** method is invoked, the user agent must statically validate the constraints of the `form` element, and return true if the constraint validation return a *positive* result, and false if it returned a *negative* result.

If the **`dispatchFormInput()`** method is invoked, the user agent must broadcast `forminput` events from the `form` element.

If the **`dispatchFormChange()`** method is invoked, the user agent must broadcast `formchange` events from the `form` element.

4.10.2 The `fieldset` element

Categories

Flow content.

Listed form-associated element.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

One `legend` element follwed by flow content.

Content attributes:

Global attributes
`disabled`
`form`
`name`

DOM interface:

```
interface HTMLFieldSetElement : HTMLElement {
           attribute boolean disabled;
  readonly attribute HTMLFormElement form;
           attribute DOMString name;

  readonly attribute DOMString type;

  readonly attribute HTMLFormControlsCollection elements;

  readonly attribute boolean willValidate;
  readonly attribute ValidityState validity;
  readonly attribute DOMString validationMessage;
  boolean checkValidity();
  void setCustomValidity(in DOMString error);
};
```

The `fieldset` element represents a set of form controls grouped under a common name.

The name of the group is given by the first `legend` element that is a child of the `fieldset` element. The remainder of the descendants form the group.

The **`disabled`** attribute, when specified, causes all the form control descendants of the `fieldset` element to be disabled.

The `form` attribute is used to explicitly associate the `fieldset` element with its form owner. The `name` attribute represents the element's name.

fieldset **`.type`**

Returns the string "fieldset".

fieldset **`.elements`**

Returns an `HTMLCollection` of the form controls in the element.

The **`disabled`** DOM attribute must reflect the content attribute of the same name.

The **`type`** DOM attribute must return the string "`fieldset`".

The **`elements`** DOM attribute must return an `HTMLFormControlsCollection` rooted at the `fieldset` element, whose filter matches listed elements.

The `willValidate`, `validity`, and `validationMessage` attributes, and the `checkValidity()` and

`setCustomValidity()` methods, are part of the constraint validation API.

Constraint validation: `fieldset` elements are always barred from constraint validation.

4.10.3 The `label` element

Categories

Flow content.
Phrasing content.
Interactive content.
Form-associated element.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content, but with no descendant labelable form-associated elements unless it is the element's labeled control, and no descendant `label` elements.

Content attributes:

Global attributes
`form`
`for`

DOM interface:

```
interface HTMLLabelElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute DOMString htmlFor;
  readonly attribute HTMLElement control;
};
```

The `label` represents a caption in a user interface. The caption can be associated with a specific form control, known as the `label` element's **labeled control**, either using `for` attribute, or by putting the form control inside the `label` element itself.

Unless otherwise specified by the following rules, a `label` element has no labeled control.

The **`for`** attribute may be specified to indicate a form control with which the caption is to be associated. If the attribute is specified, the attribute's value must be the ID of a labelable form-associated element in the same `Document` as the `label` element. If the attribute is specified and there is an element in the `Document` whose ID is equal to the value of the `for` attribute, and the first such element is a labelable form-associated element, then that element is the `label` element's labeled control.

If the `for` attribute is not specified, but the `label` element has a labelable form-associated element descendant, then the first such descendant in tree order is the `label` element's labeled control.

The `label` element's exact default presentation and behavior, in particular what its activation behavior might be, if anything, should match the platform's label behavior.

> For example, on platforms where clicking a checkbox label checks the checkbox, clicking the `label` in the following snippet could trigger the user agent to run synthetic click activation steps on the `input` element, as if the element itself had been triggered

by the user:

```
<label><input type=checkbox name=lost> Lost</label>
```

On other platforms, the behavior might be just to focus the control, or do nothing.

***label* . `control`**

Returns the form control that is associated with this element.

The `form` attribute is used to explicitly associate the `label` element with its form owner.

The **`htmlFor`** DOM attribute must reflect the `for` content attribute.

The **`control`** DOM attribute must return the `label` element's labeled control, if any, or null if there isn't one.

***control* . `labels`**

Returns a `NodeList` of all the `label` elements that the form control is associated with.

Labelable form-associated elements have a `NodeList` object associated with them that represents the list of `label` elements, in tree order, whose labeled control is the element in question. The **`labels`** DOM attribute of labelable form-associated elements, on getting, must return that `NodeList` object.

4.10.4 The `input` element

Categories

Flow content.
Phrasing content.
If the `type` attribute is *not* in the Hidden state: Interactive content.
Listed, labelable, submittable, and resettable form-associated element.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Empty.

Content attributes:

Global attributes
`accept`
`alt`
`autocomplete`
`autofocus`
`checked`
`disabled`

form
formaction
formenctype
formmethod
formnovalidate
formtarget
height
list
max
maxlength
min
multiple
name
pattern
placeholder
readonly
required
size
src
step
type
value
width

DOM interface:

```
interface HTMLInputElement : HTMLElement {
           attribute DOMString accept;
           attribute DOMString alt;
           attribute boolean autocomplete;
           attribute boolean autofocus;
           attribute boolean defaultChecked;
           attribute boolean checked;
           attribute boolean disabled;
  readonly attribute HTMLFormElement form;
           attribute DOMString formAction;
           attribute DOMString formEnctype;
           attribute DOMString formMethod;
           attribute boolean formNoValidate;
           attribute DOMString formTarget;
           attribute DOMString height;
           attribute boolean indeterminate;
  readonly attribute HTMLElement list;
           attribute DOMString max;
           attribute long maxLength;
           attribute DOMString min;
           attribute boolean multiple;
           attribute DOMString name;
           attribute DOMString pattern;
           attribute DOMString placeholder;
           attribute boolean readOnly;
           attribute boolean required;
           attribute unsigned long size;
           attribute DOMString src;
           attribute DOMString step;
           attribute DOMString type;
           attribute DOMString defaultValue;
           attribute DOMString value;
           attribute Date valueAsDate;
           attribute float valueAsNumber;
  readonly attribute HTMLOptionElement selectedOption;
           attribute DOMString width;
```

```
  void stepUp(in long n);
  void stepDown(in long n);

  readonly attribute boolean willValidate;
  readonly attribute ValidityState validity;
  readonly attribute DOMString validationMessage;
  boolean checkValidity();
  void setCustomValidity(in DOMString error);

  readonly attribute NodeList labels;

  void select();
       attribute unsigned long selectionStart;
       attribute unsigned long selectionEnd;
  void setSelectionRange(in unsigned long start, in unsigned long end);
};
```

The `input` element represents a typed data field, usually with a form control to allow the user to edit the data.

The **`type`** attribute controls the data type (and associated control) of the element. It is an enumerated attribute. The following table lists the keywords and states for the attribute — the keywords in the left column map to the states in the cell in the second column on the same row as the keyword.

Keyword	State	Data type	Control type
`hidden`	Hidden	An arbitrary string	n/a
`text`	Text	Text with no line breaks	Text field
`search`	Search	Text with no line breaks	Search field
`url`	URL	An absolute IRI	A text field
`email`	E-mail	An e-mail address or list of e-mail addresses	A text field
`password`	Password	Text with no line breaks (sensitive information)	Text field that obscures data entry
`datetime`	Date and Time	A date and time (year, month, day, hour, minute, second, fraction of a second) with the time zone set to UTC	A date and time control
`date`	Date	A date (year, month, day) with no time zone	A date control
`month`	Month	A date consisting of a year and a month with no time zone	A month control
`week`	Week	A date consisting of a week-year number and a week number with no time zone	A week control
`time`	Time	A time (hour, minute, seconds, fractional seconds) with no time zone	A time control
`datetime-local`	Local Date and Time	A date and time (year, month, day, hour, minute, second, fraction of a second) with no time zone	A date and time control
`number`	Number	A numerical value	A text field or spinner control

Keyword	State	Data type	Control type
`range`	Range	A numerical value, with the extra semantic that the exact value is not important	A slider control or similar
`color`	Color	An sRGB color with 8-bit red, green, and blue components	A color well
`checkbox`	Checkbox	A set of zero or more values from a predefined list	A checkbox
`radio`	Radio Button	An enumerated value	A radio button
`file`	File Upload	Zero or more files each with a MIME type and optionally a file name	A label and a button
`submit`	Submit Button	An enumerated value, with the extra semantic that it must be the last value selected and initiates form submission	A button
`image`	Image Button	A coordinate, relative to a particular image's size, with the extra semantic that it must be the last value selected and initiates form submission	Either a clickable image, or a button
`reset`	Reset Button	n/a	A button
`button`	Button	n/a	A button

The *missing value default* is the Text state.

Which of the `accept`, `alt`, `autocomplete`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `list`, `max`, `maxlength`, `min`, `multiple`, `pattern`, `readonly`, `required`, `size`, `src`, `step`, and `width` attributes apply to an `input` element depends on the state of its `type` attribute. Similarly, the `checked`, `valueAsDate`, `valueAsNumber`, `list`, and `selectedOption` DOM attributes, and the `stepUp()` and `stepDown()` methods, are specific to certain states. The following table is non-normative and summarises which content attributes, DOM attrbutes, and methods apply to each state:

	Hidden	Text, Search, URL	E-mail	Password	Date and Time, Date, Month, Week, Time	Local Date and Tir Number
`accept`	.	.	.	.	.	.
`alt`	.	.	.	.	.	.
`autocomplete`	.	Yes	Yes	Yes	Yes	Yes
`checked`	.	.	.	.	.	.
`formaction`	.	.	.	.	.	.
`formenctype`	.	.	.	.	.	.
`formmethod`	.	.	.	.	.	.
`formnovalidate`	.	.	.	.	.	.

	Hidden	Text, Search, URL	E-mail	Password	Date and Time, Date, Month, Week, Time	Local Date and Tir Number
formtarget	.	.	.	.	.	.
height	.	.	.	.	.	.
list	.	Yes	Yes	.	Yes	Yes
max	.	.	.	.	Yes	Yes
maxlength	.	Yes	Yes	Yes	.	.
min	.	.	.	.	Yes	Yes
multiple	.	.	Yes	.	.	.
pattern	.	Yes	Yes	Yes	.	.
placeholder	.	Yes	Yes	Yes	.	.
readonly	.	Yes	Yes	Yes	Yes	Yes
required	.	Yes	Yes	Yes	Yes	Yes
size	.	Yes	Yes	Yes	.	.
src	.	.	.	.	.	.
step	.	.	.	.	Yes	Yes
width	.	.	.	.	.	.
checked	.	.	.	.	.	.
value	value	value	value	value	value	value
valueAsDate	.	.	.	.	Yes	.
valueAsNumber	.	.	.	.	Yes	Yes
list	.	Yes	Yes	.	Yes	Yes
selectedOption	.	Yes	Yes	.	Yes	Yes
select()	.	Yes	Yes	Yes	.	.
selectionStart	.	Yes	Yes	Yes	.	.
selectionEnd	.	Yes	Yes	Yes	.	.
setSelectionRange()	.	Yes	Yes	Yes	.	.
stepDown()	.	.	.	.	Yes	Yes
stepUp()	.	.	.	.	Yes	Yes
input **event**	.	Yes	Yes	Yes	Yes	Yes
change **event**	.	Yes	Yes	Yes	Yes	Yes

When an input element's type attribute changes state, and when the element is first created, the element's rendering and behaviour must change to the new state's accordingly and the **value sanitization algorithm**, if one is defined for the type attribute's new state, must be invoked.

Each input element has a value, which is exposed by the value DOM attribute. Some states

define an **algorithm to convert a string to a number**, an **algorithm to convert a number to a string**, an **algorithm to convert a string to a `Date` object**, and an **algorithm to convert a `Date` object to a string**, which are used by `max`, `min`, `step`, `valueAsDate`, `valueAsNumber`, `stepDown()`, and `stepUp()`.

Each `input` element has a boolean **dirty value flag**. When it is true, the element is said to have a ***dirty value***. The dirty value flag must be initially set to false when the element is created, and must be set to true whenever the user interacts with the control in a way that changes the value.

The **`value`** content attribute gives the default value of the `input` element. When the `value` content attribute is added, set, or removed, if the control does not have a *dirty value*, the user agent must set the value of the element to the value of the `value` content attribute, if there is one, or the empty string otherwise, and then run the current value sanitization algorithm, if one is defined.

Each `input` element has a checkedness, which is exposed by the `checked` DOM attribute.

Each `input` element has a boolean **dirty checkedness flag**. When it is true, the element is said to have a ***dirty checkedness***. The dirty checkedness flag must be initially set to false when the element is created, and must be set to true whenever the user interacts with the control in a way that changes the checkedness.

The **`checked`** content attribute gives the default checkedness of the `input` element. When the `checked` content attribute is added, if the control does not have *dirty checkedness*, the user agent must set the checkedness of the element to true; when the `checked` content attribute is removed, if the control does not have *dirty checkedness*, the user agent must set the checkedness of the element to false.

The reset algorithm for `input` elements is to set the dirty value flag and dirty checkedness flag back to false, set the value of the element to the value of the `value` content attribute, if there is one, or the empty string otherwise, set the checkedness of the element to true if the element has a `checked` content attribute and false if it does not, and then invoke the value sanitization algorithm, if the `type` attribute's current state defines one.

Each `input` element has a boolean **mutability flag**. When it is true, the element is said to be ***mutable***, and when it is false the element is ***immutable***. Unless otherwise specified, an `input` element is always *mutable*. Unless otherwise specified, the user agent should not allow the user to modify the element's value or checkedness.

When an `input` element is disabled, it is *immutable*.

When an `input` element does not have a `Document` node as one of its ancestors (i.e. when it is not in the document), it is *immutable*.

> ***Note: The `readonly` attribute can also in some cases (e.g. for the Date state, but not the Checkbox state) make an `input` element immutable.***

The `form` attribute is used to explicitly associate the `input` element with its form owner. The `name` attribute represents the element's name. The `disabled` attribute is used to make the control non-interactive and to prevent its value from being submitted. The `autofocus` attribute controls focus.

The **indeterminate** DOM attribute must initially be set to false. On getting, it must return the last value it was set to. On setting, it must be set to the new value. It has no effect except for changing the appearance of checkbox controls.

The **accept**, **alt**, **autocomplete**, **max**, **min**, **multiple**, **pattern**, **placeholder**, **required**, **size**, **src**, **step**, and **type** DOM attributes must reflect the respective content attributes of the same name. The **maxLength** DOM attribute must reflect the `maxlength` content attribute. The **readOnly** DOM attribute must reflect the `readonly` content attribute. The **defaultChecked** DOM attribute must reflect the `checked` content attribute. The **defaultValue** DOM attribute must reflect the `value` content attribute.

The `willValidate`, `validity`, and `validationMessage` attributes, and the `checkValidity()` and `setCustomValidity()` methods, are part of the constraint validation API. The `labels` attribute provides a list of the element's `label`s. The `select()`, `selectionStart`, `selectionEnd`, and `setSelectionRange()` methods and attributes expose the element's text selection.

4.10.4.1 States of the `type` attribute

4.10.4.1.1 Hidden state

When an `input` element's `type` attribute is in the Hidden state, the rules in this section apply.

The `input` element represents a value that is not intended to be examined or manipulated by the user.

Constraint validation: If an `input` element's `type` attribute is in the Hidden state, it is barred from constraint validation.

If the `name` attribute is present and has a value that is a case-sensitive match for the string "`_charset_`", then the element's `value` attribute must be omitted.

Bookkeeping details

- The `value` DOM attribute applies to this element and is in mode value.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `autocomplete`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `list`, `max`, `maxlength`, `min`, `multiple`, `pattern`, `placeholder`, `readonly`, `required`, `size`, `src`, `step`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `list`, `selectedOption`, `selectionStart`, `selectionEnd`, `valueAsDate`, and `valueAsNumber` DOM attributes; `select()`, `setSelectionRange()`, `stepDown()`, and `stepUp()` methods.
- The `input` and `change` events do not apply.

4.10.4.1.2 Text state and Search state

When an `input` element's `type` attribute is in the Text state or the Search state, the rules in this section apply.

The `input` element represents a one line plain text edit control for the element's value.

If the element is *mutable*, its value should be editable by the user. User agents must not allow users to insert U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters

into the element's value.

The value attribute, if specified, must have a value that contains no U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters.

The value sanitization algorithm is as follows: Strip line breaks from the value.

Bookkeeping details

- The following common input element content attributes, DOM attributes, and methods apply to the element: autocomplete, list, maxlength, pattern, placeholder, readonly, required, and size content attributes; list, selectedOption, selectionStart, selectionEnd, and value DOM attributes; select() and setSelectionRange() methods.
- The value DOM attribute is in mode value.
- The input and change events apply.
- The following content attributes must not be specified and do not apply to the element: accept, alt, checked, formaction, formenctype, formmethod, formnovalidate, formtarget, height, max, min, multiple, src, step, and width.
- The following DOM attributes and methods do not apply to the element: checked, valueAsDate, and valueAsNumber DOM attributes; stepDown() and stepUp() methods.

4.10.4.1.3 URL STATE

When an input element's type attribute is in the URL state, the rules in this section apply.

The input element represents a control for editing a single absolute URL given in the element's value.

If the is *mutable*, the user agent should allow the user to change the URL represented by its value. User agents may allow the user to set the value to a string that is not a valid absolute URL, but may also or instead automatically escape characters entered by the user so that the value is always a valid absolute URL (even if that isn't the actual value seen and edited by the user in the interface). User agents should allow the user to set the value to the empty string. User agents must not allow users to insert U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters into the value.

The value attribute, if specified, must have a value that is a valid absolute URL.

The value sanitization algorithm is as follows: Strip line breaks from the value.

Constraint validation: While the value of the element is not a valid absolute URL, the element is suffering from a type mismatch.

Bookkeeping details

- The following common input element content attributes, DOM attributes, and methods apply to the element: autocomplete, list, maxlength, pattern, placeholder, readonly, required, and size content attributes; list, selectedOption, selectionStart, selectionEnd, and value DOM attributes; select() and setSelectionRange() methods.
- The value DOM attribute is in mode value.
- The input and change events apply.
- The following content attributes must not be specified and do not apply to the element: accept, alt, checked, formaction, formenctype, formmethod, formnovalidate, formtarget, height, max, min, multiple, src, step, and width.

- The following DOM attributes and methods do not apply to the element: `checked`, `valueAsDate`, and `valueAsNumber` DOM attributes; `stepDown()` and `stepUp()` methods.

4.10.4.1.4 E-MAIL STATE

When an `input` element's `type` attribute is in the E-mail state, the rules in this section apply.

The `input` element represents a control for editing a list of e-mail addresses given in the element's value.

If the element is *mutable*, the user agent should allow the user to change the e-mail addresses represented by its value. If the `multiple` attribute is specified, then the user agent should allow the user to select or provide multiple addresses; otherwise, the user agent should act in a manner consistent with expecting the user to provide a single e-mail address. User agents may allow the user to set the value to a string that is not an valid e-mail address list. User agents should allow the user to set the value to the empty string. User agents must not allow users to insert U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters into the value. User agents may transform the value for display and editing (e.g. converting punycode in the value to IDN in the display and vice versa).

If the `multiple` attribute is specified on the element, then the `value` attribute, if specified, must have a value that is a valid e-mail address list; otherwise, the `value` attribute, if specified, must have a value that is a single valid e-mail address.

The value sanitization algorithm is as follows: Strip line breaks from the value.

Constraint validation: If the `multiple` attribute is specified on the element, then, while the value of the element is not a valid e-mail address list, the element is suffering from a type mismatch; otherwise, while the value of the element is not a single valid e-mail address, the element is suffering from a type mismatch.

A **valid e-mail address list** is a set of comma-separated tokens, where each token is itself a valid e-mail address. To obtain the list of tokens from a valid e-mail address list, the user agent must split the string on commas.

A **valid e-mail address** is a string that matches the production `dot-atom "@" dot-atom` where `dot-atom` is defined in RFC 2822 section 3.2.4, excluding the `CFWS` production everywhere. [RFC2822]

Bookkeeping details

- The following common `input` element content attributes, DOM attributes, and methods apply to the element: `autocomplete`, `list`, `maxlength`, `multiple`, `pattern`, `placeholder`, `readonly`, `required`, and `size` content attributes; `list`, `selectedOption`, `selectionStart`, `selectionEnd`, and `value` DOM attributes; `select()` and `setSelectionRange()` methods.
- The `value` DOM attribute is in mode value.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `max`, `min`, `src`, `step`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `valueAsDate`, and `valueAsNumber` DOM attributes; `stepDown()` and `stepUp()` methods.

4.10.4.1.5 Password state

When an `input` element's `type` attribute is in the Password state, the rules in this section apply.

The `input` element represents a one line plain text edit control for the element's value. The user agent should obscure the value so that people other than the user cannot see it.

If the element is *mutable*, its value should be editable by the user. User agents must not allow users to insert U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters into the value.

The `value` attribute, if specified, must have a value that contains no U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters.

The value sanitization algorithm is as follows: Strip line breaks from the value.

Bookkeeping details

- The following common `input` element content attributes, DOM attributes, and methods apply to the element: `autocomplete`, `maxlength`, `pattern`, `placeholder`, `readonly`, `required`, and `size` content attributes; `selectionStart`, `selectionEnd`, and `value` DOM attributes; `select()`, and `setSelectionRange()` methods.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `list`, `max`, `min`, `multiple`, `src`, `step`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `list`, `selectedOption`, `valueAsDate`, and `valueAsNumber` DOM attributes; `stepDown()` and `stepUp()` methods.

4.10.4.1.6 Date and Time state

When an `input` element's `type` attribute is in the Date and Time state, the rules in this section apply.

The `input` element represents a control for setting the element's value to a string representing a specific global date and time. User agents may display the date and time in whatever timezone is appropriate for the user.

If the element is *mutable*, the user agent should allow the user to change the global date and time represented by its value, as obtained by parsing a global date and time from it. User agents must not allow the user to set the value to a string that is not a valid global date and time string expressed in UTC, though user agents may allow the user to set and view the time in another timezone and silently translate the time to and from the UTC timezone in the value. If the user agent provides a user interface for selecting a global date and time, then the value must be set to a valid global date and time string expressed in UTC representing the user's selection. User agents should allow the user to set the value to the empty string.

The `value` attribute, if specified, must have a value that is a valid global date and time string.

The value sanitization algorithm is as follows: If the value of the element is a valid global date and time string, then adjust the time so that the value represents the same point in time

but expressed in the UTC timezone, otherwise, set it to the empty string instead.

The `min` attribute, if specified, must have a value that is a valid global date and time string. The `max` attribute, if specified, must have a value that is a valid global date and time string.

The `step` attribute is expressed in seconds. The step scale factor is 1000 (which converts the seconds to milliseconds, as used in the other algorithms). The default step is 60 seconds.

When the element is suffering from a step mismatch, the user agent may round the element's value to the nearest global date and time for which the element would not suffer from a step mismatch.

The algorithm to convert a string to a number, given a string *input*, is as follows: If parsing a global date and time from *input* results in an error, then return an error; otherwise, return the number of milliseconds elapsed from midnight UTC on the morning of 1970-01-01 (the time represented by the value "`1970-01-01T00:00:00.0Z`") to the parsed global date and time, ignoring leap seconds.

The algorithm to convert a number to a string, given a number *input*, is as follows: Return a valid global date and time string expressed in UTC that represents the global date and time that is *input* milliseconds after midnight UTC on the morning of 1970-01-01 (the time represented by the value "`1970-01-01T00:00:00.0Z`").

The algorithm to convert a string to a `Date` object, given a string *input*, is as follows: If parsing a global date and time from *input* results in an error, then return an error; otherwise, return a `Date` object representing the parsed global date and time, expressed in UTC.

The algorithm to convert a `Date` object to a string, given a `Date` object *input*, is as follows: Return a valid global date and time string expressed in UTC that represents the global date and time that is represented by *input*.

Bookkeeping details

- The following common `input` element content attributes, DOM attributes, and methods apply to the element: `autocomplete`, `list`, `max`, `min`, `readonly`, `required`, and `step` content attributes; `list`, `value`, `valueAsDate`, `valueAsNumber`, and `selectedOption` DOM attributes; `stepDown()` and `stepUp()` methods.
- The `value` DOM attribute is in mode value.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `maxlength`, `multiple`, `pattern`, `placeholder`, `size`, `src`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `selectionStart`, and `selectionEnd` DOM attributes; `select()` and `setSelectionRange()` methods.

4.10.4.1.7 Date state

When an `input` element's `type` attribute is in the Date state, the rules in this section apply.

The `input` element represents a control for setting the element's value to a string representing a specific date.

If the element is *mutable*, the user agent should allow the user to change the date represented by its value, as obtained by parsing a date from it. User agents must not allow

the user to set the value to a string that is not a valid date string. If the user agent provides a user interface for selecting a date, then the value must be set to a valid date string representing the user's selection. User agents should allow the user to set the value to the empty string.

The `value` attribute, if specified, must have a value that is a valid date string.

The value sanitization algorithm is as follows: If the value of the element is not a valid date string, then set it to the empty string instead.

The `min` attribute, if specified, must have a value that is a valid date string. The `max` attribute, if specified, must have a value that is a valid date string.

The `step` attribute is expressed in days. The step scale factor is 86,400,000 (which converts the days to milliseconds, as used in the other algorithms). The default step is 1 day.

When the element is suffering from a step mismatch, the user agent may round the element's value to the nearest date for which the element would not suffer from a step mismatch.

The algorithm to convert a string to a number, given a string *input*, is as follows: If parsing a date from *input* results in an error, then return an error; otherwise, return the number of milliseconds elapsed from midnight UTC on the morning of 1970-01-01 (the time represented by the value "`1970-01-01T00:00:00.0Z`") to midnight UTC on the morning of the parsed date, ignoring leap seconds.

The algorithm to convert a number to a string, given a number *input*, is as follows: Return a valid date string that represents the date that, in UTC, is current *input* milliseconds after midnight UTC on the morning of 1970-01-01 (the time represented by the value "`1970-01-01T00:00:00.0Z`").

The algorithm to convert a string to a `Date` object, given a string *input*, is as follows: If parsing a date from *input* results in an error, then return an error; otherwise, return a `Date` object representing midnight UTC on the morning of the parsed date.

The algorithm to convert a `Date` object to a string, given a `Date` object *input*, is as follows: Return a valid date string that represents the date current at the time represented by *input* in the UTC timezone.

Bookkeeping details

- The following common `input` element content attributes, DOM attributes, and methods apply to the element: `autocomplete`, `list`, `max`, `min`, `readonly`, `required`, and `step` content attributes; `list`, `value`, `valueAsDate`, `valueAsNumber`, and `selectedOption` DOM attributes; `stepDown()` and `stepUp()` methods.
- The `value` DOM attribute is in mode value.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `maxlength`, `multiple`, `pattern`, `placeholder`, `size`, `src`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `selectionStart`, and `selectionEnd` DOM attributes; `select()` and `setSelectionRange()` methods.

4.10.4.1.8 Month state

When an input element's type attribute is in the Month state, the rules in this section apply.

The input element represents a control for setting the element's value to a string representing a specific month.

If the element is *mutable*, the user agent should allow the user to change the month represented by its value, as obtained by parsing a month from it. User agents must not allow the user to set the value to a string that is not a valid month string. If the user agent provides a user interface for selecting a month, then the value must be set to a valid month string representing the user's selection. User agents should allow the user to set the value to the empty string.

The value attribute, if specified, must have a value that is a valid month string.

The value sanitization algorithm is as follows: If the value of the element is not a valid month string, then set it to the empty string instead.

The min attribute, if specified, must have a value that is a valid month string. The max attribute, if specified, must have a value that is a valid month string.

The step attribute is expressed in months. The step scale factor is 1 (there is no conversion needed as the algorithms use months). The default step is 1 month.

When the element is suffering from a step mismatch, the user agent may round the element's value to the nearest month for which the element would not suffer from a step mismatch.

The algorithm to convert a string to a number, given a string *input*, is as follows: If parsing a month time from *input* results in an error, then return an error; otherwise, return the number of months between January 1970 and the parsed month.

The algorithm to convert a number to a string, given a number *input*, is as follows: Return a valid month string that represents the month that has *input* months between it and January 1970.

The algorithm to convert a string to a `Date` object, given a string *input*, is as follows: If parsing a month from *input* results in an error, then return an error; otherwise, return a `Date` object representing midnight UTC on the morning of the first day of the parsed month.

The algorithm to convert a `Date` object to a string, given a `Date` object *input*, is as follows: Return a valid month string that represents the month current at the time represented by *input* in the UTC timezone.

Bookkeeping details

- The following common input element content attributes, DOM attributes, and methods apply to the element: autocomplete, list, max, min, readonly, required, and step content attributes; list, value, valueAsDate, valueAsNumber, and selectedOption DOM attributes; stepDown() and stepUp() methods.
- The value DOM attribute is in mode value.
- The input and change events apply.
- The following content attributes must not be specified and do not apply to the element: accept, alt, checked, formaction, formenctype, formmethod, formnovalidate, formtarget, height, maxlength, multiple, pattern, placeholder, size, src, and width.
- The following DOM attributes and methods do not apply to the element: checked, selectionStart, and

selectionEnd DOM attributes; select() and setSelectionRange() methods.

4.10.4.1.9 Week state

When an input element's type attribute is in the Week state, the rules in this section apply.

The input element represents a control for setting the element's value to a string representing a specific week.

If the element is *mutable*, the user agent should allow the user to change the week represented by its value, as obtained by parsing a week from it. User agents must not allow the user to set the value to a string that is not a valid week string. If the user agent provides a user interface for selecting a week, then the value must be set to a valid week string representing the user's selection. User agents should allow the user to set the value to the empty string.

The value attribute, if specified, must have a value that is a valid week string.

The value sanitization algorithm is as follows: If the value of the element is not a valid week string, then set it to the empty string instead.

The min attribute, if specified, must have a value that is a valid week string. The max attribute, if specified, must have a value that is a valid week string.

The step attribute is expressed in weeks. The step scale factor is 604,800,000 (which converts the weeks to milliseconds, as used in the other algorithms). The default step is 1 week.

When the element is suffering from a step mismatch, the user agent may round the element's value to the nearest week for which the element would not suffer from a step mismatch.

The algorithm to convert a string to a number, given a string *input*, is as follows: If parsing a week string from *input* results in an error, then return an error; otherwise, return the number of milliseconds elapsed from midnight UTC on the morning of 1970-01-01 (the time represented by the value "`1970-01-01T00:00:00.0Z`") to midnight UTC on the morning of the Monday of the parsed week, ignoring leap seconds.

The algorithm to convert a number to a string, given a number *input*, is as follows: Return a valid week string that represents the week that, in UTC, is current *input* milliseconds after midnight UTC on the morning of 1970-01-01 (the time represented by the value "`1970-01-01T00:00:00.0Z`").

The algorithm to convert a string to a `Date` object, given a string *input*, is as follows: If parsing a week from *input* results in an error, then return an error; otherwise, return a `Date` object representing midnight UTC on the morning of the Monday of the parsed week.

The algorithm to convert a `Date` object to a string, given a `Date` object *input*, is as follows: Return a valid week string that represents the week current at the time represented by *input* in the UTC timezone.

Bookkeeping details

- The following common input element content attributes, DOM attributes, and methods apply to the element:

`autocomplete`, `list`, `max`, `min`, `readonly`, `required`, and `step` content attributes; `list`, `value`, `valueAsDate`, `valueAsNumber`, and `selectedOption` DOM attributes; `stepDown()` and `stepUp()` methods.

- The `value` DOM attribute is in mode value.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `maxlength`, `multiple`, `pattern`, `placeholder`, `size`, `src`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `selectionStart`, and `selectionEnd` DOM attributes; `select()` and `setSelectionRange()` methods.

4.10.4.1.10 Time state

When an `input` element's `type` attribute is in the Time state, the rules in this section apply.

The `input` element represents a control for setting the element's value to a string representing a specific time.

If the element is *mutable*, the user agent should allow the user to change the time represented by its value, as obtained by parsing a time from it. User agents must not allow the user to set the value to a string that is not a valid time string. If the user agent provides a user interface for selecting a time, then the value must be set to a valid time string representing the user's selection. User agents should allow the user to set the value to the empty string.

The `value` attribute, if specified, must have a value that is a valid time string.

The value sanitization algorithm is as follows: If the value of the element is not a valid time string, then set it to the empty string instead.

The `min` attribute, if specified, must have a value that is a valid time string. The `max` attribute, if specified, must have a value that is a valid time string.

The `step` attribute is expressed in seconds. The step scale factor is 1000 (which converts the seconds to milliseconds, as used in the other algorithms). The default step is 60 seconds.

When the element is suffering from a step mismatch, the user agent may round the element's value to the nearest time for which the element would not suffer from a step mismatch.

The algorithm to convert a string to a number, given a string *input*, is as follows: If parsing a time from *input* results in an error, then return an error; otherwise, return the number of milliseconds elapsed from midnight to the parsed time on a day with no time changes.

The algorithm to convert a number to a string, given a number *input*, is as follows: Return a valid time string that represents the time that is *input* milliseconds after midnight on a day with no time changes.

The algorithm to convert a string to a `Date` object, given a string *input*, is as follows: If parsing a time from *input* results in an error, then return an error; otherwise, return a `Date` object representing the parsed time in UTC on 1970-01-01.

The algorithm to convert a `Date` object to a string, given a `Date` object *input*, is as follows: Return a valid time string that represents the UTC time component that is

represented by *input*.

Bookkeeping details

- The following common `input` element content attributes, DOM attributes, and methods apply to the element: `autocomplete`, `list`, `max`, `min`, `readonly`, `required`, and `step` content attributes; `list`, `value`, `valueAsDate`, `valueAsNumber`, and `selectedOption` DOM attributes; `stepDown()` and `stepUp()` methods.
- The `value` DOM attribute is in mode value.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `maxlength`, `multiple`, `pattern`, `placeholder`, `size`, `src`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `selectionStart`, and `selectionEnd` DOM attributes; `select()` and `setSelectionRange()` methods.

4.10.4.1.11 Local Date and Time state

When an `input` element's `type` attribute is in the Local Date and Time state, the rules in this section apply.

The `input` element represents a control for setting the element's value to a string representing a local date and time, with no time zone information.

If the element is *mutable*, the user agent should allow the user to change the date and time represented by its value, as obtained by parsing a date and time from it. User agents must not allow the user to set the value to a string that is not a valid local date and time string. If the user agent provides a user interface for selecting a local date and time, then the value must be set to a valid local date and time string representing the user's selection. User agents should allow the user to set the value to the empty string.

The `value` attribute, if specified, must have a value that is a valid local date and time string.

The value sanitization algorithm is as follows: If the value of the element is not a valid local date and time string, then set it to the empty string instead.

The `min` attribute, if specified, must have a value that is a valid local date and time string. The `max` attribute, if specified, must have a value that is a valid local date and time string.

The `step` attribute is expressed in seconds. The step scale factor is 1000 (which converts the seconds to milliseconds, as used in the other algorithms). The default step is 60 seconds.

When the element is suffering from a step mismatch, the user agent may round the element's value to the nearest local date and time for which the element would not suffer from a step mismatch.

The algorithm to convert a string to a number, given a string *input*, is as follows: If parsing a date and time from *input* results in an error, then return an error; otherwise, return the number of milliseconds elapsed from midnight on the morning of 1970-01-01 (the time represented by the value "`1970-01-01T00:00:00.0`") to the parsed local date and time, ignoring leap seconds.

The algorithm to convert a number to a string, given a number *input*, is as follows: Return a valid local date and time string that represents the date and time that is *input*

milliseconds after midnight on the morning of 1970-01-01 (the time represented by the value "1970-01-01T00:00:00.0").

Bookkeeping details

- The following common input element content attributes, DOM attributes, and methods apply to the element: autocomplete, list, max, min, readonly, required, and step content attributes; list, value, valueAsNumber, and selectedOption DOM attributes; stepDown() and stepUp() methods.
- The value DOM attribute is in mode value.
- The input and change events apply.
- The following content attributes must not be specified and do not apply to the element: accept, alt, checked, formaction, formenctype, formmethod, formnovalidate, formtarget, height, maxlength, multiple, pattern, placeholder, size, src, and width.
- The following DOM attributes and methods do not apply to the element: checked, selectionStart, selectionEnd, and valueAsDate DOM attributes; select() and setSelectionRange() methods.

4.10.4.1.12 Number state

When an input element's type attribute is in the Number state, the rules in this section apply.

The input element represents a control for setting the element's value to a string representing a number.

If the element is *mutable*, the user agent should allow the user to change the number represented by its value, as obtained from applying the rules for parsing floating point number values to it. User agents must not allow the user to set the value to a string that is not a valid floating point number. If the user agent provides a user interface for selecting a number, then the value must be set to the best representation of the floating point number representing the user's selection. User agents should allow the user to set the value to the empty string.

The value attribute, if specified, must have a value that is a valid floating point number.

The value sanitization algorithm is as follows: If the value of the element is not a valid floating point number, then set it to the empty string instead.

The min attribute, if specified, must have a value that is a valid floating point number. The max attribute, if specified, must have a value that is a valid floating point number.

The step scale factor is 1. The default step is 1 (allowing only integers, unless the min attribute has a non-integer value).

When the element is suffering from a step mismatch, the user agent may round the element's value to the nearest number for which the element would not suffer from a step mismatch.

The algorithm to convert a string to a number, given a string *input*, is as follows: If applying the rules for parsing floating point number values to *input* results in an error, then return an error; otherwise, return the resulting number.

The algorithm to convert a number to a string, given a number *input*, is as follows: Return a valid floating point number that represents *input*.

Bookkeeping details

- The following common input element content attributes, DOM attributes, and methods apply to the element:

`autocomplete`, `list`, `max`, `min`, `readonly`, `required`, and `step` content attributes; `list`, `value`, `valueAsNumber`, and `selectedOption` DOM attributes; `stepDown()` and `stepUp()` methods.

- The `value` DOM attribute is in mode value.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `maxlength`, `multiple`, `pattern`, `placeholder`, `size`, `src`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `selectionStart`, `selectionEnd`, and `valueAsDate` DOM attributes; `select()` and `setSelectionRange()` methods.

4.10.4.1.13 Range state

When an `input` element's `type` attribute is in the Range state, the rules in this section apply.

The `input` element represents a control for setting the element's value to a string representing a number, but with the caveat that the exact value is not important, letting UAs provide a simpler interface than they do for the Number state.

> ***Note: In this state, the range and step constraints are enforced even during user input, and there is no way to set the value to the empty string.***

If the element is *mutable*, the user agent should allow the user to change the number represented by its value, as obtained from applying the rules for parsing floating point number values to it. User agents must not allow the user to set the value to a string that is not a valid floating point number. If the user agent provides a user interface for selecting a number, then the value must be set to a best representation of the floating point number representing the user's selection. User agents must not allow the user to set the value to the empty string.

The `value` attribute, if specified, must have a value that is a valid floating point number.

The value sanitization algorithm is as follows: If the value of the element is not a valid floating point number, then set it to a valid floating point number that represents the default value.

The `min` attribute, if specified, must have a value that is a valid floating point number. The default minimum is 0. The `max` attribute, if specified, must have a value that is a valid floating point number. The default maximum is 100.

The **default value** is the minimum plus half the difference between the minimum and the maximum, unless the maximum is less than the minimum, in which case the default value is the minimum.

When the element is suffering from an underflow, the user agent must set the element's value to a valid floating point number that represents the minimum.

When the element is suffering from an overflow, if the maximum is not less than the minimum, the user agent must set the element's value to a valid floating point number that represents the maximum.

The step scale factor is 1. The default step is 1 (allowing only integers, unless the `min` attribute has a non-integer value).

When the element is suffering from a step mismatch, the user agent must round the element's value to the nearest number for which the element would not suffer from a step mismatch, and which is greater than or equal to the minimum, and, if the maximum is not less than the minimum, which is less than or equal to the maximum.

The algorithm to convert a string to a number, given a string *input*, is as follows: If applying the rules for parsing floating point number values to *input* results in an error, then return an error; otherwise, return the resulting number.

The algorithm to convert a number to a string, given a number *input*, is as follows: Return a valid floating point number that represents *input*.

Bookkeeping details

- The following common `input` element content attributes, DOM attributes, and methods apply to the element: `autocomplete`, `list`, `max`, `min`, and `step` content attributes; `list`, `value`, `valueAsNumber`, and `selectedOption` DOM attributes; `stepDown()` and `stepUp()` methods.
- The `value` DOM attribute is in mode value.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `maxlength`, `multiple`, `pattern`, `placeholder`, `readonly`, `required`, `size`, `src`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `selectionStart`, `selectionEnd`, and `valueAsDate` DOM attributes; `select()` and `setSelectionRange()` methods.

4.10.4.1.14 Color state

When an `input` element's `type` attribute is in the Color state, the rules in this section apply.

The `input` element represents a color well control, for setting the element's value to a string representing a simple color.

> ***Note: In this state, there is always a color picked, and there is no way to set the value to the empty string.***

If the element is *mutable*, the user agent should allow the user to change the color represented by its value, as obtained from applying the rules for parsing simple color values to it. User agents must not allow the user to set the value to a string that is not a valid lowercase simple color. If the user agent provides a user interface for selecting a color, then the value must be set to the result of using the rules for serialising simple color values to the user's selection. User agents must not allow the user to set the value to the empty string.

The `value` attribute, if specified, must have a value that is a valid simple color.

The value sanitization algorithm is as follows: If the value of the element is a valid simple color, then set it to the value of the element converted to lowercase; otherwise, set it to the string "`#000000`".

Bookkeeping details

- The following common `input` element content attributes, DOM attributes, and methods apply to the element: `autocomplete` and `list` content attributes; `list`, `value`, and `selectedOption` DOM attributes.

- The `value` DOM attribute is in mode value.
- The `input` and `change` events apply.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `maxlength`, `max`, `min`, `multiple`, `pattern`, `placeholder`, `readonly`, `required`, `size`, `src`, `step`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `selectionStart`, `selectionEnd`, `valueAsDate`, and `valueAsNumber` DOM attributes; `select()`, `setSelectionRange()`, `stepDown()`, and `stepUp()` methods.

4.10.4.1.15 Checkbox state

When an `input` element's `type` attribute is in the Checkbox state, the rules in this section apply.

The `input` element represents a two-state control that represents the element's checkedness state. If the element's checkedness state is true, the control represents a positive selection, and if it is false, a negative selection. If the element's `indeterminate` DOM attribute is set to true, then the control's selection should be obscured as if the control was in a third, indeterminate, state.

> ***Note: The control is never a true tri-state control, even if the element's `indeterminate` DOM attribute is set to true. The `indeterminate` DOM attribute only gives the appearance of a third state.***

If the element is *mutable*, then: The pre-click activation steps consist of setting the element's checkedness to its opposite value (i.e. true if it is false, false if it is true), and of setting the element's `indeterminate` DOM attribute to false. The canceled activation steps consist of setting the checkedness and the element's `indeterminate` DOM attribute back to the values they had before the pre-click activation steps were run. The activation behavior is to fire a simple event called `change` at the element, then broadcast `formchange` events at the element's form owner.

Constraint validation: If the element is *required* and its checkedness is false, then the element is suffering from being missing.

> ***input* . `indeterminate` [= *value*]**
>
> When set, overrides the rendering of checkbox controls so that the current value is not visible.

Bookkeeping details

- The following common `input` element content attributes and DOM attributes apply to the element: `checked`, and `required` content attributes; `checked` and `value` DOM attributes.
- The `value` DOM attribute is in mode default/on.
- The `change` event applies.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `autocomplete`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `list`, `max`, `maxlength`, `min`, `multiple`, `pattern`, `placeholder`, `readonly`, `size`, `src`, `step`, and `width`.

- The following DOM attributes and methods do not apply to the element: `list`, `selectedOption`, `selectionStart`, `selectionEnd`, `valueAsDate`, and `valueAsNumber` DOM attributes; `select()`, `setSelectionRange()`, `stepDown()`, and `stepUp()` methods.
- The `input` event does not apply.

4.10.4.1.16 Radio Button state

When an `input` element's `type` attribute is in the Radio Button state, the rules in this section apply.

The `input` element represents a control that, when used in conjunction with other `input` elements, forms a *radio button group* in which only one control can have its checkedness state set to true. If the element's checkedness state is true, the control represents the selected control in the group, and if it is false, it indicates a control in the group that is not selected.

The ***radio button group*** that contains an `input` element *a* also contains all the other `input` elements *b* that fulfill all of the following conditions:

- The `input` element *b*'s `type` attribute is in the Radio Button state.
- Either neither *a* nor *b* have a form owner, or they both have one and it is the same for both.
- They both have a `name` attribute, and the value of *a*'s `name` attribute is a compatibility caseless match for the value of *b*'s `name` attribute.

A document must not contain an `input` element whose *radio button group* contains only that element.

When any of the following events occur, if the element's checkedness state is true after the event, the checkedness state of all the other elements in the same *radio button group* must be set to false:

- The element's checkedness state is set to true (for whatever reason).
- The element's `name` attribute is added, removed, or changes value.
- The element's form owner changes.

If the element is *mutable*, then: The pre-click activation steps consist of setting the element's checkedness to true. The canceled activation steps consist of setting the element's checkedness to false. The activation behavior is to fire a simple event called `change` at the element, then broadcast `formchange` events at the element's form owner.

Constraint validation: If the element is *required* and all of the `input` elements in the *radio button group* have a checkedness that is false, then the element is suffering from being missing.

> ***Note: If none of the radio buttons in a radio button group are checked when they are inserted into the document, then they will all be initially unchecked in the interface, until such time as one of them is checked (either by the user or***

by script).

Bookkeeping details

- The following common `input` element content attributes and DOM attributes apply to the element: `checked` and `required` content attributes; `checked` and `value` DOM attributes.
- The `value` DOM attribute is in mode default/on.
- The `change` event applies.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `autocomplete`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `list`, `max`, `maxlength`, `min`, `multiple`, `pattern`, `placeholder`, `readonly`, `size`, `src`, `step`, and `width`.
- The following DOM attributes and methods do not apply to the element: `list`, `selectedOption`, `selectionStart`, `selectionEnd`, `valueAsDate`, and `valueAsNumber` DOM attributes; `select()`, `setSelectionRange()`, `stepDown()`, and `stepUp()` methods.
- The `input` event does not apply.

4.10.4.1.17 File Upload state

When an `input` element's `type` attribute is in the File Upload state, the rules in this section apply.

The `input` element represents a list of **selected files**, each file consisting of a file name, a file type, and a file body (the contents of the file).

If the element is *mutable*, the user agent should allow the user to change the files on the list, e.g. adding or removing files. Files can be from the filesystem or created on the fly, e.g. a picture taken from a camera connected to the user's device.

Constraint validation: If the element is *required* and the list of selected files is empty, then the element is suffering from being missing.

Unless the `multiple` attribute is set, there must be no more than one file in the list of selected files.

The **`accept`** attribute may be specified to provide user agents with a hint of what file types the server will be able to accept.

If specified, the attribute must consist of a set of comma-separated tokens, each of which must be an ASCII case-insensitive match for one of the following:

The string `audio/*`
: Indicates that sound files are accepted.

The string `video/*`
: Indicates that video files are accepted.

The string `image/*`
: Indicates that image files are accepted.

A valid MIME type, with no parameters
: Indicates that files of the specified type are accepted. [RFC2046]

The tokens must not be ASCII case-insensitive matches for any of the other tokens (i.e. duplicates are not allowed). To obtain the list of tokens from the attribute, the user agent must

split the attribute value on commas.

User agents should prevent the user from selecting files that are not accepted by one (or more) of these tokens.

Bookkeeping details

- The following common `input` element content attributes apply to the element: `accept`, `multiple`, and `required`.
- The `value` DOM attribute is in mode filename.
- The `change` event applies.
- The following content attributes must not be specified and do not apply to the element: `alt`, `autocomplete`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `list`, `max`, `maxlength`, `min`, `pattern`, `placeholder`, `readonly`, `size`, `src`, `step`, and `width`.
- The element's `value` attribute must be omitted.
- The following DOM attributes and methods do not apply to the element: `checked`, `list`, `selectedOption`, `selectionStart`, `selectionEnd`, `valueAsDate`, and `valueAsNumber` DOM attributes; `select()`, `setSelectionRange()`, `stepDown()`, and `stepUp()` methods.
- The `input` event does not apply.

4.10.4.1.18 Submit Button state

When an `input` element's `type` attribute is in the Submit Button state, the rules in this section apply.

The `input` element represents a button that, when activated, submits the form. If the element has a `value` attribute, the button's label must be the value of that attribute; otherwise, it must be an implementation-defined string that means "Submit" or some such. The element is a button, specifically a submit button.

If the element is *mutable*, the user agent should allow the user to activate the element.

The element's activation behavior, if the element has a form owner, is to submit the form owner from the `input` element; otherwise, it is to do nothing.

The `formaction`, `formenctype`, `formmethod`, `formnovalidate`, and `formtarget` attributes are attributes for form submission.

Bookkeeping details

- The following common `input` element content attributes and DOM attributes apply to the element: `formaction`, `formenctype`, `formmethod`, `formnovalidate`, and `formtarget` content attributes; `value` DOM attribute.
- The `value` DOM attribute is in mode default.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `autocomplete`, `checked`, `height`, `list`, `max`, `maxlength`, `min`, `multiple`, `pattern`, `placeholder`, `readonly`, `required` `size`, `src`, `step`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `list`, `selectedOption`, `selectionStart`, `selectionEnd`, `valueAsDate`, and `valueAsNumber` DOM attributes; `select()`, `setSelectionRange()`, `stepDown()`, and `stepUp()` methods.
- The `input` and `change` events do not apply.

4.10.4.1.19 Image Button state

When an `input` element's `type` attribute is in the Image Button state, the rules in this section apply.

The `input` element represents either an image from which a user can select a coordinate and submit the form, or alternatively a button from which the user can submit the form. The element is a button, specifically a submit button.

The image is given by the **`src`** attribute. The `src` attribute must be present, and must contain a valid URL referencing a non-interactive, optionally animated, image resource that is neither paged nor scripted.

When any of the following events occur, unless the user agent cannot support images, or its support for images has been disabled, or the user agent only fetches elements on demand, the user agent must resolve the value of the `src` attribute, relative to the element, and if that is successful, must fetch the resulting absolute URL:

- The `input` element's `type` attribute is first set to the Image Button state (possibly when the element is first created), and the `src` attribute is present.
- The `input` element's `type` attribute is changed back to the Image Button state, and the `src` attribute is present, and its value has changed since the last time the `type` attribute was in the Image Button state.
- The `input` element's `type` attribute is in the Image Button state, and the `src` attribute is set or changed.

Fetching the image must delay the `load` event.

If the image was successfully obtained, with no network errors, and the image's type is a supported image type, and the image is a valid image of that type, then the image is said to be *available*. If this is true before the image is completely downloaded, each task that is queued by the networking task source while the image is being fetched must update the presentation of the image appropriately.

The user agents should apply the image sniffing rules to determine the type of the image, with the image's associated Content-Type headers giving the *official type*. If these rules are not applied, then the type of the image must be the type given by the image's associated Content-Type headers.

User agents must not support non-image resources with the `input` element. User agents must not run executable code embedded in the image resource. User agents must only display the first page of a multipage resource. User agents must not allow the resource to act in an interactive fashion, but should honour any animation in the resource.

The task that is queued by the networking task source once the resource has been fetched, must, if the download was successful and the image is *available*, queue a task to fire a `load` event on the `input` element; and otherwise, if the fetching process fails without a response from the remote server, or completes but the image is not a valid or supported image, queue a task to fire an `error` event on the `input` element.

The **alt** attribute provides the textual label for the alternative button for users and user agents who cannot use the image. The alt attribute must also be present, and must contain a non-empty string.

The input element supports dimension attributes.

If the src attribute is set, and the image is *available* and the user agent is configured to display that image, then: The element represents a control for selecting a coordinate from the image specified by the src attribute; if the element is *mutable*, the user agent should allow the user to select this coordinate. The activation behavior in this case consists of taking the user's selected coordinate, and then, if the element has a form owner, submitting the input element's form owner from the input element. If the user activates the control without explicitly selecting a coordinate, then the coordinate (0,0) must be assumed.

Otherwise, the element represents a submit button whose label is given by the value of the alt attribute; if the element is *mutable*, the user agent should allow the user to activate the button. The activation behavior in this case consists of setting the selected coordinate to (0,0), and then, if the element has a form owner, submitting the input element's form owner from the input element.

The **selected coordinate** must consist of an *x*-component and a *y*-component. The *x*-component must be greater than or equal to zero, and less than or equal to the rendered width, in CSS pixels, of the image, plus the widths of the left and right borders rendered around the image, if any. The *y*-component must be greater than or equal to zero, and less than or equal to the rendered height, in CSS pixels, of the image, plus the widths of the top and bottom bordered rendered around the image, if any. The coordinates must be relative to the image's borders, where there are any, and the edge of the image otherwise.

The formaction, formenctype, formmethod, formnovalidate, and formtarget attributes are attributes for form submission.

Bookkeeping details

- The following common input element content attributes and DOM attributes apply to the element: alt, formaction, formenctype, formmethod, formnovalidate, formtarget, height, src, and width content attributes; value DOM attribute.
- The value DOM attribute is in mode default.
- The following content attributes must not be specified and do not apply to the element: accept, autocomplete, checked, list, max, maxlength, min, multiple, pattern, placeholder, readonly, required size, and step.
- The element's value attribute must be omitted.
- The following DOM attributes and methods do not apply to the element: checked, list, selectedOption, selectionStart, selectionEnd, valueAsDate, and valueAsNumber DOM attributes; select(), setSelectionRange(), stepDown(), and stepUp() methods.
- The input and change events do not apply.

Note: Many aspects of this state's behavior are similar to the behavior of the img element. Readers are encouraged to read that section, where many of the same requirements are described in more detail.

4.10.4.1.20 Reset Button state

When an `input` element's `type` attribute is in the Reset Button state, the rules in this section apply.

The `input` element represents a button that, when activated, resets the form. If the element has a `value` attribute, the button's label must be the value of that attribute; otherwise, it must be an implementation-defined string that means "Reset" or some such. The element is a button.

If the element is *mutable*, the user agent should allow the user to activate the element.

The element's activation behavior, if the element has a form owner, is to reset the form owner; otherwise, it is to do nothing.

Constraint validation: The element is barred from constraint validation.

Bookkeeping details

- The `value` DOM attribute applies to this element and is in mode default.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `autocomplete`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `list`, `max`, `maxlength`, `min`, `multiple`, `pattern`, `placeholder`, `readonly`, `required` `size`, `src`, `step`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `list`, `selectedOption`, `selectionStart`, `selectionEnd`, `valueAsDate`, and `valueAsNumber` DOM attributes; `select()`, `setSelectionRange()`, `stepDown()`, and `stepUp()` methods.
- The `input` and `change` events do not apply.

4.10.4.1.21 Button state

When an `input` element's `type` attribute is in the Button state, the rules in this section apply.

The `input` element represents a button with no default behavior. If the element has a `value` attribute, the button's label must be the value of that attribute; otherwise, it must be the empty string. The element is a button.

If the element is *mutable*, the user agent should allow the user to activate the element. The element's activation behavior is to do nothing.

Constraint validation: The element is barred from constraint validation.

Bookkeeping details

- The `value` DOM attribute applies to this element and is in mode default.
- The following content attributes must not be specified and do not apply to the element: `accept`, `alt`, `autocomplete`, `checked`, `formaction`, `formenctype`, `formmethod`, `formnovalidate`, `formtarget`, `height`, `list`, `max`, `maxlength`, `min`, `multiple`, `pattern`, `placeholder`, `readonly`, `required` `size`, `src`, `step`, and `width`.
- The following DOM attributes and methods do not apply to the element: `checked`, `list`, `selectedOption`, `selectionStart`, `selectionEnd`, `valueAsDate`, and `valueAsNumber` DOM attributes; `select()`, `setSelectionRange()`, `stepDown()`, and `stepUp()` methods.
- The `input` and `change` events do not apply.

4.10.4.2 Common `input` element attributes

These attributes only apply to an `input` element if its `type` attribute is in a state whose definition declares that the attribute applies. When an attribute doesn't apply to an `input` element, user agents must ignore the attribute.

4.10.4.2.1 THE `autocomplete` ATTRIBUTE

The **`autocomplete`** attribute is an enumerated attribute. The attribute has three states. The `on` keyword maps to the **on** state, and the `off` keyword maps to the **off** state. The attribute may also be omitted. The *missing value default* is the **default** state.

The off state indicates that the control's input data is either particularly sensitive (for example the activation code for a nuclear weapon) or is a value that will never be reused (for example a one-time-key for a bank login) and the user will therefore have to explicitly enter the data each time, instead of being able to rely on the UA to prefill the value for him.

Conversely, the on state indicates that the value is not particularly sensitive and the user can expect to be able to rely on his user agent to remember values he has entered for that control.

The default state indicates that the user agent is to use the `autocomplete` attribute on the element's form owner instead.

Each `input` element has a **resulting autocompletion state**, which is either *on* or *off*.

When an `input` element's `autocomplete` attribute is in the on state, when an `input` element's `autocomplete` attribute is in the default state, and the element has no form owner, and when an `input` element's `autocomplete` attribute is in the default state, and the element's form owner's `autocomplete` attribute is in the on state, the `input` element's resulting autocompletion state is *on*. Otherwise, the `input` element's resulting autocompletion state is *off*.

When an `input` element's resulting autocompletion state is *on*, the user agent may store the value entered by the user so that if the user returns to the page, the UA can prefill the form. Otherwise, the user agent should not remember the control's value.

The autocompletion mechanism must be implemented by the user agent acting as if the user had modified the element's value, and must be done at a time where the element is *mutable* (e.g. just after the element has been inserted into the document, or when the user agent stops parsing).

> Banks frequently do not want UAs to prefill login information:
>
> ```
> Account: <input type="text" name="ac" autocomplete="off">
> PIN: <input type="text" name="pin" autocomplete="off">
> ```

A user agent may allow the user to override the resulting autocompletion state and set it to always *on*, always allowing values to be remembered and prefilled), or always *off*, never remembering values. However, the ability to override the resulting autocompletion state to *on* should not be trivially accessible, as there are significant security implications for the user if all values are always remembered, regardless of the site's preferences.

4.10.4.2.2 The `list` attribute

The **`list`** attribute is used to identify an element that lists predefined options suggested to the user.

If present, its value must be the ID of a `datalist` element in the same document.

The **suggestions source element** is the first element in the document in tree order to have an ID equal to the value of the `list` attribute, if that element is a `datalist` element. If there is no `list` attribute, or if there is no element with that ID, or if the first element with that ID is not a `datalist` element, then there is no suggestions source element.

If there is a suggestions source element, then each `option` element that is a descendant of the suggestions source element, that is not disabled, and whose value is a string that isn't the empty string and that the user would be allowed to enter as the `input` element's value, represents a suggestion. Each suggestion has a value and a label.

When the user agent is allowing the user to edit the `input` element's value, the user agent should offer the suggestions to the user in a manner suitable for the type of control used. The user agent may use the suggestion's label to identify the suggestion if appropriate. If the user selects a suggestion, then the `input` element's value must be set to the selected suggestion's value, as if the user had written that value himself.

User agents should filter the suggestions to hide suggestions that would cause the element to not satisfy its constraints.

If the `list` attribute does not apply, there is no suggestions source element.

4.10.4.2.3 The `readonly` attribute

The **`readonly`** attribute is a boolean attribute that controls whether or not the use can edit the form control. When specified, the element is *immutable*.

Constraint validation: If the `readonly` attribute is specified on an `input` element, the element is barred from constraint validation.

4.10.4.2.4 The `size` attribute

The **`size`** attribute gives the number of characters that, in a visual rendering, the user agent is to allow the user to see while editing the element's value.

The `size` attribute, if specified, must have a value that is a valid non-negative integer greater than zero.

If the attribute is present, then its value must be parsed using the rules for parsing non-negative integers, and if the result is a number greater than zero, then the user agent should ensure that at least that many characters are visible.

The `size` DOM attribute limited to only positive non-zero numbers.

4.10.4.2.5 The `required` attribute

The **`required`** attribute is a boolean attribute. When specified, the element is ***required***.

Constraint validation: If the element is *required*, and its `value` DOM attribute applies and is in the mode value, and the element is *mutable*, and the element's value is the empty string, then the element is suffering from being missing.

4.10.4.2.6 The `multiple` attribute

The **`multiple`** attribute is a boolean attribute that indicates whether the user is to be allowed to specify more than one value.

4.10.4.2.7 The `maxlength` attribute

The **`maxlength`** attribute, when it applies, is a form control `maxlength` attribute controlled by the `input` element's dirty value flag.

If the `input` element has a maximum allowed value length, then the codepoint length of the value of the element's `value` attribute must be equal to or less than the element's maximum allowed value length.

4.10.4.2.8 The `pattern` attribute

The **`pattern`** attribute specifies a regular expression against which the control's value is to be checked.

If specified, the attribute's value must match the *Pattern* production of ECMA 262's grammar. [ECMA262]

Constraint validation: If the element's value is not the empty string, and the element's `pattern` attribute is specified and the attribute's value, when compiled as an ECMA 262 regular expression with the `global`, `ignoreCase`, and `multiline` flags *disabled* (see ECMA 262, sections 15.10.7.2 through 15.10.7.4), compiles successfully but the resulting regular expression does not match the entirety of the element's value, then the element is suffering from a pattern mismatch. [ECMA262]

> ***Note: This implies that the regular expression language used for this attribute is the same as that defined in ECMA 262, except that the `pattern` attribute must match the entire value, not just any subset (somewhat as if it implied a `^(?:` at the start of the pattern and a `)$` at the end).***

When an `input` element has a `pattern` attribute specified, authors should include a `title` attribute to give a description of the pattern. User agents may use the contents of this attribute, if it is present, when informing the user that the pattern is not matched, or at any other suitable time, such as in a tooltip or read out by assistive technology when the control

gains focus.

> For example, the following snippet:
>
> ```
> <label> Part number:
>  <input pattern="[0-9][A-Z]{3}" name="part"
>         title="A part number is a digit followed by three uppercase letters.",
> </label>
> ```
>
> ...could cause the UA to display an alert such as:
>
> ```
>  part number is a digit followed by three uppercase letters.
> You cannot complete this form until the field is correct.
> ```

When a control has a `pattern` attribute, the `title` attribute, if used, must describe the pattern. Additional information could also be included, so long as it assists the user in filling in the control. Otherwise, assistive technology would be impaired.

> For instance, if the title attribute contained the caption of the control, assistive technology could end up saying something like `The text you have entered does not match the required pattern. Birthday`, which is not useful.

UAs may still show the `title` in non-error situations (for example, as a tooltip when hovering over the control), so authors should be careful not to word `title`s as if an error has necessarily occurred.

4.10.4.2.9 The `min` and `max` attributes

The **`min`** and **`max`** attributes indicate the allowed range of values for the element.

Their syntax is defined by the section that defines the `type` attribute's current state.

If the element has a `min` attribute, and the result of applying the algorithm to convert a string to a number to the value of the `min` attribute is a a number, then that number is the element's **minimum**; otherwise, if the the `type` attribute's current state defines a **default minimum**, then that is the minimum; otherwise, the element has no minimum.)

Constraint validation: When the element has a minimum, and the result of applying the algorithm to convert a string to a number to the string given by the element's value is a number, and the number obtained from that algorithm is less than the minimum, the element is suffering from an underflow.

The `min` attribute also defines the step base.

If the element has a `max` attribute, and the result of applying the algorithm to convert a string to a number to the value of the `max` attribute is a a number, then that number is the element's **maximum**; otherwise, if the the `type` attribute's current state defines a **default maximum**, then that is the maximum; otherwise, the element has no maximum.)

Constraint validation: When the element has a maximum, and the result of applying the algorithm to convert a string to a number to the string given by the element's value is a number, and the number obtained from that algorithm is more than the maximum, the element is suffering from an overflow.

The max attribute's value (the maximum) must not be less than the min attribute's value (its minimum).

Note: If an element has a maximum that is less than its minimum, then so long as the element has a value, it will either be suffering from an underflow or suffering from an overflow.

4.10.4.2.10 The step attribute

The **step** attribute indicates the granularity that is expected (and required) of the value, by limiting the allowed values. The section that defines the type attribute's current state also defines the **default step** and the **step scale factor**, which are used in processing the attribute as described below.

The step attribute, if specified, must either have a value that is a valid floating point number that parses to a number that is greater than zero, or must have a value that is an ASCII case-insensitive match for the string "any".

The attribute provides the **allowed value step** for the element, as follows:

1. If the attribute is absent, then the allowed value step is the default step multiplied by the step scale factor.
2. Otherwise, if the attribute's value is an ASCII case-insensitive match for the string "any", then there is no allowed value step.
3. Otherwise, if the rules for parsing floating point number values, when they are applied to the attribute's value, return an error, zero, or a number less than zero, then the allowed value step is the default step multiplied by the step scale factor.
4. Otherwise, the allowed value step is the number returned by the rules for parsing floating point number values when they are applied to the attribute's value, multiplied by the step scale factor.

The **step base** is the result of applying the algorithm to convert a string to a number to the value of the min attribute, unless the element does not have a min attribute specified or the result of applying that algorithm is an error, in which case the step base is zero.

Constraint validation: When the element has an allowed value step, and the result of applying the algorithm to convert a string to a number to the string given by the element's value is a number, and that number subtracted from the step base is not an integral multiple of the allowed value step, the element is suffering from a step mismatch.

4.10.4.2.11 The placeholder attribute

The **placeholder** attribute represents a *short* hint (a word or short phrase) intended to aid the user with data entry. A hint could be a sample value or a brief description of the expected format. The attribute, if specified, must have a value that contains no U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters.

Note: For a longer hint or other advisory text, the `title` attribute is more appropriate.

The `placeholder` attribute should not be used as an alternative to a `label`.

User agents should present this hint to the user, after having stripped line breaks from it, when the element's value is the empty string and the control is not focused (e.g. by displaying it inside a blank unfocused control).

> Here is an example of a mail configuration user interface that uses the `placeholder` attribute:
>
> ```
> <fieldset>
>  <legend>Mail Account</legend>
>  <label>Name: <input type="text" name="fullname" placeholder="John Ratzenb
>  <label>Address: <input type="email" name="address" placeholder="john@exam
>  <label>Password: <input type="password" name="password"></label>
>  <label>Description: <input type="text" name="desc" placeholder="My Email
> </fieldset>
> ```

4.10.4.3 Common `input` element APIs

input **. `value` [= *value*]**

Returns the current value of the form control.

Can be set, to change the value.

Throws an `INVALID_ACCESS_ERR` exception if it is set when the control is a file upload control.

input **. `checked` [= *value*]**

Returns the current checkedness of the form control.

Can be set, to change the checkedness.

input **. `valueAsDate` [= *value*]**

Returns a `Date` object representing the form control's value, if applicable; otherwise, returns null.

Can be set, to change the value.

Throws an `INVALID_ACCESS_ERR` exception if the control isn't date- or time-based.

input **. `valueAsNumber` [= *value*]**

Returns a number representing the form control's value, if applicable; otherwise, returns null.

Can be set, to change the value.

Throws an `INVALID_ACCESS_ERR` exception if the control is neither date- or time-based nor numeric.

input **. stepUp()**
input **. stepDown()**

Changes the the form control's value by the value given in the `step` attribute.

Throws `INVALID_ACCESS_ERR` exception if the control is neither date- or time-based nor numeric, if the `step` attribute's value is "`any`", if the current value could not be parsed, or if stepping in the given direction would take the value out of range.

input **. list**

Returns the `datalist` element indicated by the `list` attribute.

input **. selectedOption**

Returns the `option` element from the `datalist` element indicated by the `list` attribute that matches the form control's value.

The **`value`** DOM attribute allows scripts to manipulate the value of an `input` element. The attribute is in one of the following modes, which define its behavior:

value
: On getting, it must return the current value of the element. On setting, it must set the element's value to the new value, set the element's dirty value flag to true, and then invoke the value sanitization algorithm, if the element's `type` attribute's current state defines one.

default
: On getting, if the element has a `value` attribute, it must return that attribute's value; otherwise, it must return the empty string. On setting, it must set the element's `value` attribute to the new value.

default/on
: On getting, if the element has a `value` attribute, it must return that attribute's value; otherwise, it must return the string "`on`". On setting, it must set the element's `value` attribute to the new value.

filename
: On getting, it must return the string "`C:\fakepath\`" followed by the filename of the first file in the list of selected files, if any, or the empty string if the list is empty. On setting, it must throw an `INVALID_ACCESS_ERR` exception.

The **`checked`** DOM attribute allows scripts to manipulate the checkedness of an `input` element. On getting, it must return the current checkedness of the element; and on setting, it must set the element's checkedness to the new value and set the element's dirty checkedness flag to true.

The **`valueAsDate`** DOM attribute represents the value of the element, interpreted as a date.

On getting, if the `valueAsDate` attribute does not apply, as defined for the `input` element's `type` attribute's current state, then return null. Otherwise, run the algorithm to convert a string to a `Date` object defined for that state; if the algorithm returned a `Date` object, then return it, otherwise, return null.

On setting, if the `valueAsDate` attribute does not apply, as defined for the `input` element's `type` attribute's current state, then throw an `INVALID_ACCESS_ERR` exception; otherwise, if the new value is null, then set the value of the element to the empty string; otherwise, run the algorithm to convert a `Date` object to a string, as defined for that state, on the new value, and set the value of the element to resulting string.

The **`valueAsNumber`** DOM attribute represents the value of the element, interpreted as a number.

On getting, if the `valueAsNumber` attribute does not apply, as defined for the `input` element's `type` attribute's current state, then return a Not-a-Number (NaN) value. Otherwise, if the `valueAs`*`Date`* attribute applies, run the algorithm to convert a string to a `Date` object defined for that state; if the algorithm returned a `Date` object, then return the *time value* of the object (the number of milliseconds from midnight UTC the morning of 1970-01-01 to the time represented by the `Date` object), otherwise, return a Not-a-Number (NaN) value. Otherwise, run the algorithm to convert a string to a number defined for that state; if the algorithm returned a number, then return it, otherwise, return a Not-a-Number (NaN) value.

On setting, if the `valueAsNumber` attribute does not apply, as defined for the `input` element's `type` attribute's current state, then throw an `INVALID_ACCESS_ERR` exception. Otherwise, if the `valueAs`*`Date`* attribute applies, run the algorithm to convert a `Date` object to a string defined for that state, passing it a `Date` object whose *time value* is the new value, and set the value of the element to resulting string. Otherwise, run the algorithm to convert a number to a string, as defined for that state, on the new value, and set the value of the element to resulting string.

The **`stepDown()`** and **`stepUp()`** methods, when invoked, must run the following algorithm:

1. If the `stepDown()` and `stepUp()` methods do not apply, as defined for the `input` element's `type` attribute's current state, then throw an `INVALID_ACCESS_ERR` exception, and abort these steps.

2. If the element has no allowed value step, then throw an `INVALID_ACCESS_ERR` exception, and abort these steps.

3. If applying the algorithm to convert a string to a number to the string given by the element's value results in an error, then throw an `INVALID_ACCESS_ERR` exception, and abort these steps; otherwise, let *value* be the result of that algorithm.

4. Let *delta* be the allowed value step.

5. If the method invoked was the `stepDown()` method, negate *delta*.

6. Let *value* be the result of adding *delta* to *value*.

7. If the element has a minimum, and the *value* is less than that minimum, then throw a `INVALID_ACCESS_ERR` exception.

8. If the element has a maximum, and the *value* is greater than that maximum, then throw a `INVALID_ACCESS_ERR` exception.

9. Let *value as string* be the result of running the algorithm to convert a number to a string, as defined for the `input` element's `type` attribute's current state, on *value*.

10. Set the value of the element to *value as string*.

The **`list`** DOM attribute must return the current suggestions source element, if any, or null otherwise.

The **`selectedOption`** DOM attribute must return the first `option` element, in tree order, to be a child of the suggestions source element and whose value matches the `input` element's value, if any. If there is no suggestions source element, or if it contains no matching `option` element, then the `selectedOption` attribute must return null.

4.10.4.4 Common event behaviors

When the **`input`** event applies, any time the user causes the element's value to change, the user agent must queue a task to fire a simple event called `input` at the `input` element, then broadcast `forminput` events at the `input` element's form owner. User agents may wait for a suitable break in the user's interaction before queuing the task; for example, a user agent could wait for the user to have not hit a key for 100ms, so as to only fire the event when the user pauses, instead of continuously for each keystroke.

> Examples of a user changing the element's value would include the user typing into a text field, pasting a new value into the field, or undoing an edit in that field. Some user interactions do not cause changes to the value, e.g. hitting the "delete" key in an empty text field, or replacing some text in the field with text from the clipboard that happens to be exactly the same text.

When the **`change`** event applies, if the element does not have an activation behavior defined but uses a user interface that involves an explicit commit action, then any time the user commits a change to the element's value or list of selected files, the user agent must queue a task to fire a simple event called `change` at the `input` element, then broadcast `formchange` events at the `input` element's form owner.

> An example of a user interface with a commit action would be a File Upload control that consists of a single button that brings up a file selection dialog: when the dialog is closed, if that the file selection changed as a result, then the user has commited a new file selection.

> Another example of a user interface with a commit action would be a Date control that allows both text-based user input and user selection from a drop-down calendar: while text input might not have an explicit commit step, selecting a date from the drop down calendar and then dismissing the drop down would be a commit action.

When the user agent changes the element's value on behalf of the user (e.g. as part of a form prefilling feature), the user agent must follow these steps:

1. If the `input` event applies, queue a task to fire a simple event called `input` at the `input` element.
2. If the `input` event applies, broadcast `forminput` events at the `input` element's form owner.
3. If the `change` event applies, queue a task to fire a simple event called `change` at the `input` element.

4. If the `change` event applies, broadcast `formchange` events at the `input` element's form owner.

Note: In addition, when the `change` event applies, `change` events can also be fired as part of the element's activation behavior and as part of the unfocusing steps.

The task source for these task is the user interaction task source.

4.10.5 The `button` element

Categories

Flow content.
Phrasing content.
Interactive content.
Listed, labelable, and submittable form-associated element.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content, but there must be no interactive content descendant.

Content attributes:

Global attributes
`autofocus`
`disabled`
`form`
`formaction`
`formenctype`
`formmethod`
`formnovalidate`
`formtarget`
`name`
`type`
`value`

DOM interface:

```
interface HTMLButtonElement : HTMLElement {
           attribute boolean autofocus;
           attribute boolean disabled;
  readonly attribute HTMLFormElement form;
           attribute DOMString formaction;
           attribute DOMString formenctype;
           attribute DOMString formmethod;
           attribute DOMString formnoValidate;
           attribute DOMString formtarget;
           attribute DOMString name;
           attribute DOMString type;
           attribute DOMString value;

  readonly attribute boolean willValidate;
  readonly attribute ValidityState validity;
  readonly attribute DOMString validationMessage;
  boolean checkValidity();
  void setCustomValidity(in DOMString error);
```

```
  readonly attribute NodeList labels;
};
```

The `button` element represents a button. If the element is not disabled, then the user agent should allow the user to activate the button.

The element is a button.

The **`type`** attribute controls the behavior of the button when it is activated. It is an enumerated attribute. The following table lists the keywords and states for the attribute — the keywords in the left column map to the states in the cell in the second column on the same row as the keyword.

Keyword	State	Brief description
`submit`	Submit Button	Submits the form.
`reset`	Reset Button	Resets the form.
`button`	Button	Does nothing.

The *missing value default* is the Submit Button state.

If the `type` attribute is in the Submit Button state, the element is specifically a submit button.

If the element is not disabled, the activation behavior of the `button` element is to run the steps defined in the following list for the current state of the element's `type` attribute.

Submit Button
: If the element has a form owner, the element must submit the form owner from the `button` element.

Reset Button
: If the element has a form owner, the element must reset the form owner.

Button
: Do nothing.

The `form` attribute is used to explicitly associate the `button` element with its form owner. The `name` attribute represents the element's name. The `disabled` attribute is used to make the control non-interactive and to prevent its value from being submitted. The `autofocus` attribute controls focus. The `formaction`, `formenctype`, `formmethod`, `formnovalidate`, and `formtarget` attributes are attributes for form submission.

The **`value`** attribute gives the element's value for the purposes of form submission. The `value` attribute must not be present unless the `form` attribute is present. The element's value is the value of the element's `value` attribute, if there is one, or the empty string otherwise.

> ***Note: A button (and its value) is only included in the form submission if the button itself was used to initiate the form submission.***

The **`value`** and **`type`** DOM attributes must reflect the respective content attributes of the same name.

The `willValidate`, `validity`, and `validationMessage` attributes, and the `checkValidity()` and `setCustomValidity()` methods, are part of the constraint validation API. The `labels` attribute provides a list of the element's `label`s.

4.10.6 The `select` element

Categories

Flow content.
Phrasing content.
Interactive content.
Listed, labelable, submittable, and resettable form-associated element.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Zero or more `option` or `optgroup` elements.

Content attributes:

Global attributes
`autofocus`
`disabled`
`form`
`multiple`
`name`
`size`

DOM interface:

```
[Callable=namedItem]
interface HTMLSelectElement : HTMLElement {
           attribute boolean autofocus;
           attribute boolean disabled;
  readonly attribute HTMLFormElement form;
           attribute boolean multiple;
           attribute DOMString name;
           attribute boolean size;

  readonly attribute DOMString type;

  readonly attribute HTMLOptionsCollection options;
           attribute unsigned long length;
  [IndexGetter] any item(in DOMString name);
  [NameGetter] any namedItem(in DOMString name);
  void add(in HTMLElement element, [Optional] in HTMLElement before);
  void add(in HTMLElement element, in long before);
  void remove(in long index);

  readonly attribute HTMLCollection selectedOptions;
           attribute long selectedIndex;
           attribute DOMString value;

  readonly attribute boolean willValidate;
  readonly attribute ValidityState validity;
  readonly attribute DOMString validationMessage;
  boolean checkValidity();
  void setCustomValidity(in DOMString error);

  readonly attribute NodeList labels;
};
```

The select element represents a control for selecting amongst a set of options.

The **multiple** attribute is a boolean attribute. If the attribute is present, then the select element represents a control for selecting zero or more options from the list of options. If the attribute is absent, then the select element represents a control for selecting a single option from the list of options.

The **list of options** for a select element consists of all the option element children of the select element, and all the option element children of all the optgroup element children of the select element, in tree order.

The **size** attribute gives the number of options to show to the user. The size attribute, if specified, must have a value that is a valid non-negative integer greater than zero. If the multiple attribute is present, then the size attribute's default value is 4. If the multiple attribute is absent, then the size attribute's default value is 1.

If the multiple attribute is absent, and the element is not disabled, then the user agent should allow the user to pick an option element in its list of options that is itself not disabled. Upon this option element being **picked** (either through a click, or through unfocusing the element after changing its value, or through a menu command, or through any other mechanism), and before the relevant user interaction event is queued (e.g. before the click event), the user agent must set the selectedness of the picked option element to true and then queue a task to fire a simple event called change at the select element, using the user interaction task source as the task source, then broadcast formchange events at the element's form owner.

If the multiple attribute is absent, whenever an option element in the select element's list of options has its selectedness set to true, and whenever an option element with its selectedness set to true is added to the select element's list of options, the user agent must set the selectedness of all the other option element in its list of options to false.

If the multiple attribute is absent, whenever there are no option elements in the select element's list of options that have their selectedness set to true, the user agent must set the selectedness of the first option element in the list of options in tree order that is not disabled, if any, to true.

If the multiple attribute is present, and the element is not disabled, then the user agent should allow the user to **toggle** the selectedness of the option elements in its list of options that are themselves not disabled (either through a click, or through a menu command, or any other mechanism). Upon the selectedness of one or more option elements being changed by the user, and before the relevant user interaction event is queued (e.g. before a related click event), the user agent must queue a task to fire a simple event called change at the select element, using the user interaction task source as the task source, then broadcast formchange events at the element's form owner.

The reset algorithm for select elements is to go through all the option elements in the element's list of options, and set their selectedness to true if the option element has a selected attribute, and false otherwise.

The form attribute is used to explicitly associate the select element with its form owner. The name attribute represents the element's name. The disabled attribute is used to make the control non-interactive and to prevent its value from being submitted. The autofocus attribute controls focus.

select **.** **`type`**

Returns "`select-multiple`" if the element has a `multiple` attribute, and "`select-one`" otherwise.

select **.** **`options`**

Returns an `HTMLOptionsCollection` of the list of options.

select **.** **`length`** **[= *value*]**

Returns the number of elements in the list of options.

When set to a smaller number, truncates the number of `option` elements in the `select`.

When set to a greater number, adds new blank `option` elements to the `select`.

element **= *select*** **.** **`item`**(***index***)
select**[*index*]**

Returns the item with index *index* from the list of options. The items are sorted in tree order.

Returns null if *index* is out of range.

element **= *select*** **.** **`namedItem`**(***name***)
select**[*name*]**

Returns the item with ID or `name` *name* from the list of options.

If there are multiple matching items, then a `NodeList` object containing all those elements is returned.

Returns null if no element with that ID could be found.

select **.** **`add`**(***element*** **[,** ***before*** **])**

Inserts *element* before the node given by *before*.

The *before* argument can be a number, in which case *element* is inserted before the item with that number, or an element from the list of options, in which case *element* is inserted before that element.

If *before* is omitted, null, or a number out of range, then *element* will be added at the end of the list.

This method will throw a `HIERARCHY_REQUEST_ERR` exception if *element* is an ancestor of the element into which it is to be inserted. If *element* is not an `option` or `optgroup` element, then the method does nothing.

select **.** **`selectedOptions`**

Returns an `HTMLCollection` of the list of options that are selected.

select **.** **`selectedIndex`** **[= *value*]**

Returns the index of the first selected item, if any, or −1 if there is no selected item.

Can be set, to change the selection.

select **.** `value` **[=** ***value*** **]**

Returns the value of the first selected item, if any, or the empty string if there is no selected item.

Can be set, to change the selection.

The **`type`** DOM attribute, on getting, must return the string "`select-one`" if the `multiple` attribute is absent, and the string "`select-multiple`" if the `multiple` attribute is present.

The **`options`** DOM attribute must return an `HTMLOptionsCollection` rooted at the `select` node, whose filter matches the elements in the list of options.

The `options` collection is also mirrored on the `HTMLSelectElement` object. The indices of the supported indexed properties at any instant are the indicies supported by the object returned by the `options` attribute at that instant. The names of the supported named properties at any instant are the names supported by the object returned by the `options` attribute at that instant.

The **`length`** DOM attribute must return the number of nodes represented by the `options` collection. On setting, it must act like the attribute of the same name on the `options` collection.

The **`item(`*`index`*`)`** method must return the value returned by the method of the same name on the `options` collection, when invoked with the same argument.

The **`namedItem(`*`name`*`)`** method must return the value returned by the method of the same name on the `options` collection, when invoked with the same argument.

Similarly, the **`add()`** and **`remove()`** methods must act like their namesake methods on that same `options` collection.

The **`selectedOptions`** DOM attribute must return an `HTMLCollection` rooted at the `select` node, whose filter matches the elements in the list of options that have their selectedness set to true.

The **`selectedIndex`** DOM attribute, on getting, must return the index of the first `option` element in the list of options in tree order that has its selectedness set to true, if any. If there isn't one, then it must return −1.

On setting, the `selectedIndex` attribute must set the selectedness of all the `option` elements in the list of options to false, and then the `option` element in the list of options whose index is the given new value, if any, must have its selectedness set to true.

The **`value`** DOM attribute, on getting, must return the value of the first `option` element in the list of options in tree order that has its selectedness set to true, if any. If there isn't one, then it must return the empty string.

On setting, the `value` attribute must set the selectedness of all the `option` elements in the list of options to false, and then first the `option` element in the list of options, in tree order, whose value is equal to the given new value, if any, must have its selectedness set to true.

The **multiple** and **size** DOM attributes must reflect the respective content attributes of the same name. The size DOM attribute limited to only positive non-zero numbers.

The willValidate, validity, and validationMessage attributes, and the checkValidity() and setCustomValidity() methods, are part of the constraint validation API. The labels attribute provides a list of the element's labels.

4.10.7 The datalist element

Categories
Flow content.
Phrasing content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Either: phrasing content.
Or: Zero or more option elements.
Content attributes:
Global attributes
DOM interface:

```
interface HTMLDataListElement : HTMLElement {
  readonly attribute HTMLCollection options;
};
```

The datalist element represents a set of option elements that represent predefined options for other controls. The contents of the element represents fallback content for legacy user agents, intermixed with option elements that represent the predefined options. In the rendering, the datalist element represents nothing and it, along with its children, should be hidden.

The datalist element is hooked up to an input element using the list attribute on the input element.

datalist **. options**

Returns an HTMLCollection of the options elements of the table.

The **options** DOM attribute must return an HTMLCollection rooted at the datalist node, whose filter matches option elements.

Constraint validation: If an element has a datalist element ancestor, it is barred from constraint validation.

4.10.8 The optgroup element

Categories
None.
Contexts in which this element may be used:
As a child of a `select` element.
Content model:
Zero or more `option` elements.
Content attributes:
Global attributes
`disabled`
`label`
DOM interface:

```
interface HTMLOptGroupElement : HTMLElement {
           attribute boolean disabled;
           attribute DOMString label;
};
```

The `optgroup` element represents a group of `option` elements with a common label.

The element's group of `option` elements consists of the `option` elements that are children of the `optgroup` element.

When showing `option` elements in `select` elements, user agents should show the `option` elements of such groups as being related to each other and separate from other `option` elements.

The **`disabled`** attribute is a boolean attribute and can be used to disable a group of `option` elements together.

The **`label`** attribute must be specified. Its value gives the name of the group, for the purposes of the user interface. User agents should use this attribute's value when labelling the group of `option` elements in a `select` element.

The **`disabled`** and **`label`** attributes must reflect the respective content attributes of the same name.

4.10.9 The `option` element

Categories
None.
Contexts in which this element may be used:
As a child of a `select` element.
As a child of a `datalist` element.
As a child of an `optgroup` element.
Content model:
Text.
Content attributes:
Global attributes
`disabled`
`label`
`selected`

value

DOM interface:

```
[NamedConstructor=Option(),
 NamedConstructor=Option(in DOMString text),
 NamedConstructor=Option(in DOMString text, in DOMString value),
 NamedConstructor=Option(in DOMString text, in DOMString value, in boolean
 NamedConstructor=Option(in DOMString text, in DOMString value, in boolean
interface HTMLOptionElement : HTMLElement {
           attribute boolean disabled;
  readonly attribute HTMLFormElement form;
           attribute DOMString label;
           attribute boolean defaultSelected;
           attribute boolean selected;
           attribute DOMString value;

  readonly attribute DOMString text;
  readonly attribute long index;
};
```

The `option` element represents an option in a `select` element or as part of a list of suggestions in a `datalist` element.

The **`disabled`** attribute is a boolean attribute. An `option` element is **disabled** if its `disabled` attribute is present or if it is a child of an `optgroup` element whose `disabled` attribute is present.

An `option` element that is disabled must prevent any `click` events that are queued on the user interaction task source from being dispatched on the element.

The **`label`** attribute provides a label for element. The **label** of an `option` element is the value of the `label` attribute, if there is one, or the `textContent` of the element, if there isn't.

The **`value`** attribute provides a value for element. The **value** of an `option` element is the value of the `value` attribute, if there is one, or the `textContent` of the element, if there isn't.

The **`selected`** attribute represents the default selectedness of the element.

The **selectedness** of an `option` element is a boolean state, initially false. If the element is disabled, then the element's selectedness is always false and cannot be set to true. Unless otherwise specified, when the element is created, its selectedness must be set to true if the element has a `selected` attribute. Whenever an `option` element's `selected` attribute is added, its selectedness must be set to true.

> ***Note: The `Option()` constructor with two or more arguments overrides the initial state of the selectedness state to always be false even if the third argument is true (implying that a `selected` attribute is to be set).***

An `option` element's **index** is the number of `option` element that are in the same list of options but that come before it in tree order. If the `option` element is not in a list of options, then the `option` element's index is zero.

***option* . `selected`**

Returns true if the element is selected, and false otherwise.

***option* . `index`**

Returns the index of the element in its `select` element's `options` list.

***option* . `form`**

Returns the element's `form` element, if any, or null otherwise.

***option* = new `Option`([*text* [, *value* [, *defaultSelected* [, *selected*]]]])**

Returns a new `option` element.

The *text* argument sets the contents of the element.

The *value* argument sets the `value` attribute.

The *defaultSelected* argument sets the `selected` attribute.

The *selected* argument sets whether or not the element is selected. If it is omitted, even if the *defaultSelected* argument is true, the element is not selected.

The **`disabled`**, **`label`**, and **`value`** DOM attributes must reflect the respective content attributes of the same name. The **`defaultSelected`** DOM attribute must reflect the `selected` content attribute.

The **`selected`** DOM attribute must return true if the element's selectedness is true, and false otherwise.

The **`index`** DOM attribute must return the element's index.

The **`text`** DOM attribute must return the same value as the `textContent` DOM attribute on the element.

The **`form`** DOM attribute's behavior depends on whether the `option` element is in a `select` element or not. If the `option` has a `select` element as its parent, or has a `colgroup` element as its parent and that `colgroup` element has a `select` element as its parent, then the `form` DOM attribute must return the same value as the `form` DOM attribute on that `select` element. Otherwise, it must return null.

Several constructors are provided for creating `HTMLOptionElement` objects (in addition to the factory methods from DOM Core such as `createElement()`): **`Option()`**, **`Option(`*`text`*`)`**, **`Option(`*`text`*`, `*`value`*`)`**, **`Option(`*`text`*`, `*`value`*`, `*`defaultSelected`*`)`**, and **`Option(`*`text`*`, `*`value`*`, `*`defaultSelected`*`, `*`selected`*`)`**. When invoked as constructors, these must return a new `HTMLOptionElement` object (a new `option` element). If the *text* argument is present, the new object must have as its only child a `Node` with node type `TEXT_NODE` (3) whose data is the value of that argument. If the *value* argument is present, the new object must have a `value` attribute set with the value of the argument as its value. If the *defaultSelected* argument is present and true, the new object must have a `selected` attribute set with no value. If the *selected*

argument is present and true, the new object must have its selectedness set to true; otherwise the fouth argument is absent or false, and the selectedness must be set to false, even if the *defaultSelected* argument is present and true.

4.10.10 The `textarea` element

Categories

Flow content.
Phrasing content.
Interactive content.
Listed, labelable, submittable, and resettable form-associated element.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Text.

Content attributes:

Global attributes
`autofocus`
`cols`
`disabled`
`form`
`maxlength`
`name`
`readonly`
`required`
`rows`
`wrap`

DOM interface:

```
interface HTMLTextAreaElement : HTMLElement {
           attribute boolean autofocus;
           attribute unsigned long cols;
           attribute boolean disabled;
  readonly attribute HTMLFormElement form;
           attribute long maxLength;
           attribute DOMString name;
           attribute boolean readOnly;
           attribute boolean required;
           attribute unsigned long rows;
           attribute DOMString wrap;

  readonly attribute DOMString type;
           attribute DOMString defaultValue;
           attribute DOMString value;

  readonly attribute boolean willValidate;
  readonly attribute ValidityState validity;
  readonly attribute DOMString validationMessage;
  boolean checkValidity();
  void setCustomValidity(in DOMString error);

  readonly attribute NodeList labels;

  void select();
           attribute unsigned long selectionStart;
           attribute unsigned long selectionEnd;
  void setSelectionRange(in unsigned long start, in unsigned long end);
```

```
};
```

The `textarea` element represents a multiline plain text edit control for the element's **raw value**. The contents of the control represent the control's default value.

The raw value of a `textarea` control must be initially the empty string.

The **`readonly`** attribute is a boolean attribute used to control whether the text can be edited by the user or not.

Constraint validation: If the `readonly` attribute is specified on a `textarea` element, the element is barred from constraint validation.

A `textarea` element is **mutable** if it is neither disabled nor has a `readonly` attribute specified.

When a `textarea` is mutable, its raw value should be editable by the user. Any time the user causes the element's raw value to change, the user agent must queue a task to fire a simple event called `input` at the `textarea` element, then broadcast `forminput` events at the `textarea` element's form owner. User agents may wait for a suitable break in the user's interaction before queuing the task; for example, a user agent could wait for the user to have not hit a key for 100ms, so as to only fire the event when the user pauses, instead of continuously for each keystroke.

A `textarea` element has a **dirty value flag**, which must be initially set to false, and must be set to true whenever the user interacts with the control in a way that changes the raw value.

When the `textarea` element's `textContent` DOM attribute changes value, if the element's dirty value flag is false, then the element's raw value must be set to the value of the element's `textContent` DOM attribute.

The reset algorithm for `textarea` elements is to set the element's value to the value of the element's `textContent` DOM attribute.

The **`cols`** attribute specifies the expected maximum number of characters per line. If the `cols` attribute is specified, its value must be a valid non-negative integer greater than zero. If applying the rules for parsing non-negative integers to the attribute's value results in a number greater than zero, then the element's **character width** is that value; otherwise, it is 20.

The user agent may use the `textarea` element's character width as a hint to the user as to how many characters the server prefers per line (e.g. for visual user agents by making the width of the control be that many characters). In visual renderings, the user agent should wrap the user's input in the rendering so that each line is no wider than this number of characters.

The **`rows`** attribute specifies the number of lines to show. If the `rows` attribute is specified, its value must be a valid non-negative integer greater than zero. If applying the rules for parsing non-negative integers to the attribute's value results in a number greater than zero, then the element's **character height** is that value; otherwise, it is 2.

Visual user agents should set the height of the control to the number of lines given by character height.

The **`wrap`** attribute is an enumerated attribute with two keywords and states: the **`soft`** keyword which maps to the **Soft** state, and the the **`hard`** keyword which maps to the **Hard** state. The

missing value default is the Soft state.

If the element's `wrap` attribute is in the Hard state, the `cols` attribute must be specified.

The element's value is defined to be the element's raw value with the following transformation applied:

1. Replace every occurance of a U+000D CARRIAGE RETURN (CR) character not followed by a U+000A LINE FEED (LF) character, and every occurance of a U+000A LINE FEED (LF) character not proceeded by a U+000D CARRIAGE RETURN (CR) character, by a two-character string consisting of a U+000D CARRIAGE RETURN - U+000A LINE FEED (CRLF) character pair.

2. If the element's `wrap` attribute is in the Hard state, insert U+000D CARRIAGE RETURN - U+000A LINE FEED (CRLF) character pairs into the string using a UA-defined algorithm so that each line so that each line has no more than character width characters. The the purposes of this requirement, lines are delimited by the start of the string, the end of the string, and U+000D CARRIAGE RETURN - U+000A LINE FEED (CRLF) character pairs.

The **`maxlength`** attribute is a form control `maxlength` attribute controlled by the `textarea` element's dirty value flag.

If the `textarea` element has a maximum allowed value length, then the element's children must be such that the codepoint length of the value of the element's `textContent` DOM attribute is equal to or less than the element's maximum allowed value length.

The **`required`** attribute is a boolean attribute. When specified, the user will be required to enter a value before submitting the form.

Constraint validation: If the element has its `required` attribute specified, and the element is mutable, and the element's value is the empty string, then the element is suffering from being missing.

The `form` attribute is used to explicitly associate the `textarea` element with its form owner. The `name` attribute represents the element's name. The `disabled` attribute is used to make the control non-interactive and to prevent its value from being submitted. The `autofocus` attribute controls focus.

textarea **`.type`**

Returns the string "`textarea`".

textarea **`.value`**

Returns the current value of the element.

Can be set, to change the value.

The **`cols`**, **`required`**, **`rows`**, and **`wrap`** attributes must reflect the respective content attributes of the same name. The `cols` and `rows` attributes are limited to only positive non-zero numbers. The **`maxLength`** DOM attribute must reflect the `maxlength` content attribute. The **`readOnly`** DOM

attribute must reflect the `readonly` content attribute.

The **`type`** DOM attribute must return the value "`textarea`".

The **`defaultValue`** DOM attribute must act like the element's `textContent` DOM attribute.

The **`value`** attribute must, on getting, return the element's raw value; on setting, it must set the element's raw value to the new value.

The `willValidate`, `validity`, and `validationMessage` attributes, and the `checkValidity()` and `setCustomValidity()` methods, are part of the constraint validation API. The `labels` attribute provides a list of the element's `label`s. The `select()`, `selectionStart`, `selectionEnd`, and `setSelectionRange()` methods and attributes expose the element's text selection.

4.10.11 The `output` element

Categories

Flow content.
Phrasing content.
Listed and resettable form-associated element.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Content attributes:

Global attributes
`for`
`form`
`name`

DOM interface:

```
interface HTMLOutputElement : HTMLElement {
           attribute DOMString htmlFor;
  readonly attribute HTMLFormElement form;
           attribute DOMString name;

  readonly attribute DOMString type;
           attribute DOMString defaultValue;
           attribute DOMString value;

  readonly attribute boolean willValidate;
  readonly attribute ValidityState validity;
  readonly attribute DOMString validationMessage;
  boolean checkValidity();
  void setCustomValidity(in DOMString error);
};
```

The `output` element represents the result of a calculation.

The **`for`** content attribute allows an explicit relationship to be made between the result of a calculation and the elements that represent the values that went into the calculation or that otherwise influenced the calculation. The `for` attribute, if specified, must contain a string consisting of an unordered set of unique space-separated tokens, each of which must have the value of an ID of an element in the same `Document`.

The `form` attribute is used to explicitly associate the `output` element with its form owner. The `name` attribute represents the element's name.

The element has a **value mode flag** which is either *value* or *default*. Initially the value mode flag must be set to *default*.

When the to value mode flag is in mode *default*, the contents of the element represent both the value of the element and its default value. When the value mode flag is in mode *value*, the contents of the element represent the value of the element only, and the default value is only accessible using the `defaultValue` DOM attribute.

The element also has a **default value**. Initially, the default value must be the empty string.

Whenever the element's descendants are changed in any way, if the value mode flag is in mode *default*, the element's default value must be set to the value of the element's `textContent` DOM attribute.

The reset algorithm for `textarea` elements is to set the element's `textContent` DOM attribute to the value of the element's `defaultValue` DOM attribute (thus replacing the element's child nodes), and then to set the element's value mode flag to *default*.

output **.** **`value`** **[= *value*]**

Returns the element's current value.

Can be set, to change the value.

output **.** **`defaultValue`** **[= *value*]**

Returns the element's current default value.

Can be set, to change the default value.

output **.** **`type`**

Returns the string "`output`".

The **`value`** DOM attribute must act like the element's `textContent` DOM attribute, except that on setting, in addition, before the child nodes are changed, the element's value mode flag must be set to *value*.

The **`defaultValue`** DOM attribute, on getting, must return the element's default value. On setting, the attribute must set the element's default value, and, if the element's value mode flag is in the mode *default*, set the element's `textContent` DOM attribute as well.

The **`type`** attribute must return the string "`output`".

The **`htmlFor`** DOM attribute must reflect the `for` content attribute.

The `willValidate`, `validity`, and `validationMessage` attributes, and the `checkValidity()` and `setCustomValidity()` methods, are part of the constraint validation API.

Constraint validation: `output` elements are always barred from constraint validation.

4.10.12 Association of controls and forms

A form-associated element can have a relationship with a `form` element, which is called the element's **form owner**. If a form-associated element is not associated with a `form` element, its form owner is said to be null.

A form-associated element is, by default, associated with its nearest ancestor `form` element (as described below), but may have a **`form`** attribute specified to override this.

If a form-associated element has a `form` attribute specified, then its value must be the ID of a `form` element in the element's owner `Document`.

When a form-associated element is created, its form owner must be initialized to null (no owner).

When a form-associated element is to be **associated** with a form, its form owner must be set to that form.

When a form-associated element's ancestor chain changes, e.g. because it or one of its ancestors was inserted or removed from a `Document`, then the user agent must reset the form owner of that element.

When a form-associated element's `form` attribute is added, removed, or has its value changed, then the user agent must reset the form owner of that element.

When a form-associated element has a `form` attribute and the ID of any of the `form` elements in the `Document` changes, then the user agent must reset the form owner of that form-associated element.

When the user agent is to **reset the form owner** of a form-associated element, it must run the following steps:

1. If the element's form owner is not null, and the element's `form` content attribute is not present, and the element's form owner is one of the ancestors of the element after the change to the ancestor chain, then do nothing, and abort these steps.

2. Let the element's form owner be null.

3. If the element has a `form` content attribute, then run these substeps:

    1. If the first element in the `Document` to have an ID that is equal to the element's `form` content attribute's value is a `form` element, then associate the form-associated element with that `form` element.

    2. Abort the "reset the form owner" steps.

4. Otherwise, if the form-associated element in question has an ancestor `form` element, then associate the form-associated element with the nearest such ancestor `form` element.

5. Otherwise, the element is left unassociated.

element **.** **`form`**

Returns the element's form owner.

Returns null if there isn't one.

Form-associated elements have a **`form`** DOM attribute, which, on getting, must return the element's form owner, or null if there isn't one.

Constraint validation: If an element has no form owner, it is barred from constraint validation.

4.10.13 Attributes common to form controls

4.10.13.1 Naming form controls

The **`name`** content attribute gives the name of the form control, as used in form submission and in the `form` element's `elements` object. If the attribute is specified, its value must not be the empty string.

Constraint validation: If an element does not have a `name` attribute specified, or its `name` attribute's value is the empty string, then it is barred from constraint validation.

The **`name`** DOM attribute must reflect the `name` content attribute.

4.10.13.2 Enabling and disabling form controls

The **`disabled`** content attribute is a boolean attribute.

A form control is **disabled** if its `disabled` attribute is set, or if it is a descendant of a `fieldset` element whose `disabled` attribute is set.

A form control that is disabled must prevent any `click` events that are queued on the user interaction task source from being dispatched on the element.

Constraint validation: If an element is disabled, it is barred from constraint validation.

The **`disabled`** DOM attribute must reflect the `disabled` content attribute.

4.10.13.3 A form control's value

Form controls have a **value** and a **checkedness**. (The latter is only used by `input` elements.) These are used to describe how the user interacts with the control.

4.10.13.4 Autofocusing a form control

The **autofocus** content attribute allows the user to indicate that a control is to be focused as soon as the page is loaded, allowing the user to just start typing without having to manually focus the main control.

The autofocus attribute is a boolean attribute.

There must not be more than one element in the document with the autofocus attribute specified.

Whenever an element with the autofocus attribute specified is inserted into a document, the user agent should queue a task that checks to see if the element is focusable, and if so, runs the focusing steps for that element. User agents may also change the scrolling position of the document, or perform some other action that brings the element to the user's attention. The task source for this task is the DOM manipulation task source.

User agents may ignore this attribute if the user has indicated (for example, by starting to type in a form control) that he does not wish focus to be changed.

> ***Note: Focusing the control does not imply that the user agent must focus the browser window if it has lost focus.***

The **autofocus** DOM attribute must reflect the content attribute of the same name.

> In the following snippet, the text control would be focused when the document was loaded.
>
> ```
> <input maxlength="256" name="q" value="" autofocus>
> <input type="submit" value="Search">
> ```

4.10.13.5 Limiting user input length

A **form control maxlength attribute**, controlled by a *dirty value flag* declares a limit on the number of characters a user can input.

If an element has its form control maxlength attribute specified, the attribute's value must be a valid non-negative integer. If the attribute is specified and applying the rules for parsing non-negative integers to its value results in a number, then that number is the element's **maximum allowed value length**. If the attribute is omitted or parsing its value results in an error, then there is no maximum allowed value length.

Constraint validation: If an element has a maximum allowed value length, and its *dirty value flag* is false, and the codepoint length of the element's value is greater than the element's maximum allowed value length, then the element is suffering from being too long.

User agents may prevent the user from causing the element's value to be set to a value whose codepoint length is greater than the element's maximum allowed value length.

4.10.13.6 Form submission

Attributes for form submission can be specified both on form elements and on submit

button (elements that represent buttons that submit forms, e.g. an `input` element whose `type` attribute is in the Submit Button state).

The attributes for form submission that may be specified on `form` elements are `action`, `enctype`, `method`, `novalidate`, and `target`.

The corresponding attributes for form submission that may be specified on submit button are `formaction`, `formenctype`, `formmethod`, `formnovalidate`, and `formtarget`. When omitted, they default to the values given on the corresponding attributes on the `form` element.

The **`action`** and **`formaction`** content attributes, if specified, must have a value that is a valid URL.

The **action** of an element is the value of the element's `formaction` attribute, if the element is a submit button and has such an attribute, or the value of its form owner's `action` attribute, if *it* has one, or else the empty string.

The **`method`** and **`formmethod`** content attributes are enumerated attributes with the following keywords and states:

- The keyword **`GET`**, mapping to the state **GET**, indicating the HTTP GET method.
- The keyword **`POST`**, mapping to the state **POST**, indicating the HTTP POST method.
- The keyword **`PUT`**, mapping to the state **PUT**, indicating the HTTP PUT method.
- The keyword **`DELETE`**, mapping to the state **DELETE**, indicating the HTTP DELETE method.

The *missing value default* for these attributes is the GET state.

The **method** of an element is one of those four states. If the element is a submit button and has a `formmethod` attribute, then the element's method is that attribute's state; otherwise, it is the form owner's `method` attribute's state.

The **`enctype`** and **`formenctype`** content attributes are enumerated attributes with the following keywords and states:

- The "**`application/x-www-form-urlencoded`**" keyword and corresponding state.
- The "**`multipart/form-data`**" keyword and corresponding state.
- The "**`text/plain`**" keyword and corresponding state.

The *missing value default* for these attributes is the `application/x-www-form-urlencoded` state.

The **enctype** of an element is one of those three states. If the element is a submit button and has a `formenctype` attribute, then the element's enctype is that attribute's state; otherwise, it is the form owner's `enctype` attribute's state.

The **target** and **formtarget** content attributes, if specified, must have values that are valid browsing context names or keywords.

The **target** of an element is the value of the element's `formtarget` attribute, if the element is a submit button and has such an attribute; or the value of its form owner's `target` attribute, if *it* has such an attribute; or, if one of the child nodes of the `head` element is a `base` element with a `target` attribute, then the value of the `target` attribute of the first such `base` element; or, if there is no such element, the empty string.

The **novalidate** and **formnovalidate** content attributes are boolean attributes. If present, they indicate that the form is not to be validated during submission.

The **no-validate state** of an element is true if the element is a submit button and the element's `formnovalidate` attribute is present, or if the element's form owner's `novalidate` attribute is present, and false otherwise.

The **action**, **method**, **enctype**, and **target** DOM attributes must reflect the respective content attributes of the same name. The **noValidate** DOM attribute must reflect the `novalidate` content attribute. The **formAction** DOM attribute must reflect the `formaction` content attribute. The **formEnctype** DOM attribute must reflect the `formenctype` content attribute. The **formMethod** DOM attribute must reflect the `formmethod` content attribute. The **formNoValidate** DOM attribute must reflect the `formnovalidate` content attribute. The **formTarget** DOM attribute must reflect the `formtarget` content attribute.

4.10.14 Constraints

4.10.14.1 Definitions

A listed form-associated element is a **candidate for constraint validation** unless a condition has **barred the element from constraint validation**. (For example, an element is barred from constraint validation if it is an `output` or `fieldset` element.)

An element can have a **custom validity error message** defined. Initially, an element must have its custom validity error message set to the empty string. When its value is not the empty string, the element is suffering from a custom error. It can be set using the `setCustomValidity()` method. The user agent should use the custom validity error message when alerting the user to the problem with the control.

An element can be constrained in various ways. The following is the list of **validity states** that a form control can be in, making the control invalid for the purposes of constraint validation. (The definitions below are non-normative; other parts of this specification define more precisely when each state applies or does not.)

Suffering from being missing

> ***Note: When a control has no value but has a `required` attribute (`input required`, `textarea required`).***

Suffering from a type mismatch

Note: When a control that allows arbitrary user input has a value that is not in the correct syntax (E-mail, URL).

Suffering from a pattern mismatch

Note: When a control has a value that doesn't satisfy the `pattern` attribute.

Suffering from being too long

Note: When a control has a value that is too long for the form control `maxlength` attribute (`input` `maxlength`, `textarea` `maxlength`).

Suffering from an underflow

Note: When a control has a value that is too low for the `min` attribute.

Suffering from an overflow

Note: When a control has a value that is too high for the `max` attribute.

Suffering from a step mismatch

Note: When a control has a value that doesn't fit the rules given by the `step` attribute.

Suffering from a custom error

Note: When a control's custom validity error message (as set by the element's `setCustomValidity()` method) is not the empty string.

Note: An element can still suffer from these states even when the element is disabled; thus these states can be represented in the DOM even if validating the form during submission wouldn't indicate a problem to the user.

An element **satisfies its constraints** if it is not suffering from any of the above validity states.

4.10.14.2 Constraint validation

When the user agent is required to **statically validate the constraints** of `form` element *form*, it must run the following steps, which return either a *positive* result (all the controls in the form are valid) or a *negative* result (there are invalid controls) along with a (possibly empty) list of elements that are invalid and for which no script has claimed responsibility:

1. Let *controls* be a list of all the submittable elements whose form owner is *form*, in tree order.
2. Let *invalid controls* be an initially empty list of elements.
3. For each element *field* in *controls*, in tree order, run the following substeps:

1. If *field* is not a candidate for constraint validation, then move on to the next element.

2. Otherwise, if *field* satisfies its constaints, then move on to the next element.

3. Otherwise, add *field* to *invalid controls*.

4. If *invalid controls* is empty, then return a *positive* result and abort these steps.

5. Let *unhandled invalid controls* be an initially empty list of elements.

6. For each element *field* in *invalid controls*, if any, in tree order, run the following substeps:

    1. Fire a simple event named `invalid` at *field*.

    2. If the event was not canceled, then add *field* to *unhandled invalid controls*.

7. Return a *negative* result with the list of elements in the *unhandled invalid controls* list.

If a user agent is to **interactively validate the constraints** of `form` element *form*, then the user agent must run the following steps:

1. Statically validate the constraints of *form*, and let *unhandled invalid controls* be the list of elements returned if the result was *negative*.

2. If the result was *positive*, then return that result and abort these steps.

3. Report the problems with the constraints of at least one of the elements given in *unhandled invalid controls* to the user. User agents may focus one of those elements in the process, by running the focusing steps for that element, and may change the scrolling position of the document, or perform some other action that brings the element to the user's attention. User agents may report more than one constraint violation. User agents may coalesce related constraint violation reports if appropriate (e.g. if multiple radio buttons in a group are marked as required, only one error need be reported). If one of the controls is not visible to the user (e.g. it has the `hidden` attribute set) then user agents may report a script error.

4. Return a *negative* result.

4.10.14.3 The constraint validation API

***element*`.willValidate`**

Returns true if the element will be validated when the form is submitted; false otherwise.

***element*`.setCustomValidity(`*message*`)`**

Sets a custom error, so that the elemnt would fail to validate. The given message is the message to be shown to the user when reporting the problem to the user.

If the argument is the empty string, clears the custom error.

element **. validity . valueMissing**

Returns true if the element has no value but is a required field; false otherwise.

element **. validity . typeMismatch**

Returns true if the element's value is not in the correct syntax; false otherwise.

element **. validity . patternMismatch**

Returns true if the element's value doesn't match the provided pattern; false otherwise.

element **. validity . tooLong**

Returns true if the element's value is longer than the provided maximum length; false otherwise.

element **. validity . rangeUnderflow**

Returns true if the element's value is lower than the provided minimum; false otherwise.

element **. validity . rangeOverflow**

Returns true if the element's value is higher than the provided maximum; false otherwise.

element **. validity . stepMismatch**

Returns true if the element's value doesn't fit the rules given by the `step` attribute; false otherwise.

element **. validity . customError**

Returns true if the element has a custom error; false otherwise.

element **. validity . valid**

Returns true if the element's value has no validity problems; false otherwise.

valid **=** ***element*** **. checkValidity()**

Returns true if the element's value has no validity problems; false otherwise. Fires an `invalid` event at the element in the latter case.

element **. validationMessage**

Returns the error message that would be shown to the user if the element was to be checked for validity.

The **`willValidate`** attribute must return true if an element is a candidate for constraint validation, and false otherwise (i.e. false if any conditions are barring it from constraint validation).

The **`setCustomValidity(`*`message`*`)`**, when invoked, must set the custom validity error message to the value of the given *message* argument.

The **`validity`** attribute must return a `ValidityState` object that represents the validity states of the element. This object is live, and the same object must be returned each time the element's `validity` attribute is retrieved.

```
interface ValidityState {
  readonly attribute boolean valueMissing;
  readonly attribute boolean typeMismatch;
  readonly attribute boolean patternMismatch;
  readonly attribute boolean tooLong;
  readonly attribute boolean rangeUnderflow;
  readonly attribute boolean rangeOverflow;
  readonly attribute boolean stepMismatch;
  readonly attribute boolean customError;
  readonly attribute boolean valid;
};
```

A `ValidityState` object has the following attributes. On getting, they must return true if the corresponding condition given in the following list is true, and false otherwise.

`valueMissing`
: The control is suffering from being missing.

`typeMismatch`
: The control is suffering from a type mismatch.

`patternMismatch`
: The control is suffering from a pattern mismatch.

`tooLong`
: The control is suffering from being too long.

`rangeUnderflow`
: The control is suffering from an underflow.

`rangeOverflow`
: The control is suffering from an overflow.

`stepMismatch`
: The control is suffering from a step mismatch.

`customError`
: The control is suffering from a custom error.

`valid`
: None of the other conditions are true.

When the **`checkValidity()`** method is invoked, if the element is a candidate for constraint validation and does not satisfy its constaints, the user agent must fire a simple event named `invalid` at the element and return false. Otherwise, it must only return true without doing anything else.

The **`validationMessage`** attribute must return the empty string if the element is not a candidate for constraint validation or if it is one but it satisfies its constaints; otherwise, it must return a

suitably localised message that the user agent would show the user if this were the only form with a validity constraint problem. If the element is suffering from a custom error, then the custom validity error message should be present in the return value.

4.10.14.4 Security

Servers should not rely on client-side validation. Client-side validation can be intentionally bypassed by hostile users, and unintentionally bypassed by users of older user agents or automated tools that do not implement these features. The constraint validation features are only intended to improve the user experience, not to provide any kind of security mechanism.

4.10.15 Form submission

4.10.15.1 Introduction

This section is non-normative.

...

4.10.15.2 Implicit submission

User agents may establish a button in each form as being the form's **default button**. This should be the first submit button in tree order whose form owner is that `form` element, but user agents may pick another button if another would be more appropriate for the platform. If the platform supports letting the user submit a form implicitly (for example, on some platforms hitting the "enter" key while a text field is focused implicitly submits the form), then doing so must cause the form's default button's activation behavior, if any, to be run.

> ***Note: Consequently, if the default button is disabled, the form is not submitted when such an implicit submission mechanism is used. (A button has no activation behavior when disabled.)***

If the form has no submit button, then the implicit submission mechanism must just submit the `form` element from the `form` element itself.

4.10.15.3 Form submission algorithm

When a form *form* is **submitted** from an element *submitter* (typically a button), the user agent must run the following steps:

1. If *form* is in a `Document` that has no associated browsing context or whose browsing context has its sandboxed forms browsing context flag set, then abort these steps without doing anything.

2. If *form* is already being submitted (i.e. the form was submitted again while processing

the events fired from the next two steps, probably from a script redundantly calling the `submit()` method on *form*), then abort these steps. This doesn't affect the earlier instance of this algorithm.

3. If the *submitter* is anything but a `form` element, and the *submitter* element's no-validate state is false, then interactively validate the constraints of *form* and examine the result: if the result is negative (the constraint validation concluded that there were invalid fields and probably informed the user of this) then abort these steps.

4. If the *submitter* is anything but a `form` element, then fire a simple event that bubbles, named `submit`, at *form*. If the event's default action is prevented (i.e. if the event is canceled) then abort these steps. Otherwise, continue (effectively the default action is to perform the submission).

5. Let *controls* be a list of all the submittable elements whose form owner is *form*, in tree order.

6. Let the *form data set* be a list of name-value-type tuples, initially empty.

7. **Constructing the form data set**. For each element *field* in *controls*, in tree order, run the following substeps:

    1. If any of the following conditions are met, then skip these substeps for this element:

        - The *field* element has a `datalist` element ancestor.
        - The *field* element is disabled.
        - The *field* element is a button but it is not *submitter*.
        - The *field* element is an `input` element whose `type` attribute is in the Checkbox state and whose checkedness is false.
        - The *field* element is an `input` element whose `type` attribute is in the Radio Button state and whose checkedness is false.
        - The *field* element is an `input` element whose `type` attribute is in the File Upload state but the control does not have any files selected.
        - The *field* element is an `object` element that is not using a plugin.

        Otherwise, process *field* as follows:

    2. Let *type* be the value of the `type` DOM attribute of *field*.

    3. If the *field* element is an `input` element whose `type` attribute is in the Image Button state, then run these further nested substeps:

        1. If the *field* element has an `name` attribute specified and value is not the empty string, let *name* be that value followed by a single U+002E FULL STOP (.) character. Otherwise, let *name* be the empty string.

        2. Let $name_x$ be the string consisting of the concatenation of *name* and a single

U+0078 LATIN SMALL LETTER X (x) character.

3. Let *name*$_Y$ be the string consisting of the concatenation of *name* and a single U+0079 LATIN SMALL LETTER Y (y) character.

4. The *field* element is *submitter*, and before this algorithm was invoked the user indicated a coordinate. Let *x* be the *x*-component of the coordindate selected by the user, and let *y* be the *y*-component of the coordinate selected by the user.

5. Append an entry in the *form data set* with the name *name*$_X$, the value *x*, and the type *type*.

6. Append an entry in the *form data set* with the name *name*$_Y$ and the value *y*, and the type *type*.

7. Skip the remaining substeps for this element: if there are any more elements in *controls*, return to the top of the constructing the form data set step, otherwise, jump to the next step in the overall form submission algorithm.

4. If the *field* element does not have a `name` attribute specified, or its `name` attribute's value is the empty string, skip these substeps for this element: if there are any more elements in *controls*, return to the top of the constructing the form data set step, otherwise, jump to the next step in the overall form submission algorithm.

5. Let *name* be the value of the *field* element's `name` attribute.

6. If the *field* element is a `select` element, then for each `option` element in the `select` element whose selectedness is true, append an entry in the *form data set* with the *name* as the name, the value of the `option` element as the value, and *type* as the type.

7. Otherwise, if the *field* element is an `input` element whose `type` attribute is in the Checkbox state or the Radio Button state, then then run these further nested substeps:

    1. If the *field* element has a `value` attribute specified, then let *value* be the value of that attribute; otherwise, let *value* be the string "`on`".

    2. Append an entry in the *form data set* with *name* as the name, *value* as the value, and *type* as the type.

8. Otherwise, if the *field* element is an `input` element whose `type` attribute is in the File Upload state, then for each file selected in the `input` element, append an entry in the *form data set* with the *name* as the name, the file (consisting of the name, the type, and the body) as the value, and *type* as the type.

9. Otherwise, if the *field* element is an `object` element: try to obtain a form submission value from the plugin, and if that is successful, append an entry in the *form data set* with *name* as the name, the returned form submission value as the value, and the string "`object`" as the type.

10. Otherwise, append an entry in the *form data set* with *name* as the name, the value of the *field* element as the value, and *type* as the type.

8. Let *action* be the *submitter* element's action.

9. If *action* is the empty string, let *action* be the document's address.

   Note: This step is a willful violation of RFC 3986. [RFC3986]

10. Resolve the URL *action*, relative to the *submitter* element. If this fails, abort these steps. Otherwise, let *action* be the resulting absolute URL.

11. Let *scheme* be the <scheme> of the resulting absolute URL.

12. Let *enctype* be the *submitter* element's enctype.

13. Let *method* be the *submitter* element's method.

14. Let *target* be the *submitter* element's target.

15. Select the appropriate row in the table below based on the value of *scheme* as given by the first cell of each row. Then, select the appropriate cell on that row based on the value of *method* as given in the first cell of each column. Then, jump to the steps named in that cell and defined below the table.

	GET	**POST**	**PUT**	**DELETE**
`http`	Mutate action	Submit as entity body	Submit as entity body	Delete action
`https`	Mutate action	Submit as entity body	Submit as entity body	Delete action
`ftp`	Get action	Get action	Get action	Get action
`javascript`	Get action	Get action	Get action	Get action
`data`	Get action	Post to data:	Put to data:	Get action
`mailto`	Mail with headers	Mail as body	Mail with headers	Mail with headers

If *scheme* is not one of those listed in this table, then the behavior is not defined by this specification. User agents should, in the absence of another specification defining this, act in a manner analogous to that defined in this specification for similar schemes.

The behaviors are as follows:

Mutate action

Let *query* be the result of encoding the *form data set* using the `application/x-www-form-urlencoded` encoding algorithm, interpreted as a US-ASCII string.

Let *destination* be a new URL that is equal to the *action* except that its <query> component is replaced by *query* (adding a U+003F QUESTION MARK (?) character if appropriate).

Let *target browsing context* be the form submission target browsing context.

Navigate *target browsing context* to *destination*. If *target browsing context* was newly created for this purpose by the steps above, then it must be navigated with replacement enabled.

Submit as entity body

Let *entity body* be the result of encoding the *form data set* using the appropriate form encoding algorithm.

Let *target browsing context* be the form submission target browsing context.

Let *MIME type* be determined as follows:

If *enctype* is `application/x-www-form-urlencoded`
Let *MIME type* be "`application/x-www-form-urlencoded`".

If *enctype* is `multipart/form-data`
Let *MIME type* be "`multipart/form-data`".

If *enctype* is `text/plain`
Let *MIME type* be "`text/plain`".

Navigate *target browsing context* to *action* using the HTTP method given by *method* and with *entity body* as the entity body, of type *MIME type*. If *target browsing context* was newly created for this purpose by the steps above, then it must be navigated with replacement enabled.

Delete action

Let *target browsing context* be the form submission target browsing context.

Navigate *target browsing context* to *action* using the DELETE method. If *target browsing context* was newly created for this purpose by the steps above, then it must be navigated with replacement enabled.

Get action

Let *target browsing context* be the form submission target browsing context.

Navigate *target browsing context* to *action*. If *target browsing context* was newly created for this purpose by the steps above, then it must be navigated with replacement enabled.

Post to data:

Let *data* be the result of encoding the *form data set* using the appropriate form encoding algorithm.

If *action* contains the string "`%%%%`" (four U+0025 PERCENT SIGN characters), then %-escape all bytes in *data* that, if interpreted as US-ASCII, do not match the `unreserved` production in the URI Generic Syntax, and then, treating the result as a US-ASCII string, further %-escape all the U+0025 PERCENT SIGN characters in the resulting string and replace the first occurance of "`%%%%`" in *action* with the resulting double-escaped string. [RFC3986]

Otherwise, if *action* contains the string "`%%`" (two U+0025 PERCENT SIGN characters in a row, but not four), then %-escape all characters in *data* that, if interpreted as US-ASCII, do not match the `unreserved` production in the URI Generic Syntax, and then, treating the result as a US-ASCII string, replace the first

occurance of "%%" in *action* with the resulting escaped string. [RFC3986]

Let *target browsing context* be the form submission target browsing context.

Navigate *target browsing context* to the potentially modified *action*. If *target browsing context* was newly created for this purpose by the steps above, then it must be navigated with replacement enabled.

Put to data:

Let *data* be the result of encoding the *form data set* using the appropriate form encoding algorithm.

Let *MIME type* be determined as follows:

If *enctype* is `application/x-www-form-urlencoded`
Let *MIME type* be "`application/x-www-form-urlencoded`".

If *enctype* is `multipart/form-data`
Let *MIME type* be "`multipart/form-data`".

If *enctype* is `text/plain`
Let *MIME type* be "`text/plain`".

Let *destination* be the result of concatenating the following:

1. The string "`data:`".
2. The value of *MIME type*.
3. The string "`base64,`".
4. A base-64 encoded representation of *data*. [RFC2045]

Let *target browsing context* be the form submission target browsing context.

Navigate *target browsing context* to *destination*. If *target browsing context* was newly created for this purpose by the steps above, then it must be navigated with replacement enabled.

Mail with headers

Let *headers* be the resulting encoding the *form data set* using the `application/x-www-form-urlencoded` encoding algorithm, interpreted as a US-ASCII string.

Replace occurances of U+002B PLUS SIGN characters (+) in *headers* with the string "`%20`".

Let *destination* consist of all the characters from the first character in *action* to the character immediately before the first U+003F QUESTION MARK character (?), if any, or the end of the string if there are none.

Append a single U+003F QUESTION MARK character (?) to *destination*.

Append *headers* to *destination*.

Let *target browsing context* be the form submission target browsing context.

Navigate *target browsing context* to *destination*. If *target browsing context* was newly created for this purpose by the steps above, then it must be navigated with

replacement enabled.

Mail as body

Let *body* be the resulting encoding the *form data set* using the appropriate form encoding algorithm and then %-escaping all the bytes in the resulting byte string that, when interpreted as US-ASCII, do not match the `unreserved` production in the URI Generic Syntax. [RFC3986]

Let *destination* have the same value as *action*.

If *destination* does not contain a U+003F QUESTION MARK character (?), append a single U+003F QUESTION MARK character (?) to *destination*. Otherwise, append a single U+0026 AMPERSAND character (&).

Append the string "`body=`" to *destination*.

Append *body*, interpreted as a US-ASCII string, to *destination*.

Let *target browsing context* be the form submission target browsing context.

Navigate *target browsing context* to *destination*. If *target browsing context* was newly created for this purpose by the steps above, then it must be navigated with replacement enabled.

The form submission target browsing context is obtained, when needed by the behaviors described above, as follows: If the user indicated a specific browsing context to use when submitting the form, then that is the target browsing context. Otherwise, apply the rules for choosing a browsing context given a browsing context name using *target* as the name and the browsing context of *form* as the context in which the algorithm is executed; the resulting browsing context is the target browsing context.

The **appropriate form encoding algorithm** is determined as follows:

If *enctype* is `application/x-www-form-urlencoded`
: Use the `application/x-www-form-urlencoded` encoding algorithm.

If *enctype* is `multipart/form-data`
: Use the `multipart/form-data` encoding algorithm.

If *enctype* is `text/plain`
: Use the `text/plain` encoding algorithm.

4.10.15.4 URL-encoded form data

The **`application/x-www-form-urlencoded` encoding algorithm** is as follows:

1. Let *result* be the empty string.

2. If the `form` element has an `accept-charset` attribute, then, taking into account the characters found in the *form data set*'s names and values, and the character encodings supported by the user agent, select a character encoding from the list given in the `form`'s `accept-charset` attribute that is an ASCII-compatible character encoding. If none of the encodings are supported, then let the selected character encoding be UTF-8.

Otherwise, if the document's character encoding is an ASCII-compatible character encoding, then that is the selected character encoding.

Otherwise, let the selected character encoding be UTF-8.

3. Let *charset* be the preferred MIME name of the selected character encoding.
4. If the entry's name is "`_charset_`" and its type is "`hidden`", replace its value with *charset*.
5. If the entry's type is "`file`", replace its value with the file's filename only.
6. For each entry in the *form data set*, perform these substeps:
    1. For each character in the entry's name and value that cannot be expressed using the selected character encoding, replace the character by a string consisting of a U+0026 AMPERSAND character (&), one of more characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9) representing the Unicode codepoint of the character in base ten, and finally a U+003B SEMICOLON character (;).
    2. For each character in the entry's name and value, apply the following subsubsteps:
        1. If the character isn't in the range U+0020, U+002A, U+002D, U+002E, U+0030 .. U+0039, U+0041 .. U+005A, U+005F, U+0061 .. U+007A then replace the character with a string formed as follows: Start with the empty string, and then, taking each byte of the character when expressed in the selected character encoding in turn, append to the string a U+0025 PERCENT SIGN character (%) followed by two characters in the ranges U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9) and U+0041 LATIN CAPITAL LETTER A to U+005A LATIN CAPITAL LETTER Z representing the hexadecimal value of the byte (zero-padded if necessary).
        2. If the character is a U+0020 SPACE character, replace it with a single U+002B PLUS SIGN character (+).
    3. If the entry's name is "`isindex`", its type is "`text`", and this is the first entry in the *form data set*, then append the value to *result* and skip the rest of the substeps for this entry, moving on to the next entry, if any, or the next step in the overall algorithm otherwise.
    4. If this is not the first entry, append a single U+0026 AMPERSAND character (&) to *result*.
    5. Append the entry's name to *result*.
    6. Append a single U+003D EQUALS SIGN character (=) to *result*.
    7. Append the entry's value to *result*.
7. Encode *result* as US-ASCII and return the resulting byte stream.

4.10.15.5 Multipart form data

The **`multipart/form-data` encoding algorithm** is to encode the *form data set* using the rules described by RFC2388, *Returning Values from Forms:* `multipart/form-data`, and return the resulting byte stream. [RFC2388]

Each entry in the *form data set* is a *field*, the name of the entry is the *field name* and the value of the entry is the *field value*.

The order of parts must be the same as the order of fields in the *form data set*. Multiple entries with the same name must be treated as distinct fields.

4.10.15.6 Plain text form data

The **`text/plain` encoding algorithm** is as follows:

1. Let *result* be the empty string.
2. If the `form` element has an `accept-charset` attribute, then, taking into account the characters found in the *form data set*'s names and values, and the character encodings supported by the user agent, select a character encoding from the list given in the `form`'s `accept-charset` attribute. If none of the encodings are supported, then let the selected character encoding be UTF-8.

   Otherwise, the selected character encoding is the document's character encoding.
3. Let *charset* be the preferred MIME name of the selected character encoding.
4. If the entry's name is "`_charset_`" and its type is "`hidden`", replace its value with *charset*.
5. If the entry's type is "`file`", replace its value with the file's filename only.
6. For each entry in the *form data set*, perform these substeps:
   1. Append the entry's name to *result*.
   2. Append a single U+003D EQUALS SIGN character (=) to *result*.
   3. Append the entry's value to *result*.
   4. Append a U+000D CARRIAGE RETURN (CR) U+000A LINE FEED (LF) character pair to *result*.
7. Encode *result* using the selected character encoding and return the resulting byte stream.

4.10.16 Resetting a form

When a form *form* is **reset**, the user agent must invoke the reset algorithm of each resettable elements whose form owner is *form*, and must then broadcast `formchange` events from *form*.

Each resettable element defines its own **reset algorithm**. Changes made to form controls as part of these algorithms do not count as changes caused by the user (and thus, e.g., do not cause `input` events to fire).

4.10.17 Event dispatch

When the user agent is to **broadcast `forminput` events** or **broadcast `formchange` events** from a `form` element *form*, it must run the following steps:

1. Let *controls* be a list of all the resettable elements whose form owner is *form*.
2. If the user agent was to broadcast `forminput` events, let *event name* be `forminput`. Otherwise the user agent was to broadcast `formchange` events; let *event name* be `formchange`.
3. For each element in *controls*, in tree order, fire a simple event named *event name* at the element.

4.11 Interactive elements

4.11.1 The `details` element

Categories
Flow content.
Interactive content.

Contexts in which this element may be used:
Where flow content is expected.

Content model:
One `legend` element followed by flow content.

Content attributes:
Global attributes
`open`

DOM interface:

```
interface HTMLDetailsElement : HTMLElement {
           attribute boolean open;
};
```

The `details` element represents additional information or controls which the user can obtain on demand.

> ***Note: The `details` element is not appropriate for footnotes. Please see the section on footnotes for details on how to mark up footnotes.***

The first element child of a `details` element, if it is a `legend` element, represents the summary of the details.

If the first element is not a `legend` element, the UA should provide its own legend (e.g. "Details").

The **`open`** content attribute is a boolean attribute. If present, it indicates that the details are to

be shown to the user. If the attribute is absent, the details are not to be shown.

If the attribute is removed, then the details should be hidden. If the attribute is added, the details should be shown.

The user agent should allow the user to request that the details be shown or hidden. To honor a request for the details to be shown, the user agent must set the `open` attribute on the element to the value `open`. To honour a request for the details to be hidden, the user agent must remove the `open` attribute from the element.

The **open** attribute must reflect the `open` content attribute.

4.11.2 The `datagrid` element

The API here needs rewriting. Don't implement this right now.

Categories

Flow content.
Interactive content.
Sectioning root.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Either: Nothing.
Or: Flow content, but where the first element child node, if any, is not a `table`, `select`, or `datalist` element.
Or: A single `table` element.
Or: A single `select` element.
Or: A single `datalist` element.

Content attributes:

Global attributes
`multiple`
`disabled`

DOM interface:

```
interface HTMLDataGridElement : HTMLElement {
           attribute DataGridDataProvider data;
  readonly attribute DataGridSelection selection;
           attribute boolean multiple;
           attribute boolean disabled;
  void updateEverything();
  void updateRowsChanged(in RowSpecification row, in unsigned long count);
  void updateRowsInserted(in RowSpecification row, in unsigned long count)
  void updateRowsRemoved(in RowSpecification row, in unsigned long count);
  void updateRowChanged(in RowSpecification row);
  void updateColumnChanged(in unsigned long column);
  void updateCellChanged(in RowSpecification row, in unsigned long column)
};
```

One possible thing to be added is a way to detect when a row/selection has been deleted, activated, etc, by the user (delete key, enter key, etc).

The datagrid element represents an interactive representation of tree, list, or tabular data.

The data being presented can come either from the content, as elements given as children of the datagrid element, or from a scripted data provider given by the data DOM attribute.

The multiple and disabled attributes are boolean attributes. Their effects are described in the processing model sections below.

The **multiple** and **disabled** DOM attributes must reflect the multiple and disabled content attributes respectively.

4.11.2.1 The datagrid data model

This section is non-normative.

In the datagrid data model, data is structured as a set of rows representing a tree, each row being split into a number of columns. The columns are always present in the data model, although individual columns may be hidden in the presentation.

Each row can have child rows. Child rows may be hidden or shown, by closing or opening (respectively) the parent row.

Rows are referred to by the path along the tree that one would take to reach the row, using zero-based indices. Thus, the first row of a list is row "0", the second row is row "1"; the first child row of the first row is row "0,0", the second child row of the first row is row "0,1"; the fourth child of the seventh child of the third child of the tenth row is "9,2,6,3", etc.

The columns can have captions. Those captions are not considered a row in their own right, they are obtained separately.

Selection of data in a datagrid operates at the row level. If the multiple attribute is present, multiple rows can be selected at once, otherwise the user can only select one row at a time.

The datagrid element can be disabled entirely by setting the disabled attribute.

Columns, rows, and cells can each have specific flags, known as classes, applied to them by the data provider. These classes affect the functionality of the datagrid element, and are also passed to the style system. They are similar in concept to the class attribute, except that they are not specified on elements but are given by scripted data providers.

4.11.2.2 How rows are identified

The chains of numbers that give a row's path, or identifier, are represented by objects that implement the RowSpecification interface.

```
[NoInterfaceObject] interface RowSpecification {
  // binding-specific interface
};
```

In ECMAScript, two classes of objects are said to implement this interface: Numbers representing non-negative integers, and homogeneous arrays of Numbers representing non-negative integers. Thus, `[1,0,9]` is a RowSpecification, as is `1` on its own. However,

`[1,0.2,9]` is not a RowSpecification object, since its second value is not an integer.

User agents must always represent `RowSpecification`s in ECMAScript by using arrays, even if the path only has one number.

The root of the tree is represented by the empty path; in ECMAScript, this is the empty array (`[]`). Only the `getRowCount()` and `GetChildAtPosition()` methods ever get called with the empty path.

4.11.2.3 The data provider interface

The conformance criteria in this section apply to any implementation of the `DataGridDataProvider`, including (and most commonly) the content author's implementation(s).

The API here needs rewriting. Don't implement this right now.

```
// To be implemented by Web authors as a JS object
[NoInterfaceObject] interface DataGridDataProvider {
  void initialize(in HTMLDataGridElement datagrid);
  unsigned long getRowCount(in RowSpecification row);
  unsigned long getChildAtPosition(in RowSpecification parentRow, in unsigned long
  unsigned long getColumnCount();
  DOMString getCaptionText(in unsigned long column);
  void getCaptionClasses(in unsigned long column, in DOMTokenList classes);
  DOMString getRowImage(in RowSpecification row);
  HTMLMenuElement getRowMenu(in RowSpecification row);
  void getRowClasses(in RowSpecification row, in DOMTokenList classes);
  DOMString getCellData(in RowSpecification row, in unsigned long column);
  void getCellClasses(in RowSpecification row, in unsigned long column, in DOMToke
  void toggleColumnSortState(in unsigned long column);
  void setCellCheckedState(in RowSpecification row, in unsigned long column, in lo
  void cycleCell(in RowSpecification row, in unsigned long column);
  void editCell(in RowSpecification row, in unsigned long column, in DOMString dat
};
```

The `DataGridDataProvider` interface represents the interface that objects must implement to be used as custom data views for `datagrid` elements.

Not all the methods are required. The minimum number of methods that must be implemented in a useful view is two: the `getRowCount()` and `getCellData()` methods.

Once the object is written, it must be hooked up to the `datagrid` using the **`data`** DOM attribute.

The following methods may be usefully implemented:

`initialize(datagrid)`
: Called by the `datagrid` element (the one given by the *datagrid* argument) after it has first populated itself. This would typically be used to set the initial selection of the `datagrid` element when it is first loaded. The data provider could also use this method call to register a `select` event handler on the `datagrid` in order to monitor selection changes.

`getRowCount(row)`
: Must return the number of rows that are children of the specified *row*, including rows that are off-screen. If *row* is empty, then the number of rows at the top level must be

returned. If the value that this method would return for a given *row* changes, the relevant update methods on the `datagrid` must be called first. Otherwise, this method must always return the same number. For a list (as opposed to a tree), this method must return 0 whenever it is called with a *row* identifier that is not empty.

getChildAtPosition(parentRow, position)

Must return the index of the row that is a child of *parentRow* and that is to be positioned as the *position*th row under *parentRow* when rendering the children of *parentRow*. If *parentRow* is empty, then *position* refers to the *position*th row at the top level of the data grid. May be omitted if the rows are always to be sorted in the natural order. (The natural order is the one where the method always returns *position*.) For a given *parentRow*, this method must never return the same value for different values of *position*. The returned value *x* must be in the range $0 \leq x < n$, where *n* is the value returned by `getRowCount(parentRow)`.

getColumnCount()

Must return the number of columns currently in the data model (including columns that might be hidden). May be omitted if there is only one column. If the value that this method would return changes, the `datagrid`'s `updateEverything()` method must be called.

getCaptionText(column)

Must return the caption, or label, for column *column*. May be omitted if the columns have no captions. If the value that this method would return changes, the `datagrid`'s `updateColumnChanged()` method must be called with the appropriate column index.

getCaptionClasses(column, classes)

Must add the classes that apply to column *column* to the *classes* object. May be omitted if the columns have no special classes. If the classes that this method would add changes, the `datagrid`'s `updateColumnChanged()` method must be called with the appropriate column index. Some classes have predefined meanings.

getRowImage(row)

Must return a URL giving the address of an image that represents row *row*, or the empty string if there is no applicable image. May be omitted if no rows have associated images. If the value that this method would return changes, the `datagrid`'s update methods must be called to update the row in question.

getRowMenu(row)

Must return an `HTMLMenuElement` object that is to be used as a context menu for row *row*, or null if there is no particular context menu. May be omitted if none of the rows have a special context menu. As this method is called immediately before showing the menu in question, no precautions need to be taken if the return value of this method changes.

getRowClasses(row, classes)

Must add the classes that apply to row *row* to the *classes* object. May be omitted if the rows have no special classes. If the classes that this method would add changes, the `datagrid`'s update methods must be called to update the row in question. Some classes have predefined meanings.

getCellData(row, column)

Must return the value of the cell on row *row* in column *column*. For text cells, this must be the text to show for that cell. For progress bar cells, this must be either a floating point number in the range 0.0 to 1.0 (converted to a string representation), indicating the fraction of the progress bar to show as full (1.0 meaning complete), or the empty string, indicating an indeterminate progress bar. If the value that this method would return changes, the `datagrid`'s update methods must be called to update the rows that changed. If only one cell changed, the `updateCellChanged()` method may be used.

getCellClasses(row, column, classes)

Must add the classes that apply to the cell on row *row* in column *column* to the *classes*

object. May be omitted if the cells have no special classes. If the classes that this method would add changes, the `datagrid`'s update methods must be called to update the rows or cells in question. Some classes have predefined meanings.

toggleColumnSortState(column)

Called by the `datagrid` when the user tries to sort the data using a particular column *column*. The data provider must update its state so that the `GetChildAtPosition()` method returns the new order, and the classes of the columns returned by `getCaptionClasses()` represent the new sort status. There is no need to tell the `datagrid` that it the data has changed, as the `datagrid` automatically assumes that the entire data model will need updating.

setCellCheckedState(row, column, state)

Called by the `datagrid` when the user changes the state of a checkbox cell on row *row*, column *column*. The checkbox should be toggled to the state given by *state*, which is a positive integer (1) if the checkbox is to be checked, zero (0) if it is to be unchecked, and a negative number (−1) if it is to be set to the indeterminate state. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

cycleCell(row, column)

Called by the `datagrid` when the user changes the state of a cyclable cell on row *row*, column *column*. The data provider should change the state of the cell to the new state, as appropriate. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

editCell(row, column, data)

Called by the `datagrid` when the user edits the cell on row *row*, column *column*. The new value of the cell is given by *data*. The data provider should update the cell accordingly. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

The following classes (for rows, columns, and cells) may be usefully used in conjunction with this interface:

Class name	Applies to	Description
checked	Cells	The cell has a checkbox and it is checked. (The `cyclable` and `progress` classes override this, though.)
cyclable	Cells	The cell can be cycled through multiple values. (The `progress` class overrides this, though.)
editable	Cells	The cell can be edited. (The `cyclable`, `progress`, `checked`, `unchecked` and `indeterminate` classes override this, though.)
header	Rows	The row is a heading, not a data row.
indeterminate	Cells	The cell has a checkbox, and it can be set to an indeterminate state. If neither the `checked` nor `unchecked` classes are present, then the checkbox is in that state, too. (The `cyclable` and `progress` classes override this, though.)
initially-hidden	Columns	The column will not be shown when the `datagrid` is initially rendered. If this class is not present on the column when the `datagrid` is initially rendered, the column will be visible if space allows.

initially-closed	Rows	The row will be closed when the datagrid is initially rendered. If neither this class nor the initially-open class is present on the row when the datagrid is initially rendered, the initial state will depend on platform conventions.
initially-open	Rows	The row will be opened when the datagrid is initially rendered. If neither this class nor the initially-closed class is present on the row when the datagrid is initially rendered, the initial state will depend on platform conventions.
progress	Cells	The cell is a progress bar.
reversed	Columns	If the cell is sorted, the sort direction is descending, instead of ascending.
selectable-separator	Rows	The row is a normal, selectable, data row, except that instead of having data, it only has a separator. (The header and separator classes override this, though.)
separator	Rows	The row is a separator row, not a data row. (The header class overrides this, though.)
sortable	Columns	The data can be sorted by this column.
sorted	Columns	The data is sorted by this column. Unless the reversed class is also present, the sort direction is ascending.
unchecked	Cells	The cell has a checkbox and, unless the checked class is present as well, it is unchecked. (The cyclable and progress classes override this, though.)

4.11.2.4 The default data provider

The user agent must supply a default data provider for the case where the datagrid's data attribute is null. It must act as described in this section.

The behavior of the default data provider depends on the nature of the first element child of the datagrid.

↪ **While the first element child is a table element**

getRowCount(*row*): The number of rows returned by the default data provider for the root of the tree (when *row* is empty) must be the total number of tr elements that are children of tbody elements that are children of the table, if there are any such child tbody elements. If there are no such tbody elements then the number of rows returned for the root must be the number of tr elements that are children of the table.

When *row* is not empty, the number of rows returned must be zero.

Note: The table-based default data provider cannot represent a tree.

Note: Rows in thead elements do not contribute to the number of rows returned, although they do affect the columns and column

captions. Rows in `tfoot` elements are ignored completely by this algorithm.

`getChildAtPosition(row, i)`: The default data provider must return the mapping appropriate to the current sort order.

`getColumnCount()`: The number of columns returned must be the number of `td` element children in the first `tr` element child of the first `tbody` element child of the `table`, if there are any such `tbody` elements. If there are no such `tbody` elements, then it must be the number of `td` element children in the first `tr` element child of the `table`, if any, or otherwise 1. If the number that would be returned by these rules is 0, then 1 must be returned instead.

`getCaptionText(i)`: If the `table` has no `thead` element child, or if its first `thead` element child has no `tr` element child, the default data provider must return the empty string for all captions. Otherwise, the value of the `textContent` attribute of the *i*th `th` element child of the first `tr` element child of the first `thead` element child of the `table` element must be returned. If there is no such `th` element, the empty string must be returned.

`getCaptionClasses(i, classes)`: If the `table` has no `thead` element child, or if its first `thead` element child has no `tr` element child, the default data provider must not add any classes for any of the captions. Otherwise, each class in the `class` attribute of the *i*th `th` element child of the first `tr` element child of the first `thead` element child of the `table` element must be added to the *classes*. If there is no such `th` element, no classes must be added. The user agent must then:

1. Remove the `sorted` and `reversed` classes.
2. If the `table` element has a `class` attribute that includes the `sortable` class, add the `sortable` class.
3. If the column is the one currently being used to sort the data, add the `sorted` class.
4. If the column is the one currently being used to sort the data, and it is sorted in descending order, add the `reversed` class as well.

The various row- and cell- related methods operate relative to a particular element, the element of the row or cell specified by their arguments.

For rows: Since the default data provider for a `table` always returns 0 as the number of children for any row other than the root, the path to the row passed to these methods will always consist of a single number. In the prose below, this number is referred to as *i*.

If the `table` has `tbody` element children, the element for the *i*th row is the *i*th `tr` element that is a child of a `tbody` element that is a child of the `table` element. If the `table` does not have `tbody` element children, then the element for the *i*th real row is the *i*th `tr` element that is a child of the `table` element.

For cells: Given a row and its element, the row's *i*th cell's element is the *i*th `td` element child of the row element.

Note: The `colspan` and `rowspan` attributes are ignored by this algorithm.

`getRowImage(i)`: The URL of the row's image is the absolute URL obtained by resolving the value of the `src` attribute of the first `img` element child of the row's first cell's element, relative to that element, if there is one and resolving its attribute is successful. Otherwise, the URL of the row's image is the empty string.

`getRowMenu(i)`: If the row's first cell's element has a `menu` element child, then the row's menu is the first `menu` element child of the row's first cell's element. Otherwise, the row has no menu.

`getRowClasses(i, classes)`: The default data provider must never add a class to the row's classes.

`toggleColumnSortState(i)`: If the data is already being sorted on the given column, then the user agent must change the current sort mapping to be the inverse of the current sort mapping; if the sort order was ascending before, it is now descending, otherwise it is now ascending. Otherwise, if the current sort column is another column, or the data model is currently not sorted, the user agent must create a new mapping, which maps rows in the data model to rows in the DOM so that the rows in the data model are sorted by the specified column, in ascending order. (Which sort comparison operator to use is left up to the UA to decide.)

When the sort mapping is changed, the values returned by the `getChildAtPosition()` method for the default data provider will change appropriately.

`getCellData(i, j)`, **`getCellClasses(i, j, classes)`**, **`getCellCheckedState(i, j, state)`**, **`cycleCell(i, j)`**, **and** **`editCell(i, j, data)`**: See the common definitions below.

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate. For example, if a `tr` is removed, then the `updateRowsRemoved()` methods would probably need to be invoked, and any change to a cell or its descendants must cause the cell to be updated. If the `table` element stops being the first child of the `datagrid`, then the data provider must call the `updateEverything()` method on the `datagrid`. Any change to a cell that is in the column that the data provider is currently using as its sort column must also cause the sort to be reperformed, with a call to `updateEverything()` if the change did affect the sort order.

↪ **While the first element child is a `select` or `datalist` element**

The default data provider must return 1 for the column count, the empty string for the column's caption, and must not add any classes to the column's classes.

For the rows, assume the existence of a node filter view of the descendants of the first element child of the `datagrid` element (the `select` or `datalist` element), that skips all nodes other than `optgroup` and `option` elements, as well as any descendants of any `option` elements.

Given a path *row*, the corresponding element is the one obtained by drilling into the

view, taking the child given by the path each time.

> Given the following XML markup:
>
> ```
> <datagrid>
>  <select>
>   <!-- the options and optgroups have had their labels and values re
>        to make the underlying structure clearer -->
>   <optgroup>
>    <option/>
>    <option/>
>   </optgroup>
>   <optgroup>
>    <option/>
>    <optgroup id="a">
>     <option/>
>     <option/>
>     <bogus/>
>     <option id="b"/>
>    </optgroup>
>    <option/>
>   </optgroup>
>  </select>
> </datagrid>
> ```
>
> The path "1,1,2" would select the element with ID "b". In the filtered view, the text nodes, comment nodes, and bogus elements are ignored; so for instance, the element with ID "a" (path "1,1") has only 3 child nodes in the view.

`getRowCount(`*`row`*`)` must drill through the view to find the element corresponding to the method's argument, and return the number of child nodes in the filtered view that the corresponding element has. (If the *row* is empty, the corresponding element is the `select` element at the root of the filtered view.)

`getChildAtPosition(`*`row, position`*`)` must return *position*. (The `select`/`datalist` default data provider does not support sorting the data grid.)

`getRowImage(`*`i`*`)` must return the empty string, `getRowMenu(`*`i`*`)` must return null.

`getRowClasses(`*`row, classes`*`)` must add the classes from the following list to *classes* when their condition is met:

- If the *row*'s corresponding element is an `optgroup` element: `header`
- If the *row*'s corresponding element contains other elements that are also in the view, and the element's `class` attribute contains the `closed` class: `initially-closed`
- If the *row*'s corresponding element contains other elements that are also in the view, and the element's `class` attribute contains the `open` class: `initially-open`

The `getCellData(`*`row, cell`*`)` method must return the value of the `label` attribute if the *row*'s corresponding element is an `optgroup` element, otherwise, if the *row*'s corresponding element is an `option`element, its `label` attribute if it has one, otherwise the value of its `textContent` DOM attribute.

The `getCellClasses(`*`row, cell, classes`*`)` method must add no classes.

autoselect some rows when initialized, reflect the selection in the select, reflect the multiple attribute somehow.

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate.

↪ **While the first element child is another element**

The default data provider must return 1 for the column count, the empty string for the column's caption, and must not add any classes to the column's classes.

For the rows, assume the existence of a node filter view of the descendants of the `datagrid` that skips all nodes other than `li`, `h1`–`h6`, and `hr` elements, and skips any descendants of `menu` elements.

Given this view, each element in the view represents a row in the data model. The element corresponding to a path *row* is the one obtained by drilling into the view, taking the child given by the path each time. The element of the row of a particular method call is the element given by drilling into the view along the path given by the method's arguments.

`getRowCount(row)` must return the number of child elements in this view for the given row, or the number of elements at the root of the view if the *row* is empty.

> In the following example, the elements are identified by the paths given by their child text nodes:
>
> ```
> <datagrid>
>  <ol>
>   <li> row 0 </li>
>   <li> row 1
>    <ol>
>     <li> row 1,0 </li>
>    </ol>
>   </li>
>   <li> row 2 </li>
>  </ol>
> </datagrid>
> ```
>
> In this example, only the `li` elements actually appear in the data grid; the `ol` element does not affect the data grid's processing model.

`getChildAtPosition(row, position)` must return *position*. (The generic default data provider does not support sorting the data grid.)

`getRowImage(i)` must return the absolute URL obtained from resolving the value of the `src` attribute of the first `img` element descendant (in the real DOM) of the row's element, that is not also a descendant of another element in the filtered view that is a descendant of the row's element, if such an element exists and resolving its attribute relative to the element itself, is successful. Otherwise, it must return the empty string.

> In the following example, the row with path "1,0" returns "http://example.com/a" as its image URL, and the other rows (including the row with path "1") return the empty string:
>
> ```
> <datagrid>
> ```

```
<ol>
 <li> row 0 </li>
 <li> row 1
  <ol>
   <li> row 1,0 <img src="http://example.com/a" alt=""> </li>
  </ol>
 </li>
 <li> row 2 </li>
</ol>
</datagrid>
```

`getRowMenu(i)` must return the first `menu` element descendant (in the real DOM) of the row's element, that is not also a descendant of another element in the filtered view that is a descendant of the row's element. (This is analogous to the image case above.)

`getRowClasses(i, classes)` must add the classes from the following list to *classes* when their condition is met:

- If the row's element contains other elements that are also in the view, and the element's `class` attribute contains the `closed` class: `initially-closed`
- If the row's element contains other elements that are also in the view, and the element's `class` attribute contains the `open` class: `initially-open`
- If the row's element is an `h1`–`h6` element: `header`
- If the row's element is an `hr` element: `separator`

The `getCellData(i, j)`, `getCellClasses(i, j, classes)`, `getCellCheckedState(i, j, state)`, `cycleCell(i, j)`, and `editCell(i, j, data)` methods must act as described in the common definitions below, treating the row's element as being the cell's element.

selection handling?

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate.

↪ **Otherwise, while there is no element child**

The data provider must return 0 for the number of rows, 1 for the number of columns, the empty string for the first column's caption, and must add no classes when asked for that column's classes. If the `datagrid`'s child list changes such that there is a first element child, then the data provider must call the `updateEverything()` method on the `datagrid`.

4.11.2.4.1 Common default data provider method definitions for cells

These definitions are used for the cell-specific methods of the default data providers (other than in the `select`/`datalist` case). How they behave is based on the contents of an element that represents the cell given by their first two arguments. Which element that is is defined in the previous section.

Cyclable cells

If the first element child of a cell's element is a `select` element that has a no `multiple` attribute and has at least one `option` element descendant, then the cell acts as a cyclable cell.

The "current" `option` element is the selected `option` element, or the first `option` element if none is selected.

The `getCellData()` method must return the `textContent` of the current `option` element (the `label` attribute is ignored in this context as the `optgroup`s are not displayed).

The `getCellClasses()` method must add the `cyclable` class and then all the classes of the current `option` element.

The `cycleCell()` method must change the selection of the `select` element such that the next `option` element after the current `option` element is the only one that is selected (in tree order). If the current `option` element is the last `option` element descendant of the `select`, then the first `option` element descendant must be selected instead.

The `setCellCheckedState()` and `editCell()` methods must do nothing.

Progress bar cells

If the first element child of a cell's element is a `progress` element, then the cell acts as a progress bar cell.

The `getCellData()` method must return the value returned by the `progress` element's `position` DOM attribute.

The `getCellClasses()` method must add the `progress` class.

The `setCellCheckedState()`, `cycleCell()`, and `editCell()` methods must do nothing.

Checkbox cells

If the first element child of a cell's element is an `input` element that has a `type` attribute with the value `checkbox`, then the cell acts as a check box cell.

The `getCellData()` method must return the `textContent` of the cell element.

The `getCellClasses()` method must add the `checked` class if the `input` element's checkedness is true, and the `unchecked` class otherwise.

The `setCellCheckedState()` method must set the `input` element's checkbox checkedness to true if the method's third argument is 1, and to false otherwise.

The `cycleCell()` and `editCell()` methods must do nothing.

Editable cells

If the first element child of a cell's element is an `input` element that has a `type` attribute with the value `text` or that has no `type` attribute at all, then the cell acts as an editable cell.

The `getCellData()` method must return the `value` of the `input` element.

The `getCellClasses()` method must add the `editable` class.

The `editCell()` method must set the `input` element's `value` DOM attribute to the value

of the third argument to the method.

The `setCellCheckedState()` and `cycleCell()` methods must do nothing.

4.11.2.5 Populating the `datagrid` element

A `datagrid` must be disabled until its end tag has been parsed (in the case of a `datagrid` element in the original document markup) or until it has been inserted into the document (in the case of a dynamically created element). After that point, the element must fire a single `load` event at itself, which doesn't bubble and cannot be canceled.

The end-tag parsing thing should be moved to the parsing section.

The `datagrid` must then populate itself using the data provided by the data provider assigned to the `data` DOM attribute. After the view is populated (using the methods described below), the `datagrid` must invoke the `initialize()` method on the data provider specified by the `data` attribute, passing itself (the `HTMLDataGridElement` object) as the only argument.

When the `data` attribute is null, the `datagrid` must use the default data provider described in the previous section.

To obtain data from the data provider, the element must invoke methods on the data provider object in the following ways:

To determine the total number of columns
: Invoke the `getColumnCount()` method with no arguments. The return value is the number of columns. If the return value is zero or negative, not an integer, or simply not a numeric type, or if the method is not defined, then 1 must be used instead.

To get the captions to use for the columns
: Invoke the `getCaptionText()` method with the index of the column in question. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of columns. The return value is the string to use when referring to that column. If the method returns null or the empty string, the column has no caption. If the method is not defined, then none of the columns have any captions.

To establish what classes apply to a column
: Invoke the `getCaptionClasses()` method with the index of the column in question, and an object implementing the `DOMTokenList` interface, associated with an anonymous empty string. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of columns. The tokens contained in the string underlying `DOMTokenList` object when the method returns represent the classes that apply to the given column. If the method is not defined, no classes apply to the column.

To establish whether a column should be initially included in the visible columns
: Check whether the `initially-hidden` class applies to the column. If it does, then the column should not be initially included; if it does not, then the column should be initially included.

To establish whether the data can be sorted relative to a particular column
: Check whether the `sortable` class applies to the column. If it does, then the user agent should offer the user the option to have the data displayed sorted by that column; if it does not, then the user agent must not allow the user to ask for the data to be sorted by that column.

To establish if a column is a sorted column

If the user agent can handle multiple columns being marked as sorted simultaneously: Check whether the `sorted` class applies to the column. If it does, then that column is the sorted column, otherwise it is not.

If the user agent can only handle one column being marked as sorted at a time: Check each column in turn, starting with the first one, to see whether the `sorted` class applies to that column. The first column that has that class, if any, is the sorted column. If none of the columns have that class, there is no sorted column.

To establish the sort direction of a sorted column

Check whether the `reversed` class applies to the column. If it does, then the sort direction is descending (down; first rows have the highest values), otherwise it is ascending (up; first rows have the lowest values).

To determine the total number of rows

Determine the number of rows for the root of the data grid, and determine the number of child rows for each open row. The total number of rows is the sum of all these numbers.

To determine the number of rows for the root of the data grid

Invoke the `getRowCount()` method with a `RowSpecification` object representing the empty path as its only argument. The return value is the number of rows at the top level of the data grid. If the return value of the method is negative, not an integer, or simply not a numeric type, or if the method is not defined, then zero must be used instead.

To determine the number of child rows for a row

Invoke the `getRowCount()` method with a `RowSpecification` object representing the path to the row in question. The return value is the number of child rows for the given row. If the return value of the method is negative, not an integer, or simply not a numeric type, or if the method is not defined, then zero must be used instead.

To determine what order to render rows in

Invoke the `getChildAtPosition()` method with a `RowSpecification` object representing the path to the parent of the rows that are being rendered as the first argument, and the position that is being rendered as the second argument. The return value is the index of the row to render in that position.

> If the rows are:
>
> 1. Row "0"
>   1. Row "0,0"
>   2. Row "0,1"
> 2. Row "1"
>   1. Row "1,0"
>   2. Row "1,1"
>
> ...and the `getChildAtPosition()` method is implemented as follows:
>
> ```
> function getChildAtPosition(parent, child) {
>   // always return the reverse order
>   return getRowCount(parent)-child-1;
> }
> ```
>
> ...then the rendering would actually be:
>
> 1. Row "1"

1. Row "1,1"
2. Row "1,0"

2. Row "0"
    1. Row "0,1"
    2. Row "0,0"

If the return value of the method is negative, larger than the number of rows that the `getRowCount()` method reported for that parent, not an integer, or simply not a numeric type, then the entire data grid should be disabled. Similarly, if the method returns the same value for two or more different values for the second argument (with the same first argument, and assuming that the data grid hasn't had relevant update methods invoked in the meantime), then the data grid should be disabled. Instead of disabling the data grid, the user agent may act as if the `getChildAtPosition()` method was not defined on the data provider (thus disabling sorting for that data grid, but still letting the user interact with the data). If the method is not defined, then the return value must be assumed to be the same as the second argument (an identity transform; the data is rendered in its natural order).

To establish what classes apply to a row

Invoke the `getRowClasses()` method with a `RowSpecification` object representing the row in question, and a `DOMTokenList` associated with an empty string. The tokens contained in the `DOMTokenList` object's underlying string when the method returns represent the classes that apply to the row in question. If the method is not defined, no classes apply to the row.

To establish whether a row is a data row or a special row

Examine the classes that apply to the row. If the `header` class applies to the row, then it is not a data row, it is a subheading. The data from the first cell of the row is the text of the subheading, the rest of the cells must be ignored. Otherwise, if the `separator` class applies to the row, then in the place of the row, a separator should be shown. Otherwise, if the `selectable-separator` class applies to the row, then the row should be a data row, but represented as a separator. (The difference between a `separator` and a `selectable-separator` is that the former is not an item that can be actually selected, whereas the second can be selected and thus has a context menu that applies to it, and so forth.) For both kinds of separator rows, the data of the rows' cells must all be ignored. If none of those three classes apply then the row is a simple data row.

To establish whether a row is openable

Determine the number of child rows for that row. If there are one or more child rows, then the row is openable.

To establish whether a row should be initially open or closed

If the row is openable, examine the classes that apply to the row. If the `initially-open` class applies to the row, then it should be initially open. Otherwise, if the `initially-closed` class applies to the row, then it must be initially closed. Otherwise, if neither class applies to the row, or if the row is not openable, then the initial state of the row should be based on platform conventions.

To obtain a URL identifying an image representing a row

Invoke the `getRowImage()` method with a `RowSpecification` object representing the row in question. The return value is a URL. Immediately resolve that URL, relative to the `datagrid` element, to obtain an absolute URL identifying the image that represents the row. If the method returns the empty string, null, or if the method is not defined, then the

row has no associated image.

To obtain a context menu appropriate for a particular row

Invoke the `getRowMenu()` method with a `RowSpecification` object representing the row in question. The return value is a reference to an object implementing the `HTMLMenuElement` interface, i.e. a `menu` element DOM node. (This element must then be interpreted as described in the section on context menus to obtain the actual context menu to use.) If the method returns something that is not an `HTMLMenuElement`, or if the method is not defined, then the row has no associated context menu. User agents may provide their own default context menu, and may add items to the author-provided context menu. For example, such a menu could allow the user to change the presentation of the `datagrid` element.

To establish the value of a particular cell

Invoke the `getCellData()` method with the first argument being a `RowSpecification` object representing the row of the cell in question and the second argument being the index of the cell's column. The second argument must be a non-negative integer less than the total number of columns. The return value is the value of the cell. If the return value is null or the empty string, or if the method is not defined, then the cell has no data. (For progress bar cells, the cell's value must be further interpreted, as described below.)

To establish what classes apply to a cell

Invoke the `getCellClasses()` method with the first argument being a `RowSpecification` object representing the row of the cell in question, the second argument being the index of the cell's column, and the third being an object implementing the `DOMTokenList` interface, associated with an empty string. The second argument must be a non-negative integer less than the total number of columns. The tokens contained in the `DOMTokenList` object's underlying string when the method returns represent the classes that apply to that cell. If the method is not defined, no classes apply to the cell.

To establish the type of a cell

Examine the classes that apply to the cell. If the `progress` class applies to the cell, it is a progress bar. Otherwise, if the `cyclable` class applies to the cell, it is a cycling cell whose value can be cycled between multiple states. Otherwise, none of these classes apply, and the cell is a simple text cell.

To establish the value of a progress bar cell

If the value *x* of the cell is a string that can be converted to a floating-point number in the range $0.0 \leq x \leq 1.0$, then the progress bar has that value (0.0 means no progress, 1.0 means complete). Otherwise, the progress bar is an indeterminate progress bar.

To establish how a simple text cell should be presented

Check whether one of the `checked`, `unchecked`, or `indeterminate` classes applies to the cell. If any of these are present, then the cell has a checkbox, otherwise none are present and the cell does not have a checkbox. If the cell has no checkbox, check whether the `editable` class applies to the cell. If it does, then the cell value is editable, otherwise the cell value is static.

To establish the state of a cell's checkbox, if it has one

Check whether the `checked` class applies to the cell. If it does, the cell is checked. Otherwise, check whether the `unchecked` class applies to the cell. If it does, the cell is unchecked. Otherwise, the `indeterminate` class applies to the cell and the cell's checkbox is in an indeterminate state. When the `indeterminate` class applies to the cell, the checkbox is a tristate checkbox, and the user can set it to the indeterminate state. Otherwise, only the `checked` and/or `unchecked` classes apply to the cell, and the cell can only be toggled between those two states.

If the data provider ever raises an exception while the `datagrid` is invoking one of its methods,

the `datagrid` must act, for the purposes of that particular method call, as if the relevant method had not been defined.

A `RowSpecification` object *p* with *n* path components passed to a method of the data provider must fulfill the constraint $0 \leq p_i < m\text{-}1$ for all integer values of *i* in the range $0 \leq i < n\text{-}1$, where *m* is the value that was last returned by the `getRowCount()` method when it was passed the `RowSpecification` object *q* with *i*-1 items, where $p_i = q_i$ for all integer values of *i* in the range $0 \leq i < n\text{-}1$, with any changes implied by the update methods taken into account.

The data model is considered stable: user agents may assume that subsequent calls to the data provider methods will return the same data, until one of the update methods is called on the `datagrid` element. If a user agent is returned inconsistent data, for example if the number of rows returned by `getRowCount()` varies in ways that do not match the calls made to the update methods, the user agent may disable the `datagrid`. User agents that do not disable the `datagrid` in inconsistent cases must honor the most recently returned values.

User agents may cache returned values so that the data provider is never asked for data that could contradict earlier data. User agents must not cache the return value of the `getRowMenu` method.

The exact algorithm used to populate the data grid is not defined here, since it will differ based on the presentation used. However, the behavior of user agents must be consistent with the descriptions above. For example, it would be non-conformant for a user agent to make cells have both a checkbox and be editable, as the descriptions above state that cells that have a checkbox cannot be edited.

4.11.2.6 Updating the `datagrid`

Whenever the `data` attribute is set to a new value, the `datagrid` must clear the current selection, remove all the displayed rows, and plan to repopulate itself using the information from the new data provider at the earliest opportunity.

There are a number of update methods that can be invoked on the `datagrid` element to cause it to refresh itself in slightly less drastic ways:

When the **`updateEverything()`** method is called, the user agent must repopulate the entire `datagrid`. If the number of rows decreased, the selection must be updated appropriately. If the number of rows increased, the new rows should be left unselected.

When the **`updateRowsChanged(`*`row, count`*`)`** method is called, the user agent must refresh the rendering of the rows starting from the row specified by *row*, and including the *count* next siblings of the row (or as many next siblings as it has, if that is less than *count*), including all descendant rows.

When the **`updateRowsInserted(`*`row, count`*`)`** method is called, the user agent must assume that *count* new rows have been inserted, such that the first new row is identified by *row*. The user agent must update its rendering and the selection accordingly. The new rows should not be selected.

When the **`updateRowsRemoved(`*`row, count`*`)`** method is called, the user agent must assume that *count* rows have been removed starting from the row that used to be identifier by *row*. The user agent must update its rendering and the selection accordingly.

The **updateRowChanged(*row*)** method must be exactly equivalent to calling updateRowsChanged(*row*, 1).

When the **updateColumnChanged(*column*)** method is called, the user agent must refresh the rendering of the specified column *column*, for all rows.

When the **updateCellChanged(*row*, *column*)** method is called, the user agent must refresh the rendering of the cell on row *row*, in column *column*.

Any effects the update methods have on the datagrid's selection is not considered a change to the selection, and must therefore not fire the select event.

These update methods should be called only by the data provider, or code acting on behalf of the data provider. In particular, calling the updateRowsInserted() and updateRowsRemoved() methods without actually inserting or removing rows from the data provider is likely to result in inconsistent renderings, and the user agent is likely to disable the data grid.

4.11.2.7 Requirements for interactive user agents

This section only applies to interactive user agents.

If the datagrid element has a **disabled** attribute, then the user agent must disable the datagrid, preventing the user from interacting with it. The datagrid element should still continue to update itself when the data provider signals changes to the data, though. Obviously, conformance requirements stating that datagrid elements must react to users in particular ways do not apply when one is disabled.

If a row is openable, then the user agent should offer to the user the option of toggling the row's open/closed state. When a row's open/closed state changes, the user agent must update the rendering to match the new state.

If a cell is a cell whose value can be cycled between multiple states, then the user agent should allow the user to activate the cell to cycle its value. When the user activates this "cycling" behavior of a cell, then the datagrid must invoke the data provider's cycleCell() method, with a RowSpecification object representing the cell's row as the first argument and the cell's column index as the second. The datagrid must then act as if the datagrid's updateCellChanged() method had been invoked with those same arguments.

When a cell has a checkbox, the user agent should allow the user to set the checkbox's state. When the user changes the state of a checkbox in such a cell, the datagrid must invoke the data provider's setCellCheckedState() method, with a RowSpecification object representing the cell's row as the first argument, the cell's column index as the second, and the checkbox's new state as the third. The state should be represented by the number 1 if the new state is checked, 0 if the new state is unchecked, and −1 if the new state is indeterminate (which must be possible only if the cell has the indeterminate class set). The datagrid must then act as if the datagrid's updateCellChanged() method had been invoked, specifying the same cell.

If a cell is editable, the user agent should allow the user to edit the data for that cell, and doing so must cause the user agent to invoke the editCell() method of the data provider with three arguments: a RowSpecification object representing the cell's row, the cell's column's index, and the new text entered by the user. The user agent must then act as if the updateCellChanged() method had been invoked, with the same row and column specified.

4.11.2.8 The selection

This section only applies to interactive user agents. For other user agents, the `selection` attribute must return null.

```
interface DataGridSelection {
  readonly attribute unsigned long length;
  [IndexGetter] RowSpecification item(in unsigned long index);
  boolean isSelected(in RowSpecification row);
  void setSelected(in RowSpecification row, in boolean selected);

  void selectAll();
  void invert();
  void clear();
};
```

Each `datagrid` element must keep track of which rows are currently selected. Initially no rows are selected, but this can be changed via the methods described in this section.

The selection of a `datagrid` is represented by its **`selection`** DOM attribute, which must be a `DataGridSelection` object.

`DataGridSelection` objects represent the rows in the selection. In the selection the rows must be ordered in the natural order of the data provider (and not, e.g., the rendered order). Rows that are not rendered because one of their ancestors is closed must share the same selection state as their nearest rendered ancestor. Such rows are not considered part of the selection for the purposes of iterating over the selection.

> ***Note: This selection API doesn't allow for hidden rows to be selected because it is trivial to create a data provider that has infinite depth, which would then require the selection to be infinite if every row, including every hidden row, was selected.***

The **`length`** attribute must return the number of rows currently present in the selection. This is the *length*.

The object's indices of the supported indexed properties are the numbers in the range zero to *length*-1, unless the *length* is zero, in which case there are no supported indexed properties.

The **`item(`*`index`*`)`** method must return the *index*th row in the selection. If the argument is out of range (less than zero or greater than the number of selected rows minus one), then it must raise an `INDEX_SIZE_ERR` exception. [DOM3CORE]

The **`isSelected()`** method must return the selected state of the row specified by its argument. If the specified row exists and is selected, it must return true, otherwise it must return false.

The **`setSelected()`** method takes two arguments, *row* and *selected*. When invoked, it must set the selection state of row *row* to selected if *selected* is true, and unselected if it is false. If *row* is not a row in the data grid, the method must raise an `INDEX_SIZE_ERR` exception. If the specified row is not rendered because one of its ancestors is closed, the method must do nothing.

The **`selectAll()`** method must mark all the rows in the data grid as selected. After a call to `selectAll()`, the `length` attribute will return the number of rows in the data grid, not counting

children of closed rows.

The **invert()** method must cause all the rows in the selection that were marked as selected to now be marked as not selected, and vice versa.

The **clear()** method must mark all the rows in the data grid to be marked as not selected. After a call to clear(), the length attribute will return zero.

If the datagrid element has a **multiple** attribute, then the user agent should allow the user to select any number of rows (zero or more). If the attribute is not present, then the user agent must not allow the user to select more than a single row at a time, and selecting another one must unselect all the other rows.

> ***Note: This only applies to the user. Scripts can select multiple rows even when the multiple attribute is absent.***

Whenever the selection of a datagrid changes, whether due to the user interacting with the element, or as a result of calls to methods of the selection object, a **select** event that bubbles but is not cancelable must be fired on the datagrid element. If changes are made to the selection via calls to the object's methods during the execution of a script, then the select events must be coalesced into one, which must then be fired when the script execution has completed.

> ***Note: The DataGridSelection interface has no relation to the Selection interface.***

4.11.2.9 Columns and captions

This section only applies to interactive user agents.

Each datagrid element must keep track of which columns are currently being rendered. User agents should initially show all the columns except those with the initially-hidden class, but may allow users to hide or show columns. User agents should initially display the columns in the order given by the data provider, but may allow this order to be changed by the user.

If columns are not being used, as might be the case if the data grid is being presented in an icon view, or if an overview of data is being read in an aural context, then the text of the first column of each row should be used to represent the row.

If none of the columns have any captions (i.e. if the data provider does not provide a getCaptionText() method), then user agents may avoid showing the column headers at all. This may prevent the user from performing actions on the columns (such as reordering them, changing the sort column, and so on).

> ***Note: Whatever the order used for rendering, and irrespective of what columns are being shown or hidden, the "first column" as referred to in this specification is always the column with index zero, and the "last column" is always the column with the index one less than the value returned by the getColumnCount() method of the data provider.***

If a column is sortable, then the user agent should allow the user to request that the data be sorted using that column. When the user does so, then the `datagrid` must invoke the data provider's `toggleColumnSortState()` method, with the column's index as the only argument. The `datagrid` must *then* act as if the `datagrid`'s `updateEverything()` method had been invoked.

4.11.3 The `command` element

Categories

Metadata content.
Flow content.
Phrasing content.

Contexts in which this element may be used:

Where metadata content is expected.
Where phrasing content is expected.

Content model:

Empty.

Content attributes:

Global attributes
`type`
`label`
`icon`
`disabled`
`checked`
`radiogroup`
`default`

Also, the `title` attribute has special semantics on this element.

DOM interface:

```
interface HTMLCommandElement : HTMLElement {
           attribute DOMString type;
           attribute DOMString label;
           attribute DOMString icon;
           attribute boolean disabled;
           attribute boolean checked;
           attribute DOMString radiogroup;
           attribute boolean default;
 void click(); // shadows HTMLElement.click()
};
```

The `Command` interface must also be implemented by this element.

The `command` element represents a command that the user can invoke.

The **`type`** attribute indicates the kind of command: either a normal command with an associated action, or a state or option that can be toggled, or a selection of one item from a list of items.

The attribute is an enumerated attribute with three keywords and states. The keyword "**`command`**" maps to the Command state, the **`checkbox`**" maps to the Checkbox, and the "**`radio`**" keyword maps to the Radio state. The *missing value default* is the Command state.

The *Command* state
The element represents a normal command with an associated action.

The *Checkbox* state
The element represents a state or option that can be toggled.

The *Radio* state
The element represents a selection of one item from a list of items.

The **`label`** attribute gives the name of the command, as shown to the user.

The **`title`** attribute gives a hint describing the command, which might be shown to the user to help him.

The **`icon`** attribute gives a picture that represents the command. If the attribute is specified, the attribute's value must contain a valid URL. To obtain the absolute URL of the icon, the attribute's value must be resolved relative to the element.

The **`disabled`** attribute is a boolean attribute that, if present, indicates that the command is not available in the current state.

> ***Note: The distinction between `disabled` and `hidden` is subtle. A command would be disabled if, in the same context, it could be enabled if only certain aspects of the situation were changed. A command would be marked as hidden if, in that situation, the command will never be enabled. For example, in the context menu for a water faucet, the command "open" might be disabled if the faucet is already open, but the command "eat" would be marked hidden since the faucet could never be eaten.***

The **`checked`** attribute is a boolean attribute that, if present, indicates that the command is selected. The attribute must be omitted unless the `type` attribute is in either the Checkbox state or the Radio state.

The **`radiogroup`** attribute gives the name of the group of commands that will be toggled when the command itself is toggled, for commands whose `type` attribute has the value "`radio`". The scope of the name is the child list of the parent element. The attribute must be omitted unless the `type` attribute is in the Radio state.

If the `command` element is used when generating a context menu, then the **`default`** attribute indicates, if present, that the command is the one that would have been invoked if the user had directly activated the menu's subject instead of using its context menu. The `default` attribute is a boolean attribute. The attribute must be omitted unless the `type` attribute is in the Command state.

> Need an example that shows an element that, if double-clicked, invokes an action, but that also has a context menu, showing the various `command` attributes off, and that has a default command.

The **`type`**, **`label`**, **`icon`**, **`disabled`**, **`checked`**, **`radiogroup`**, and **`default`** DOM attributes must reflect the respective content attributes of the same name.

The **`click()`** method's behavior depends on the value of the `type` attribute of the element, as

follows:

↪ **If the `type` attribute is in the Checkbox state**
If the element has a `checked` attribute, the UA must remove that attribute. Otherwise, the UA must add a `checked` attribute, with the literal value `checked`. The UA must then fire a `click` event at the element.

↪ **If the `type` attribute is in the Radio state**
If the element has a parent, then the UA must walk the list of child nodes of that parent element, and for each node that is a `command` element, if that element has a `radiogroup` attribute whose value exactly matches the current element's (treating missing `radiogroup` attributes as if they were the empty string), and has a `checked` attribute, must remove that attribute and fire a `click` event at the element.

Then, the element's `checked` attribute attribute must be set to the literal value `checked` and a `click` event must be fired at the element.

↪ **Otherwise**
The UA must fire a `click` event at the element.

Note: Firing a synthetic `click` event at the element does not cause any of the actions described above to happen.

Note: `command` elements are not rendered unless they form part of a menu.

4.11.4 The `bb` element

Categories
Flow content.
Phrasing content.
Interactive content.
Contexts in which this element may be used:
Where phrasing content is expected.
Content model:
Phrasing content, but there must be no interactive content descendant.
Content attributes:
Global attributes
`type`
DOM interface:

```
interface HTMLBrowserButtonElement : HTMLElement {
           attribute DOMString type;
  readonly attribute boolean supported;
  readonly attribute boolean disabled;
};
```

The `Command` interface must also be implemented by this element.

The `bb` element represents a user agent command that the user can invoke.

The **type** attribute indicates the kind of command. The type attribute is an enumerated attribute. The following table lists the keywords and states for the attribute — the keywords in the left column map to the states listed in the cell in the second column on the same row as the keyword.

Keyword	State
makeapp	*make application*

The *missing value default* state is the *null* state.

Each state has an *action* and a *relevance*, defined in the following sections.

When the attribute is in the ***null*** state, the *action* is to not do anything, and the *relevance* is unconditionally false.

A bb element whose type attribute is in a state whose *relevance* is true must be enabled. Conversely, a bb element whose type attribute is in a state whose *relevance* is false must be disabled.

> ***Note: If a bb element is enabled, it will match the :enabled pseudo-class; otherwise, it will match the :disabled pseudo-class.***

User agents should allow users to invoke bb elements when they are enabled. When a user invokes a bb element, its type attribute's state's *action* must be invoked.

When the element has no descendant element children and has no descendant text node children of non-zero length, the element represents a browser button with a user-agent-defined icon or text representing the type attribute's state's *action* and *relevance* (enabled vs disabled). Otherwise, the element represents its children.

bb **.supported**

Returns true if the value in the type attribute is a value that the user agent supports. Otherwise, returns false.

bb **.disabled**

Returns false if the user can invoke the element's *action* (i.e. if the element's *relevance* is true). Otherwise, returns true.

The **type** DOM attribute must reflect the content attribute of the same name.

The **supported** DOM attribute must return true if the type attribute is in a state other than the *null* state and the user agent supports that state's *action* (i.e. when the attribute's value is one that the user agent recognises and supports), and false otherwise.

The **disabled** DOM attribute must return true if the element is disabled, and false otherwise (i.e. it returns the opposite of the type attribute's state's *relevance*).

4.11.4.1 Browser button types

4.11.4.1.1 THE *MAKE APPLICATION* STATE

Some user agents support making sites accessible as independent applications, as if they were not Web sites at all. The *make application* state exists to allow Web pages to offer themselves to the user as targets for this mode of operation.

The *action* of the *make application* state is to confirm the user's intent to use the current site in a standalone fashion, and, provided the user's intent is confirmed, offer the user a way to make the resource identified by the document's address available in such a fashion.

?Warning! ***The confirmation is needed because it is relatively easy to trick users into activating buttons. The confirmation could, e.g. take the form of asking the user where to "save" the application, or non-modal information panel that is clearly from the user agent and gives the user the opportunity to drag an icon to their system's application launcher.***

The *relevance* of the *make application* state is false if the user agent is already handling the site in such a fashion, or if the user agent doesn't support making the site available in that fashion, and true otherwise.

In the following example, a few links are listed on an application's page, to allow the user perform certain actions, including making the application standalone:

```
<menu>
 <li><a href="settings.html" onclick="panels.show('settings')">Settings</a>
 <li><bb type="makeapp">Download standalone application</bb>
 <li><a href="help.html" onclick="panels.show('help')">Help</a>
 <li><a href="logout.html" onclick="panels.show('logout')">Sign out</a>
</menu>
```

With the following stylesheet, it could be make to look like a single line of text with vertical bars separating the options, with the "make app" option disappearing when it's not supported or relevant:

```
menu li { display: none; }
menu li:enabled { display: inline; }
menu li:not(:first-child)::before { content: ' | '; }
```

This could look like this:

Settings | Download standalone application | Help | Sign out

The following example shows another way to do the same thing as the previous one, this time not relying on CSS support to hide the "make app" link if it doesn't apply:

```
<menu>
 <a href="settings.html" onclick="panels.show('settings')">Settings</a> |
 <bb type="makeapp" id="makeapp"> </bb>
 <a href="help.html" onclick="panels.show('help')">Help</a> |
 <a href="logout.html" onclick="panels.show('logout')">Sign out</a>
</menu>
<script>
 var bb = document.getElementById('makeapp');
```

```
      if (bb.supported && bb.enabled) {
        bb.parentNode.nextSibling.textContent = ' | ';
        bb.textContent = 'Download standalone application';
      } else {
        bb.parentNode.removeChild(bb);
      }
    </script>
```

4.11.5 The `menu` element

Categories

Flow content.

If the element's `type` attribute is in the tool bar state: Interactive content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Either: Zero or more `li` elements.

Or: Flow content.

Content attributes:

Global attributes

`type`

`label`

DOM interface:

```
interface HTMLMenuElement : HTMLElement {
           attribute DOMString type;
           attribute DOMString label;
};
```

The `menu` element represents a list of commands.

The **`type`** attribute is an enumerated attribute indicating the kind of menu being declared. The attribute has three states. The `context` keyword maps to the **context menu** state, in which the element is declaring a context menu. The `toolbar` keyword maps to the **tool bar** state, in which the element is declaring a tool bar. The attribute may also be omitted. The *missing value default* is the **list** state, which indicates that the element is merely a list of commands that is neither declaring a context menu nor defining a tool bar.

If a `menu` element's `type` attribute is in the context menu state, then the element represents the commands of a context menu, and the user can only interact with the commands if that context menu is activated.

If a `menu` element's `type` attribute is in the tool bar state, then the element represents a list of active commands that the user can immediately interact with.

If a `menu` element's `type` attribute is in the list state, then the element either represents an unordered list of items (each represented by an `li` element), each of which represents a command that the user can perform or activate, or, if the element has no `li` element children, flow content describing available commands.

The **`label`** attribute gives the label of the menu. It is used by user agents to display nested menus in the UI. For example, a context menu containing another menu would use the nested menu's `label` attribute for the submenu's menu label.

The `type` and `label` DOM attributes must reflect the respective content attributes of the same name.

4.11.5.1 Introduction

This section is non-normative.

...

4.11.5.2 Building menus and tool bars

A menu (or tool bar) consists of a list of zero or more of the following components:

- Commands, which can be marked as default commands
- Separators
- Other menus (which allows the list to be nested)

The list corresponding to a particular `menu` element is built by iterating over its child nodes. For each child node in tree order, the required behavior depends on what the node is, as follows:

↪ **An element that defines a command**
Append the command to the menu, respecting its facets. If the element is a `command` element with a `default` attribute, mark the command as being a default command.

↪ **An `hr` element**

↪ **An `option` element that has a `value` attribute set to the empty string, and has a `disabled` attribute, and whose `textContent` consists of a string of one or more hyphens (U+002D HYPHEN-MINUS)**
Append a separator to the menu.

↪ **An `li` element**
Iterate over the children of the `li` element.

↪ **A `menu` element with no `label` attribute**

↪ **A `select` element**
Append a separator to the menu, then iterate over the children of the `menu` or `select` element, then append another separator.

↪ **A `menu` element with a `label` attribute**

↪ **An `optgroup` element**
Append a submenu to the menu, using the value of the element's `label` attribute as the label of the menu. The submenu must be constructed by taking the element and creating a new menu for it using the complete process described in this section.

↪ **Any other node**
Ignore the node.

We should support `label` in the algorithm above -- just iterate through the contents like with `li`, to support `input` elements in `label` elements. Also, `optgroup` elements without labels should be ignored (maybe? or at least should say they have no label so that they are dropped below), and `select` elements inside `label` elements may need special processing.

Once all the nodes have been processed as described above, the user agent must the post-process the menu as follows:

1. Any menu item with no label, or whose label is the empty string, must be removed.
2. Any sequence of two or more separators in a row must be collapsed to a single separator.
3. Any separator at the start or end of the menu must be removed.

4.11.5.3 Context menus

The **`contextmenu`** attribute gives the element's context menu. The value must be the ID of a `menu` element in the DOM. If the node that would be obtained by the invoking the `getElementById()` method using the attribute's value as the only argument is null or not a `menu` element, then the element has no assigned context menu. Otherwise, the element's assigned context menu is the element so identified.

When an element's context menu is requested (e.g. by the user right-clicking the element, or pressing a context menu key), the UA must fire a `contextmenu` event on the element for which the menu was requested.

> ***Note: Typically, therefore, the firing of the `contextmenu` event will be the default action of a `mouseup` or `keyup` event. The exact sequence of events is UA-dependent, as it will vary based on platform conventions.***

The default action of the `contextmenu` event depends on whether the element has a context menu assigned (using the `contextmenu` attribute) or not. If it does not, the default action must be for the user agent to show its default context menu, if it has one.

Context menus should inherit (so clicking on a span in a paragraph with a context menu should show the menu).

If the element *does* have a context menu assigned, then the user agent must fire a `show` event on the relevant `menu` element.

The default action of *this* event is that the user agent must show a context menu built from the `menu` element.

The user agent may also provide access to its default context menu, if any, with the context menu shown. For example, it could merge the menu items from the two menus together, or provide the page's context menu as a submenu of the default menu.

If the user dismisses the menu without making a selection, nothing in particular happens.

If the user selects a menu item that represents a command, then the UA must invoke that command's Action.

Context menus must not, while being shown, reflect changes in the DOM; they are constructed as the default action of the `show` event and must remain like that until dismissed.

User agents may provide means for bypassing the context menu processing model, ensuring that the user can always access the UA's default context menus. For example, the user agent could handle right-clicks that have the Shift key depressed in such a way that it does not fire the `contextmenu` event and instead always shows the default context menu.

The **`contextMenu`** attribute must reflect the `contextmenu` content attribute.

4.11.5.4 Tool bars

When a `menu` element has a `type` attribute in the tool bar state, then the user agent must build the menu for that `menu` element, and use the result in the rendering.

The user agent must reflect changes made to the `menu`'s DOM, by immediately rebuilding the menu.

4.11.6 Commands

A **command** is the abstraction behind menu items, buttons, and links. Once a command is defined, other parts of the interface can refer to the same command, allowing many access points to a single feature to share aspects such as the disabled state.

Commands are defined to have the following **facets**:

Type
: The kind of command: "command", meaning it is a normal command; "radio", meaning that triggering the command will, amongst other things, set the Checked State to true (and probably uncheck some other commands); or "checkbox", meaning that triggering the command will, amongst other things, toggle the value of the Checked State.

ID
: The name of the command, for referring to the command from the markup or from script. If a command has no ID, it is an **anonymous command**.

Label
: The name of the command as seen by the user.

Hint
: A helpful or descriptive string that can be shown to the user.

Icon
: An absolute URL identifying a graphical image that represents the action. A command might not have an Icon.

Hidden State
: Whether the command is hidden or not (basically, whether it should be shown in menus).

Disabled State
: Whether the command is relevant and can be triggered or not.

Checked State
: Whether the command is checked or not.

Action
: The actual effect that triggering the command will have. This could be a scripted event handler, a URL to which to navigate, or a form submission.

Commands are represented by elements in the DOM. Any element that can define a

command also implements the `Command` interface:

Actually even better would be to just mix it straight into those interfaces somehow.

```
[NoInterfaceObject] interface Command {
  readonly attribute DOMString commandType;
  readonly attribute DOMString id;
  readonly attribute DOMString label;
  readonly attribute DOMString title;
  readonly attribute DOMString icon;
  readonly attribute boolean hidden;
  readonly attribute boolean disabled;
  readonly attribute boolean checked;
  void click();
};
```

The `Command` interface must be implemented by any element capable of defining a command. All the attributes of the `Command` interface are read-only. Elements implementing this interface might implement other interfaces that have attributes with identical names but that are mutable; in bindings that flatten all supported interfaces on the object, the mutable attributes must shadow the readonly attributes defined in the `Command` interface.

***element*.commandType**

Exposes the Type facet of the command.

***element*.id**

Exposes the ID facet of the command.

***element*.label**

Exposes the Label facet of the command.

***element*.title**

Exposes the Hint facet of the command.

***element*.icon**

Exposes the Icon facet of the command.

***element*.hidden**

Exposes the Hidden State facet of the command.

***element*.disabled**

Exposes the Disabled State facet of the command.

***element*.checked**

Exposes the Checked State facet of the command.

***element*.click**

Triggers the Action of the command.

The **commandType** attribute must return a string whose value is either "command", "radio", or "checked", depending on whether the Type of the command defined by the element is "command", "radio", or "checked" respectively. If the element does not define a command, it must return null.

The **id** attribute must return the command's ID, or null if the element does not define a command or defines an anonymous command. This attribute will be shadowed by the id DOM attribute on the HTMLElement interface.

The **label** attribute must return the command's Label, or null if the element does not define a command or does not specify a Label. This attribute will be shadowed by the label DOM attribute on option and command elements.

The **title** attribute must return the command's Hint, or null if the element does not define a command or does not specify a Hint. This attribute will be shadowed by the title DOM attribute on the HTMLElement interface.

The **icon** attribute must return the absolute URL of the command's Icon. If the element does not specify an icon, or if the element does not define a command, then the attribute must return null. This attribute will be shadowed by the icon DOM attribute on command elements.

The **hidden** attribute must return true if the command's Hidden State is that the command is hidden, and false if it is that the command is not hidden. If the element does not define a command, the attribute must return false. This attribute will be shadowed by the hidden DOM attribute on the HTMLElement interface.

The **disabled** attribute must return true if the command's Disabled State is that the command is disabled, and false if the command is not disabled. This attribute is not affected by the command's Hidden State. If the element does not define a command, the attribute must return false. This attribute will be shadowed by the disabled attribute on button, input, option, and command elements.

The **checked** attribute must return true if the command's Checked State is that the command is checked, and false if it is that the command is not checked. If the element does not define a command, the attribute must return false. This attribute will be shadowed by the checked attribute on input and command elements.

The **click()** method must trigger the Action for the command. If the element does not define a command, this method must do nothing. This method will be shadowed by the click() method on HTML elements, and is included only for completeness.

The **commands** attribute of the document's HTMLDocument interface must return an HTMLCollection rooted at the Document node, whose filter matches only elements that define commands and have IDs.

The following elements can define commands: a, button, input, option, command, bb.

4.11.6.1 Using the a element to define a command

An a element with an href attribute defines a command.

The Type of the command is "command".

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the string given by the element's `textContent` DOM attribute.

The Hint of the command is the value of the `title` attribute of the element. If the attribute is not present, the Hint is the empty string.

The Icon of the command is the absolute URL obtained from resolving the value of the `src` attribute of the first `img` element descendant of the element, relative to that element, if there is such an element and resolving its attribute is successful. Otherwise, there is no Icon for the command.

The Hidden State of the command is true (hidden) if the element has a `hidden` attribute, and false otherwise.

The Disabled State facet of the command is always false. (The command is always enabled.)

The Checked State of the command is always false. (The command is never checked.)

The Action of the command is to fire a `click` event at the element.

4.11.6.2 Using the `button` element to define a command

A `button` element always defines a command.

The Type, ID, Label, Hint, Icon, Hidden State, Checked State, and Action facets of the command are determined as for `a` elements (see the previous section).

The Disabled State of the command mirrors the disabled state of the button.

4.11.6.3 Using the `input` element to define a command

An `input` element whose `type` attribute is in one of the Submit Button, Reset Button, Button, Radio Button, or Checkbox states defines a command.

The Type of the command is "radio" if the `type` attribute is in the `Radio Button` state, "checkbox" if the `type` attribute is in the `Checkbox` state, and "command" otherwise.

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command depends on the Type of the command:

If the Type is "command", then it is the string given by the `value` attribute, if any, and a UA-dependent value that the UA uses to label the button itself if the attribute is absent.

Otherwise, the Type is "radio" or "checkbox". If the element is a labeled control, the `textContent` of the first `label` element in tree order whose labeled control is the element in question is the Label (in DOM terms, this the string given by

element.labels[0].textContent). Otherwise, the value of the value attribute, if present, is the Label. Otherwise, the Label is the empty string.

The Hint of the command is the value of the title attribute of the input element. If the attribute is not present, the Hint is the empty string.

There is no Icon for the command.

The Hidden State of the command is true (hidden) if the element has a hidden attribute, and false otherwise.

The Disabled State of the command mirrors the disabled state of the control.

The Checked State of the command is true if the command is of Type "radio" or "checkbox" and the element is checked attribute, and false otherwise.

The Action of the command is to fire a click event at the element.

4.11.6.4 Using the option element to define a command

An option element with an ancestor select element and either no value attribute or a value attribute that is not the empty string defines a command.

The Type of the command is "radio" if the option's nearest ancestor select element has no multiple attribute, and "checkbox" if it does.

The ID of the command is the value of the id attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the value of the option element's label attribute, if there is one, or the value of the option element's textContent DOM attribute if there isn't.

The Hint of the command is the string given by the element's title attribute, if any, and the empty string if the attribute is absent.

There is no Icon for the command.

The Hidden State of the command is true (hidden) if the element has a hidden attribute, and false otherwise.

The Disabled State of the command is true (disabled) if the element is disabled or if its nearest ancestor select element is disabled, and false otherwise.

The Checked State of the command is true (checked) if the element's selectedness is true, and false otherwise.

The Action of the command depends on its Type. If the command is of Type "radio" then it must pick the option element. Otherwise, it must toggle the option element.

4.11.6.5 Using the command element to define a command

A command element defines a command.

The Type of the command is "radio" if the `command`'s `type` attribute is "`radio`", "checkbox" if the attribute's value is "`checkbox`", and "command" otherwise.

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the value of the element's `label` attribute, if there is one, or the empty string if it doesn't.

The Hint of the command is the string given by the element's `title` attribute, if any, and the empty string if the attribute is absent.

The Icon for the command is the absolute URL obtained from resolving the value of the element's `icon` attribute, relative to the element, if it has such an attribute and resolving it is successful. Otherwise, there is no Icon for the command.

The Hidden State of the command is true (hidden) if the element has a `hidden` attribute, and false otherwise.

The Disabled State of the command is true (disabled) if the element has a `disabled` attribute, and false otherwise.

The Checked State of the command is true (checked) if the element has a `checked` attribute, and false otherwise.

The Action of the command is to invoke the behavior described in the definition of the `click()` method of the `HTMLCommandElement` interface.

4.11.6.6 Using the `bb` element to define a command

A `bb` element always defines a command.

The Type of the command is "command".

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the string given by the element's `textContent` DOM attribute, if that is not the empty string, or a user-agent-defined string appropriate for the `bb` element's `type` attribute's state.

The Hint of the command is the value of the `title` attribute of the element. If the attribute is not present, the Hint is a user-agent-defined string appropriate for the `bb` element's `type` attribute's state.

The Icon of the command is the absolute URL obtained from resolving the value of the `src` attribute of the first `img` element descendant of the element, relative to that element, if there is such an element and resolving its attribute is successful. Otherwise, the Icon is a user-agent-defined image appropriate for the `bb` element's `type` attribute's state.

The Hidden State facet of the command is true (hidden) if the `bb` element's `type` attribute's state is *null* or if the element has a `hidden` attribute, and false otherwise.

The Disabled State facet of the command is true if the `bb` element's `type` attribute's state's *relevance* is false, and true otherwise.

The Checked State of the command is always false. (The command is never checked.)

The Action of the command is to perform the *action* of the `bb` element's `type` attribute's state.

4.12 Miscellaneous elements

4.12.1 The `legend` element

Categories
None.
Contexts in which this element may be used:
As the first child of a `fieldset` element.
As the first child of a `details` element.
As a child of a `figure` element, if there are no other `legend` element children of that element.
Content model:
Phrasing content.
Content attributes:
Global attributes
DOM interface:

```
interface HTMLLegendElement : HTMLElement {
  readonly attribute HTMLFormElement form;
};
```

The `legend` element represents a title or explanatory caption for the rest of the contents of the `legend` element's parent element.

***legend* . `form`**

Returns the element's `form` element, if any, or null otherwise.

The **`form`** DOM attribute's behavior depends on whether the `legend` element is in a `fieldset` element or not. If the `legend` has a `fieldset` element as its parent, then the `form` DOM attribute must return the same value as the `form` DOM attribute on that `fieldset` element. Otherwise, it must return null.

4.12.2 The `div` element

Categories
Flow content.
Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Content attributes:

Global attributes

DOM interface:

Uses `HTMLElement`.

The `div` element has no special meaning at all. It represents its children. It can be used with the `class`, `lang`/`xml:lang`, and `title` attributes to mark up semantics common to a group of consecutive elements.

Allowing `div` elements to contain phrasing content makes it easy for authors to abuse `div`, using it with the `class=""` attribute to the point of not having any other elements in the markup. This is a disaster from an accessibility point of view, and it would be nice if we could somehow make such pages non-compliant without preventing people from using `div`s as the extension mechanism that they are, to handle things the spec can't otherwise do (like making new widgets).

4.13 Matching HTML elements using selectors

There are a number of dynamic selectors that can be used with HTML. This section defines when these selectors match HTML elements.

:link

:visited

All `a` elements that have an `href` attribute, all `area` elements that have an `href` attribute, and all `link` elements that have an `href` attribute, must match one of `:link` and `:visited`.

:active

The `:active` pseudo-class must match the following elements between the time the user begins to activate the element and the time the users stops activating the element:

- `a` elements that have an `href` attribute
- `area` elements that have an `href` attribute
- `link` elements that have an `href` attribute
- `bb` elements whose `type` attribute is in a state whose *relevance* is true
- `button` elements that are not disabled
- `input` elements whose `type` attribute is in the Submit Button, Image Button, Reset Button, or Button state
- `command` elements that do not have a `disabled` attribute
- any other element, if it is specially focusable

> For example, if the user is using a keyboard to push a `button` element by pressing the space bar, the element would match this pseudo-class in between the time that the element received the `keydown` event and the time the element received the `keyup` event.

:enabled

The `:enabled` pseudo-class must match the following elements:

- `a` elements that have an `href` attribute
- `area` elements that have an `href` attribute
- `link` elements that have an `href` attribute
- `bb` elements whose `type` attribute is in a state whose *relevance* is true
- `button` elements that are not disabled
- `input` elements whose `type` attribute are not in the Hidden state and that are not disabled
- `select` elements that are not disabled
- `textarea` elements that are not disabled
- `option` elements that do not have a `disabled` attribute
- `command` elements that do not have a `disabled` attribute
- `li` elements that are children of `menu` elements, and that have a child element that defines a command, if the first such element's Disabled State facet is false (not disabled)
- any other element, if it is specially focusable

:disabled

The `:disabled` pseudo-class must match the following elements:

- `bb` elements whose `type` attribute is in a state whose *relevance* is false
- `button` elements that are disabled
- `input` elements whose `type` attribute are not in the Hidden state and that are disabled
- `select` elements that are disabled
- `textarea` elements that are disabled
- `option` elements that have a `disabled` attribute
- `command` elements that have a `disabled` attribute
- `li` elements that are children of `menu` elements, and that have a child element that defines a command, if the first such element's Disabled State facet is true

(disabled)

:checked

The `:checked` pseudo-class must match the following elements:

- `input` elements whose `type` attribute is in the Checkbox state and whose checkedness state is true
- `input` elements whose `type` attribute is in the Radio Button state and whose checkedness state is true
- `command` elements whose `type` attribute is in the Checkbox state and that have a `checked` attribute
- `command` elements whose `type` attribute is in the Radio state and that have a `checked` attribute

:indeterminate

The `:indeterminate` pseudo-class must match `input` elements whose `type` attribute is in the Checkbox state and whose `indeterminate` DOM attribute is set to true.

:default

The `:default` pseudo-class must match the following elements:

- `button` elements that are their form's default button
- `input` elements whose `type` attribute is in the Submit Button or Image Button state, and that are their form's default button
- `command` elements that have a `default` attribute

:valid

The `:valid` pseudo-class must match all elements that are candidates for constraint validation and that satisfy their constraints.

:invalid

The `:invalid` pseudo-class must match all elements that are candidates for constraint validation but that do not satisfy their constraints.

:in-range

The `:in-range` pseudo-class must match all elements that are candidates for constraint validation and that are neither suffering from an underflow nor suffering from an overflow.

:out-of-range

The `:out-of-range` pseudo-class must match all elements that are candidates for constraint validation and that are suffering from an underflow or suffering from an overflow.

:required

The `:required` pseudo-class must match the following elements:

- `input` elements that are *required*

- `textarea` elements that have a `required` attribute

`:optional`

The `:optional` pseudo-class must match the following elements:

- `button` elements
- `input` elements that are not ***required***
- `select` elements
- `textarea` elements that do not have a `required` attribute

`:read-only`

`:read-write`

The `:read-write` pseudo-class must match the following elements:

- `input` elements to which the `readonly` attribute applies, but that are not ***immutable*** (i.e. that do not have the `readonly` attribute specified and that are not disabled)
- `textarea` elements that do not have a `readonly` attribute, and that are not disabled
- any element that is editable

The `:read-only` pseudo-class must match all other HTML elements.

Note: Another section of this specification defines the target element used with the `:target` pseudo-class.

Note: This specification does not define when an element matches the `:hover`, `:focus`, or `:lang()` dynamic pseudo-classes, as those are all defined in sufficient detail in a language-agnostic fashion in the Selectors specification. [SELECTORS]

5 Web browsers

This section describes features that apply most directly to Web browsers. Having said that, unless specified elsewhere, the requirements defined in this section *do* apply to all user agents, whether they are Web browsers or not.

5.1 Browsing contexts

A **browsing context** is an environment in which `Document` objects are presented to the user.

> ***Note: A tab or window in a Web browser typically contains a browsing context, as does an `iframe` or `frame`s in a `frameset`.***

Each browsing context has a corresponding `WindowProxy` object.

The collection of `Document`s is the browsing context's session history. At any time, one `Document` in each browsing context is designated the **active document**.

Each `Document` has a collection of one or more views.

A **view** is a user agent interface tied to a particular media used for the presentation of a particular `Document` object in some media. A view may be interactive. Each view is represented by an `AbstractView` object. [DOM2VIEWS]

The main view through which a user primarily interacts with a user agent is the **default view**. The `AbstractView` object that represents this view must also implement the `Window` interface, and is referred to as the `Document`'s `Window` object. `WindowProxy` objects forward everything to the active document's default view's `Window` object.

The `defaultView` attribute on the `Document` object's `DocumentView` interface must return the browsing context's `WindowProxy` object, not the actual `AbstractView` object of the default view. [DOM3VIEWS]

> ***Note: The `document` attribute of an `AbstractView` object representing a view gives the view's corresponding `Document` object. [DOM2VIEWS]***

Events that use the `UIEvent` interface are related to a specific view (the view in which the event happened); when that view is the default view, the event object's `view` attribute's must return the `WindowProxy` object of the browsing context of that view, not the actual `AbstractView` object of the default view. [DOM3EVENTS]

> ***Note: A typical Web browser has one obvious view per `Document`: the browser's window (screen media). This is typically the default view. If a page is printed, however, a second view becomes evident, that of the print media. The two***

views always share the same underlying `Document` object, but they have a different presentation of that object. A speech browser might have a different default view, using the speech media.

Note: A `Document` does not necessarily have a browsing context associated with it. In particular, data mining tools are likely to never instantiate browsing contexts.

A browsing context can have a **creator browsing context**, the browsing context that was responsible for its creation. Unless otherwise specified, a browsing context has no creator browsing context.

If a browsing context *A* has a creator browsing context, then the `Document` that was the active document of that creator browsing context at the time *A* was created is the **creator `Document`**.

When a browsing context is first created, it must be created with a single `Document` in its session history, whose address is `about:blank`, which is marked as being an HTML document, and whose character encoding is UTF-8. The `Document` must have a single child `html` node, which itself has a single child `body` node. If the browsing context is created specifically to be immediately navigated, then that initial navigation will have replacement enabled.

The origin of the `about:blank` `Document` is set when the `Document` is created. If the new browsing context has a creator browsing context, then the origin of the `about:blank` `Document` is the origin of the creator `Document`. Otherwise, the origin of the `about:blank` `Document` is a globally unique identifier assigned when the new browsing context is created.

5.1.1 Nested browsing contexts

Certain elements (for example, `iframe` elements) can instantiate further browsing contexts. These are called **nested browsing contexts**. If a browsing context *P* has an element *E* in one of its `Document`s *D* that nests another browsing context *C* inside it, then *P* is said to be the **parent browsing context** of *C*, *C* is said to be a **child browsing context** of *P*, *C* is said to be **nested through** *D*, and *E* is said to be the **browsing context container** of *C*.

A browsing context *A* is said to be an ancestor of a browsing context *B* if there exists a browsing context *A'* that is a child browsing context of *A* and that is itself an ancestor of *B*, or if there is a browsing context *P* that is a child browsing context of *A* and that is the parent browsing context of *B*.

The browsing context with no parent browsing context is the **top-level browsing context** of all the browsing contexts nested within it (either directly or indirectly through other nested browsing contexts).

The transitive closure of parent browsing contexts for a nested browsing context gives the list of **ancestor browsing contexts**.

A `Document` is said to be **fully active** when it is the active document of its browsing context, and either its browsing context is a top-level browsing context, or the `Document` through which that browsing context is nested is itself fully active.

Because they are nested through an element, child browsing contexts are always tied to a specific `Document` in their parent browsing context. User agents must not allow the user to interact with child browsing contexts of elements that are in `Document`s that are not themselves fully active.

A nested browsing context can have a seamless browsing context flag set, if it is embedded through an `iframe` element with a `seamless` attribute.

5.1.1.1 Navigating nested browsing contexts in the DOM

window **`.top`**

Returns the `WindowProxy` for the top-level browsing context.

window **`.parent`**

Returns the `WindowProxy` for the parent browsing context.

window **`.frameElement`**

Returns the `Element` for the browsing context container.

Returns null if there isn't one.

Throws a `SECURITY_ERR` exception in cross-origin situations.

The **`top`** DOM attribute on the `Window` object of a `Document` in a browsing context *b* must return the `WindowProxy` object of its top-level browsing context (which would be its own `WindowProxy` object if it was a top-level browsing context itself).

The **`parent`** DOM attribute on the `Window` object of a `Document` in a browsing context *b* must return the `WindowProxy` object of the parent browsing context, if there is one (i.e. if *b* is a child browsing context), or the `WindowProxy` object of the browsing context *b* itself, otherwise (i.e. if it is a top-level browsing context).

The **`frameElement`** DOM attribute on the `Window` object of a `Document` *d*, on getting, must run the following algorithm:

1. If *d* is not a `Document` in a child browsing context, return null and abort these steps.
2. If the parent browsing context's active document does not have the same effective script origin as the script that is accessing the `frameElement` attribute, then throw a `SECURITY_ERR` exception.
3. Otherwise, return the browsing context container for *b*.

5.1.2 Auxiliary browsing contexts

It is possible to create new browsing contexts that are related to a top level browsing context

without being nested through an element. Such browsing contexts are called **auxiliary browsing contexts**. Auxiliary browsing contexts are always top-level browsing contexts.

An auxiliary browsing context has an **opener browsing context**, which is the browsing context from which the auxiliary browsing context was created, and it has a **furthest ancestor browsing context**, which is the top-level browsing context of the opener browsing context when the auxiliary browsing context was created.

5.1.2.1 Navigating auxiliary browsing contexts in the DOM

The **`opener`** DOM attribute on the `Window` object must return the `WindowProxy` object of the browsing context from which the current browsing context was created (its opener browsing context), if there is one and it is still available.

5.1.3 Secondary browsing contexts

User agents may support **secondary browsing contexts**, which are browsing contexts that form part of the user agent's interface, apart from the main content area.

5.1.4 Security

A browsing context *A* is **allowed to navigate** a second browsing context *B* if one of the following conditions is true:

- Either the origin of the active document of *A* is the same as the origin of the active document of *B*, or
- The browsing context *A* is a nested browsing context and its top-level browsing context is *B*, or
- The browsing context *B* is an auxiliary browsing context and *A* is allowed to navigate *B*'s opener browsing context, or
- The browsing context *B* is not a top-level browsing context, but there exists an ancestor browsing context of *B* whose active document has the same origin as the active document of *A* (possibly in fact being *A* itself).

5.1.5 Groupings of browsing contexts

Each browsing context is defined as having a list of zero or more **directly reachable browsing contexts**. These are:

- All the browsing context's child browsing contexts.
- The browsing context's parent browsing context.
- All the browsing contexts that have the browsing context as their opener browsing context.

- The browsing context's opener browsing context.

The transitive closure of all the browsing contexts that are directly reachable browsing contexts forms a **unit of related browsing contexts**.

Each unit of related browsing contexts is then further divided into the smallest number of groups such that every member of each group has an effective script origin that, through appropriate manipulation of the `document.domain` attribute, could be made to be the same as other members of the group, but could not be made the same as members of any other group. Each such group is a **unit of related similar-origin browsing contexts**.

Each unit of related similar-origin browsing contexts can have a **first script** which is used to obtain, amongst other things, the script's base URL to resolve relative URLs used in scripts running in that unit of related similar-origin browsing contexts. Initially there is no first script.

5.1.6 Browsing context names

Browsing contexts can have a **browsing context name**. By default, a browsing context has no name (its name is not set).

A **valid browsing context name** is any string with at least one character that does not start with a U+005F LOW LINE character. (Names starting with an underscore are reserved for special keywords.)

A **valid browsing context name or keyword** is any string that is either a valid browsing context name or that is an ASCII case-insensitive match for one of: `_blank`, `_self`, `_parent`, or `_top`.

The rules for choosing a browsing context given a browsing context name are as follows. The rules assume that they are being applied in the context of a browsing context.

1. If the given browsing context name is the empty string or `_self`, then the chosen browsing context must be the current one.

2. If the given browsing context name is `_parent`, then the chosen browsing context must be the *parent* browsing context of the current one, unless there isn't one, in which case the chosen browsing context must be the current browsing context.

3. If the given browsing context name is `_top`, then the chosen browsing context must be the most top-level browsing context of the current one.

4. If the given browsing context name is not `_blank` and there exists a browsing context whose name is the same as the given browsing context name, and the current browsing context is allowed to navigate that browsing context, and the user agent determines that the two browsing contexts are related enough that it is ok if they reach each other, then that browsing context must be the chosen one. If there are multiple matching browsing contexts, the user agent should select one in some arbitrary consistent manner, such as the most recently opened, most recently focused, or more closely related.

5. Otherwise, a new browsing context is being requested, and what happens depends on the user agent's configuration and/or abilities:

   If the current browsing context has the sandboxed navigation browsing context

flag set.

The user agent may offer to create a new top-level browsing context or reuse an existing top-level browsing context. If the user picks one of those options, then the designated browsing context must be the chosen one (the browsing context's name isn't set to the given browsing context name). Otherwise (if the user agent doesn't offer the option to the user, or if the user declines to allow a browsing context to be used) there must not be a chosen browsing context.

If the user agent has been configured such that in this instance it will create a new browsing context, and the browsing context is being requested as part of following a hyperlink whose link types include the `noreferrer` keyword

A new top-level browsing context must be created. If the given browsing context name is not `_blank`, then the new top-level browsing context's name must be the given browsing context name (otherwise, it has no name). The chosen browsing context must be this new browsing context. If it is immediately navigated, then the navigation will be done with replacement enabled.

If the user agent has been configured such that in this instance it will create a new browsing context, and the `noreferrer` keyword doesn't apply

A new auxiliary browsing context must be created, with the opener browsing context being the current one. If the given browsing context name is not `_blank`, then the new auxiliary browsing context's name must be the given browsing context name (otherwise, it has no name). The chosen browsing context must be this new browsing context. If it is immediately navigated, then the navigation will be done with replacement enabled.

If the user agent has been configured such that in this instance it will reuse the current browsing context

The chosen browsing context is the current browsing context.

If the user agent has been configured such that in this instance it will not find a browsing context

There must not be a chosen browsing context.

User agent implementors are encouraged to provide a way for users to configure the user agent to always reuse the current browsing context.

5.2 The `WindowProxy` object

As mentioned earlier, each browsing context has a **WindowProxy** object. This object is unusual in that it must proxy all operations to the `Window` object of the browsing context's active document. It is thus indistinguishable from that `Window` object in every way, except that it is not equal to it.

5.3 The `Window` object

```
[IndexGetter, NameGetter=OverrideBuiltins]
interface Window {
  // the current browsing context
  readonly attribute WindowProxy window;
  readonly attribute WindowProxy self;
           attribute DOMString name;
  [PutForwards=href] readonly attribute Location location;
  readonly attribute History history;
```

```
    readonly attribute UndoManager undoManager;
    Selection getSelection();
    [Replaceable] readonly attribute BarProp locationbar;
    [Replaceable] readonly attribute BarProp menubar;
    [Replaceable] readonly attribute BarProp personalbar;
    [Replaceable] readonly attribute BarProp scrollbars;
    [Replaceable] readonly attribute BarProp statusbar;
    [Replaceable] readonly attribute BarProp toolbar;
    void close();
    void focus();
    void blur();

    // other browsing contexts
    readonly attribute WindowProxy frames;
    readonly attribute unsigned long length;
    readonly attribute WindowProxy top;
    [Replaceable] readonly attribute WindowProxy opener;
    readonly attribute WindowProxy parent;
    readonly attribute Element frameElement;
    WindowProxy open([Optional] in DOMString url, [Optional] in DOMString target, [O

    // the user agent
    readonly attribute Navigator navigator;
    readonly attribute Storage localStorage;
    readonly attribute Storage sessionStorage;
    Database openDatabase(in DOMString name, in DOMString version, in DOMString disp
    readonly attribute ApplicationCache applicationCache;

    // user prompts
    void alert(in DOMString message);
    boolean confirm(in DOMString message);
    DOMString prompt(in DOMString message, [Optional] in DOMString default);
    void print();
    any showModalDialog(in DOMString url, [Optional] in any argument);

    // cross-document messaging
    void postMessage(in any message, in DOMString targetOrigin);
    void postMessage(in any message, in MessagePortArray ports, in DOMString targetO

    // event handler DOM attributes
             attribute Function onabort;
             attribute Function onbeforeunload;
             attribute Function onblur;
             attribute Function onchange;
             attribute Function onclick;
             attribute Function oncontextmenu;
             attribute Function ondblclick;
             attribute Function ondrag;
             attribute Function ondragend;
             attribute Function ondragenter;
             attribute Function ondragleave;
             attribute Function ondragover;
             attribute Function ondragstart;
             attribute Function ondrop;
             attribute Function onerror;
             attribute Function onfocus;
             attribute Function onhashchange;
             attribute Function onkeydown;
             attribute Function onkeypress;
             attribute Function onkeyup;
             attribute Function onload;
             attribute Function onmessage;
             attribute Function onmousedown;
             attribute Function onmousemove;
             attribute Function onmouseout;
             attribute Function onmouseover;
             attribute Function onmouseup;
             attribute Function onmousewheel;
             attribute Function ononline;
```

```
            attribute Function onoffline;
            attribute Function onpopstate;
            attribute Function onresize;
            attribute Function onscroll;
            attribute Function onselect;
            attribute Function onstorage;
            attribute Function onsubmit;
            attribute Function onunload;
};
```

window **.** **`window`**
window **.** **`frames`**
window **.** **`self`**

These attributes all return *window*.

The `Window` object must also implement the `EventTarget` interface.

The **`window`**, **`frames`**, and **`self`** DOM attributes must all return the `Window` object's browsing context's `WindowProxy` object.

5.3.1 Security

User agents must raise a `SECURITY_ERR` exception whenever any of the members of a `Window` object are accessed by scripts whose effective script origin is not the same as the `Window` object's `Document`'s effective script origin, with the following exceptions:

- The `location` object
- The `postMessage()` method with two arguments
- The `postMessage()` method with three arguments
- The `frames` attribute
- The dynamic nested browsing context properties

User agents must not allow scripts to override the `location` object's setter.

5.3.2 APIs for creating and navigating browsing contexts by name

window **=** ***window*** **.** **open(** **[** ***url*** **[,** ***target*** **[,** ***features*** **[,** ***replace*** **]]]])**

Opens a window to show *url* (defaults to `about:blank`), and returns it. The *target* argument gives the name of the new window. If a window exists with that name already, it is reused. The *replace* attribute, if true, means that whatever page is currently open in that window will be removed from the window's session history. The *features* argument is ignored.

window **.** **`name`** **[=** ***value*** **]**

Returns the name of the window.

Can be set, to change the name.

window **.** **`close`****()**

Closes the window.

The **`open()`** method on `Window` objects provides a mechanism for navigating an existing browsing context or opening and navigating an auxiliary browsing context.

The method has four arguments, though they are all optional.

The first argument, *url*, must be a valid URL for a page to load in the browsing context. If no arguments are provided, or if the first argument is the empty string, then the *url* argument defaults to "`about:blank`". The argument must be resolved to an absolute URL (or an error), relative to the first script's base URL, when the method is invoked.

The second argument, *target*, specifies the name of the browsing context that is to be navigated. It must be a valid browsing context name or keyword. If fewer than two arguments are provided, then the *name* argument defaults to the value "`_blank`".

The third argument, *features*, has no effect and is supported for historical reasons only.

The fourth argument, *replace*, specifies whether or not the new page will replace the page currently loaded in the browsing context, when *target* identifies an existing browsing context (as opposed to leaving the current page in the browsing context's session history). When three or fewer arguments are provided, *replace* defaults to false.

When the method is invoked, the user agent must first select a browsing context to navigate by applying the rules for choosing a browsing context given a browsing context name using the *target* argument as the name and the browsing context of the script as the context in which the algorithm is executed, unless the user has indicated a preference, in which case the browsing context to navigate may instead be the one indicated by the user.

> For example, suppose there is a user agent that supports control-clicking a link to open it in a new tab. If a user clicks in that user agent on an element whose `onclick` handler uses the `window.open()` API to open a page in an iframe, but, while doing so, holds the control key down, the user agent could override the selection of the target browsing context to instead target a new tab.

Then, the user agent must navigate the selected browsing context to the absolute URL (or error) obtained from resolving *url* earlier. If the *replace* is true, then replacement must be enabled; otherwise, it must not be enabled unless the browsing context was just created as part of the the rules for choosing a browsing context given a browsing context name. The navigation must be done with the browsing context of the script that invoked the method as the source browsing context.

The method must return the `WindowProxy` object of the browsing context that was navigated, or null if no browsing context was navigated.

The **name** attribute of the Window object must, on getting, return the current name of the browsing context, and, on setting, set the name of the browsing context to the new value.

> ***Note: The name gets reset when the browsing context is navigated to another domain.***

The **close()** method on Window objects should, if the corresponding browsing context *A* is an auxiliary browsing context that was created by a script (as opposed to by an action of the user), and if the browsing context of the script that invokes the method is allowed to navigate the browsing context *A*, close the browsing context *A* (and may discard it too).

5.3.3 Accessing other browsing contexts

***window*. length**

Returns the number of child browsing contexts.

***window*[*index*]**

Returns the indicated child browsing context.

The **length** DOM attribute on the Window interface must return the number of child browsing contexts of the Document.

The indices of the supported indexed properties on the Window object at any instant are the numbers in the range 0 .. *n*-1, where *n* is the number of child browsing contexts of the Document. If *n* is zero then there are no supported indexed properties.

When a Window object is **indexed to retrieve an indexed property** *index*, the value returned must be the *index*th child browsing context of the Document, sorted in the tree order of the elements nesting those browsing contexts.

These properties are the **dynamic nested browsing context properties**.

5.3.4 Named access on the Window object

***window*[*name*]**

Returns the indicated child browsing context.

The Window interface supports named properties. The names of the supported named properties at any moment consist of:

- The value of the name content attribute for all a, applet, area, embed, frame, frameset,

`form`, `iframe`, `img`, and `object` elements in the active document that have a `name` content attribute, and,

- The value of the `id` content attribute of any HTML element in the active document with an `id` content attribute.

When **the `Window` object is indexed for property retrieval** using a name *name*, then the user agent must return the value obtained using the following steps:

1. Let *elements* be the list of named elements with the name *name* in the active document.

   Note: There will be at least one such element, by definition.

2. If *elements* contains an `iframe` element, then return the `WindowProxy` object of the nested browsing context represented by the first such `iframe` element in tree order, and abort these steps.

3. Otherwise, if *elements* has only one element, return that element and abort these steps.

4. Otherwise return an `HTMLCollection` rooted at the `Document` node, whose filter matches only named elements with the name *name*.

Named elements with the name *name*, for the purposes of the above algorithm, are those that are either:

- `a`, `applet`, `area`, `embed`, `form`, `frame`, `frameset`, `iframe`, `img`, or `object` elements that have a `name` content attribute whose value is *name*, or

- HTML elements elements that have an `id` content attribute whose value is *name*.

5.3.5 Garbage collection and browsing contexts

A browsing context has a strong reference to each of its `Document`s and its `WindowProxy` object, and the user agent itself has a strong reference to its top-level browsing contexts.

A `Document` has a strong reference to each of its views and their `AbstractView` objects.

When a browsing context is to **discard a `Document`**, that means that it is to lose the strong reference from the `Document`'s browsing context to the `Document`.

Note: The browsing context's default view's `Window` object has a strong reference of its own to its `Document` object.

When **a *browsing context* is discarded**, the strong reference from the user agent itself to the browsing context must be severed, and all the `Document` objects for all the entries in the browsing context's session history must be discarded as well.

User agents may discard top-level browsing contexts at any time (typically, in response to user requests, e.g. when a user closes a window containing one or more top-level browsing contexts). Other browsing contexts must be discarded once their `WindowProxy` object is eligible for garbage collection.

5.3.6 Browser interface elements

To allow Web pages to integrate with Web browsers, certain Web browser interface elements are exposed in a limited way to scripts in Web pages.

Each interface element is represented by a `BarProp` object:

```
interface BarProp {
          attribute boolean visible;
};
```

window **`.locationbar.visible`**

Returns true if the location bar is visible; otherwise, returns false.

window **`.menubar.visible`**

Returns true if the menu bar is visible; otherwise, returns false.

window **`.personalbar.visible`**

Returns true if the personal bar is visible; otherwise, returns false.

window **`.scrollbars.visible`**

Returns true if the scroll bars are visible; otherwise, returns false.

window **`.statusbar.visible`**

Returns true if the status bar is visible; otherwise, returns false.

window **`.toolbar.visible`**

Returns true if the tool bar is visible; otherwise, returns false.

The **visible** attribute, on getting, must return either true or a value determined by the user agent to most accurately represent the visibility state of the user interface element that the object represents, as described below. On setting, the new value must be discarded.

The following `BarProp` objects exist for each `Document` object in a browsing context. Some of the user interface elements represented by these objects might have no equivalent in some user agents; for those user agents, unless otherwise specified, the object must act as if it was present and visible (i.e. its `visible` attribute must return true).

The location bar `BarProp` object

Represents the user interface element that contains a control that displays the URL of the active document, or some similar interface concept.

The menu bar `BarProp` object

Represents the user interface element that contains a list of commands in menu form, or some similar interface concept.

The personal bar `BarProp` object

Represents the user interface element that contains links to the user's favorite pages, or some similar interface concept.

The scrollbar `BarProp` object

Represents the user interface element that contains a scrolling mechanism, or some similar interface concept.

The status bar `BarProp` object

Represents a user interface element found immediately below or after the document, as appropriate for the default view's media. If the user agent has no such user interface element, then the object may act as if the corresponding user interface element was absent (i.e. its `visible` attribute may return false).

The tool bar `BarProp` object

Represents the user interface element found immediately above or before the document, as appropriate for the default view's media. If the user agent has no such user interface element, then the object may act as if the corresponding user interface element was absent (i.e. its `visible` attribute may return false).

The **`locationbar`** attribute must return the location bar `BarProp` object.

The **`menubar`** attribute must return the menu bar `BarProp` object.

The **`personalbar`** attribute must return the personal bar `BarProp` object.

The **`scrollbars`** attribute must return the scrollbar `BarProp` object.

The **`statusbar`** attribute must return the status bar `BarProp` object.

The **`toolbar`** attribute must return the tool bar `BarProp` object.

5.4 Origin

The **origin** of a resource and the **effective script origin** of a resource are both either opaque identifiers or tuples consisting of a scheme component, a host component, a port component, and optionally extra data.

> ***Note: The extra data could include the certificate of the site when using encrypted connections, to ensure that if the site's secure certificate changes, the origin is considered to change as well.***

These characteristics are defined as follows:

For URLs

The origin and effective script origin of the URL is whatever is returned by the following algorithm:

1. Let *url* be the URL for which the origin is being determined.
2. Parse *url*.
3. If *url* does not use a server-based naming authority, or if parsing *url* failed, or if *url* is not an absolute URL, then return a new globally unique identifier.

4. Let *scheme* be the <scheme> component of *url*, converted to lowercase.

5. If the UA doesn't support the protocol given by *scheme*, then return a new globally unique identifier.

6. If *scheme* is "`file`", then the user agent may return a UA-specific value.

7. Let *host* be the <host> component of *url*.

8. Apply the IDNA ToASCII algorithm to *host*, with both the AllowUnassigned and UseSTD3ASCIIRules flags set. Let *host* be the result of the ToASCII algorithm.

   If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, then return a new globally unique identifier. [RFC3490]

9. Let *host* be the result of converting *host* to lowercase.

10. If there is no <port> component, then let *port* be the default port for the protocol given by *scheme*. Otherwise, let *port* be the <port> component of *url*.

11. Return the tuple (*scheme*, *host*, *port*).

In addition, if the URL is in fact associated with a `Document` object that was created by parsing the resource obtained from fetching URL, and this was done over a secure connection, then the server's secure certificate may be added to the origin as additional data.

For scripts

The origin and effective script origin of a script are determined from another resource, called the *owner*:

- **If a script is in a `script` element**
  The owner is the `Document` to which the `script` element belongs.
- **If a script is in an event handler content attribute**
  The owner is the `Document` to which the attribute node belongs.
- **If a script is a function or other code reference created by another script**
  The owner is the script that created it.
- **If a script is a `javascript:` URL that was returned as the location of an HTTP redirect (or equivalent in other protocols)**
  The owner is the URL that redirected to the `javascript:` URL.
- **If a script is a `javascript:` URL in an attribute**
  The owner is the `Document` of the element on which the attribute is found.
- **If a script is a `javascript:` URL in a style sheet**
  The owner is the URL of the style sheet.
- **If a script is a `javascript:` URL to which a browsing context is being navigated, the URL having been provided by the user (e.g. by using a *bookmarklet*)**
  The owner is the `Document` of the browsing context's active document.
- **If a script is a `javascript:` URL to which a browsing context is being navigated, the URL having been declared in markup**
  The owner is the `Document` of the element (e.g. an `a` or `area` element) that

declared the URL.

↪ **If a script is a `javascript:` URL to which a browsing context is being navigated, the URL having been provided by script**

The owner is the script that provided the URL.

The origin of the script is then equal to the origin of the owner, and the effective script origin of the script is equal to the effective script origin of the owner.

For `Document` objects and images

↪ **If a `Document` is in a browsing context whose sandboxed origin browsing context flag was set when the `Document` was created**

The origin is a globally unique identifier assigned when the `Document` is created.

↪ **If a `Document` or image was returned by the `XMLHttpRequest` API**

The origin and effective script origin are equal to the origin and effective script origin of the `Document` object of the `Window` object from which the `XMLHttpRequest` constructor was invoked. (That is, they track the `Document` to which the `XMLHttpRequest` object's `Document` pointer pointed when it was created.) [XHR]

↪ **If a `Document` or image was generated from a `javascript:` URL**

The origin is equal to the origin of the script of that `javascript:` URL.

↪ **If a `Document` or image was served over the network and has an address that uses a URL scheme with a server-based naming authority**

The origin is the origin of the address of the `Document` or the URL of the image, as appropriate.

↪ **If a `Document` or image was generated from a `data:` URL that was returned as the location of an HTTP redirect (or equivalent in other protocols)**

The origin is the origin of the URL that redirected to the `data:` URL.

↪ **If a `Document` or image was generated from a `data:` URL found in another `Document` or in a script**

The origin is the origin of the `Document` or script in which the `data:` URL was found.

↪ **If a `Document` has the address "`about:blank`"**

The origin of the `Document` is the origin it was assigned when its browsing context was created.

↪ **If a `Document` or image was obtained in some other manner (e.g. a `data:` URL typed in by the user, a `Document` created using the `createDocument()` API, a `data:` URL returned as the location of an HTTP redirect, etc)**

The origin is a globally unique identifier assigned when the `Document` or image is created.

When a `Document` is created, unless stated otherwise above, its effective script origin is initialized to the origin of the `Document`. However, the `document.domain` attribute can be used to change it.

The **Unicode serialization of an origin** is the string obtained by applying the following algorithm to the given origin:

1. If the origin in question is not a scheme/host/port tuple, then return the literal string "`null`" and abort these steps.

2. Otherwise, let *result* be the scheme part of the origin tuple.

3. Append the string "`://`" to *result*.

4. Apply the IDNA ToUnicode algorithm to each component of the host part of the origin tuple, and append the results — each component, in the same order, separated by U+002E FULL STOP characters (".") — to *result*.

5. If the port part of the origin tuple gives a port that is different from the default port for the protocol given by the scheme part of the origin tuple, then append a U+003A COLON character (":") and the given port, in base ten, to *result*.

6. Return *result*.

The **ASCII serialization of an origin** is the string obtained by applying the following algorithm to the given origin:

1. If the origin in question is not a scheme/host/port tuple, then return the literal string "`null`" and abort these steps.

2. Otherwise, let *result* be the scheme part of the origin tuple.

3. Append the string "`://`" to *result*.

4. Apply the IDNA ToASCII algorithm the host part of the origin tuple, with both the AllowUnassigned and UseSTD3ASCIIRules flags set, and append the results *result*.

   If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, then return the empty string and abort these steps. [RFC3490]

5. If the port part of the origin tuple gives a port that is different from the default port for the protocol given by the scheme part of the origin tuple, then append a U+003A COLON character (":") and the given port, in base ten, to *result*.

6. Return *result*.

Two origins are said to be the **same origin** if the following algorithm returns true:

1. Let *A* be the first origin being compared, and *B* be the second origin being compared.

2. If *A* and *B* are both opaque identifiers, and their value is equal, then return true.

3. Otherwise, if either *A* or *B* or both are opaque identifiers, return false.

4. If *A* and *B* have scheme components that are not identical, return false.

5. If *A* and *B* have host components that are not identical, return false.

6. If *A* and *B* have port components that are not identical, return false.

7. If either *A* or *B* have additional data, but that data is not identical for both, return false.

8. Return true.

5.4.1 Relaxing the same-origin restriction

document **.** `domain` **[=** ***domain*** **]**

Returns the current domain used for security checks.

Can be set to a value that removes subdomains, to allow pages on other subdomains of the same domain (if they do the same thing) to access each other.

The **`domain`** attribute on `Document` objects must be initialized to the document's domain, if it has one, and the empty string otherwise. If the value is an IPv6 address, then the square brackets from the host portion of the <host> component must be omitted from the attribute's value.

On getting, the attribute must return its current value, unless the document was created by `XMLHttpRequest`, in which case it must throw an `INVALID_ACCESS_ERR` exception.

On setting, the user agent must run the following algorithm:

1. If the document was created by `XMLHttpRequest`, throw an `INVALID_ACCESS_ERR` exception and abort these steps.

2. If the new value is an IP address, let *new value* be the new value. Otherwise, apply the IDNA ToASCII algorithm to the new value, with both the AllowUnassigned and UseSTD3ASCIIRules flags set, and let *new value* be the result of the ToASCII algorithm.

   If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, then throw a `SECURITY_ERR` exception and abort these steps. [RFC3490]

3. If *new value* is not exactly equal to the current value of the `document.domain` attribute, then run these substeps:

   1. If the current value is an IP address, throw a `SECURITY_ERR` exception and abort these steps.

   2. If *new value*, prefixed by a U+002E FULL STOP ("."), does not exactly match the end of the current value, throw a `SECURITY_ERR` exception and abort these steps.

   3. If *new value* matches a suffix in the Public Suffix List, or, if *new value*, prefixed by a U+002E FULL STOP ("."), matches the end of a suffix in the Public Suffix List, then throw a `SECURITY_ERR` exception and abort these steps. [PSL]

      Suffixes must be compared after applying the IDNA ToASCII algorithm to them, with both the AllowUnassigned and UseSTD3ASCIIRules flags set, in an ASCII case-insensitive manner. [RFC3490]

4. Set the attribute's value to *new value*.

5. Set the host part of the effective script origin tuple of the `Document` to *new value*.

6. Set the port part of the effective script origin tuple of the `Document` to "manual override" (a value that, for the purposes of comparing origins, is identical to "manual override" but not identical to any other value).

The **domain** of a `Document` is the host part of the document's origin, if that is a scheme/host/port tuple. If it isn't, then the document does not have a domain.

Note: The `domain` attribute is used to enable pages on different hosts of a domain to access each others' DOMs.

5.5 Scripting

5.5.1 Introduction

Various mechanisms can cause author-provided executable code to run in the context of a document. These mechanisms include, but are probably not limited to:

- Processing of `script` elements.
- Processing of inline `javascript:` URLs (e.g. the `src` attribute of `img` elements, or an `@import` rule in a CSS `style` element block).
- Event handlers, whether registered through the DOM using `addEventListener()`, by explicit event handler content attributes, by event handler DOM attributes, or otherwise.
- Processing of technologies like XBL or SVG that have their own scripting features.

5.5.2 Enabling and disabling scripting

Scripting is enabled in a *browsing context* when all of the following conditions are true:

- The user agent supports scripting.
- The user has not disabled scripting for this browsing context at this time. (User agents may provide users with the option to disable scripting globally, or in a finer-grained manner, e.g. on a per-origin basis.)
- The browsing context does not have the sandboxed scripts browsing context flag set.

Scripting is disabled in a browsing context when any of the above conditions are false (i.e. when scripting is not enabled).

Scripting is enabled for a *node* if the `Document` object of the node (the node itself, if it is itself a `Document` object) has an associated browsing context, and scripting is enabled in that browsing context.

Scripting is disabled for a node if there is no such browsing context, or if scripting is disabled in that browsing context.

5.5.3 Processing model

5.5.3.1 Definitions

A **script** has:

A *script execution environment*
The characteristics of the script execution environment depend on the language, and are not defined by this specification.

> In JavaScript, the script execution environment consists of the interpreter, the stack of *execution contexts*, the *global code* and *function code* and the Function objects resulting, and so forth.

A *list of code entry-points*
Each code entry-point represents a block of executable code that the script exposes to other scripts and to the user agent.

> Each Function object in a JavaScript script execution environment has a corresponding code entry-point, for instance.

The main program code of the script, if any, is the ***initial code entry-point***. Typically, the code corresponding to this entry-point is executed immediately after the script is parsed.

> In JavaScript, this corresponds to the execution context of the global code.

A relationship with the *script's global object*
An object that provides the APIs that the code can use.

> This is typically a `Window` object. In JavaScript, this corresponds to the *global object*.

Note: When a script's global object is an empty object, it can't do anything that interacts with the environment.

If the script's global object is a `Window` object, then in JavaScript, the `this` keyword in the global scope must, contrary to the ECMAScript specification, return the `Window` object's `WindowProxy` object.

Note: This is a willful violation of ECMAScript edition 3. [ECMA262]

A relationship with the *script's browsing context*
A browsing context that is assigned responsibility for actions taken by the script.

> When a script creates and navigates a new top-level browsing context, the `opener` attribute of the new browsing context's `Window` object will be set to the script's browsing context's `WindowProxy` object.

A *URL character encoding*

A character encoding, set when the script is created, used to encode URLs. If the character encoding is set from another source, e.g. a document's character encoding, then the script's URL character encoding must follow the source, so that if the source's changes, so does the script's.

A *base URL*

A URL, set when the script is created, used to resolve relative URLs. If the base URL is set from another source, e.g. a document base URL, then the script's base URL must follow the source, so that if the source's changes, so does the script's.

5.5.3.2 Calling scripts

When a user agent is to **jump to a code entry-point** for a script, for example to invoke an event listener defined in that script, the user agent must run the following steps:

1. If the script's global object is a `Window` object whose `Document` object is not fully active, then abort these steps without doing anything. The callback is not fired.
2. Set the first script to be the script being invoked.
3. Make the script execution environment for the script execute the code for the given code entry-point.
4. Set the first script back to whatever it was when this algorithm started.

This algorithm is not invoked by one script calling another.

5.5.3.3 Creating scripts

When the specification says that a script is to be **created**, given some script source, its scripting language, a global object, a browsing context, a character encoding, and a base URL, the user agent must run the following steps:

1. If scripting is disabled for browsing context passed to this algorithm, then abort these steps, as if the script did nothing but return void.
2. Set up a script execution environment as appropriate for the scripting language.
3. Parse/compile/initialize the source of the script using the script execution environment, as appropriate for the scripting language, and thus obtain the list of code entry-points for the script. If the semantics of the scripting language and the given source code are such that there is executable code to be immediately run, then the *initial code entry-point* is the entry-point for that code.
4. Set up the script's global object, the script's browsing context, the script's URL character encoding, and the script's base URL from the settings passed to this algorithm.
5. Jump to the script's *initial code entry-point*.

When the user agent is to **create an impotent script**, given some script source, its scripting

language, and a browsing context, the user agent must create a script, using the given script source and scripting language, using a new empty object as the global object, and using the given browsing context as the browsing context. The character encoding and base URL for the resulting script are not important as no APIs are exposed to the script.

When the specification says that a script is to be **created from a node** *node*, given some script source and its scripting language, the user agent must create a script, using the given script source and scripting language, and using the script settings determined from the node *node*.

The script settings determined from the node *node* are computed as follows:

1. Let *document* be the `Document` of *node* (or *node* itself if it is a `Document`).
2. The browsing context is the browsing context of *document*.
3. The global object is the `Window` object of *document*.
4. The character encoding is the character encoding of *document*. (This is a reference, not a copy.)
5. The base URL is the base URL of *document*. (This is a reference, not a copy.)

5.5.3.4 Killing scripts

User agents may impose resource limitations on scripts, for example CPU quotas, memory limits, total execution time limits, or bandwidth limitations. When a script exceeds a limit, the user agent may either throw a `QUOTA_EXCEEDED_ERR` exception, abort the script without an exception, prompt the user, or throttle script execution.

> For example, the following script never terminates. A user agent could, after waiting for a few seconds, prompt the user to either terminate the script or let it continue.
>
> ```
> <script>
>  while (true) { /* loop */ }
> </script>
> ```

User agents are encouraged to allow users to disable scripting whenever the user is prompted either by a script (e.g. using the `window.alert()` API) or because of a script's actions (e.g. because it has exceeded a time limit).

If scripting is disabled while a script is executing, the script should be terminated immediately.

5.5.4 Event loops

5.5.4.1 Definitions

To coordinate events, user interaction, scripts, rendering, networking, and so forth, user agents must use **event loops** as described in this section.

There must be at least one event loop per user agent, and at most one event loop per unit of related similar-origin browsing contexts.

An event loop always has at least one browsing context. If an event loop's browsing contexts all go away, then the event loop goes away as well. A browsing context always has an event loop coordinating its activities.

> ***Note: Other specifications can define new kinds of event loops that aren't associated with browsing contexts.***

An event loop has one or more **task queues**. A task queue is an ordered list of **tasks**, which can be:

Events
Asynchronously dispatching an `Event` object at a particular `EventTarget` object is a task.

> ***Note: Not all events are dispatched using the task queue, many are dispatched synchronously during other tasks.***

Parsing
The HTML parser tokenising a single byte, and then processing any resulting tokens, is a task.

Callbacks
Calling a callback asynchronously is a task.

Using a resource
When an algorithm fetches a resource, if the fetching occurs asynchronously then the processing of the resource once some or all of the resource is available is a task.

Reacting to DOM manipulation
Some elements have tasks that trigger in response to DOM manipulation, e.g. when that element is inserted into the document.

Each task is associated with a `Document`; if the task was queued in the context of an element, then it is the element's `Document`; if the task was queued in the context of a browsing context, then it is the browsing context's active document at the time the task was queued; if the task was queued by or for a script then the document is the script's script's browsing context's active document at the time the task was queued.

When a user agent is to **queue a task**, it must add the given task to one of the task queues of the relevant event loop. All the tasks from one particular **task source** (e.g. the callbacks generated by timers, the events dispatched for mouse movements, the tasks queued for the parser) must always be added to the same task queue, but tasks from different task sources may be placed in different task queues.

> For example, a user agent could have one task queue for mouse and key events (the user interaction task source), and another for everything else. The user agent could then give keyboard and mouse events preference over other tasks three quarters of the time, keeping the interface responsive but not starving other task queues, and never processing events from any one task source out of order.

A user agent is required to have one **storage mutex**. This mutex is used to control access to shared state like cookies. At any one point, the storage mutex is either free, or owned by a particular event loop or instance of the fetching algorithm.

5.5.4.2 Processing model

An event loop must continually run through the following steps for as long as it exists:

1. Run the oldest task on one of the event loop's task queues, ignoring tasks whose associated `Document`s are not fully active. The user agent may pick any task queue.
2. If the storage mutex is now owned by the event loop, release it so that it is once again free.
3. Remove that task from its task queue.
4. If necessary, update the rendering or user interface of any `Document` or browsing context to reflect the current state.
5. Return to the first step of the event loop.

Some of the algorithms in this specification, for historical reasons, require the user agent to **pause** while running a task until some condition has been met. While a user agent has a paused task, the corresponding event loop must not run further tasks, and any script in the currently running task must block. User agents should remain responsive to user input while paused, however, albeit in a reduced capacity since the event loop will not be doing anything.

When a user agent is to **obtain the storage mutex** as part of running a task, it must run through the following steps:

1. If the storage mutex is already owned by this task's event loop, then abort these steps.
2. Otherwise, pause until the storage mutex can be taken by the event loop.
3. Take ownership of the storage mutex.

5.5.4.3 Generic task sources

The following task sources are used by a number of mostly unrelated features in this and other specifications.

The *DOM manipulation task source*

This task source is used for features that react to DOM manipulations, such as things that happen asynchronously when an element is inserted into the document.

Asynchronous mutation events must be dispatched using tasks queued with the DOM manipulation task source. [DOMEVENTS]

The *user interaction task source*

This task source is used for features that react to user interaction, for example keyboard or mouse input.

Asynchronous events sent in response to user input (e.g. click events) must be dispatched using tasks queued with the user interaction task source. [DOMEVENTS]

The *networking task source*

This task source is used for features that trigger in response to network activity.

5.5.5 The `javascript:` protocol

When a URL using the `javascript:` protocol is **dereferenced**, the user agent must run the following steps:

1. Let the script source be the string obtained using the content retrieval operation defined for `javascript:` URLs. [JSURL]

2. Use the appropriate step from the following list:

   If a browsing context is being navigated to a `javascript:` URL, and the active document of that browsing context has the same origin as the script given by that URL

   Let *address* be the address of the active document of the browsing context being navigated.

   If *address* is `about:blank`, and the browsing context being navigated has a creator browsing context, then let *address* be the address of the creator `Document` instead.

   Create a script from the `Document` node of the active document, using the aforementioned script source, and assuming the scripting language is JavaScript.

   Let *result* be the return value of the *initial code entry-point* of this script. If an exception was raised, let *result* be void instead. (The result will be void also if scripting is disabled.)

   When it comes time to set the document's address in the navigation algorithm, use *address* as the override URL.

   If the `Document` object of the element, attribute, or style sheet from which the `javascript:` URL was reached has an associated browsing context

   Create an impotent script using the aforementioned script source, with the scripting language set to JavaScript, and with the `Document`'s object's browsing context as the browsing context.

   Let *result* be the return value of the *initial code entry-point* of this script. If an exception was raised, let *result* be void instead. (The result will be void also if scripting is disabled.)

   Otherwise

   Let *result* be void.

3. If the result of executing the script is void (there is no return value), then the URL must be treated in a manner equivalent to an HTTP resource with an HTTP 204 No Content response.

   Otherwise, the URL must be treated in a manner equivalent to an HTTP resource with a 200 OK response whose Content-Type metadata is `text/html` and whose response body is the return value converted to a string value.

Note: Certain contexts, in particular `img` elements, ignore the Content-Type metadata.

> So for example a `javascript:` URL for a `src` attribute of an `img` element would be evaluated in the context of an empty object as soon as the attribute is set; it would then be sniffed to determine the image type and decoded as an image.
>
> A `javascript:` URL in an `href` attribute of an `a` element would only be evaluated when the link was followed.
>
> The `src` attribute of an `iframe` element would be evaluated in the context of the `iframe`'s own browsing context; once evaluated, its return value (if it was not void) would replace that browsing context's document, thus changing the variables visible in that browsing context.

5.5.6 Events

5.5.6.1 Event handler attributes

Many objects can have **event handler attributes** specified. These act as bubbling event listeners for the element on which they are specified.

An event handler attribute, unless otherwise specified, can either have the value null or be set to a `Function` object. Initially, an event handler attribute must be set to null.

Event handler attributes are exposed in one or two ways.

The first way, common to all event handler attributes, is as an event handler DOM attribute.

The second way is as an event handler content attribute. Event handlers on HTML elements and some of the event handlers on `Window` objects are exposed in this way.

Event handler DOM attributes, on setting, must set the corresponding event handler attribute to their new value, and on getting, must return whatever the current value of the corresponding event handler attribute is (possibly null).

Note: Certain event handler DOM attributes have additional requirements, in particular the `onmessage` attribute of `MessagePort` objects.

Event handler content attributes, when specified, must contain valid ECMAScript code matching the ECMAScript `FunctionBody` production. [ECMA262]

When an event handler content attribute is set, if the element is owned by a `Document` that is in a browsing context, and scripting is enabled for that browsing context, the user agent must run the following steps to create a script after setting the content attribute to its new value:

1. Set up a script execution environment for JavaScript.

2. Using this script execution environment, interpret the attribute's new value as the body of an anonymous function, with the function's arguments set as follows:

   ↪ **If the attribute is the `onerror` attribute of the `Window` object**
   Let the function have three arguments, named `event`, `source`, and `fileno`.

   ↪ **Otherwise**
   Let the function have a single argument called `event`.

   Link the new function's scope chain from the activation object of the handler, to the element's object, to the element's form owner, if it has one, to the element's `Document` object, to the `Window` object of that `Document`. Set the function's `this` parameter to the `Element` object representing the element. Let this function be the only entry in the script's list of code entry-points.

   Note: See ECMA262 Edition 3, sections 10.1.6 and 10.2.3, for more details on activation objects. [ECMA262]

3. If the previous steps failed to compile the script, then set the corresponding event handler attribute to null and abort these steps.

4. Set up the script's global object, the script's browsing context, the script's URL character encoding, and the script's base URL from the script settings determined from the node on which the attribute is being set.

5. Set the corresponding event handler attribute to the aforementioned function.

Note: When an event handler content attribute is set on an element owned by a `Document` that is not in a browsing context, the corresponding event handler attribute is not changed.

Note: Removing an event handler content attribute does not reset the corresponding event handler attribute.

How do we allow non-JS event handlers?

All event handler attributes on an element, whether set to null or to a `Function` object, must be registered as event listeners on the element, as if the `addEventListenerNS()` method on the `Element` object's `EventTarget` interface had been invoked when the event handler attribute's element or object was created, with the event type (*type* argument) equal to the type described for the event handler attribute in the list above, the namespace (*namespaceURI* argument) set to null, the listener set to be a target and bubbling phase listener (*useCapture* argument set to false), the event group set to the default group (*evtGroup* argument set to null), and the event listener itself (*listener* argument) set to do nothing while the event handler attribute's value is not a `Function` object, and set to invoke the `call()` callback of the `Function` object associated with the event handler attribute otherwise.

Note: The listener argument is emphatically not the event handler attribute itself.

When an event handler attribute's `Function` objectw is invoked, its `call()` callback must be invoked with one argument, set to the `Event` object of the event in question.

The handler's return value must then be processed as follows:

- **If the event type is `mouseover`**
  If the return value is a boolean with the value true, then the event must be canceled.

- **If the event object is a `BeforeUnloadEvent` object**
  If the return value is a string, and the event object's `returnValue` attribute's value is the empty string, then set the `returnValue` attribute's value to the return value.

- **Otherwise**
  If the return value is a boolean with the value false, then the event must be canceled.

The `Function` interface represents a function in the scripting language being used. It is represented in IDL as follows:

```
[Callback=FunctionOnly, NoInterfaceObject]
interface Function {
  any call([Variadic] in any arguments);
};
```

The **`call(...)`** method is the object's callback.

Note: In JavaScript, any `Function` object implements this interface.

5.5.6.2 Event handler attributes on elements and on `Window` objects

The following are the event handler attributes that must be supported by all HTML elements, as both content attributes and DOM attributes, and on `Window` objects, as DOM attributes:

`onabort`
Must be invoked whenever an `abort` event is targeted at or bubbles through the element or object.

`onblur`
Must be invoked whenever a `blur` event is targeted at or bubbles through the element or object.

`onchange`
Must be invoked whenever a `change` event is targeted at or bubbles through the element or object.

`onclick`
Must be invoked whenever a `click` event is targeted at or bubbles through the element or object.

`oncontextmenu`

Must be invoked whenever a `contextmenu` event is targeted at or bubbles through the element or object.

ondblclick
Must be invoked whenever a `dblclick` event is targeted at or bubbles through the element or object.

ondrag
Must be invoked whenever a `drag` event is targeted at or bubbles through the element or object.

ondragend
Must be invoked whenever a `dragend` event is targeted at or bubbles through the element or object.

ondragenter
Must be invoked whenever a `dragenter` event is targeted at or bubbles through the element or object.

ondragleave
Must be invoked whenever a `dragleave` event is targeted at or bubbles through the element or object.

ondragover
Must be invoked whenever a `dragover` event is targeted at or bubbles through the element or object.

ondragstart
Must be invoked whenever a `dragstart` event is targeted at or bubbles through the element or object.

ondrop
Must be invoked whenever a `drop` event is targeted at or bubbles through the element or object.

onfocus
Must be invoked whenever a `focus` event is targeted at or bubbles through the element or object.

onkeydown
Must be invoked whenever a `keydown` event is targeted at or bubbles through the element or object.

onkeypress
Must be invoked whenever a `keypress` event is targeted at or bubbles through the element or object.

onkeyup
Must be invoked whenever a `keyup` event is targeted at or bubbles through the element or object.

onmousedown
Must be invoked whenever a `mousedown` event is targeted at or bubbles through the element or object.

onmousemove
: Must be invoked whenever a `mousemove` event is targeted at or bubbles through the element or object.

onmouseout
: Must be invoked whenever a `mouseout` event is targeted at or bubbles through the element or object.

onmouseover
: Must be invoked whenever a `mouseover` event is targeted at or bubbles through the element or object.

onmouseup
: Must be invoked whenever a `mouseup` event is targeted at or bubbles through the element or object.

onmousewheel
: Must be invoked whenever a `mousewheel` event is targeted at or bubbles through the element or object.

onscroll
: Must be invoked whenever a `scroll` event is targeted at or bubbles through the element or object.

onselect
: Must be invoked whenever a `select` event is targeted at or bubbles through the element or object.

onsubmit
: Must be invoked whenever a `submit` event is targeted at or bubbles through the element or object.

The following are the event handler attributes that must be supported by all HTML elements other than `body`, as both content attributes and DOM attributes:

onerror
: Must be invoked whenever a `error` event is targeted at or bubbles through the element.

onload
: Must be invoked whenever a `load` event is targeted at or bubbles through the element.

The following are the event handler attributes that must be supported by `Window` objects, as DOM attributes on the `Window` object, and with corresponding content attributes and DOM attributes exposed on the `body` element:

onbeforeunload
: Must be invoked whenever a `beforeunload` event is targeted at or bubbles through the object.

onerror
: Must be invoked whenever an `error` event is targeted at or bubbles through the object.

Note: The `onerror` handler is also used for reporting script errors.

`onhashchange`
: Must be invoked whenever a `hashchange` event is targeted at or bubbles through the object.

`onload`
: Must be invoked whenever a `load` event is targeted at or bubbles through the object.

`onmessage`
: Must be invoked whenever a `message` event is targeted at or bubbles through the object.

`onoffline`
: Must be invoked whenever a `offline` event is targeted at or bubbles through the object.

`ononline`
: Must be invoked whenever a `online` event is targeted at or bubbles through the object.

`onpopstate`
: Must be invoked whenever a `popstate` event is targeted at or bubbles through the object.

`onresize`
: Must be invoked whenever a `resize` event is targeted at or bubbles through the object.

`onstorage`
: Must be invoked whenever a `storage` event is targeted at or bubbles through the object.

`onunload`
: Must be invoked whenever an `unload` event is targeted at or bubbles through the object.

5.5.6.3 Event firing

maybe this should be moved higher up (terminology? conformance? DOM?) Also, the whole terminology thing should be changed so that we don't define any specific events here, we only define 'simple event', 'progress event', 'mouse event', 'key event', and the like, and have the actual dispatch use those generic terms when firing events.

Certain operations and methods are defined as firing events on elements. For example, the `click()` method on the `HTMLElement` interface is defined as firing a `click` event on the element. [DOM3EVENTS]

Firing a `click` event means that a `click` event with no namespace, which bubbles and is cancelable, and which uses the `MouseEvent` interface, must be dispatched at the given target. The event object must have its `screenX`, `screenY`, `clientX`, `clientY`, and `button` attributes set to 0, its `ctrlKey`, `shiftKey`, `altKey`, and `metaKey` attributes set according to the current state of the key input device, if any (false for any keys that are not available), its `detail` attribute set to 1, and its `relatedTarget` attribute set to null. The `getModifierState()` method on the object must return values appropriately describing the state of the key input device at the time the event is created.

Firing a `contextmenu` event means that a `contextmenu` event with no namespace, which bubbles and is cancelable, and which uses the `Event` interface, must be dispatched at the given target.

Firing a simple event called *e* means that an event with the name *e*, with no namespace, which does not bubble but is cancelable (unless otherwise stated), and which uses the `Event` interface, must be dispatched at the given target.

Firing a `show` event means firing a simple event called `show`. Actually this should fire an event that has modifier information (shift/ctrl etc), as well as having a pointer to the node on which the menu was fired, and with which the menu was associated (which could be an ancestor of the former).

Firing a `load` event means firing a simple event called `load`. **Firing an `error` event** means firing a simple event called `error`.

Firing a progress event called *e* means something that hasn't yet been defined, in the [PROGRESS] spec.

The default action of these event is to do nothing unless otherwise stated.

If you dispatch a custom "click" event at an element that would normally have default actions, should they get triggered? If so, we need to go through the entire spec and make sure that any default actions are defined in terms of *any* event of the right type on that element, not those that are dispatched in expected ways.

5.5.6.4 Events and the `Window` object

When an event is dispatched at a DOM node in a `Document` in a browsing context, if the event is not a `load` event, the user agent must also dispatch the event to the `Window`, as follows:

1. In the capture phase, the event must be dispatched to the `Window` object before being dispatched to any of the nodes.

2. In the bubble phase, the event must be dispatched to the `Window` object at the end of the phase, unless bubbling has been prevented.

5.5.6.5 Runtime script errors

This section only applies to user agents that support scripting in general and ECMAScript in particular.

Whenever a runtime script error occurs in one of the scripts associated with a `Document`, the

user agent must report the error using the `onerror` event handler attribute of the script's global object. If the error is still *not handled* after this, then the error should be reported to the user.

When the user agent is required to **report an error** *error* using the event handler attribute *onerror*, it must run these steps, after which the error is either *handled* or *not handled*:

↪ **If the value of *onerror* is a `Function`**

The function must be invoked with three arguments. The three arguments passed to the function are all `DOMString`s; the first must give the message that the UA is considering reporting, the second must give the absolute URL of the resource in which the error occurred, and the third must give the line number in that resource on which the error occurred.

If the function returns false, then the error is *handled*. Otherwise, the error is *not handled*.

Any exceptions thrown or errors caused by this function must be reported to the user immediately after the error that the function was called for, without using the report an error algorithm again.

↪ **Otherwise**

The error is *not handled*.

5.6 User prompts

5.6.1 Simple dialogs

window **.** `alert`**(*message*)**

Displays a modal alert with the given message, and waits for the user to dismiss it.

A call to the `navigator.getStorageUpdates()` method is implied when this method is invoked.

result **= *window*** **.** `confirm`**(*message*)**

Displays a modal OK/Cancel prompt with the given message, waits for the user to dismiss it, and returns true if the user clicks OK and false if the user clicks Cancel.

A call to the `navigator.getStorageUpdates()` method is implied when this method is invoked.

result **= *window*** **.** `prompt`**(*message* [, *default*])**

Displays a modal text field prompt with the given message, waits for the user to dismiss it, and returns the value that the user entered. If the user cancels the prompt, then returns null instead. If the second argument is present, then the given value is used as a default.

A call to the `navigator.getStorageUpdates()` method is implied when this

method is invoked.

The **`alert(`*`message`*`)`** method, when invoked, must release the storage mutex and show the given *message* to the user. The user agent may make the method wait for the user to acknowledge the message before returning; if so, the user agent must pause while the method is waiting.

The **`confirm(`*`message`*`)`** method, when invoked, must release the storage mutex andshow the given *message* to the user, and ask the user to respond with a positive or negative response. The user agent must then pause as the method waits for the user's response. If the user responds positively, the method must return true, and if the user responds negatively, the method must return false.

The **`prompt(`*`message, default`*`)`** method, when invoked, must release the storage mutex, show the given *message* to the user, and ask the user to either respond with a string value or abort. The user agent must then pause as the method waits for the user's response. The second argument is optional. If the second argument (*default*) is present, then the response must be defaulted to the value given by *default*. If the user aborts, then the method must return null; otherwise, the method must return the string that the user responded with.

5.6.2 Printing

window **`.print()`**

Prompts the user to print the page.

A call to the `navigator.getStorageUpdates()` method is implied when this method is invoked.

The **`print()`** method, when invoked, must run the printing steps.

User agents should also run the printing steps whenever the user asks for the opportunity to obtain a physical form (e.g. printed copy), or the representation of a physical form (e.g. PDF copy), of a document.

The **printing steps** are as follows:

1. The user agent may display a message to the user and/or may abort these steps.

   > For instance, a kiosk browser could silently ignore any invocations of the `print()` method.

   > For instance, a browser on a mobile device could detect that there are no printers in the vicinity and display a message saying so before continuing to offer a "save to PDF" option.

2. The user agent must fire a simple event called `beforeprint` at the `Window` object of the `Document` that is being printed, as well as any nested browsing contexts in it.

> The `beforeprint` event can be used to annotate the printed copy, for instance adding the time at which the document was printed.

3. The user agent must release the storage mutex.

4. The user agent should offer the user the opportunity to obtain a physical form (or the representation of a physical form) of the document. The user agent may wait for the user to either accept or decline before returning; if so, the user agent must pause while the method is waiting. Even if the user agent doesn't wait at this point, the user agent must use the state of the relevant documents as they are at this point in the algorithm if and when it eventually creates the alternate form.

5. The user agent must fire a simple event called `afterprint` at the `Window` object of the `Document` that is being printed, as well as any nested browsing contexts in it.

> The `afterprint` event can be used to revert annotations added in the earlier event, as well as showing post-printing UI. For instance, if a page is walking the user through the steps of applying for a home loan, the script could automatically advance to the next step after having printed a form or other.

5.6.3 Dialogs implemented using separate documents

***result* = *window* . `showModalDialog`(*url* [, *argument*])**

Prompts the user with the given page, waits for that page to close, and returns the return value.

A call to the `navigator.getStorageUpdates()` method is implied when this method is invoked.

The **`showModalDialog(`*`url, argument`*`)`** method, when invoked, must cause the user agent to run the following steps:

1. Resolve *url* relative to the first script's base URL.

   If this fails, then throw a `SYNTAX_ERR` exception and abort these steps.

2. Release the storage mutex.

3. If the user agent is configured such that this invocation of `showModalDialog()` is somehow disabled, then the method returns the empty string; abort these steps.

   Note: User agents are expected to disable this method in certain cases to avoid user annoyance (e.g. as part of their popup blocker feature). For instance, a user agent could require that a site be white-listed before enabling this method, or the user agent could be configured to only allow one modal dialog at a time.

4. Let *the list of background browsing contexts* be a list of all the browsing contexts that:

- are part of the same unit of related browsing contexts as the browsing context of the `Window` object on which the `showModalDialog()` method was called, and that

- have an active document whose origin is the same as the origin of the script that called the `showModalDialog()` method at the time the method was called,

...as well as any browsing contexts that are nested inside any of the browsing contexts matching those conditions.

5. Disable the user interface for all the browsing contexts in *the list of background browsing contexts*. This should prevent the user from navigating those browsing contexts, causing events to to be sent to those browsing context, or editing any content in those browsing contexts. However, it does not prevent those browsing contexts from receiving events from sources other than the user, from running scripts, from running animations, and so forth.

6. Create a new auxiliary browsing context, with the opener browsing context being the browsing context of the `Window` object on which the `showModalDialog()` method was called. The new auxiliary browsing context has no name.

   Note: This browsing context's `Document`s' `Window` objects all implement the `WindowModal` interface.

7. Let the dialog arguments of the new browsing context be set to the value of *argument*, or the 'undefined' value if the argument was omitted.

8. Let the dialog arguments' origin be the origin of the script that called the `showModalDialog()` method.

9. Navigate the new browsing context to the absolute URL that resulted from resolving *url* earlier, with replacement enabled, and with the browsing context of the script that invoked the method as the source browsing context.

10. Wait for the browsing context to be closed. (The user agent must allow the user to indicate that the browsing context is to be closed.)

11. Reenable the user interface for all the browsing contexts in *the list of background browsing contexts*.

12. Return the auxiliary browsing context's return value.

The `Window` objects of `Document`s hosted by browsing contexts created by the above algorithm must all implement the `WindowModal` interface:

Really I want the Window object to just gain these attributes, as if they were on the Window prototype. That's the XXX below.

```
[NoInterfaceObject, ImplementedOn=Window, XXX] interface WindowModal {
  readonly attribute any dialogArguments;
           attribute DOMString returnValue;
};
```

window **.** **`dialogArguments`**

Returns the *argument* argument that was passed to the `showModalDialog()` method.

window **.** **`returnValue`** **[= *value*]**

Returns the current return value for the window.

Can be set, to change the value that will be returned by the `showModalDialog()` method.

Such browsing contexts have associated **dialog arguments**, which are stored along with the **dialog arguments' origin**. These values are set by the `showModalDialog()` method in the algorithm above, when the browsing context is created, based on the arguments provided to the method.

The **`dialogArguments`** DOM attribute, on getting, must check whether its browsing context's active document's origin is the same as the dialog arguments' origin. If it is, then the browsing context's dialog arguments must be returned unchanged. Otherwise, if the dialog arguments are an object, then the empty string must be returned, and if the dialog arguments are not an object, then the stringification of the dialog arguments must be returned.

These browsing contexts also have an associated **return value**. The return value of a browsing context must be initialized to the empty string when the browsing context is created.

The **`returnValue`** DOM attribute, on getting, must return the return value of its browsing context, and on setting, must set the return value to the given new value.

Note: The `window.close()` method can be used to close the browsing context.

5.7 System state and capabilities

The **`navigator`** attribute of the `Window` interface must return an instance of the `Navigator` interface, which represents the identity and state of the user agent (the client), and allows Web pages to register themselves as potential protocol and content handlers:

```
interface Navigator {
  // objects implementing this interface also implement the interfaces given below
};

[NoInterfaceObject, ImplementedOn=Navigator] interface NavigatorID {
  readonly attribute DOMString appName;
  readonly attribute DOMString appVersion;
  readonly attribute DOMString platform;
  readonly attribute DOMString userAgent;
};

[NoInterfaceObject, ImplementedOn=Navigator] interface NavigatorOnLine {
  readonly attribute boolean onLine;
```

```
};

[NoInterfaceObject, ImplementedOn=Navigator] interface NavigatorAbilities {
  // content handler registration
  void registerProtocolHandler(in DOMString protocol, in DOMString url, in DOMStri
  void registerContentHandler(in DOMString mimeType, in DOMString url, in DOMStrin
  void getStorageUpdates();
};
```

Objects implementing the `Navigator` interface must also implement the NavigatorID, NavigatorOnLine, and NavigatorAbilities interfaces. (These interfaces are defined separately so that other specifications can re-use parts of the `Navigator` interface.)

5.7.1 Client identification

In certain cases, despite the best efforts of the entire industry, Web browsers have bugs and limitations that Web authors are forced to work around.

This section defines a collection of attributes that can be used to determine, from script, the kind of user agent in use, in order to work around these issues.

Client detection should always be limited to detecting known current versions; future versions and unknown versions should always be assumed to be fully compliant.

window **`.navigator.appName`**

Returns the name of the browser.

window **`.navigator.appVersion`**

Returns the version of the browser.

window **`.navigator.platform`**

Returns the name of the platform.

window **`.navigator.userAgent`**

Returns the complete User-Agent header.

`appName`

Must return either the string "`Netscape`" or the full name of the browser, e.g. "`Mellblom Browsernator`".

`appVersion`

Must return either the string "`4.0`" or a string representing the version of the browser in detail, e.g. "`1.0 (VMS; en-US) Mellblomenator/9000`".

`platform`

Must return either the empty string or a string representing the platform on which the browser is executing, e.g. "`MacIntel`", "`Win32`", "`FreeBSD i386`", "`WebTV OS`".

`userAgent`

Must return the string used for the value of the "`User-Agent`" header in HTTP requests, or the empty string if no such header is ever sent.

5.7.2 Custom protocol and content handlers

The **`registerProtocolHandler()`** method allows Web sites to register themselves as possible handlers for particular protocols. For example, an online fax service could register itself as a handler of the `fax:` protocol ([RFC2806]), so that if the user clicks on such a link, he is given the opportunity to use that Web site. Analogously, the **`registerContentHandler()`** method allows Web sites to register themselves as possible handlers for content in a particular MIME type. For example, the same online fax service could register itself as a handler for `image/g3fax` files ([RFC1494]), so that if the user has no native application capable of handling G3 Facsimile byte streams, his Web browser can instead suggest he use that site to view the image.

window **`.navigator.registerProtocolHandler(`*protocol*`, `*url*`, `*title*`)**
window **`.navigator.registerContentHandler(`*mimeType*`, `*url*`, `*title*`)**

Registers a handler for the given protocol or content type, at the given URL, with the given title.

The string "`%s`" in the URL is used as a placeholder for where to put the URL of the content to be handled.

Throws a `SECURITY_ERR` exception if the user agent blocks the registration (this might happen if trying to register as a handler for "http", for instance).

Throws a `SYNTAX_ERR` if the "`%s`" string is missing in the URL.

User agents may, within the constraints described in this section, do whatever they like when the methods are called. A UA could, for instance, prompt the user and offer the user the opportunity to add the site to a shortlist of handlers, or make the handlers his default, or cancel the request. UAs could provide such a UI through modal UI or through a non-modal transient notification interface. UAs could also simply silently collect the information, providing it only when relevant to the user.

There is an example of how these methods could be presented to the user below.

The arguments to the methods have the following meanings and corresponding implementaion requirements:

protocol **(`registerProtocolHandler()` only)**
A scheme, such as `ftp` or `fax`. The scheme must be compared in an ASCII case-insensitive manner by user agents for the purposes of comparing with the scheme part of URLs that they consider against the list of registered handlers.

The *protocol* value, if it contains a colon (as in "`ftp:`"), will never match anything, since schemes don't contain colons.

mimeType **(`registerContentHandler()` only)**
A MIME type, such as `model/vrml` or `text/richtext`. The MIME type must be compared

in an ASCII case-insensitive manner by user agents for the purposes of comparing with MIME types of documents that they consider against the list of registered handlers.

User agents must compare the given values only to the MIME type/subtype parts of content types, not to the complete type including parameters. Thus, if *mimeType* values passed to this method include characters such as commas or whitespace, or include MIME parameters, then the handler being registered will never be used.

url

The URL of the page that will handle the requests.

When the user agent uses this URL, it must replace the first occurrence of the exact literal string "`%s`" with an escaped version of the absolute URL of the content in question (as defined below), then resolve the resulting URL, relative to the base URL of the first script at the time the `registerContentHandler()` or `registerProtocolHandler()` methods were invoked, and then fetch the resulting URL using the GET method (or equivalent for non-HTTP URLs).

To get the escaped version of the absolute URL of the content in question, the user agent must replace every character in that absolute URL that doesn't match the <query> production defined in RFC 3986 by the percent-encoded form of that character. [RFC3986]

> If the user had visited a site at `http://example.com/` that made the following call:
>
> ```
> navigator.registerContentHandler('application/x-soup', 'soup?url=%s', 'S
> ```
>
> ...and then, much later, while visiting `http://www.example.net/`, clicked on a link such as:
>
> ```
> <a href="chickenkïwi.soup">Download our Chicken Kiwi soup!</a>
> ```
>
> ...then, assuming this `chickenkïwi.soup` file was served with the MIME type `application/x-soup`, the UA might navigate to the following URL:
>
> ```
> http://example.com/soup?url=http://www.example.net/chickenk%C3%AFwi.soup
> ```
>
> This site could then fetch the `chickenkïwi.soup` file and do whatever it is that it does with soup (synthesize it and ship it to the user, or whatever).

title

A descriptive title of the handler, which the UA might use to remind the user what the site in question is.

User agents should raise `SECURITY_ERR` exceptions if the methods are called with *protocol* or *mimeType* values that the UA deems to be "privileged". For example, a site attempting to register a handler for `http` URLs or `text/html` content in a Web browser would likely cause an exception to be raised.

User agents must raise a `SYNTAX_ERR` exception if the *url* argument passed to one of these methods does not contain the exact literal string "`%s`", or if resolving the *url* argument with the first occurance of the string "`%s`" removed, relative to the first script's base URL, is not successful.

User agents must not raise any other exceptions (other than binding-specific exceptions, such

as for an incorrect number of arguments in an ECMAScript implementation).

This section does not define how the pages registered by these methods are used, beyond the requirements on how to process the *url* value (see above). To some extent, the processing model for navigating across documents defines some cases where these methods are relevant, but in general UAs may use this information wherever they would otherwise consider handing content to native plugins or helper applications.

UAs must not use registered content handlers to handle content that was returned as part of a non-GET transaction (or rather, as part of any non-idempotent transaction), as the remote site would not be able to fetch the same data.

5.7.2.1 Security and privacy

These mechanisms can introduce a number of concerns, in particular privacy concerns.

Hijacking all Web usage. User agents should not allow protocols that are key to its normal operation, such as `http` or `https`, to be rerouted through third-party sites. This would allow a user's activities to be trivially tracked, and would allow user information, even in secure connections, to be collected.

Hijacking defaults. It is strongly recommended that user agents do not automatically change any defaults, as this could lead the user to send data to remote hosts that the user is not expecting. New handlers registering themselves should never automatically cause those sites to be used.

Registration spamming. User agents should consider the possibility that a site will attempt to register a large number of handlers, possibly from multiple domains (e.g. by redirecting through a series of pages each on a different domain, and each registering a handler for `video/mpeg` — analogous practices abusing other Web browser features have been used by pornography Web sites for many years). User agents should gracefully handle such hostile attempts, protecting the user.

Misleading titles. User agents should not rely wholly on the *title* argument to the methods when presenting the registered handlers to the user, since sites could easily lie. For example, a site `hostile.example.net` could claim that it was registering the "Cuddly Bear Happy Content Handler". User agents should therefore use the handler's domain in any UI along with any title.

Hostile handler metadata. User agents should protect against typical attacks against strings embedded in their interface, for example ensuring that markup or escape characters in such strings are not executed, that null bytes are properly handled, that over-long strings do not cause crashes or buffer overruns, and so forth.

Leaking Intranet URLs. The mechanism described in this section can result in secret Intranet URLs being leaked, in the following manner:

1. The user registers a third-party content handler as the default handler for a content type.
2. The user then browses his corporate Intranet site and accesses a document that uses that content type.

3. The user agent contacts the third party and hands the third party the URL to the Intranet content.

No actual confidential file data is leaked in this manner, but the URLs themselves could contain confidential information. For example, the URL could be `http://www.corp.example.com/upcoming-aquisitions/the-sample-company.egf`, which might tell the third party that Example Corporation is intending to merge with The Sample Company. Implementors might wish to consider allowing administrators to disable this feature for certain subdomains, content types, or protocols.

Leaking secure URLs. User agents should not send HTTPS URLs to third-party sites registered as content handlers, in the same way that user agents do not send `Referer` headers from secure sites to third-party sites.

Leaking credentials. User agents must never send username or password information in the URLs that are escaped and included sent to the handler sites. User agents may even avoid attempting to pass to Web-based handlers the URLs of resources that are known to require authentication to access, as such sites would be unable to access the resources in question without prompting the user for credentials themselves (a practice that would require the user to know whether to trust the third-party handler, a decision many users are unable to make or even understand).

5.7.2.2 Sample user interface

This section is non-normative.

A simple implementation of this feature for a desktop Web browser might work as follows.

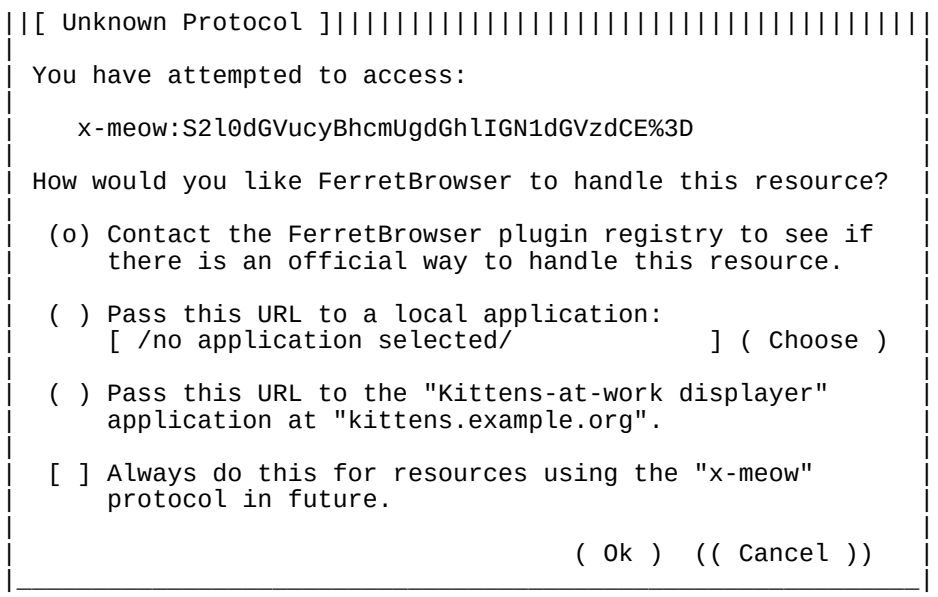
The `registerProtocolHandler()` method could display a modal dialog box:

```
||[ Protocol Handler Registration ]||||||||||||||||||||||||||||
|                                                              |
| This Web page:                                               |
|                                                              |
|    Kittens at work                                           |
|    http://kittens.example.org/                               |
|                                                              |
| ...would like permission to handle the protocol "x-meow:"    |
| using the following Web-based application:                   |
|                                                              |
|    Kittens-at-work displayer                                 |
|    http://kittens.example.org/?show=%s                       |
|                                                              |
| Do you trust the administrators of the "kittens.example.     |
| org" domain?                                                 |
|                                                              |
|              ( Trust kittens.example.org )  (( Cancel ))     |
|______________________________________________________________|
```

...where "Kittens at work" is the title of the page that invoked the method, "http://kittens.example.org/" is the URL of that page, "x-meow" is the string that was passed to the `registerProtocolHandler()` method as its first argument (*protocol*), "http://kittens.example.org/?show=%s" was the second argument (*url*), and "Kittens-at-work displayer" was the third argument (*title*).

If the user clicks the Cancel button, then nothing further happens. If the user clicks the "Trust" button, then the handler is remembered.

When the user then attempts to fetch a URL that uses the "x-meow:" scheme, then it might display a dialog as follows:



...where the third option is the one that was primed by the site registering itself earlier.

If the user does select that option, then the browser, in accordance with the requirements described in the previous two sections, will redirect the user to "http://kittens.example.org/?show=x-meow%3AS2l0dGVucyBhcmUgdGhlIGN1dGVzdCE%253D'

The `registerContentHandler()` method would work equivalently, but for unknown MIME types instead of unknown protocols.

5.7.3 Manually releasing the storage mutex

window **. navigator . getStorageUpdates()**

If a script uses the `document.cookie` API, or the `localStorage` API, the browser will block other scripts from accessing cookies or storage until the first script finishes.

Calling the `navigator.getStorageUpdates()` method tells the user agent to unblock any other scripts that may be blocked, even though the script hasn't returned.

Values of cookies and items in the `Storage` objects of `localStorage` attributes can change after calling this method, whence its name.

The **`getStorageUpdates()`** method, when invoked, must, if the storage mutex is owned by the event loop of the task that resulted in the method being called, release the storage mutex so that it is once again free. Otherwise, it must do nothing.

5.8 Offline Web applications

5.8.1 Introduction

This section is non-normative.

...

5.8.2 Application caches

An **application cache** is a set of cached resources consisting of:

- One of more resources (including their out-of-band metadata, such as HTTP headers, if any), identified by URLs, each falling into one (or more) of the following categories:

  Master entries
  : Documents that were added to the cache because a browsing context was navigated to that document and the document indicated that this was its cache, using the `manifest` attribute.

  The manifest
  : The resource corresponding to the URL that was given in a master entry's `html` element's `manifest` attribute. The manifest is fetched and processed during the application cache update process. All the master entries have the same origin as the manifest.

  Explicit entries
  : Resources that were listed in the cache's manifest. Explicit entries can also be marked as **foreign**, which means that they have a `manifest` attribute but that it doesn't point at this cache's manifest.

  Fallback entries
  : Resources that were listed in the cache's manifest as fallback entries.

  Note: A URL in the list can be flagged with multiple different types, and thus an entry can end up being categorized as multiple entries. For example, an entry can be a manfiest entry and an explicit entry at the same time, if the manifest is listed within the manifest.

- Zero or more **fallback namespaces**: URLs, used as prefix match patterns, each of which is mapped to a fallback entry. Each namespace URL has the same origin as the manifest.

- Zero or more URLs that form the **online whitelist namespaces**.

Each application cache has a **completeness flag**, which is either *complete* or *incomplete*.

An **application cache group** is a group of application caches, identified by the absolute URL of a resource manifest which is used to populate the caches in the group.

An application cache is **newer** than another if it was created after the other (in other words, application caches in an application cache group have a chronological order).

Only the newest application cache in an application cache group can have its completeness flag set to *incomplete*, the others are always all *complete*.

Each application cache group has an **update status**, which is one of the following: *idle*, *checking*, *downloading*.

A **relevant application cache** is an application cache that is the newest in its group to be *complete*.

Each application cache group has a **list of pending master entries**. Each entry in this list consists of a resource and a corresponding `Document` object. It is used during the update process to ensure that new master entries are cached.

An application cache group can be marked as **obsolete**, meaning that it must be ignored when looking at what application cache groups exist.

A **cache host** is a `Document` or a `SharedWorkerGlobalScope` object. A cache host can be associated with an application cache.

A `Document` initially is not associated with an application cache, buu can become associated with one early during the page load process, when steps in the parser and in the navigation sections that cause cache selection to occur.

A `SharedWorkerGlobalScope` is can be associated with an application cache when it is created.

Each cache host has an associated `ApplicationCache` object.

Multiple application caches in different application cache groups can contain the same resource, e.g. if the manifests all reference that resource. If the user agent is to **select an application cache** from a list of relevant application caches that contain a resource, that the user agent must use the application cache that the user most likely wants to see the resource from, taking into account the following:

- which application cache was most recently updated,
- which application cache was being used to display the resource from which the user decided to look at the new resource, and
- which application cache the user prefers.

5.8.3 The cache manifest syntax

5.8.3.1 A sample manifest

This section is non-normative.

This example manifest requires two images and a style sheet to be cached and whitelists a CGI script.

```
CACHE MANIFEST
# the above line is required

# this is a comment
# there can be as many of these anywhere in the file
# they are all ignored
  # comments can have spaces before them
  # but must be alone on the line

# blank lines are ignored too

# these are files that need to be cached they can either be listed
# first, or a "CACHE:" header could be put before them, as is done
# lower down.
images/sound-icon.png
images/background.png
# note that each file has to be put on its own line

# here is a file for the online whitelist -- it isn't cached, and
# references to this file will bypass the cache, always hitting the
# network (or trying to, if the user is offline).
NETWORK:
comm.cgi

# here is another set of files to cache, this time just the CSS file.
CACHE:
style/default.css
```

5.8.3.2 Writing cache manifests

Manifests must be served using the `text/cache-manifest` MIME type. All resources served using the `text/cache-manifest` MIME type must follow the syntax of application cache manifests, as described in this section.

An application cache manifest is a text file, whose text is encoded using UTF-8. Data in application cache manifests is line-based. Newlines must be represented by U+000A LINE FEED (LF) characters, U+000D CARRIAGE RETURN (CR) characters, or U+000D CARRIAGE RETURN (CR) U+000A LINE FEED (LF) pairs.

Note: This is a willful double violation of RFC2046. [RFC2046]

The first line of an application cache manifest must consist of the string "CACHE", a single U+0020 SPACE character, the string "MANIFEST", and either a U+0020 SPACE character, a U+0009 CHARACTER TABULATION (tab) character, a U+000A LINE FEED (LF) character, or a U+000D CARRIAGE RETURN (CR) character. The first line may optionally be preceded by a U+FEFF BYTE ORDER MARK (BOM) character. If any other text is found on the first line, it is ignored.

Subsequent lines, if any, must all be one of the following:

A blank line

Blank lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters only.

A comment

Comment lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, followed by a single U+0023 NUMBER

SIGN (#) character, followed by zero or more characters other than U+000A LINE FEED (LF) and U+000D CARRIAGE RETURN (CR) characters.

Note: Comments must be on a line on their own. If they were to be included on a line with a URL, the "#" would be mistaken for part of a fragment identifier.

A section header

Section headers change the current section. There are three possible section headers:

`CACHE:`
: Switches to the explicit section.

`FALLBACK:`
: Switches to the fallback section.

`NETWORK:`
: Switches to the online whitelist section.

Section header lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, followed by one of the names above (including the U+003A COLON (:) character) followed by zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters.

Ironically, by default, the current section is the explicit section.

Data for the current section

The format that data lines must take depends on the current section.

When the current section is the explicit section or the online whitelist section, data lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, a valid URL identifying a resource other than the manifest itself, and then zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters.

When the current section is the fallback section, data lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, a valid URL identifying a resource other than the manifest itself, one or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, another valid URL identifying a resource other than the manifest itself, and then zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters.

Note: The URLs in data lines can't be empty strings, since those would be relative URLs to the manifest itself. Such lines would be confused with blank or invalid lines, anyway.

Manifests may contain sections more than once. Sections may be empty.

URLs that are to be fallback pages associated with fallback namespaces, and those namespaces themselves, must be given in fallback sections, with the namespace being the first URL of the data line, and the corresponding fallback page being the second URL. All the other pages to be cached must be listed in explicit sections.

Fallback namespaces and fallback entries must have the same origin as the manifest itself.

A fallback namespace must not be listed more than once.

URLs that the user agent is to put into the online whitelist must all be specified in online whitelist sections. (This is needed for any URL that the page is intending to use to communicate back to the server.)

Relative URLs must be given relative to the manifest's own URL.

URLs in manifests must not have fragment identifiers (i.e. the U+0023 NUMBER SIGN character isn't allowed in URLs in manifests).

5.8.3.3 Parsing cache manifests

When a user agent is to **parse a manifest**, it means that the user agent must run the following steps:

1. The user agent must decode the byte stream corresponding with the manifest to be parsed, treating it as UTF-8. Bytes or sequences of bytes that are not valid UTF-8 sequences must be interpreted as a U+FFFD REPLACEMENT CHARACTER.
2. Let *base URL* be the absolute URL representing the manifest.
3. Let *explicit URLs* be an initially empty list of explicit entries.
4. Let *fallback URLs* be an initially empty mapping of fallback namespaces to fallback entries.
5. Let *online whitelist URLs* be an initially empty list of URLs for a online whitelist.
6. Let *input* be the decoded text of the manifest's byte stream.
7. Let *position* be a pointer into *input*, initially pointing at the first character.
8. If *position* is pointing at a U+FEFF BYTE ORDER MARK (BOM) character, then advance *position* to the next character.
9. If the characters starting from *position* are "CACHE", followed by a U+0020 SPACE character, followed by "MANIFEST", then advance *position* to the next character after those. Otherwise, this isn't a cache manifest; abort this algorithm with a failure while checking for the magic signature.
10. If the character at *position* is neither a U+0020 SPACE character, a U+0009 CHARACTER TABULATION (tab) character, U+000A LINE FEED (LF) character, nor a U+000D CARRIAGE RETURN (CR) character, then this isn't a cache manifest; abort this algorithm with a failure while checking for the magic signature.
11. This is a cache manifest. The algorithm cannot fail beyond this point (though bogus lines can get ignored).
12. Collect a sequence of characters that are *not* U+000A LINE FEED (LF) or U+000D

CARRIAGE RETURN (CR) characters, and ignore those characters. (Extra text on the first line, after the signature, is ignored.)

13. Let *mode* be "explicit".

14. *Start of line*: If *position* is past the end of *input*, then jump to the last step. Otherwise, collect a sequence of characters that are U+000A LINE FEED (LF), U+000D CARRIAGE RETURN (CR), U+0020 SPACE, or U+0009 CHARACTER TABULATION (tab) characters.

15. Now, collect a sequence of characters that are *not* U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters, and let the result be *line*.

16. Drop any trailing U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters at the end of *line*.

17. If *line* is the empty string, then jump back to the step labeled "start of line".

18. If the first character in *line* is a U+0023 NUMBER SIGN (#) character, then jump back to the step labeled "start of line".

19. If *line* equals "CACHE:" (the word "CACHE" followed by a U+003A COLON (:) character), then set *mode* to "explicit" and jump back to the step labeled "start of line".

20. If *line* equals "FALLBACK:" (the word "FALLBACK" followed by a U+003A COLON (:) character), then set *mode* to "fallback" and jump back to the step labeled "start of line".

21. If *line* equals "NETWORK:" (the word "NETWORK" followed by a U+003A COLON (:) character), then set *mode* to "online whitelist" and jump back to the step labeled "start of line".

22. If *line* ends with a U+003A COLON (:) character, then set *mode* to "unknown" and jump back to the step labeled "start of line".

23. This is either a data line or it is syntactically incorrect.

24. Let *position* be a pointer into *line*, initially pointing at the start of the string.

25. Let *tokens* be a list of strings, initially empty.

26. While *position* doesn't point past the end of *line*:

    1. Let *current token* be an empty string.

    2. While *position* doesn't point past the end of *line* and the character at *position* is neither a U+0020 SPACE nor a U+0009 CHARACTER TABULATION (tab) character, add the character at *position* to *current token* and advance *position* to the next character in *input*.

    3. Add *current token* to the *tokens* list.

    4. While *position* doesn't point past the end of *line* and the character at *position* is either a U+0020 SPACE or a U+0009 CHARACTER TABULATION (tab) character, advance *position* to the next character in *input*.

27. Process *tokens* as follows:

    ↪ **If *mode* is "explicit"**

    Resolve the first item in *tokens*, relative to *base URL*; ignore the rest.

    If this fails, then jump back to the step labeled "start of line".

    If the resulting absolute URL has a different <scheme> component than the manifest's URL (compared in an ASCII case-insensitive manner), then jump back to the step labeled "start of line".

    Drop the <fragment> component of the resulting absolute URL, if it has one.

    Add the resulting absolute URL to the *explicit URLs*.

    ↪ **If *mode* is "fallback"**

    Let *part one* be the first token in *tokens*, and let *part two* be the second token in *tokens*.

    Resolve *part one* and *part two*, relative to *base URL*.

    If either fails, then jump back to the step labeled "start of line".

    If the absolute URL corresponding to either *part one* or *part two* does not have the same origin as the manifest's URL, then jump back to the step labeled "start of line".

    Drop any the <fragment> components of the resulting absolute URLs.

    If the absolute URL corresponding to *part one* is already in the *fallback URLs* mapping as a fallback namespace, then jump back to the step labeled "start of line".

    Otherwise, add the absolute URL corresponding to *part one* to the *fallback URLs* mapping as a fallback namespace, mapped to the absolute URL corresponding to *part two* as the fallback entry.

    ↪ **If *mode* is "online whitelist"**

    Resolve the first item in *tokens*, relative to *base URL*; ignore the rest.

    If this fails, then jump back to the step labeled "start of line".

    If the resulting absolute URL has a different <scheme> component than the manifest's URL (compared in an ASCII case-insensitive manner), then jump back to the step labeled "start of line".

    Drop the <fragment> component of the resulting absolute URL, if it has one.

    Add the resulting absolute URL to the *online whitelist URLs*.

    ↪ **If *mode* is "unknown"**

    Do nothing. The line is ignored.

28. Jump back to the step labeled "start of line". (That step jumps to the next, and last, step

when the end of the file is reached.)

29. Return the *explicit URLs* list, the *fallback URLs* mapping, and the *online whitelist URLs*.

Note: If a resource is listed in the explicit section and matches an entry in the online whitelist, or if a resource matches both an entry in the fallback section and the online whitelist, the resource will taken from the cache, and the online whitelist entry will be ignored.

5.8.4 Updating an application cache

When the user agent is required (by other parts of this specification) to start the **application cache update process** for an absolute URL purported to identify a manifest, or for an application cache group, potentially given a particular cache host, and potentially given a new master resource, the user agent must run the following steps:

1. Atomically, so as to avoid race conditions, perform the following substeps:

    1. Pick the approprate substeps:

        ↪ **If these steps were invoked with an absolute URL purported to identify a manifest**

        Let *manifest URL* be that absolute URL.

        If there is no application cache group identified by *manifest URL*, then create a new application cache group identified by *manifest URL*. Initially it has no application caches, though one will be created later in this algorithm.

        ↪ **If these steps were invoked with an application cache group**

        Let *manifest URL* be the absolute URL of the manifest used to identify the application cache group to be updated.

    2. Let *cache group* be the application cache group identified by *manifest URL*.

    3. If these steps were invoked with a new master resource, then add the resource, along with the resource's `Document`, to *cache group*'s list of pending master entries.

    4. If these steps were invoked with a cache host, and the status of *cache group* is *checking* or *downloading*, then queue a task to fire a simple event called `checking` at the `ApplicationCache` singleton of that cache host.

    5. If these steps were invoked with a cache host, and the status of *cache group* is *downloading*, then also queue a task to fire a simple event called `downloading` at the `ApplicationCache` singleton of that cache host.

    6. If the status of the *cache group* is either *checking* or *downloading*, then abort this instance of the update process, as an update is already in progress for them.

    7. Set the status of *cache group* to *checking*.

    8. For each cache host associated with an application cache in *cache group*, queue a

task to fire a simple event called `checking` at the `ApplicationCache` singleton of the cache host. The default action of these events should be the display of some sort of user interface indicating to the user that the user agent is checking for the availability of updates.

The remainder of the steps run asychronously.

If *cache group* already has an application cache in it, then this is an **upgrade attempt**. Otherwise, this is a **cache attempt**.

2. If this is a cache attempt, then this algorithm was invoked with a cache host; queue a task to fire a simple event called `checking` at the `ApplicationCache` singleton of that cache host.

3. *Fetching the manifest*: Fetch the resource from *manifest URL*, and let *manifest* be that resource.

   If the resource is labeled with the MIME type `text/cache-manifest`, parse *manifest* according to the rules for parsing manifests, obtaining a list of explicit entries, fallback entries and the fallback namespaces that map to them, and entries for the online whitelist.

4. If *fetching the manifest* fails due to a 404 or 410 response or equivalent, then run these substeps:

   1. Mark *cache group* as obsolete. This *cache group* no longer exists for any purpose other than the processing of `Document` objects already associated with an application cache in the *cache group*.

   2. For each cache host associated with an application cache in *cache group*, queue a task to fire a simple event called `obsolete` at the `ApplicationCache` singleton of the cache host. The default action of these events should be the display of some sort of user interface indicating to the user that the application is no longer available for offline use.

   3. For each entry in *cache group*'s list of pending master entries, queue a task to fire a simple event called `error` (not `obsolete`!) at the `ApplicationCache` singleton of the cache host the `Document` for this entry, if there still is one. The default action of this event should be the display of some sort of user interface indicating to the user that the user agent failed to save the application for offline use.

   4. If *cache group* has an application cache whose completeness flag is *incomplete*, then discard that application cache.

   5. If appropriate, remove any user interface indicating that an update for this cache is in progress.

   6. Let the status of *cache group* be *idle*.

   7. Abort the update process.

5. Otherwise, if *fetching the manifest* fails in some other way (e.g. the server returns another 4xx or 5xx response or equivalent, or there is a DNS error, or the connection times out, or the user cancels the download, or the parser for manifests fails when

checking the magic signature), or if the server returned a redirect, or if the resource is labeled with a MIME type other than `text/cache-manifest`, then run the cache failure steps.

6. If this is an upgrade attempt and the newly downloaded *manifest* is byte-for-byte identical to the manifest found in the newest application cache in *cache group*, or the server reported it as "304 Not Modified" or equivalent, then run these substeps:

    1. Let *cache* be the newest application cache in *cache group*.

    2. For each entry in *cache group*'s list of pending master entries, wait for the resource for this entry to have either completely downloaded or failed.

        If the download failed (e.g. the connection times out, or the user cancels the download), then queue a task to fire a simple event called `error` at the `ApplicationCache` singleton of the cache host the `Document` for this entry, if there still is one. The default action of this event should be the display of some sort of user interface indicating to the user that the user agent failed to save the application for offline use.

        Otherwise, associate the `Document` for this entry with *cache*; store the resource for this entry in *cache*, if it isn't already there, and categorize its entry as a master entry.

        Note: HTTP caching rules, such as `Cache-Control: no-store`, are ignored for the purposes of the application cache update process.

    3. For each cache host associated with an application cache in *cache group*, queue a task to fire a simple event called `noupdate` at the `ApplicationCache` singleton of the cache host. The default action of these events should be the display of some sort of user interface indicating to the user that the application is up to date.

    4. Empty *cache group*'s list of pending master entries.

    5. If appropriate, remove any user interface indicating that an update for this cache is in progress.

    6. Let the status of *cache group* be *idle*.

    7. Abort the update process.

7. Let *new cache* be a newly created application cache in *cache group*. Set its completeness flag to *incomplete*.

8. For each entry in *cache group*'s list of pending master entries, associate the `Document` for this entry with *new cache*.

9. Set the status of *cache group* to *downloading*.

10. For each cache host associated with an application cache in *cache group*, queue a task to fire a simple event called `downloading` at the `ApplicationCache` singleton of the cache host. The default action of these events should be the display of some sort of user interface indicating to the user that a new version is being downloaded.

11. Let *file list* be an empty list of URLs with flags.

12. Add all the URLs in the list of explicit entries obtained by parsing *manifest* to *file list*, each flagged with "explicit entry".

13. Add all the URLs in the list of fallback entries obtained by parsing *manifest* to *file list*, each flagged with "fallback entry".

14. If this is an upgrade attempt, then add all the URLs of master entries in the newest application cache in *cache group* whose completeness flag is *complete* to *file list*, each flagged with "master entry".

15. If any URL is in *file list* more than once, then merge the entries into one entry for that URL, that entry having all the flags that the original entries had.

16. For each URL in *file list*, run the following steps. These steps may be run in parallel for two or more of the URLs at a time.

    1. If the resource URL being processed was flagged as neither an "explicit entry" nor or a "fallback entry", then the user agent may skip this URL.

       Note: This is intended to allow user agents to expire resources not listed in the manifest (other than those in the manifest itself) from the cache. Generally, implementors are urged to use an approach that expires lesser-used resources first.

    2. For each cache host associated with an application cache in *cache group*, queue a task to fire a simple event called `progress` at the `ApplicationCache` singleton of the cache host. The default action of these events should be the display of some sort of user interface indicating to the user that a file is being downloaded in preparation for updating the application.

    3. Fetch the resource. If this is an upgrade attempt, then use the newest application cache in *cache group* as an HTTP cache, and honor HTTP caching semantics (such as expiration, ETags, and so forth) with respect to that cache. User agents may also have other caches in place that are also honored.

       Note: If the resource in question is already being downloaded for other reasons then the existing download process can be used for the purposes of this step, as defined by the fetching algorithm.

       > An example of a resource that might already be being downloaded is a large image on a Web page that is being seen for the first time. The image would get downloaded to satisfy the `img` element on the page, as well as being listed in the cache manifest. According to the rules for fetching that image only need be downloaded once, and it can be used both for the cache and for the rendered Web page.

    4. If the previous step fails (e.g. the server returns a 4xx or 5xx response or equivalent, or there is a DNS error, or the connection times out, or the user cancels the download), or if the server returned a redirect, then run the first appropriate step from the following list:

↪ **If the URL being processed was flagged as an "explicit entry" or a "fallback entry"**

Run the cache failure steps.

Note: Redirects are fatal because they are either indicative of a network problem (e.g. a captive portal); or would allow resources to be added to the cache under URLs that differ from any URL that the networking model will allow access to, leaving orphan entries; or would allow resources to be stored under URLs different than their true URLs. All of these situations are bad.

↪ **If the error was a 404 or 410 HTTP response or equivalent**

Skip this resource. It is dropped from the cache.

↪ **Otherwise**

Copy the resource and its metadata from the newest application cache in *cache group* whose completeness flag is *complete*, and act as if that was the fetched resource, ignoring the resource obtained from the network.

User agents may warn the user of these errors as an aid to development.

Note: These rules make errors for resources listed in the manifest fatal, while making it possible for other resources to be removed from caches when they are removed from the server, without errors, and making non-manifest resources survive server-side errors.

5. Otherwise, the fetching succeeded. Store the resource in the *new cache*.
6. If the URL being processed was flagged as an "explicit entry" in *file list*, then categorize the entry as an explicit entry.
7. If the URL being processed was flagged as a "fallback entry" in *file list*, then categorize the entry as a fallback entry.
8. If the URL being processed was flagged as an "master entry" in *file list*, then categorize the entry as a master entry.
9. As an optimization, if the resource is an HTML or XML file whose root element is an `html` element with a `manifest` attribute whose value doesn't match the manifest URL of the application cache being processed, then the user agent should mark the entry as being foreign.

17. Store the list of fallback namespaces, and the URLs of the fallback entries that they map to, in *new cache*.
18. Store the URLs that form the new online whitelist in *new cache*.
19. For each entry in *cache group*'s list of pending master entries, wait for the resource for this entry to have either completely downloaded or failed.

If the download failed (e.g. the connection times out, or the user cancels the download), then run these sebsteps:

1. Unassociate the `Document` for this entry from *new cache*.

2. Queue a task to fire a simple event called `error` at the `ApplicationCache` singleton of the `Document` for this entry, if there still is one. The default action of this event should be the display of some sort of user interface indicating to the user that the user agent failed to save the application for offline use.

3. If this is a cache attempt and this entry is the last entry in *cache group*'s list of pending master entries, then run these further substeps:

    1. Discard *cache group* and its only application cache, *new cache*.

    2. If appropriate, remove any user interface indicating that an update for this cache is in progress.

    3. Abort the update process.

4. Otherwise, remove this entry from *cache group*'s list of pending master entries.

Otherwise, store the resource for this entry in *new cache*, if it isn't already there, and categorize its entry as a master entry.

20. Fetch the resource from *manifest URL* again, and let *second manifest* be that resource.

21. If the previous step failed for any reason, or if the fetching attempt involved a redirect, or if *second manifest* and *manifest* are not byte-for-byte identical, then schedule a rerun of the entire algorithm with the same parameters after a short delay, and run the cache failure steps.

22. Otherwise, store *manifest* in *new cache*, if it's not there already, and categorize its entry as the manifest.

23. Set the completeness flag of *new cache* to *complete*.

24. If this is a cache attempt, then for each cache host associated with an application cache in *cache group*, queue a task to fire a simple event called `cached` at the `ApplicationCache` singleton of the cache host. The default action of these events should be the display of some sort of user interface indicating to the user that the application has been cached and that they can now use it offline.

    Otherwise, it is an upgrade attempt. For each cache host associated with an application cache in *cache group*, queue a task to fire a simple event called `updateready` at the `ApplicationCache` singleton of the cache host. The default action of these events should be the display of some sort of user interface indicating to the user that a new version is available and that they can activate it by reloading the page.

25. If appropriate, remove any user interface indicating that an update for this cache is in progress.

26. Set the update status of *cache group* to *idle*.

The **cache failure steps** are as follows:

1. For each entry in *cache group*'s list of pending master entries, run the following further substeps. These steps may be run in parallel for two or more entries at a time.

    1. Wait for the resource for this entry to have either completely downloaded or failed.

    2. Unassociate the `Document` for this entry from its application cache, if it has one.

    3. Queue a task to fire a simple event called `error` at the `ApplicationCache` singleton of the `Document` for this entry, if there still is one. The default action of these events should be the display of some sort of user interface indicating to the user that the user agent failed to save the application for offline use.

2. For each cache host still associated with an application cache in *cache group*, queue a task to fire a simple event called `error` at the `ApplicationCache` singleton of the cache host. The default action of these events should be the display of some sort of user interface indicating to the user that the user agent failed to save the application for offline use.

3. Empty *cache group*'s list of pending master entries.

4. If *cache group* has an application cache whose completeness flag is *incomplete*, then discard that application cache.

5. If appropriate, remove any user interface indicating that an update for this cache is in progress.

6. Let the status of *cache group* be *idle*.

7. If this was a cache attempt, discard *cache group* altogether.

8. Abort the update process.

Attempts to fetch resources as part of the application cache update process may be done with cache-defeating semantics, to avoid problems with stale or inconsistent intermediary caches.

User agents may invoke the application cache update process, in the background, for any application cache, at any time (with no cache host). This allows user agents to keep caches primed and to update caches even before the user visits a site.

5.8.5 Matching a fallback namespace

A URL **matches a fallback namespace** if there exists a relevant application cache whose manifest's URL has the same origin as the URL in question, and that has a fallback namespace that is a prefix match for the URL being examined. If multiple fallback namespaces match the same URL, the longest one is the one that matches. A URL looking for an fallback namespace can match more than one application cache at a time, but only matches one namespace in each cache.

> If a manifest `http://example.com/app1/manifest` declares that `http://example.com/resources/images` is a fallback namespace, and the user

navigates to `HTTP://EXAMPLE.COM:80/resources/images/cat.png`, then the user agent will decide that the application cache identified by `http://example.com/app1/manifest` contains a namespace with a match for that URL.

5.8.6 The application cache selection algorithm

When the **application cache selection algorithm** algorithm is invoked with a `Document` *document* and optionally a manifest URL *manifest URL*, the user agent must run the first applicable set of steps from the following list:

↪ **If there is a *manifest URL*, and *document* was loaded from an application cache, and the URL of the manifest of that cache's application cache group is *not* the same as *manifest URL***

Mark the entry for the resource from which *document* was taken in the application cache from which it was loaded as foreign.

Restart the current navigation from the top of the navigation algorithm, undoing any changes that were made as part of the initial load (changes can be avoided by ensuring that the step to update the session history with the new page is only ever completed *after* this application cache selection algorithm is run, though this is not required).

Note: The navigation will not result in the same resource being loaded, because "foreign" entries are never picked during navigation.

User agents may notify the user of the inconsistency between the cache manifest and the document's own metadata, to aid in application development.

↪ **If *document* was loaded from an application cache**

Associate *document* with the application cache from which it was loaded. Invoke the application cache update process for that cache and with the browsing context being navigated.

↪ **If *document* was loaded using HTTP GET or equivalent, and, there is a *manifest URL*, and *manifest URL* has the same origin as *document***

Invoke the application cache update process for *manifest URL*, with the browsing context being navigated, and with *document* and the resource from which *document* was loaded as the new master resource.

↪ **Otherwise**

The `Document` is not associated with any application cache.

If there was a *manifest URL*, the user agent may report to the user that it was ignored, to aid in application development.

5.8.7 Changes to the networking model

When a cache host is associated with an application cache whose completeness flag is *complete*, any and all loads for resources related to that cache host other than those for child

browsing contexts must go through the following steps instead of immediately invoking the mechanisms appropriate to that resource's scheme:

1. If the resource is not to be fetched using the HTTP GET mechanism or equivalent, or if it has a `javascript:` URL, then fetch the resource normally and abort these steps.

2. If the resource's URL is a master entry, the manifest, an explicit entry, or a fallback entry in the application cache, then get the resource from the cache (instead of fetching it), and abort these steps.

3. If the resource's URL has the same origin as the manifest's URL, and there is a fallback namespace in the application cache that is a prefix match for the resource's URL, then:

   Fetch the resource normally. If this results in a redirect to a resource with another origin (indicative of a captive portal), or a 4xx or 5xx status code or equivalent, or if there were network errors (but not if the user canceled the download), then instead get, from the cache, the resource of the fallback entry corresponding to the matched namespace. Abort these steps.

4. If there is an entry in the application cache's online whitelist that has the same origin as the resource's URL and that is a prefix match for the resource's URL, then fetch the resource normally and abort these steps.

5. Fail the resource load.

Note: The above algorithm ensures that resources that are not present in the manifest will always fail to load (at least, after the application cache has been primed the first time), making the testing of offline applications simpler.

5.8.8 Application cache API

```
interface ApplicationCache {

  // update status
  const unsigned short UNCACHED = 0;
  const unsigned short IDLE = 1;
  const unsigned short CHECKING = 2;
  const unsigned short DOWNLOADING = 3;
  const unsigned short UPDATEREADY = 4;
  const unsigned short OBSOLETE = 5;
  readonly attribute unsigned short status;

  // updates
  void update();
  void swapCache();

  // events
           attribute Function onchecking;
           attribute Function onerror;
           attribute Function onnoupdate;
           attribute Function ondownloading;
           attribute Function onprogress;
           attribute Function onupdateready;
           attribute Function oncached;
           attribute Function onobsolete;

};
```

***cache* = *window* . applicationCache**

(In a window.) Returns the ApplicationCache object that applies to the active document of that Window.

***cache* = *self* . applicationCache**

(In a shared worker.) Returns the ApplicationCache object that applies to the current shared worker.

***cache* . status**

Returns the current status of the application cache, as given by the constants defined below.

***cache* . update()**

Invokes the application cache update process.

Throws an INVALID_ACCESS_ERR exception if there is no application cache to update.

***cache* . swapCache()**

Switches to the most recent application cache, if there is a newer one. If there isn't, throws an INVALID_ACCESS_ERR exception.

Objects implementing the ApplicationCache interface must also implement the EventTarget interface.

There is a one-to-one mapping from cache hosts to ApplicationCache objects. The **applicationCache** attribute on Window objects must return the ApplicationCache object associated with the Window object's active document. The **applicationCache** attribute on SharedWorkerGlobalScope objects must return the ApplicationCache object associated with the worker.

The **status** attribute, on getting, must return the current state of the application cache that the ApplicationCache object's cache host is associated with, if any. This must be the appropriate value from the following list:

UNCACHED **(numeric value 0)**
The ApplicationCache object's cache host is not associated with an application cache at this time.

IDLE **(numeric value 1)**
The ApplicationCache object's cache host is associated with an application cache whose application cache group's update status is *idle*, and that application cache is the newest cache in its application cache group, and the application cache group is not marked as obsolete.

CHECKING **(numeric value 2)**
The ApplicationCache object's cache host is associated with an application cache

whose application cache group's update status is *checking*.

DOWNLOADING **(numeric value 3)**
The `ApplicationCache` object's cache host is associated with an application cache whose application cache group's update status is *downloading*.

UPDATEREADY **(numeric value 4)**
The `ApplicationCache` object's cache host is associated with an application cache whose application cache group's update status is *idle*, and whose application cache group is not marked as obsolete, but that application cache is *not* the newest cache in its group.

OBSOLETE **(numeric value 5)**
The `ApplicationCache` object's cache host is associated with an application cache whose application cache group is marked as obsolete.

If the **`update()`** method is invoked, the user agent must invoke the application cache update process, in the background, for the application cache with which the `ApplicationCache` object's cache host is associated, but without giving that cache host to the algorithm. If there is no such application cache, or if it is marked as obsolete, then the method must raise an `INVALID_STATE_ERR` exception instead.

If the **`swapCache()`** method is invoked, the user agent must run the following steps:

1. Check that `ApplicationCache` object's cache host is associated with an application cache. If it is not, then raise an `INVALID_STATE_ERR` exception and abort these steps.

2. Let *cache* be the application cache with which the `ApplicationCache` object's cache host is associated. (By definition, this is the same as the one that was found in the previous step.)

3. If *cache*'s application cache group is marked as obsolete, then unassociate the `ApplicationCache` object's cache host from *cache* and abort these steps. (Resources will now load from the network instead of the cache.)

4. Check that there is an application cache in the same application cache group as *cache* whose completeness flag is *complete* and that is newer than *cache*. If there is not, then raise an `INVALID_STATE_ERR` exception and abort these steps.

5. Let *new cache* be the newest application cache in the same application cache group as *cache* whose completeness flag is *complete*.

6. Unassociate the `ApplicationCache` object's cache host from *cache* and instead associate it with *new cache*.

The following are the event handler attributes that must be supported, as DOM attributes, by all objects implementing the `ApplicationCache` interface:

onchecking
Must be invoked whenever an `checking` event is targeted at or bubbles through the `ApplicationCache` object.

onerror

Must be invoked whenever an `error` event is targeted at or bubbles through the `ApplicationCache` object.

onnoupdate
Must be invoked whenever an `noupdate` event is targeted at or bubbles through the `ApplicationCache` object.

ondownloading
Must be invoked whenever an `downloading` event is targeted at or bubbles through the `ApplicationCache` object.

onprogress
Must be invoked whenever an `progress` event is targeted at or bubbles through the `ApplicationCache` object.

onupdateready
Must be invoked whenever an `updateready` event is targeted at or bubbles through the `ApplicationCache` object.

oncached
Must be invoked whenever a `cached` event is targeted at or bubbles through the `ApplicationCache` object.

onobsolete
Must be invoked whenever an `obsolete` event is targeted at or bubbles through the `ApplicationCache` object.

5.8.9 Browser state

window **`.navigator.onLine`**

Returns false if the user agent is definitely offline (disconnected from the network). Returns true if the user agent might be online.

The **`navigator.onLine`** attribute must return false if the user agent will not contact the network when the user follows links or when a script requests a remote page (or knows that such an attempt would fail), and must return true otherwise.

When the value that would be returned by the `navigator.onLine` attribute of the `Window` changes from true to false, the user agent must fire a simple event called **`offline`** at the `Window` object.

On the other hand, when the value that would be returned by the `navigator.onLine` attribute of the `Window` changes from false to true, the user agent must fire a simple event called **`online`** at the `Window` object.

Note: This attribute is inherently unreliable. A computer can be connected to a network without having Internet access.

5.9 Session history and navigation

5.9.1 The session history of browsing contexts

The sequence of `Document`s in a browsing context is its **session history**.

`History` objects provide a representation of the pages in the session history of browsing contexts. Each browsing context, including nested browsing context, has a distinct session history.

Each `Document` object in a browsing context's session history is associated with a unique instance of the `History` object, although they all must model the same underlying session history.

The **`history`** attribute of the `Window` interface must return the object implementing the `History` interface for that `Window` object's `Document`.

`History` objects represent their browsing context's session history as a flat list of session history entries. Each **session history entry** consists of either a URL or a state object, or both, and may in addition have a title, a `Document` object, form data, a scroll position, and other information associated with it.

> ***Note: This does not imply that the user interface need be linear. See the notes below.***

URLs without associated state objects are added to the session history as the user (or script) navigates from page to page.

A **state object** is an object representing a user interface state.

Pages can add state objects between their entry in the session history and the next ("forward") entry. These are then returned to the script when the user (or script) goes back in the history, thus enabling authors to use the "navigation" metaphor even in one-page applications.

At any point, one of the entries in the session history is the **current entry**. This is the entry representing the active document of the browsing context. The current entry is usually an entry for the location of the `Document`. However, it can also be one of the entries for state objects added to the history by that document.

Entries that consist of state objects share the same `Document` as the entry for the page that was active when they were added.

Contiguous entries that differ just by fragment identifier also share the same `Document`.

> ***Note: All entries that share the same `Document` (and that are therefore merely different states of one particular document) are contiguous by definition.***

User agents may discard the `Document` objects of entries other than the current entry that are

not referenced from any script, reloading the pages afresh when the user or script navigates back to such pages. This specification does not specify when user agents should discard `Document` objects and when they should cache them.

Entries that have had their `Document` objects discarded must, for the purposes of the algorithms given below, act as if they had not. When the user or script navigates back or forwards to a page which has no in-memory DOM objects, any other entries that shared the same `Document` object with it must share the new object as well.

When state object entries are added, a URL can be provided. This URL is used to replace the state object entry if the `Document` is evicted.

5.9.2 The `History` interface

```
interface History {
  readonly attribute long length;
  void go([Optional] in long delta);
  void back();
  void forward();
  void pushState(in any data, in DOMString title, [Optional] in DOMString url);
  void clearState();
};
```

window **. `history` . `length`**

Returns the number of entries in the session history.

window **. `history` . `go`([*delta*])**

Goes back or forward the specified number of steps in the history.

A zero delta will reload the current page.

If the delta is out of range, does nothing.

window **. `history` . `back`()**

Goes back one step in the history.

If there is no previous page, does nothing.

window **. `history` . `forward`()**

Goes forward one step in the history.

If there is no next page, does nothing.

window **. `history` . `pushstate`(*data*, *title* [, *url*])**

Pushes the given data onto the session history, with the given title, and, if provided, the given URL.

window **. `history` . `clearState`()**

Removes all state objects for the current page from the session history.

The **`length`** attribute of the `History` interface must return the number of entries in this session history.

The actual entries are not accessible from script.

The **`go(`*`delta`*`)`** method causes the UA to move the number of steps specified by *delta* in the session history.

If the *delta* is zero, or if the argument is omitted, then the user agent must act as if the `location.reload()` method was called instead.

Otherwise, if the index of the current entry plus *delta* is less than zero or greater than or equal to the number of items in the session history, then the user agent must do nothing.

Otherwise, the user agent must cause the current browsing context to traverse the history to the specified entry. The *specified entry* is the one whose index equals the index of the current entry plus *delta*.

When the user navigates through a browsing context, e.g. using a browser's back and forward buttons, the user agent must translate this action into the equivalent invocations of the `history.go(`*`delta`*`)` method on the various affected `window` objects.

Some of the other members of the `History` interface are defined in terms of the `go()` method, as follows:

Member	Definition
`back()`	Must do the same as `go(-1)`
`forward()`	Must do the same as `go(1)`

The **`pushState(`*`data, title, url`*`)`** method adds a state object to the history.

When this method is invoked, the user agent must run the following steps:

1. Let *clone data* be a structured clone of the specified *data*. If this throws an exception, then rethrow that exception and abort these steps.

2. If a third argument is specified, run these substeps:

    1. Resolve the value of the third argument, relative to the first script's base URL.

    2. If that fails, raise a `SECURITY_ERR` exception and abort the `pushState()` steps.

    3. Compare the resulting absolute URL to the document's address. If any part of these two URLs differ other than the <path>, <query>, and <fragment> components, then raise a `SECURITY_ERR` exception and abort the `pushState()` steps.

    For the purposes of the comparison in the above substeps, the <path> and <query> components can only be the same if the URLs use a hierarchical <scheme>.

3. Remove from the session history any entries for the `Document` from the entry after the current entry up to the last entry in the session history that references the same

`Document` object, if any. If the current entry is the last entry in the session history, or if there are no entries after the current entry that reference the same `Document` object, then no entries are removed.

4. Add a state object entry to the session history, after the current entry, with *cloned data* as the state object, the given *title* as the title, and, if the third argument is present, the absolute URL that was found earlier in this algorithm as the URL of the entry.

5. Update the current entry to be the this newly added entry.

Note: The title is purely advisory. User agents might use the title in the user interface.

User agents may limit the number of state objects added to the session history per page. If a page hits the UA-defined limit, user agents must remove the entry immediately after the first entry for that `Document` object in the session history after having added the new entry. (Thus the state history acts as a FIFO buffer for eviction, but as a LIFO buffer for navigation.)

The **`clearState()`** method removes all the state objects for the `Document` object from the session history.

When this method is invoked, the user agent must remove from the session history all the entries from the first state object entry for that `Document` object up to the last entry that references that same `Document` object, if any.

Then, if the current entry was removed in the previous step, the current entry must be set to the last entry for that `Document` object in the session history.

5.9.3 Activating state object entries

When an entry in the session history is activated (which happens during session traversal, as described above), the user agent must run the following steps:

1. If the entry is a state object entry, let *state* be a structured clone of that state object. Otherwise, let *state* be null.

2. Run the appropriate according to the conditions described:

   ↪ **If the current document readiness is set to the string "complete"**
   Queue a task to fire a **`popstate`** event in no namespace on the `Window` object of the `Document`, using the `PopStateEvent` interface, with the `state` attribute set to the value of *state*. This event must bubble but not be cancelable and has no default action. The task source for this task is the DOM manipulation task source.

   ↪ **Otherwise**
   Let the `Document`'s **pending state object** be *state*. (If there was already a pending state object, the previous one is discarded.)

   Note: The event will then be fired just after the `load` event.

The pending state object must be initially null.

```
interface PopStateEvent : Event {
  readonly attribute any state;
  void initPopStateEvent(in DOMString typeArg, in boolean canBubbleArg, in boolean
  void initPopStateEventNS(in DOMString namespaceURIArg, in DOMString typeArg, in
};
```

event **.** **state**

Returns the information that was provided to `pushState()`.

The **initPopStateEvent()** and **initPopStateEventNS()** methods must initialize the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS]

The **state** attribute represents the context information for the event, or null, if the state represented is the initial state of the `Document`.

5.9.4 The `Location` interface

Each `Document` object in a browsing context's session history is associated with a unique instance of a `Location` object.

document **.** **location** **[** **=** ***value*** **]**
window **.** **location** **[** **=** ***value*** **]**

Returns a `Location` object with the current page's location.

Can be set, to navigate to another page.

The **location** attribute of the `HTMLDocument` interface must return the `Location` object for that `Document` object, if it is in a browsing context, and null otherwise.

The **location** attribute of the `Window` interface must return the `Location` object for that `Window` object's `Document`.

`Location` objects provide a representation of their document's address, and allow the current entry of the browsing context's session history to be changed, by adding or replacing entries in the `history` object.

```
interface Location {
  readonly attribute DOMString href;
  void assign(in DOMString url);
  void replace(in DOMString url);
  void reload();
```

```
  // URL decomposition attributes
           attribute DOMString protocol;
           attribute DOMString host;
           attribute DOMString hostname;
           attribute DOMString port;
           attribute DOMString pathname;
           attribute DOMString search;
           attribute DOMString hash;

  // resolving relative URLs
  DOMString resolveURL(in DOMString url);
};
```

location **.** **`href`** **[= *value*]**

Returns the current page's location.

Can be set, to navigate to another page.

location **.** **`assign`**(***url***)

Navigates to the given page.

location **.** **`replace`**(***url***)

Removes the current page from the session history and navigates to the given page.

location **.** **`reload`**()

Reloads the current page.

url* = *location **.** **`resolveURL`**(***url***)

Resolves the given relative URL to an absolute URL.

The **`href`** attribute must return the address of the associated `Document` object, as an absolute URL.

On setting, the user agent must act as if the `assign()` method had been called with the new value as its argument.

When the **`assign(`*`url`*`)`** method is invoked, the UA must resolve the argument, relative to the first script's base URL, and if that is successful, must navigate the browsing context to the specified *url*.

When the **`replace(`*`url`*`)`** method is invoked, the UA must resolve the argument, relative to the first script's base URL, and if that is successful, navigate the browsing context to the specified *url* with replacement enabled.

Navigation for the `assign()` and `replace()` methods must be done with the browsing context of the script that invoked the method as the source browsing context.

If the resolving step of the the `assign()` and `replace()` methods is not successful, then the user agent must instead throw a `SYNTAX_ERR` exception.

The `Location` interface also has the complement of URL decomposition attributes, **`protocol`**, **`host`**, **`port`**, **`hostname`**, **`pathname`**, **`search`**, and **`hash`**. These must follow the rules given for URL decomposition attributes, with the input being the address of the associated `Document` object, as an absolute URL (same as the `href` attribute), and the common setter action being the same as setting the `href` attribute to the new output value.

The **`resolveURL(`*`url`*`)`** method must resolve its *url* argument, relative to the first script's base URL, and if that succeeds, return the resulting absolute URL. If it fails, it must throw a `SYNTAX_ERR` exception instead.

5.9.4.1 Security

User agents must raise a `SECURITY_ERR` exception whenever any of the members of a `Location` object are accessed by scripts whose effective script origin is not the same as the `Location` object's associated `Document`'s effective script origin, with the following exceptions:

- The `href` setter, if the script is running in a browsing context that is allowed to navigate the browsing context with which the `Location` object is associated

User agents must not allow scripts to override the `href` attribute's setter.

5.9.5 Implementation notes for session history

This section is non-normative.

The `History` interface is not meant to place restrictions on how implementations represent the session history to the user.

For example, session history could be implemented in a tree-like manner, with each page having multiple "forward" pages. This specification doesn't define how the linear list of pages in the `history` object are derived from the actual session history as seen from the user's perspective.

Similarly, a page containing two `iframe`s has a `history` object distinct from the `iframe`s' `history` objects, despite the fact that typical Web browsers present the user with just one "Back" button, with a session history that interleaves the navigation of the two inner frames and the outer page.

Security: It is suggested that to avoid letting a page "hijack" the history navigation facilities of a UA by abusing `pushState()`, the UA provide the user with a way to jump back to the previous page (rather than just going back to the previous state). For example, the back button could have a drop down showing just the pages in the session history, and not showing any of the states. Similarly, an aural browser could have two "back" commands, one that goes back to the previous state, and one that jumps straight back to the previous page.

In addition, a user agent could ignore calls to `pushState()` that are invoked on a timer, or from event handlers that do not represent a clear user action, or that are invoked in rapid succession.

5.10 Browsing the Web

5.10.1 Navigating across documents

Certain actions cause the browsing context to *navigate* to a new resource. Navigation always involves **source browsing context**, which is the browsing context which was responsible for starting the navigation.

> For example, following a hyperlink, form submission, and the `window.open()` and `location.assign()` methods can all cause a browsing context to navigate.

A user agent may provide various ways for the user to explicitly cause a browsing context to navigate, in addition to those defined in this specification.

When a browsing context is **navigated** to a new resource, the user agent must run the following steps:

1. If the source browsing context is not the same as the browsing context being navigated, and the source browsing context is not one of the ancestor browsing contexts of the browsing context being navigated, and the source browsing context has its sandboxed navigation browsing context flag set, then abort these steps. The user agent may offer to open the new resource in a new top-level browsing context or in the top-level browsing context of the source browsing context, at the user's option, in which case the user agent must navigate that designated top-level browsing context to the new resource as if the user had requested it independently.

2. If the source browsing context is the same as the browsing context being navigated, and this browsing context has its seamless browsing context flag set, then find the nearest ancestor browsing context that does not have its seamless browsing context flag set, and continue these steps as if *that* browsing context was the one that was going to be navigated instead.

3. Cancel any preexisting attempt to navigate the browsing context.

4. *Fragment identifiers*: If the absolute URL of the new resource is the same as the address of the active document of the browsing context being navigated, ignoring any <fragment> components of those URLs, and the new resource is to be fetched using HTTP GET or equivalent, then navigate to that fragment identifier and abort these steps.

5. If the new resource is to be handled by displaying some sort of inline content, e.g. an error message because the specified scheme is not one of the supported protocols, or an inline prompt to allow the user to select a registered handler for the given scheme, then display the inline content and abort these steps.

6. If the new resource is to be handled using a mechanism that does not affect the browsing context, e.g. ignoring the navigation request altogether because the specified scheme is not one of the supported protocols, then abort these steps and proceed with that mechanism instead.

7. If the new resource is to be fetched using HTTP GET or equivalent, then check if there

are any relevant application caches that are identified by a URL with the same origin as the URL in question, and that have this URL as one of their entries, excluding entries marked as foreign. If so, then the user agent must then get the resource from the most appropriate application cache of those that match.

> For example, imagine an HTML page with an associated application cache displaying an image and a form, where the image is also used by several other application caches. If the user right-clicks on the image and chooses "View Image", then the user agent could decide to show the image from any of those caches, but it is likely that the most useful cache for the user would be the one that was used for the aforementioned HTML page. On the other hand, if the user submits the form, and the form does a POST submission, then the user agent will not use an application cache at all; the submission will be made to the network.

Otherwise, fetch the new resource, if it has not already been obtained. If the resource is being fetched using HTTP, and the method is not GET, then the user agent must include an `XXX-Origin` header whose value is determined as follows:

↪ **If the navigation algorithm has so far contacted more than one origin**

↪ **If there is no source browsing context**

The value must be the string "`null`".

↪ **Otherwise**

The value must be the ASCII serialization of the origin of the active document of the source browsing context at the time the navigation was started.

8. If fetching the resource is synchronous (i.e. for `javascript:` URLs and `about:blank`), then this must be synchronous, but if fetching the resource depends on external resources, as it usually does for URLs that use HTTP or other networking protocols, then at this point the user agents must yield to whatever script invoked the navigation steps, if they were invoked by script.

9. If fetching the resource results in a redirect, return to the step labeled "fragment identifiers" with the new resource.

   Note: Cross-origin redirects cause the `XXX-Origin` header to become "`null`" on subsequent requests in the chain.

10. Wait for one or more bytes to be available or for the user agent to establish that the resource in question is empty. During this time, the user agent may allow the user to cancel this navigation attempt or start other navigation attempts.

11. If the resource was not fetched from an application cache, and was to be fetched using HTTP GET or equivalent, and its URL matches the fallback namespace of one or more relevant application caches, and the user didn't cancel the navigation attempt during the previous step, and the navigation attempt failed (e.g. the server returned a 4xx or 5xx status code or equivalent, or there was a DNS error), then:

    Let *candidate* be the fallback resource specified for the fallback namespace in question. If multiple application caches match, the user agent must use the fallback of the most appropriate application cache of those that match.

    If *candidate* is not marked as foreign, then the user agent must discard the failed load

and instead continue along these steps using *candidate* as the resource. The document's address, if appropriate, will still be the originally requested URL, not the fallback URL, but the user agent may indicate to the user that the original page load failed, that the page used was a fallback resource, and what the URL of the fallback resource actually is.

12. If the document's out-of-band metadata (e.g. HTTP headers), not counting any type information (such as the Content-Type HTTP header), requires some sort of processing that will not affect the browsing context, then perform that processing and abort these steps.

    Such processing might be triggered by, amongst other things, the following:

    - ***HTTP status codes (e.g. 204 No Content or 205 Reset Content)***
    - ***HTTP Content-Disposition headers***
    - ***Network errors***

    HTTP 401 responses that do not include a challenge recognised by the user agent must be processed as if they had no challenge, e.g. rendering the entity body as if the response had been 200 OK.

    User agents may show the entity body of an HTTP 401 response even when the response do include a recognised challenge, with the option to login being included in a non-modal fashion, to enable the information provided by the server to be used by the user before authenticating. Similarly, user agents should allow the user to authenticate (in a non-modal fashion) against authentication challenges included in other responses such as HTTP 200 OK responses, effectively allowing resources to present HTTP login forms without requiring their use.

13. Let *type* be the sniffed type of the resource.

14. If the user agent has been configured to process resources of the given *type* using some mechanism other than rendering the content in a browsing context, then skip this step. Otherwise, if the *type* is one of the following types, jump to the appropriate entry in the following list, and process the resource as described there:

    ↪ **"text/html"**
    Follow the steps given in the HTML document section, and abort these steps.

    ↪ **Any type ending in "+xml"**

    ↪ **"application/xml"**

    ↪ **"text/xml"**
    Follow the steps given in the XML document section. If that section determines that the content is *not* to be displayed as a generic XML document, then proceed to the next step in this overall set of steps. Otherwise, abort these steps.

    ↪ **"text/plain"**
    Follow the steps given in the plain text file section, and abort these steps.

    ↪ **A supported image type**
    Follow the steps given in the image section, and abort these steps.

    ↪ **A type that will use an external application to render the content in the**

browsing context

Follow the steps given in the plugin section, and abort these steps.

Setting the document's address: If there is no **override URL**, then any `Document` created by these steps must have its address set to the URL that was originally to be fetched, ignoring any other data that was used to obtain the resource (e.g. the entity body in the case of a POST submission is not part of the document's address, nor is the URL of the fallback resource in the case of the original load having failed and that URL having been found to match a fallback namespace). However, if there *is* an override URL, then any `Document` created by these steps must have its address set to that URL instead.

Note: An override URL is set when dereferencing a `javascript:` URL.

15. *Non-document content*: If, given *type*, the new resource is to be handled by displaying some sort of inline content, e.g. a native rendering of the content, an error message because the specified type is not supported, or an inline prompt to allow the user to select a registered handler for the given type, then display the inline content and abort these steps.

16. Otherwise, the document's *type* is such that the resource will not affect the browsing context, e.g. because the resource is to be handed to an external application. Process the resource appropriately.

Some of the sections below, to which the above algorithm defers in certain cases, require the user agent to **update the session history with the new page**. When a user agent is required to do this, it must queue a task to run the following steps:

1. Unload the `Document` object of the current entry, with the *recycle* parameter set to false.

2. **If the navigation was initiated for *entry update* of an entry**

    1. Replace the entry being updated with a new entry representing the new resource and its `Document` object and related state. The user agent may propagate state from the old entry to the new entry (e.g. scroll position).

    2. Traverse the history to the new entry.

    Otherwise

    1. Remove all the entries after the current entry in the browsing context's `Document` object's `History` object.

        Note: This doesn't necessarily have to affect the user agent's user interface.

    2. Append a new entry at the end of the `History` object representing the new resource and its `Document` object and related state.

    3. Traverse the history to the new entry.

    4. If the navigation was initiated with **replacement enabled**, remove the entry

immediately before the new current entry in the session history.

3. If the document's address has a fragment identifier, then run these substeps:

    1. Wait for a user-agent defined amount of time, as desired by the user agent implementor. (This is intended to allow the user agent to optimise the user experience in the face of performance concerns.)

    2. If the `Document` object has no parser, or its parser has stopped parsing, or the user agent has reason to believe the user is no longer interested in scrolling to the fragment identifier, then abort these substeps.

    3. Scroll to the fragment identifier given in the document's address. If this fails to find an indicated part of the document, then return to the first step of these substeps.

The task source for this task is the networking task source.

5.10.2 Page load processing model for HTML files

When an HTML document is to be loaded in a browsing context, the user agent must create a `Document` object, mark it as being an HTML document, create an HTML parser, associate it with the document, and begin to use the bytes provided for the document as the input stream for that parser.

> ***Note: The input stream converts bytes into characters for use in the tokeniser. This process relies, in part, on character encoding information found in the real Content-Type metadata of the resource; the "sniffed type" is not used for this purpose.***

When no more bytes are available, an EOF character is implied, which eventually causes a `load` event to be fired.

After creating the `Document` object, but potentially before the page has finished parsing, the user agent must update the session history with the new page.

> ***Note: Application cache selection happens in the HTML parser.***

5.10.3 Page load processing model for XML files

When faced with displaying an XML file inline, user agents must first create a `Document` object, following the requirements of the XML and Namespaces in XML recommendations, RFC 3023, DOM3 Core, and other relevant specifications. [XML] [XMLNS] [RFC3023] [DOM3CORE]

The actual HTTP headers and other metadata, not the headers as mutated or implied by the algorithms given in this specification, are the ones that must be used when determining the character encoding according to the rules given in the above specifications. Once the character encoding is established, the document's character encoding must be set to that character encoding.

If the root element, as parsed according to the XML specifications cited above, is found to be an `html` element with an attribute `manifest`, then, as soon as the element is inserted into the document, the user agent must resolve the value of that attribute relative to that element, and if that is successful, must run the application cache selection algorithm with the resulting absolute URL as the manifest URL, and passing in the newly-created `Document`. Otherwise, if the attribute is absent or resolving it fails, then as soon as the root element is inserted into the document, the user agent must run the application cache selection algorithm with no manifest, and passing in the `Document`.

> ***Note: Because the processing of the `manifest` attribute happens only once the root element is parsed, any URLs referenced by processing instructions before the root element (such as `<?xml-styleesheet?>` and `<?xbl?>` PIs) will be fetched from the network and cannot be cached.***

User agents may examine the namespace of the root `Element` node of this `Document` object to perform namespace-based dispatch to alternative processing tools, e.g. determining that the content is actually a syndication feed and passing it to a feed handler. If such processing is to take place, abort the steps in this section, and jump to the next step (labeled "non-document content") in the navigate steps above.

Otherwise, then, with the newly created `Document`, the user agents must update the session history with the new page. User agents may do this before the complete document has been parsed (thus achieving *incremental rendering*).

Error messages from the parse process (e.g. XML namespace well-formedness errors) may be reported inline by mutating the `Document`.

5.10.4 Page load processing model for text files

When a plain text document is to be loaded in a browsing context, the user agent should create a `Document` object, mark it as being an HTML document, create an HTML parser, associate it with the document, act as if the tokeniser had emitted a start tag token with the tag name "pre", set the tokenization stage's content model flag to *PLAINTEXT*, and begin to pass the stream of characters in the plain text document to that tokeniser.

The rules for how to convert the bytes of the plain text document into actual characters are defined in RFC 2046, RFC 2646, and subsequent versions thereof. [RFC2046] [RFC2646]

The document's character encoding must be set to the character encoding used to decode the document.

Upon creation of the `Document` object, the user agent must run the application cache selection algorithm with no manifest, and passing in the newly-created `Document`.

When no more character are available, an EOF character is implied, which eventually causes a `load` event to be fired.

After creating the `Document` object, but potentially before the page has finished parsing, the user agent must update the session history with the new page.

User agents may add content to the `head` element of the `Document`, e.g. linking to stylesheet or an XBL binding, providing script, giving the document a `title`, etc.

5.10.5 Page load processing model for images

When an image resource is to be loaded in a browsing context, the user agent should create a `Document` object, mark it as being an HTML document, append an `html` element to the `Document`, append a `head` element and a `body` element to the `html` element, append an `img` to the `body` element, and set the `src` attribute of the `img` element to the address of the image.

Then, the user agent must act as if it had stopped parsing.

Upon creation of the `Document` object, the user agent must run the application cache selection algorithm with no manifest, and passing in the newly-created `Document`.

After creating the `Document` object, but potentially before the page has finished fully loading, the user agent must update the session history with the new page.

User agents may add content to the `head` element of the `Document`, or attributes to the `img` element, e.g. to link to stylesheet or an XBL binding, to provide a script, to give the document a `title`, etc.

5.10.6 Page load processing model for content that uses plugins

When a resource that requires an external resource to be rendered is to be loaded in a browsing context, the user agent should create a `Document` object, mark it as being an HTML document, append an `html` element to the `Document`, append a `head` element and a `body` element to the `html` element, append an `embed` to the `body` element, and set the `src` attribute of the `embed` element to the address of the resource.

Then, the user agent must act as if it had stopped parsing.

Upon creation of the `Document` object, the user agent must run the application cache selection algorithm with no manifest, and passing in the newly-created `Document`.

After creating the `Document` object, but potentially before the page has finished fully loading, the user agent must update the session history with the new page.

User agents may add content to the `head` element of the `Document`, or attributes to the `embed` element, e.g. to link to stylesheet or an XBL binding, or to give the document a `title`.

> ***Note: If the sandboxed plugins browsing context flag is set on the browsing context, the synthesized `embed` element will fail to render the content.***

5.10.7 Page load processing model for inline content that doesn't have a DOM

When the user agent is to display a user agent page inline in a browsing context, the user agent should create a `Document` object, mark it as being an HTML document, and then either associate that `Document` with a custom rendering that is not rendered using the normal `Document` rendering rules, or mutate that `Document` until it represents the content the user agent wants to render.

Once the page has been set up, the user agent must act as if it had stopped parsing.

Upon creation of the `Document` object, the user agent must run the application cache selection algorithm with no manifest, passing in the newly-created `Document`.

After creating the `Document` object, but potentially before the page has been completely set up, the user agent must update the session history with the new page.

5.10.8 Navigating to a fragment identifier

When a user agent is supposed to navigate to a fragment identifier, then the user agent must update the session history with the new page, where "the new page" has the same `Document` as before but with the URL having the newly specified fragment identifier.

Part of that algorithm involves the user agent having to scroll to the fragment identifier, which is the important part for this step.

When the user agent is required to **scroll to the fragment identifier**, it must change the scrolling position of the document, or perform some other action, such that the indicated part of the document is brought to the user's attention. If there is no indicated part, then the user agent must not scroll anywhere.

The indicated part of the document is the one that the fragment identifier, if any, identifies. The semantics of the fragment identifier in terms of mapping it to a specific DOM Node is defined by the MIME type specification of the document's MIME Type (for example, the processing of fragment identifiers for XML MIME types is the responsibility of RFC3023).

For HTML documents (and the `text/html` MIME type), the following processing model must be followed to determine what the indicated part of the document is.

1. Parse the URL, and let *fragid* be the <fragment> component of the URL.
2. If *fragid* is the empty string, then the indicated part of the document is the top of the document.
3. Let *decoded fragid* be the result of expanding any sequences of percent-encoded octets in *fragid* that are valid UTF-8 sequences into Unicode characters as defined by UTF-8. If any percent-encoded octets in that string are not valid UTF-8 sequences, then skip this step and the next one.
4. If this step was not skipped and there is an element in the DOM that has an ID exactly equal to *decoded fragid*, then the first such element in tree order is the indicated part of the document; stop the algorithm here.
5. If there is an `a` element in the DOM that has a `name` attribute whose value is exactly equal to *fragid* (*not decoded fragid*), then the first such element in tree order is the indicated part of the document; stop the algorithm here.
6. Otherwise, there is no indicated part of the document.

For the purposes of the interaction of HTML with Selectors' `:target` pseudo-class, the ***target element*** is the indicated part of the document, if that is an element; otherwise there is no *target element*. [SELECTORS]

5.10.9 History traversal

When a user agent is required to **traverse the history** to a *specified entry*, the user agent must act as follows:

1. If there is no longer a `Document` object for the entry in question, the user agent must navigate the browsing context to the location for that entry to perform an entry update of that entry, and abort these steps. The "navigate" algorithm reinvokes this "traverse" algorithm to complete the traversal, at which point there *is* a `Document` object and so this step gets skipped. The navigation must be done using the same source browsing context as was used the first time this entry was created.

2. If appropriate, update the current entry in the browsing context's `Document` object's `History` object to reflect any state that the user agent wishes to persist.

   > For example, some user agents might want to persist the scroll position, or the values of form controls.

3. If the *specified entry* has a different `Document` object than the current entry then the user agent must run the following substeps:

   1. If the browsing context is a top-level browsing context (and not an auxiliary browsing context), and the origin of the `Document` of the *specified entry* is not the same as the origin of the `Document` of the current entry, then the following sub-sub-steps must be run:
      1. The current browsing context name must be stored with all the entries in the history that are associated with `Document` objects with the same origin as the active document *and* that are contiguous with the current entry.
      2. The browsing context's browsing context name must be unset.

   2. The user agent must make the *specified entry*'s `Document` object the active document of the browsing context.

   3. If the *specified entry* has a browsing context name stored with it, then the following sub-sub-steps must be run:
      1. The browsing context's browsing context name must be set to the name stored with the specified entry.
      2. Any browsing context name stored with the entries in the history that are associated with `Document` objects with the same origin as the new active document, and that are contiguous with the specified entry, must be cleared.

4. If the *specified entry* is a state object or the first entry for a `Document`, the user agent must activate that entry.

5. If the *specified entry* has a URL that differs from the current entry's only by its fragment identifier, and the two share the same `Document` object, then first, if the `Document`'s current document readiness is the string "complete", then fire a simple event with the name `hashchange` at the browsing context's `Window` object; and second, if the new URL has a fragment identifier, scroll to the fragment identifier.

6. User agents may also update other aspects of the document view when the location changes in this way, for instance the scroll position, values of form fields, etc.

7. The current entry is now the *specified entry*.

5.10.10 Unloading documents

When a user agent is to **unload a document**, it must run the following steps. These steps are passed an argument, *recycle*, which is either true or false, indicating whether the `Document` object is going to be re-used. (This is set by the `document.open()` method.)

1. Set *salvageable* to true.
2. Let *event* be a new `BeforeUnloadEvent` event object with the name `beforeunload`, with no namespace, which does not bubble but is cancelable.
3. Dispatch *event* at the `Document`'s `Window` object.
4. If any event listeners were triggered by the previous step, then set *salvageable* to false.
5. If the `returnValue` attribute of the *event* object is not the empty string, or if the event was canceled, then the user agent should ask the user to confirm that they wish to unload the document.

   The prompt shown by the user agent may include the string of the `returnValue` attribute, or some leading subset thereof. (A user agent may want to truncate the string to 1024 characters for display, for instance.)

   The user agent must pause while waiting for the user's response.

   If the user **refused to allow the document to be unloaded** then these steps must be aborted.
6. Fire a simple event called `unload` at the `Document`'s `Window` object.
7. If any event listeners were triggered by the previous step, then set *salvageable* to false.
8. If there are any outstanding transactions that have callbacks that involve scripts whose global object is the `Document`'s `Window` object, roll them back (without invoking any of the callbacks) and set *salvageable* to false.
9. If *salvageable* and *recycle* are both false, discard the `Document`.

5.10.10.1 Event definition

```
interface BeforeUnloadEvent : Event {
           attribute DOMString returnValue;
};
```

event **.** `returnValue` **[=** ***value*** **]**

Returns the current return value of the event (the message to show the user).

Can be set, to update the message.

Note: There are no `BeforeUnloadEvent`-specific initialization methods.

The **`returnValue`** attribute represents the message to show the user. When the event is created, the attribute must be set to the empty string. On getting, it must return the last value it was set to. On setting, the attribute must be set to the new value.

5.11 Links

5.11.1 Hyperlink elements

The `a`, `area`, and `link` elements can, in certain situations described in the definitions of those elements, represent **hyperlinks**.

The **`href`** attribute on a hyperlink element must have a value that is a valid URL. This URL is the *destination resource* of the hyperlink.

The `href` attribute on `a` and `area` elements is not required; when those elements do not have `href` attributes they do not represent hyperlinks.

The `href` attribute on the `link` element is required, but whether a `link` element represents a hyperlink or not depends on the value of the `rel` attribute of that element.

The **`target`** attribute, if present, must be a valid browsing context name or keyword. It gives the name of the browsing context that will be used. User agents use this name when following hyperlinks.

The **`ping`** attribute, if present, gives the URLs of the resources that are interested in being notified if the user follows the hyperlink. The value must be a space separated list of one or more valid URLs. The value is used by the user agent for hyperlink auditing.

For `a` and `area` elements that represent hyperlinks, the relationship between the document containing the hyperlink and the destination resource indicated by the hyperlink is given by the value of the element's **`rel`** attribute, which must be a set of space-separated tokens. The allowed values and their meanings are defined below. The `rel` attribute has no default value. If the attribute is omitted or if none of the values in the attribute are recognized by the user agent, then the document has no particular relationship with the destination resource other than there being a hyperlink between the two.

The **`media`** attribute describes for which media the target document was designed. It is purely advisory. The value must be a valid media query. [MQ] The default, if the `media` attribute is omitted, is `all`.

The **`hreflang`** attribute on hyperlink elements, if present, gives the language of the linked resource. It is purely advisory. The value must be a valid RFC 3066 language code. [RFC3066] User agents must not consider this attribute authoritative — upon fetching the resource, user agents must use only language information associated with the resource to determine its language, not metadata included in the link to the resource.

The **type** attribute, if present, gives the MIME type of the linked resource. It is purely advisory. The value must be a valid MIME type, optionally with parameters. [RFC2046] User agents must not consider the `type` attribute authoritative — upon fetching the resource, user agents must not use metadata included in the link to the resource to determine its type.

5.11.2 Following hyperlinks

When a user *follows a hyperlink*, the user agent must resolve the URL given by the `href` attribute of that hyperlink, relative to the hyperlink element, and if that is successful, must navigate a browsing context to the resulting absolute URL. In the case of server-side image maps, the URL of the hyperlink must further have its *hyperlink suffix* appended to it.

If resolving the URL fails, the user agent may report the error to the user in a user-agent-specific manner, may navigate to an error page to report the error, or may ignore the error and do nothing.

If the user indicated a specific browsing context when following the hyperlink, or if the user agent is configured to follow hyperlinks by navigating a particular browsing context, then that must be the browsing context that is navigated.

Otherwise, if the hyperlink element is an `a` or `area` element that has a `target` attribute, then the browsing context that is navigated must be chosen by applying the rules for choosing a browsing context given a browsing context name, using the value of the `target` attribute as the browsing context name. If these rules result in the creation of a new browsing context, it must be navigated with replacement enabled.

Otherwise, if the hyperlink element is a sidebar hyperlink and the user agent implements a feature that can be considered a secondary browsing context, such a secondary browsing context may be selected as the browsing context to be navigated.

Otherwise, if the hyperlink element is an `a` or `area` element with no `target` attribute, but one of the child nodes of the `head` element is a `base` element with a `target` attribute, then the browsing context that is navigated must be chosen by applying the rules for choosing a browsing context given a browsing context name, using the value of the `target` attribute of the first such `base` element as the browsing context name. If these rules result in the creation of a new browsing context, it must be navigated with replacement enabled.

Otherwise, the browsing context that must be navigated is the same browsing context as the one which the hyperlink element itself is in.

The navigation must be done with the browsing context that contains the `Document` object with which the hyperlink's element in question is associated as the source browsing context.

5.11.2.1 Hyperlink auditing

If an `a` or `area` hyperlink element has a `ping` attribute, and the user follows the hyperlink, and the hyperlink's URL can be resolved, relative to the hyperlink element, without failure, then the user agent must take the `ping` attribute's value, split that string on spaces, resolve each resulting token relative to the hyperlink element, and then should send a request (as described below) to each of the resulting absolute URLs. (Tokens that fail to resolve are

ignored.) This may be done in parallel with the primary request, and is independent of the result of that request.

User agents should allow the user to adjust this behavior, for example in conjunction with a setting that disables the sending of HTTP `Referer` headers. Based on the user's preferences, UAs may either ignore the `ping` attribute altogether, or selectively ignore URLs in the list (e.g. ignoring any third-party URLs).

For URLs that are HTTP URLs, the requests must be performed by fetching the specified URLs using the POST method, with an entity body with the MIME type `text/ping` consisting of the four-character string "`PING`". All relevant cookie and HTTP authentication headers must be included in the request. Which other headers are required depends on the URLs involved.

- **If both the address of the `Document` object containing the hyperlink being audited and the ping URL have the same origin**

  The request must include a `Ping-From` HTTP header with, as its value, the address of the document containing the hyperlink, and a `Ping-To` HTTP header with, as its value, the address of the absolute URL of the target of the hyperlink. The request must not include a `Referer` HTTP header.

- **Otherwise, if the origins are different, but the document containing the hyperlink being audited was not retrieved over an encrypted connection**

  The request must include a `Referer` HTTP header [sic] with, as its value, the address of the document containing the hyperlink, a `Ping-From` HTTP header with the same value, and a `Ping-To` HTTP header with, as its value, the address of the target of the hyperlink.

- **Otherwise, the origins are different and the document containing the hyperlink being audited was retrieved over an encrypted connection**

  The request must include a `Ping-To` HTTP header with, as its value, the address of the target of the hyperlink. The request must neither include a `Referer` HTTP header nor include a `Ping-From` HTTP header.

In addition, an `XXX-Origin` header must always be included, whose value is the ASCII serialization of the origin of the the `Document` containing the hyperlink. The value of the `XXX-Origin` header must be set to "`null`" when following redirects if the origins of all the URLs involved are not the same.

> ***Note: To save bandwidth, implementors might also wish to consider omitting optional headers such as `Accept` from these requests.***

User agents must, unless otherwise specified by the user, honor the HTTP headers (including, in particular, redirects and HTTP cookie headers), but must ignore any entity bodies returned in the responses. User agents may close the connection prematurely once they start receiving an entity body. [RFC2109] [RFC2965]

For URLs that are not HTTP URLs, the requests must be performed by fetching the specified URL normally, and discarding the results.

When the `ping` attribute is present, user agents should clearly indicate to the user that following the hyperlink will also cause secondary requests to be sent in the background, possibly including listing the actual target URLs.

> For example, a visual user agent could include the hostnames of the target ping URLs

> along with the hyperlink's actual URL in a status bar or tooltip.

The `ping` attribute is redundant with pre-existing technologies like HTTP redirects and JavaScript in allowing Web pages to track which off-site links are most popular or allowing advertisers to track click-through rates.

However, the `ping` attribute provides these advantages to the user over those alternatives:

- ***It allows the user to see the final target URL unobscured.***
- ***It allows the UA to inform the user about the out-of-band notifications.***
- ***It allows the paranoid user to disable the notifications without losing the underlying link functionality.***
- ***It allows the UA to optimize the use of available network bandwidth so that the target page loads faster.***

Thus, while it is possible to track users without this feature, authors are encouraged to use the `ping` attribute so that the user agent can improve the user experience.

5.11.3 Link types

The following table summarizes the link types that are defined by this specification. This table is non-normative; the actual definitions for the link types are given in the next few sections.

In this section, the term *referenced document* refers to the resource identified by the element representing the link, and the term *current document* refers to the resource within which the element representing the link finds itself.

To determine which link types apply to a `link`, `a`, or `area` element, the element's `rel` attribute must be split on spaces. The resulting tokens are the link types that apply to that element.

Unless otherwise specified, a keyword must not be specified more than once per `rel` attribute.

The link types are ASCII case-insensitive values, and must be compared as such.

> Thus, `rel="next"` is the same as `rel="NEXT"`.

Link type	Effect on...		Brief description
	`link`	`a` and `area`	
`alternate`	Hyperlink	Hyperlink	Gives alternate representations of the current document.
`archives`	Hyperlink	Hyperlink	Provides a link to a collection of records, documents, or other materials of historical interest.
`author`	Hyperlink	Hyperlink	Gives a link to the current document's author.

Link type	Effect on...		Brief description
	`link`	`a` and `area`	
`bookmark`	*not allowed*	Hyperlink	Gives the permalink for the nearest ancestor section.
`external`	*not allowed*	Hyperlink	Indicates that the referenced document is not part of the same site as the current document.
`feed`	Hyperlink	Hyperlink	Gives the address of a syndication feed for the current document.
`first`	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the first document in the series is the referenced document.
`help`	Hyperlink	Hyperlink	Provides a link to context-sensitive help.
`icon`	External Resource	*not allowed*	Imports an icon to represent the current document.
`index`	Hyperlink	Hyperlink	Gives a link to the document that provides a table of contents or index listing the current document.
`last`	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the last document in the series is the referenced document.
`license`	Hyperlink	Hyperlink	Indicates that the current document is covered by the copyright license described by the referenced document.
`next`	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the next document in the series is the referenced document.
`nofollow`	*not allowed*	Hyperlink	Indicates that the current document's original author or publisher does not endorse the referenced document.
`noreferrer`	*not allowed*	Hyperlink	Requires that the user agent not send an HTTP `Referer` header if the user follows the hyperlink.
`pingback`	External Resource	*not allowed*	Gives the address of the pingback server that handles pingbacks to the current document.
`prefetch`	External Resource	*not allowed*	Specifies that the target resource should be preemptively cached.
`prev`	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the previous document in the series is the referenced document.
`search`	Hyperlink	Hyperlink	Gives a link to a resource that can be used to search through the current document and its related pages.
`stylesheet`	External Resource	*not allowed*	Imports a stylesheet.
`sidebar`	Hyperlink	Hyperlink	Specifies that the referenced document, if retrieved, is intended to be shown in the browser's sidebar (if it has one).

Link type	Effect on...		Brief description
	link	a and area	
tag	Hyperlink	Hyperlink	Gives a tag (identified by the given address) that applies to the current document.
up	Hyperlink	Hyperlink	Provides a link to a document giving the context for the current document.

Some of the types described below list synonyms for these values. These are to be handled as specified by user agents, but must not be used in documents.

5.11.3.1 Link type "alternate"

The alternate keyword may be used with link, a, and area elements. For link elements, if the rel attribute does not also contain the keyword stylesheet, it creates a hyperlink; but if it *does* also contain the keyword stylesheet, the alternate keyword instead modifies the meaning of the stylesheet keyword in the way described for that keyword, and the rest of this subsection doesn't apply.

The alternate keyword indicates that the referenced document is an alternate representation of the current document.

The nature of the referenced document is given by the media, hreflang, and type attributes.

If the alternate keyword is used with the media attribute, it indicates that the referenced document is intended for use with the media specified.

If the alternate keyword is used with the hreflang attribute, and that attribute's value differs from the root element's language, it indicates that the referenced document is a translation.

If the alternate keyword is used with the type attribute, it indicates that the referenced document is a reformulation of the current document in the specified format.

The media, hreflang, and type attributes can be combined when specified with the alternate keyword.

> For example, the following link is a French translation that uses the PDF format:
>
> ```
> <link rel=alternate type=application/pdf hreflang=fr href=manual-fr>
> ```

If the alternate keyword is used with the type attribute set to the value application/rss+xml or the value application/atom+xml, then the user agent must treat the link as it would if it had the feed keyword specified as well.

The alternate link relationship is transitive — that is, if a document links to two other documents with the link type "alternate", then, in addition to implying that those documents are alternative representations of the first document, it is also implying that those two documents are alternative representations of each other.

5.11.3.2 Link type "archives"

The archives keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The archives keyword indicates that the referenced document describes a collection of records, documents, or other materials of historical interest.

> A blog's index page could link to an index of the blog's past posts with `rel="archives"`.

Synonyms: For historical reasons, user agents must also treat the keyword "archive" like the archives keyword.

5.11.3.3 Link type "author"

The author keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

For a and area elements, the author keyword indicates that the referenced document provides further information about the author of the section that the element defining the hyperlink applies to.

For link elements, the author keyword indicates that the referenced document provides further information about the author for the page as a whole.

> ***Note: The "referenced document" can be, and often is, a `mailto:` URL giving the e-mail address of the author. [MAILTO]***

Synonyms: For historical reasons, user agents must also treat link, a, and area elements that have a rev attribute with the value "made" as having the author keyword specified as a link relationship.

5.11.3.4 Link type "bookmark"

The bookmark keyword may be used with a and area elements.

The bookmark keyword gives a permalink for the nearest ancestor article element of the linking element in question, or of the section the linking element is most closely associated with, if there are no ancestor article elements.

> The following snippet has three permalinks. A user agent could determine which permalink applies to which part of the spec by looking at where the permalinks are given.
>
> ```
> ...
> <body>
>  <h1>Example of permalinks</h1>
>  
>   <h2>First example</h2>
>   <a href="a.html" rel="bookmark">This</a> permalink applies to
>   only the content from the first H2 to the second H2. The DIV isn't
>   exactly that section, but it roughly corresponds to it.
>  
>  <h2>Second example</h2>
> ```

```
      <article id="b">
       <p><a href="b.html" rel="bookmark">This</a> permalink applies to
       the outer ARTICLE element (which could be, e.g., a blog post).</p>
       <article id="c">
        <p><a href="c.html" rel="bookmark">This</a> permalink applies to
        the inner ARTICLE element (which could be, e.g., a blog comment).</p>
       </article>
      </article>
     </body>
     ...
```

5.11.3.5 Link type "external"

The external keyword may be used with a and area elements.

The external keyword indicates that the link is leading to a document that is not part of the site that the current document forms a part of.

5.11.3.6 Link type "feed"

The feed keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The feed keyword indicates that the referenced document is a syndication feed. If the alternate link type is also specified, then the feed is specifically the feed for the current document; otherwise, the feed is just a syndication feed, not necessarily associated with a particular Web page.

The first link, a, or area element in the document (in tree order) that creates a hyperlink with the link type feed must be treated as the default syndication feed for the purposes of feed autodiscovery.

Note: The feed keyword is implied by the alternate link type in certain cases (q.v.).

The following two link elements are equivalent: both give the syndication feed for the current page:

```
<link rel="alternate" type="application/atom+xml" href="data.xml">

<link rel="feed alternate" href="data.xml">
```

The following extract offers various different syndication feeds:

```
 <p>You can access the planets database using Atom feeds:</p>
 <ul>
  <li><a href="recently-visited-planets.xml" rel="feed">Recently Visited Plan
  <li><a href="known-bad-planets.xml" rel="feed">Known Bad Planets</a></li>
  <li><a href="unexplored-planets.xml" rel="feed">Unexplored Planets</a></li>
 </ul>
```

5.11.3.7 Link type "help"

The help keyword may be used with link, a, and area elements. For link elements, it creates

a hyperlink.

For a and area elements, the help keyword indicates that the referenced document provides further help information for the parent of the element defining the hyperlink, and its children.

> In the following example, the form control has associated context-sensitive help. The user agent could use this information, for example, displaying the referenced document if the user presses the "Help" or "F1" key.
>
> ```
> <p><label> Topic: <input name=topic> <a href="help/topic.html" rel="help">(H
> ```

For link elements, the help keyword indicates that the referenced document provides help for the page as a whole.

5.11.3.8 Link type "icon"

The icon keyword may be used with link elements, for which it creates an external resource link.

The specified resource is an icon representing the page or site, and should be used by the user agent when representing the page in the user interface.

Icons could be auditory icons, visual icons, or other kinds of icons. If multiple icons are provided, the user agent must select the most appropriate icon according to the type, media, and sizes attributes. If there are multiple equally appropriate icons, user agents must use the last one declared in tree order. If the user agent tries to use an icon but that icon is determined, upon closer examination, to in fact be inappropriate (e.g. because it uses an unsupported format), then the user agent must try the next-most-appropriate icon as determined by the attributes.

There is no default type for resources given by the icon keyword. However, for the purposes of determining the type of the resource, user agents must expect the resource to be an image.

The **sizes** attribute gives the sizes of icons for visual media.

If specified, the attribute must have a value that is an unordered set of unique space-separated tokens. The values must all be either any or a value that consists of two valid non-negative integers that do not have a leading U+0030 DIGIT ZERO (0) character and that are separated by a single U+0078 LATIN SMALL LETTER X character.

The keywords represent icon sizes.

To parse and process the attribute's value, the user agent must first split the attribute's value on spaces, and must then parse each resulting keyword to determine what it represents.

The **any** keyword represents that the resource contains a scalable icon, e.g. as provided by an SVG image.

Other keywords must be further parsed as follows to determine what they represent:

- If the keyword doesn't contain exactly one U+0078 LATIN SMALL LETTER X character, then this keyword doesn't represent anything. Abort these steps for that keyword.

- Let *width string* be the string before the "x".
- Let *height string* be the string after the "x".
- If either *width string* or *height string* start with a U+0030 DIGIT ZERO (0) character or contain any characters other than characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), then this keyword doesn't represent anything. Abort these steps for that keyword.
- Apply the rules for parsing non-negative integers to *width string* to obtain *width*.
- Apply the rules for parsing non-negative integers to *height string* to obtain *height*.
- The keyword represents that the resource contains a bitmap icon with a width of *width* device pixels and a height of *height* device pixels.

The keywords specified on the `sizes` attribute must not represent icon sizes that are not actually available in the linked resource.

If the attribute is not specified, then the user agent must assume that the given icon is appropriate, but less appropriate than an icon of a known and appropriate size.

> The following snippet shows the top part of an application with several icons.
>
> ```
> <!DOCTYPE HTML>
> <html>
>  <head>
>   <title>lsForums — Inbox</title>
>   <link rel=icon href=favicon.png sizes="16x16">
>   <link rel=icon href=windows.ico sizes="32x32 48x48">
>   <link rel=icon href=mac.icns sizes="128x128 512x512 8192x8192 32768x32768">
>   <link rel=icon href=iphone.png sizes="59x60">
>   <link rel=icon href=gnome.svg sizes="any">
>   <link rel=stylesheet href=lsforums.css>
>   <script src=lsforums.js></script>
>   <meta name=application-name content="lsForums">
>  </head>
>  <body>
>   ...
> ```

5.11.3.9 Link type "`license`"

The `license` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `license` keyword indicates that the referenced document provides the copyright license terms under which the current document is provided.

Synonyms: For historical reasons, user agents must also treat the keyword "`copyright`" like the `license` keyword.

5.11.3.10 Link type "`nofollow`"

The `nofollow` keyword may be used with `a` and `area` elements.

The `nofollow` keyword indicates that the link is not endorsed by the original author or publisher of the page, or that the link to the referenced document was included primarily because of a commercial relationship between people affiliated with the two pages.

5.11.3.11 Link type "`noreferrer`"

The `noreferrer` keyword may be used with `a` and `area` elements.

It indicates that the no referrer information is to be leaked when following the link.

If a user agent follows a link defined by an `a` or `area` element that has the `noreferrer` keyword, the user agent must not include a `Referer` HTTP header (or equivalent for other protocols) in the request.

This keyword also causes the `opener` attribute to remain null if the hyperlink creates a new browsing context.

5.11.3.12 Link type "`pingback`"

The `pingback` keyword may be used with `link` elements, for which it creates an external resource link.

For the semantics of the `pingback` keyword, see the Pingback 1.0 specification. [PINGBACK]

5.11.3.13 Link type "`prefetch`"

The `prefetch` keyword may be used with `link` elements, for which it creates an external resource link.

The `prefetch` keyword indicates that preemptively fetching and caching the specified resource is likely to be beneficial, as it is highly likely that the user will require this resource.

There is no default type for resources given by the `prefetch` keyword.

5.11.3.14 Link type "`search`"

The `search` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `search` keyword indicates that the referenced document provides an interface specifically for searching the document and its related resources.

> ***Note: OpenSearch description documents can be used with `link` elements and the `search` link type to enable user agents to autodiscover search interfaces. [OPENSEARCH]***

5.11.3.15 Link type "`stylesheet`"

The `stylesheet` keyword may be used with `link` elements, for which it creates an external resource link that contributes to the styling processing model.

The specified resource is a resource that describes how to present the document. Exactly how the resource is to be processed depends on the actual type of the resource.

If the `alternate` keyword is also specified on the `link` element, then the link is an alternative stylesheet; in this case, the `title` attribute must be specified on the `link` element, with a non-empty value.

The default type for resources given by the `stylesheet` keyword is `text/css`.

Quirk: If the document has been set to quirks mode and the Content-Type metadata of the external resource is not a supported style sheet type, the user agent must instead assume it to be `text/css`.

5.11.3.16 Link type "`sidebar`"

The `sidebar` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `sidebar` keyword indicates that the referenced document, if retrieved, is intended to be shown in a secondary browsing context (if possible), instead of in the current browsing context.

A hyperlink element with with the `sidebar` keyword specified is a **sidebar hyperlink**.

5.11.3.17 Link type "`tag`"

The `tag` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `tag` keyword indicates that the *tag* that the referenced document represents applies to the current document.

> ***Note: Since it indicates that the tag applies to the current document, it would be inappropriate to use this keyword in the markup of a tag cloud, which lists the popular tag across a set of pages.***

5.11.3.18 Hierarchical link types

Some documents form part of a hierarchical structure of documents.

A hierarchical structure of documents is one where each document can have various subdocuments. The document of which a document is a subdocument is said to be the

document's *parent*. A document with no parent forms the top of the hierarchy.

A document may be part of multiple hierarchies.

5.11.3.18.1 LINK TYPE "index"

The index keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The index keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the top of the hierarchy. It conveys more information when used with the up keyword (q.v.).

Synonyms: For historical reasons, user agents must also treat the keywords "top", "contents", and "toc" like the index keyword.

5.11.3.18.2 LINK TYPE "up"

The up keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The up keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the parent of the current document.

The up keyword may be repeated within a rel attribute to indicate the hierarchical distance from the current document to the referenced document. Each occurrence of the keyword represents one further level. If the index keyword is also present, then the number of up keywords is the depth of the current page relative to the top of the hierarchy. Only one link is created for the set of one or more up keywords and, if present, the index keyword.

If the page is part of multiple hierarchies, then they should be described in different paragraphs. User agents must scope any interpretation of the up and index keywords together indicating the depth of the hierarchy to the paragraph in which the link finds itself, if any, or to the document otherwise.

When two links have both the up and index keywords specified together in the same scope and contradict each other by having a different number of up keywords, the link with the greater number of up keywords must be taken as giving the depth of the document.

> This can be used to mark up a navigation style sometimes known as bread crumbs. In the following example, the current page can be reached via two paths.
>
> ```
> <nav>
>  
>   <a href="/" rel="index up up up">Main</a> >
>   <a href="/products/" rel="up up">Products</a> >
>   <a href="/products/dishwashers/" rel="up">Dishwashers</a> >
>   <a>Second hand</a>
>  
>  
>   <a href="/" rel="index up up">Main</a> >
>   <a href="/second-hand/" rel="up">Second hand</a> >
>   <a>Dishwashers</a>
>  
> </nav>
> ```

Note: The `relList` DOM attribute (e.g. on the `a` element) does not currently represent multiple `up` keywords (the interface hides duplicates).

5.11.3.19 Sequential link types

Some documents form part of a sequence of documents.

A sequence of documents is one where each document can have a *previous sibling* and a *next sibling*. A document with no previous sibling is the start of its sequence, a document with no next sibling is the end of its sequence.

A document may be part of multiple sequences.

5.11.3.19.1 LINK TYPE "`first`"

The `first` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `first` keyword indicates that the document is part of a sequence, and that the link is leading to the document that is the first logical document in the sequence.

Synonyms: For historical reasons, user agents must also treat the keywords "`begin`" and "`start`" like the `first` keyword.

5.11.3.19.2 LINK TYPE "`last`"

The `last` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `last` keyword indicates that the document is part of a sequence, and that the link is leading to the document that is the last logical document in the sequence.

Synonyms: For historical reasons, user agents must also treat the keyword "`end`" like the `last` keyword.

5.11.3.19.3 LINK TYPE "`next`"

The `next` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `next` keyword indicates that the document is part of a sequence, and that the link is leading to the document that is the next logical document in the sequence.

5.11.3.19.4 LINK TYPE "`prev`"

The `prev` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates

a hyperlink.

The prev keyword indicates that the document is part of a sequence, and that the link is leading to the document that is the previous logical document in the sequence.

Synonyms: For historical reasons, user agents must also treat the keyword "previous" like the prev keyword.

5.11.3.20 Other link types

Other than the types defined above, only types defined as extensions in the WHATWG Wiki RelExtensions page may be used with the rel attribute on link, a, and area elements. [WHATWGWIKI]

Anyone is free to edit the WHATWG Wiki RelExtensions page at any time to add a type. Extension types must be specified with the following information:

Keyword
: The actual value being defined. The value should not be confusingly similar to any other defined value (e.g. differing only in case).

Effect on... link
: One of the following:

  not allowed
  : The keyword is not allowed to be specified on link elements.

  Hyperlink
  : The keyword may be specified on a link element; it creates a hyperlink link.

  External Resource
  : The keyword may be specified on a link element; it creates a external resource link.

Effect on... a and area
: One of the following:

  not allowed
  : The keyword is not allowed to be specified on a and area elements.

  Hyperlink
  : The keyword may be specified on a and area elements.

Brief description
: A short description of what the keyword's meaning is.

Link to more details
: A link to a more detailed description of the keyword's semantics and requirements. It could be another page on the Wiki, or a link to an external page.

Synonyms
: A list of other keyword values that have exactly the same processing requirements. Authors must not use the values defined to be synonyms, they are only intended to allow user agents to support legacy content.

Status

One of the following:

Proposal
: The keyword has not received wide peer review and approval. It is included for completeness because pages use the keyword. Pages should not use the keyword.

Accepted
: The keyword has received wide peer review and approval. It has a specification that unambiguously defines how to handle pages that use the keyword, including when they use them in incorrect ways. Pages may use the keyword.

Rejected
: The keyword has received wide peer review and it has been found to have significant problems. Pages must not use the keyword. When a keyword has this status, the "Effect on... `link`" and "Effect on... `a` and `area`" information should be set to "not allowed".

If a keyword is added with the "proposal" status and found to be redundant with existing values, it should be removed and listed as a synonym for the existing value. If a keyword is added with the "proposal" status and found to be harmful, then it should be changed to "rejected" status, and its "Effect on..." information should be changed accordingly.

Conformance checkers must use the information given on the WHATWG Wiki RelExtensions page to establish if a value not explicitly defined in this specification is allowed or not. When an author uses a new type not defined by either this specification or the Wiki page, conformance checkers should offer to add the value to the Wiki, with the details described above, with the "proposal" status.

This specification does not define how new values will get approved. It is expected that the Wiki will have a community that addresses this.

6 User Interaction

This section describes various features that allow authors to enable users to edit documents and parts of documents interactively.

6.1 Introduction

This section is non-normative.

Would be nice to explain how these features work together.

6.2 The `hidden` attribute

All elements may have the `hidden` content attribute set. The `hidden` attribute is a boolean attribute. When specified on an element, it indicates that the element is not yet, or is no longer, relevant. User agents should not render elements that have the `hidden` attribute specified.

> In the following skeletal example, the attribute is used to hide the Web game's main screen until the user logs in:
>
> ```
> <h1>The Example Game</h1>
> <section id="login">
>  <h2>Login</h2>
>  <form>
>   ...
>   <!-- calls login() once the user's credentials have been checked -->
>  </form>
>  <script>
>   function login() {
>     // switch screens
>     document.getElementById('login').hidden = true;
>     document.getElementById('game').hidden = false;
>   }
>  </script>
> </section>
> <section id="game" hidden>
>  ...
> </section>
> ```

The `hidden` attribute must not be used to hide content that could legitimately be shown in another presentation. For example, it is incorrect to use `hidden` to hide panels in a tabbed dialog, because the tabbed interface is merely a kind of overflow presentation — showing all the form controls in one big page with a scrollbar would be equivalent, and no less correct.

Elements in a section hidden by the `hidden` attribute are still active, e.g. scripts and form controls in such sections still render execute and submit respectively. Only their presentation to the user changes.

The **`hidden`** DOM attribute must reflect the content attribute of the same name.

6.3 Activation

element **`.click()`**

Acts as if the element was clicked.

Each element has a *click in progress* flag, initially set to false.

The **`click()`** method must run these steps:

1. If the element's *click in progress* flag is set to true, then abort these steps.
2. Set the *click in progress* flag on the element to true.
3. If the element has a defined activation behavior, run synthetic click activation steps on the element. Otherwise, fire a `click` event at the element.
4. Set the *click in progress* flag on the element to false.

6.4 Scrolling elements into view

element **`.scrollIntoView(`** **[** ***top*** **]** **`)`**

Scrolls the element into view. If the *top* argument is true, then the element will be scrolled to the top of the viewport, otherwise it'll be scrolled to the bottom. The default is the top.

The **`scrollIntoView([`*`top`*`])`** method, when called, must cause the element on which the method was called to have the attention of the user called to it.

Note: In a speech browser, this could happen by having the current playback position move to the start of the given element.

In visual user agents, if the argument is present and has the value false, the user agent should scroll the element into view such that both the bottom and the top of the element are in the viewport, with the bottom of the element aligned with the bottom of the viewport. If it isn't possible to show the entire element in that way, or if the argument is omitted or is true, then the user agent should instead align the top of the element with the top of the viewport. If the entire scrollable part of the content is visible all at once (e.g. if a page is shorter than the viewport), then the user agent should not scroll anything. Visual user agents should further

scroll horizontally as necessary to bring the element to the attention of the user.

Non-visual user agents may ignore the argument, or may treat it in some media-specific manner most useful to the user.

6.5 Focus

When an element is *focused*, key events received by the document must be targeted at that element. There may be no element focused; when no element is focused, key events received by the document must be targetted at the body element.

User agents may track focus for each browsing context or `Document` individually, or may support only one focused elment per top-level browsing context — user agents should follow platform conventions in this regard.

Which elements within a top-level browsing context currently have focus must be independent of whether or not the top-level browsing context itself has the *system focus*.

> ***Note: When an element is focused, the element matches the CSS `:focus` pseudo-class.***

6.5.1 Sequential focus navigation

The **`tabindex`** content attribute specifies whether the element is focusable, whether it can be reached using sequential focus navigation, and the relative order of the element for the purposes of sequential focus navigation. The name "tab index" comes from the common use of the "tab" key to navigate through the focusable elements. The term "tabbing" refers to moving forward through the focusable elements that can be reached using sequential focus navigation.

The `tabindex` attribute, if specified, must have a value that is a valid integer.

If the attribute is specified, it must be parsed using the rules for parsing integers. The attribute's values have the following meanings:

If the attribute is omitted or parsing the value returns an error
: The user agent should follow platform conventions to determine if the element is to be focusable and, if so, whether the element can be reached using sequential focus navigation, and if so, what its relative order should be.

If the value is a negative integer
: The user agent must allow the element to be focused, but should not allow the element to be reached using sequential focus navigation.

If the value is a zero
: The user agent must allow the element to be focused, should allow the element to be reached using sequential focus navigation, and should follow platform conventions to determine the element's relative order.

If the value is greater than zero

The user agent must allow the element to be focused, should allow the element to be reached using sequential focus navigation, and should place the element in the sequential focus navigation order so that it is:

- before any focusable element whose `tabindex` attribute has been omitted or whose value, when parsed, returns an error,
- before any focusable element whose `tabindex` attribute has a value equal to or less than zero,
- after any element whose `tabindex` attribute has a value greater than zero but less than the value of the `tabindex` attribute on the element,
- after any element whose `tabindex` attribute has a value equal to the value of the `tabindex` attribute on the element but that is earlier in the document in tree order than the element,
- before any element whose `tabindex` attribute has a value equal to the value of the `tabindex` attribute on the element but that is later in the document in tree order than the element, and
- before any element whose `tabindex` attribute has a value greater than the value of the `tabindex` attribute on the element.

An element is **specially focusable** if the `tabindex` attribute's definition above defines the element to be focusable.

An element that is specially focusable but does not otherwise have an activation behavior defined has an activation behavior that does nothing.

Note: This means that an element that is only focusable because of its `tabindex` attribute will fire a `click` event in response to a non-mouse activation (e.g. hitting the "enter" key while the element is focused).

An element is **focusable** if the user agent's default behavior allows it to be focusable or if the element is specially focusable, but only if the element is being rendered.

User agents should make the following elements focusable, unless platform conventions dictate otherwise:

- `a` elements that have an `href` attribute
- `area` elements that have an `href` attribute
- `link` elements that have an `href` attribute
- `bb` elements whose `type` attribute is in a state whose *relevance* is true
- `button` elements that are not disabled
- `input` elements whose `type` attribute are not in the Hidden state and that are not disabled

- `select` elements that are not disabled
- `textarea` elements that are not disabled
- `command` elements that do not have a `disabled` attribute

The **`tabIndex`** DOM attribute must reflect the value of the `tabindex` content attribute. If the attribute is not present, or parsing its value returns an error, then the DOM attribute must return 0 for elements that are focusable and −1 for elements that are not focusable.

6.5.2 Focus management

The **focusing steps** are as follows:

1. If focusing the element will remove the focus from another element, then run the unfocusing steps for that element.
2. Make the element the currently focused element in its top-level browsing context.

   Some elements, most notably `area`, can correspond to more than one distinct focusable area. If a particular area was indicated when the element was focused, then that is the area that must get focus; otherwise, e.g. when using the `focus()` method, the first such region in tree order is the one that must be focused.
3. Fire a simple event that doesn't bubble called `focus` at the element.

User agents must run the focusing steps for an element whenever the user moves the focus to a focusable element.

The **unfocusing steps** are as follows:

1. If the element is an `input` element, and the `change` event applies to the element, and the element does not have a defined activation behavior, and the user has changed the element's value or its list of selected files while the control was focused without committing that change, then fire a simple event called `change` at the element, then broadcast `formchange` events at the element's form owner.
2. Unfocus the element.
3. Fire a simple event that doesn't bubble called `blur` at the element.

When an element that is focused stops being a focusable element, or stops being focused without another element being explicitly focused in its stead, the user agent should run the focusing steps for the body element, if there is one; if there is not, then the user agent should run the unfocusing steps for the affected element only.

> For example, this might happen because the element is removed from its `Document`, or has a `hidden` attribute added. It would also happen to an `input` element when the element gets disabled.

6.5.3 Document-level focus APIs

***document* . `activeElement`**

Returns the currently focused element.

***document* . `hasFocus`()**

Returns true if the document has focus; otherwise, returns false.

***window* . `focus`()**

Focuses the window. Use of this method is discouraged. Allow the user to control window focus instead.

***window* . `blur`()**

Unfocuses the window. Use of this method is discouraged. Allow the user to control window focus instead.

The **`activeElement`** attribute on `DocumentHTML` objects must return the element in the document that is focused. If no element in the `Document` is focused, this must return the body element.

The **`hasFocus()`** method on `DocumentHTML` objects must return true if the document's browsing context is focused, and all its ancestor browsing contexts are also focused, and the top-level browsing context has the *system focus*.

The **`focus()`** method on the `Window` object, when invoked, provides a hint to the user agent that the script believes the user might be interested in the contents of the browsing context of the `Window` object on which the method was invoked.

User agents are encouraged to have this `focus()` method trigger some kind of notification.

The **`blur()`** method on the `Window` object, when invoked, provides a hint to the user agent that the script believes the user probably is not currently interested in the contents of the browsing context of the `Window` object on which the method was invoked, but that the contents might become interesting again in the future.

User agents are encouraged to ignore calls to this `blur()` method entirely.

Note: Historically the `focus()` and `blur()` methods actually affected the system focus, but hostile sites widely abuse this behavior to the user's detriment.

6.5.4 Element-level focus APIs

***element* . `focus`()**

Focuses the element.

***element*.`blur`()**

Unfocuses the element. Use of this method is discouraged. Focus another element instead.

The **`focus()`** method, when invoked, must run the following algorithm:

1. If the element is marked as *locked for focus*, then abort these steps.
2. If the element is not focusable, then abort these steps.
3. Mark the element as **locked for focus**.
4. If the element is not already focused, run the focusing steps for the element.
5. Unmark the element as *locked for focus*.

The **`blur()`** method, when invoked, should run the focusing steps for the body element, if there is one; if there is not, then it should run the unfocusing steps for the element on which the method was called instead. User agents may selectively or uniformly ignore calls to this method for usability reasons.

6.6 The text selection APIs

Every browsing context has **a selection**. The selection can be empty, and the selection can have more than one range (a disjointed selection). The user agent should allow the user to change the selection. User agents are not required to let the user select more than one range, and may collapse multiple ranges in the selection to a single range when the user interacts with the selection. (But, of course, the user agent may let the user create selections with multiple ranges.)

This one selection must be shared by all the content of the browsing context (though not by nested browsing contexts), including any editing hosts in the document. (Editing hosts that are not inside a document cannot have a selection.)

If the selection is empty (collapsed, so that it has only one segment and that segment's start and end points are the same) then the selection's position should equal the caret position. When the selection is not empty, this specification does not define the caret position; user agents should follow platform conventions in deciding whether the caret is at the start of the selection, the end of the selection, or somewhere else.

On some platforms (such as those using Wordstar editing conventions), the caret position is totally independent of the start and end of the selection, even when the selection is empty. On such platforms, user agents may ignore the requirement that the cursor position be linked to the position of the selection altogether.

Mostly for historical reasons, in addition to the browsing context's selection, each `textarea` and `input` element has an independent selection. These are the **text field selections**.

User agents may selectively ignore attempts to use the API to adjust the selection made after

the user has modified the selection. For example, if the user has just selected part of a word, the user agent could ignore attempts to use the API call to immediately unselect the selection altogether, but could allow attempts to change the selection to select the entire word.

User agents may also allow the user to create selections that are not exposed to the API.

The `datagrid` and `select` elements also have selections, indicating which items have been picked by the user. These are not discussed in this section.

Note: This specification does not specify how selections are presented to the user. The Selectors specification, in conjunction with CSS, can be used to style text selections using the `::selection` pseudo-element. [SELECTORS] [CSS21]

6.6.1 APIs for the browsing context selection

window **. `getSelection`()**
document **. `getSelection`()**

Returns the `Selection` object for the window, which stringifies to the text of the current selection.

The **`getSelection()`** method on the `Window` interface must return the `Selection` object representing the selection of that `Window` object's browsing context.

For historical reasons, the **`getSelection()`** method on the `HTMLDocument` interface must return the same `Selection` object.

```
[Stringifies] interface Selection {
  readonly attribute Node anchorNode;
  readonly attribute long anchorOffset;
  readonly attribute Node focusNode;
  readonly attribute long focusOffset;
  readonly attribute boolean isCollapsed;
  void collapse(in Node parentNode, in long offset);
  void collapseToStart();
  void collapseToEnd();
  void selectAllChildren(in Node parentNode);
  void deleteFromDocument();
  readonly attribute long rangeCount;
  Range getRangeAt(in long index);
  void addRange(in Range range);
  void removeRange(in Range range);
  void removeAllRanges();
};
```

The `Selection` interface is represents a list of `Range` objects. The first item in the list has index 0, and the last item has index *count*-1, where *count* is the number of ranges in the list. [DOM2RANGE]

All of the members of the `Selection` interface are defined in terms of operations on the `Range` objects represented by this object. These operations can raise exceptions, as defined for the

`Range` interface; this can therefore result in the members of the `Selection` interface raising exceptions as well, in addition to any explicitly called out below.

selection **.** **`anchorNode`**

Returns the element that contains the start of the selection.

Returns null if there's no selection.

selection **.** **`anchorOffset`**

Returns the offset of the start of the selection relative to the element that contains the start of the selection.

Returns 0 if there's no selection.

selection **.** **`focusNode`**

Returns the element that contains the end of the selection.

Returns null if there's no selection.

selection **.** **`focusOffset`**

Returns the offset of the end of the selection relative to the element that contains the end of the selection.

Returns 0 if there's no selection.

collapsed **=** ***selection*** **.** **`isCollapsed`****()**

Returns true if there's no selection or if the selection is empty. Otherwise, returns false.

selection **.** **`collapsed`****(*****parentNode*****,** ***offset*****)**

Replaces the selection with an empty one at the given position.

Throws a `WRONG_DOCUMENT_ERR` exception if the given node is in a different document.

selection **.** **`collapseToStart`****()**

Replaces the selection with an empty one at the position of the start of the current selection.

Throws an `INVALID_STATE_ERR` exception if there is no selection.

selection **.** **`collapseToEnd`****()**

Replaces the selection with an empty one at the position of the end of the current selection.

Throws an `INVALID_STATE_ERR` exception if there is no selection.

selection **.** **`selectAllChildren`****(*****parentNode*****)**

Replaces the selection with one that contains all the contents of the given element.

Throws a `WRONG_DOCUMENT_ERR` exception if the given node is in a different

document.

selection **. deleteFromDocument()**

Deletes the selection.

selection **. rangeCount**

Returns the number of ranges in the selection.

selection **. getRangeAt(*index*)**

Returns the given range.

Throws an INVALID_STATE_ERR exception if the value is out of range.

selection **. addRange(*range*)**

Adds the given range to the selection.

selection **. removeRange(*range*)**

Removes the given range from the selection, if the range was one of the ones in the selection.

selection **. removeAllRanges()**

Removes all the ranges in the selection.

The **anchorNode** attribute must return the value returned by the `startContainer` attribute of the last `Range` object in the list, or null if the list is empty.

The **anchorOffset** attribute must return the value returned by the `startOffset` attribute of the last `Range` object in the list, or 0 if the list is empty.

The **focusNode** attribute must return the value returned by the `endContainer` attribute of the last `Range` object in the list, or null if the list is empty.

The **focusOffset** attribute must return the value returned by the `endOffset` attribute of the last `Range` object in the list, or 0 if the list is empty.

The **isCollapsed** attribute must return true if there are zero ranges, or if there is exactly one range and its `collapsed` attribute is itself true. Otherwise it must return false.

The **collapse(*parentNode, offset*)** method must raise a WRONG_DOCUMENT_ERR DOM exception if *parentNode*'s `Document` is not the HTMLDocument object with which the `Selection` object is associated. Otherwise it is, and the method must remove all the ranges in the `Selection` list, then create a new `Range` object, add it to the list, and invoke its `setStart()` and `setEnd()` methods with the *parentNode* and *offset* values as their arguments.

The **collapseToStart()** method must raise an INVALID_STATE_ERR DOM exception if there are no ranges in the list. Otherwise, it must invoke the `collapse()` method with the `startContainer` and `startOffset` values of the first `Range` object in the list as the arguments.

The **collapseToEnd()** method must raise an INVALID_STATE_ERR DOM exception if there are no

ranges in the list. Otherwise, it must invoke the `collapse()` method with the `endContainer` and `endOffset` values of the last `Range` object in the list as the arguments.

The **`selectAllChildren(`*`parentNode`*`)`** method must invoke the `collapse()` method with the *parentNode* value as the first argument and 0 as the second argument, and must then invoke the `selectNodeContents()` method on the first (and only) range in the list with the *parentNode* value as the argument.

The **`deleteFromDocument()`** method must invoke the `deleteContents()` method on each range in the list, if any, from first to last.

The **`rangeCount`** attribute must return the number of ranges in the list.

The **`getRangeAt(`*`index`*`)`** method must return the *index*th range in the list. If *index* is less than zero or greater or equal to the value returned by the `rangeCount` attribute, then the method must raise an `INDEX_SIZE_ERR` DOM exception.

The **`addRange(`*`range`*`)`** method must add the given *range* Range object to the list of selections, at the end (so the newly added range is the new last range). Duplicates are not prevented; a range may be added more than once in which case it appears in the list more than once, which (for example) will cause stringification to return the range's text twice.

The **`removeRange(`*`range`*`)`** method must remove the first occurrence of *range* in the list of ranges, if it appears at all.

The **`removeAllRanges()`** method must remove all the ranges from the list of ranges, such that the `rangeCount` attribute returns 0 after the `removeAllRanges()` method is invoked (and until a new range is added to the list, either through this interface or via user interaction).

Objects implementing this interface must **stringify** to a concatenation of the results of invoking the `toString()` method of the `Range` object on each of the ranges of the selection, in the order they appear in the list (first to last).

> In the following document fragment, the emphasised parts indicate the selection.
>
> ```
> The cute girl likes the <cite>Oxford English Dictionary</cite>.
> ```
>
> If a script invoked `window.getSelection().toString()`, the return value would be "`the Oxford English`".

Note: The `Selection` interface has no relation to the `DataGridSelection` interface.

6.6.2 APIs for the text field selections

The `input` and `textarea` elements define the following members in their DOM interfaces for handling their text selection:

```
void select();
         attribute unsigned long selectionStart;
         attribute unsigned long selectionEnd;
void setSelectionRange(in unsigned long start, in unsigned long end);
```

These methods and attributes expose and control the selection of `input` and `textarea` text fields.

element **. `select`()**

Selects everything in the text field.

element **. `selectionStart` [= *value*]**

Returns the offset to the start of the selection.

Can be set, to change the start of the selection.

element **. `selectionEnd` [= *value*]**

Returns the offset to the end of the selection.

Can be set, to change the end of the selection.

element **. `setSelectionRange`(*start*, *end*)**

Changes the selection to cover the given substring.

When these methods and attributes are used with `input` elements while they don't apply, they must raise an `INVALID_STATE_ERR` exception. Otherwise, they must act as described below.

The **`select()`** method must cause the contents of the text field to be fully selected.

The **`selectionStart`** attribute must, on getting, return the offset (in logical order) to the character that immediately follows the start of the selection. If there is no selection, then it must return the offset (in logical order) to the character that immediately follows the text entry cursor.

On setting, it must act as if the `setSelectionRange()` method had been called, with the new value as the first argument, and the current value of the `selectionEnd` attribute as the second argument, unless the current value of the `selectionEnd` is less than the new value, in which case the second argument must also be the new value.

The **`selectionEnd`** attribute must, on getting, return the offset (in logical order) to the character that immediately follows the end of the selection. If there is no selection, then it must return the offset (in logical order) to the character that immediately follows the text entry cursor.

On setting, it must act as if the `setSelectionRange()` method had been called, with the current value of the `selectionStart` attribute as the first argument, and new value as the second argument.

The **`setSelectionRange(`*`start`*`, `*`end`*`)`** method must set the selection of the text field to the sequence of characters starting with the character at the *start*th position (in logical order) and ending with the character at the (*end*-1)th position. Arguments greater than the length of the value in the text field must be treated as pointing at the end of the text field. If *end* is less than or equal to *start* then the start of the selection and the end of the selection must both be placed immediately before the character with offset *end*. In UAs where there is no concept of

an empty selection, this must set the cursor to be just before the character with offset *end*.

> To obtain the currently selected text, the following JavaScript suffices:
>
> ```
> var selectionText = control.value.substring(control.selectionStart, control.s
> ```
>
> ...where *control* is the `input` or `textarea` element.

Characters with no visible rendering, such as U+200D ZERO WIDTH JOINER, still count as characters. Thus, for instance, the selection can include just an invisible character, and the text insertion cursor can be placed to one side or another of such a character.

6.7 The `contenteditable` attribute

The **`contenteditable`** attribute is an enumerated attribute whose keywords are the empty string, `true`, and `false`. The empty string and the `true` keyword map to the *true* state. The `false` keyword maps to the *false* state. In addition, there is a third state, the *inherit* state, which is the *missing value default* (and the *invalid value default*).

The *true* state indicates that the element is editable. The *inherit* state indicates that the element is editable if its parent is. The *false* state indicates that the element is not editable.

Specifically, if an HTML element has a `contenteditable` attribute set to the true state, or it has its `contenteditable` attribute set to the inherit state and if its nearest ancestor HTML element with the `contenteditable` attribute set to a state other than the inherit state has its attribute set to the true state, or if it and its ancestors all have their `contenteditable` attribute set to the inherit state but the `Document` has `designMode` enabled, then the UA must treat the element as **editable** (as described below).

Otherwise, either the HTML element has a `contenteditable` attribute set to the false state, or its `contenteditable` attribute is set to the inherit state and its nearest ancestor HTML element with the `contenteditable` attribute set to a state other than the inherit state has its attribute set to the false state, or all its ancestors have their `contenteditable` attribute set to the inherit state and the `Document` itself has `designMode` disabled; either way, the element is not editable.

***element*`.contentEditable` [= *value*]**

Returns "`true`", "`false`", or "`inherit`", based on the state of the `contenteditable` attribute.

Can be set, to change that state.

Throws a `SYNTAX_ERR` exception if the new value isn't one of those strings.

***element*`.isContentEditable`**

Returns true if the element is editable; otherwise, returns false.

The **`contentEditable`** DOM attribute, on getting, must return the string "`true`" if the content attribute is set to the true state, `false`" if the content attribute is set to the false state, and

"inherit" otherwise. On setting, if the new value is an ASCII case-insensitive match for the string "inherit" then the content attribute must be removed, if the new value is an ASCII case-insensitive match for the string "true" then the content attribute must be set to the string "true", if the new value is an ASCII case-insensitive match for the string "false" then the content attribute must be set to the string "false", and otherwise the attribute setter must raise a SYNTAX_ERR exception.

The **isContentEditable** DOM attribute, on getting, must return true if the element is editable, and false otherwise.

If an element is editable and its parent element is not, or if an element is editable and it has no parent element, then the element is an **editing host**. Editable elements can be nested. User agents must make editing hosts focusable (which typically means they enter the tab order). An editing host can contain non-editable sections, these are handled as described below. An editing host can contain non-editable sections that contain further editing hosts.

When an editing host has focus, it must have a **caret position** that specifies where the current editing position is. It may also have a selection.

Note: How the caret and selection are represented depends entirely on the UA.

6.7.1 User editing actions

There are several actions that the user agent should allow the user to perform while the user is interacting with an editing host. How exactly each action is triggered is not defined for every action, but when it is not defined, suggested key bindings are provided to guide implementors.

Move the caret

User agents must allow users to move the caret to any position within an editing host, even into nested editable elements. This could be triggered as the default action of keydown events with various key identifiers and as the default action of mousedown events.

Change the selection

User agents must allow users to change the selection within an editing host, even into nested editable elements. User agents may prevent selections from being made in ways that cross from editable elements into non-editable elements (e.g. by making each non-editable descendant atomically selectable, but not allowing text selection within them). This could be triggered as the default action of keydown events with various key identifiers and as the default action of mousedown events.

Insert text

This action must be triggered as the default action of a textInput event, and may be triggered by other commands as well. It must cause the user agent to insert the specified text (given by the event object's data attribute in the case of the textInput event) at the caret.

If the caret is positioned somewhere where phrasing content is not allowed (e.g. inside an empty ol element), then the user agent must not insert the text directly at the caret position. In such cases the behavior is UA-dependent, but user agents must not, in response to a request to insert text, generate a DOM that is less conformant than the

DOM prior to the request.

User agents should allow users to insert new paragraphs into elements that contains only content other than paragraphs.

> For example, given the markup:
>
> ```
> <section>
>  <dl>
>   <dt> Ben </dt>
>   <dd> Goat </dd>
>  </dl>
> </section>
> ```
>
> ...the user agent should allow the user to insert `p` elements before and after the `dl` element, as children of the `section` element.

Break block

UAs should offer a way for the user to request that the current paragraph be broken at the caret, e.g. as the default action of a `keydown` event whose identifier is the "Enter" key and that has no modifiers set.

The exact behavior is UA-dependent, but user agents must not, in response to a request to break a paragraph, generate a DOM that is less conformant than the DOM prior to the request.

Insert a line separator

UAs should offer a way for the user to request an explicit line break at the caret position without breaking the paragraph, e.g. as the default action of a `keydown` event whose identifier is the "Enter" key and that has a shift modifier set. Line separators are typically found within a poem verse or an address. To insert a line break, the user agent must insert a `br` element.

If the caret is positioned somewhere where phrasing content is not allowed (e.g. in an empty `ol` element), then the user agent must not insert the `br` element directly at the caret position. In such cases the behavior is UA-dependent, but user agents must not, in response to a request to insert a line separator, generate a DOM that is less conformant than the DOM prior to the request.

Delete

UAs should offer a way for the user to delete text and elements, including non-editable descendants, e.g. as the default action of `keydown` events whose identifiers are "U+0008" or "U+007F".

Five edge cases in particular need to be considered carefully when implementing this feature: backspacing at the start of an element, backspacing when the caret is immediately after an element, forward-deleting at the end of an element, forward-deleting when the caret is immediately before an element, and deleting a selection whose start and end points do not share a common parent node.

In any case, the exact behavior is UA-dependent, but user agents must not, in response to a request to delete text or an element, generate a DOM that is less conformant than the DOM prior to the request.

Insert, and wrap text in, semantic elements

UAs should offer the user the ability to mark text and paragraphs with semantics that HTML can express.

UAs should similarly offer a way for the user to insert empty semantic elements to subsequently fill by entering text manually.

UAs should also offer a way to remove those semantics from marked up text, and to remove empty semantic element that have been inserted.

In response to a request from a user to mark text up in italics, user agents should use the `i` element to represent the semantic. The `em` element should be used only if the user agent is sure that the user means to indicate stress emphasis.

In response to a request from a user to mark text up in bold, user agents should use the `b` element to represent the semantic. The `strong` element should be used only if the user agent is sure that the user means to indicate importance.

The exact behavior is UA-dependent, but user agents must not, in response to a request to wrap semantics around some text or to insert or remove a semantic element, generate a DOM that is less conformant than the DOM prior to the request.

Select and move non-editable elements nested inside editing hosts

UAs should offer a way for the user to move images and other non-editable parts around the content within an editing host. This may be done using the drag and drop mechanism. User agents must not, in response to a request to move non-editable elements nested inside editing hosts, generate a DOM that is less conformant than the DOM prior to the request.

Edit form controls nested inside editing hosts

When an editable form control is edited, the changes must be reflected in both its current value *and* its default value. For `input` elements this means updating the `defaultValue` DOM attribute as well as the `value` DOM attribute; for `select` elements it means updating the `option` elements' `defaultSelected` DOM attribute as well as the `selected` DOM attribute; for `textarea` elements this means updating the `defaultValue` DOM attribute as well as the `value` DOM attribute. (Updating the `default*` DOM attributes causes content attributes to be updated as well.)

User agents may perform several commands per user request; for example if the user selects a block of text and hits `Enter`, the UA might interpret that as a request to delete the content of the selection followed by a request to break the paragraph at that position.

6.7.2 Making entire documents editable

Documents have a **`designMode`**, which can be either enabled or disabled.

document **. `designMode` [= *value*]**

Returns "`on`" if the document is editable, and "`off`" if it isn't.

Can be set, to change the document's current state.

The designMode DOM attribute on the Document object takes two values, "on" and "off". When it is set, the new value must be compared in an ASCII case-insensitive manner to these two values. If it matches the "on" value, then designMode must be enabled, and if it matches the "off" value, then designMode must be disabled. Other values must be ignored.

When designMode is enabled, the DOM attribute must return the value "on", and when it is disabled, it must return the value "off".

The last state set must persist until the document is destroyed or the state is changed. Initially, documents must have their designMode disabled.

6.8 Spelling and grammar checking

User agents can support the checking of spelling and grammar of editable text, either in form controls (such as the value of textarea elements), or in elements in an editing host (using contenteditable).

For each element, user agents must establish a **default behavior**, either through defaults or through preferences expressed by the user. There are three possible default behaviors for each element:

true-by-default
: The element will be checked for spelling and grammar if its contents are editable.

false-by-default
: The element will never be checked for spelling and grammar.

inherit-by-default
: The element's default behavior is the same as its parent element's. Elements that have no parent element cannot have this as their default behavior.

The **spellcheck** attribute is an enumerated attribute whose keywords are true and false. The true keyword map to the *true* state. The false keyword maps to the *false* state. In addition, there is a third state, the *inherit* state, which is the *missing value default* (and the *invalid value default*).

The *true* state indicates that the element is to have its spelling and grammar checked. The *inherit* state indicates that the element is to act the same way as its parent. The *false* state indicates that the element is not to be checked.

element **.** **spellcheck** **[** **=** ***value*** **]**

Returns true if the element is to have its spelling and grammar checked; otherwise, returns false.

Can be set, to override the default and set the spellcheck content attribute.

The **spellcheck** DOM attribute, on getting, must return true if the element's spellcheck content attribute is in the *true* state, or if the element's spellcheck content attribute is in the

inherit state and the element's default behavior is true-by-default, or if the element's `spellcheck` content attribute is in the *inherit* state and the element's default behavior is inherit-by-default and the element's parent element's `spellcheck` DOM attribute would return true; otherwise, if none of those conditions applies, then the attribute must instead return false.

> ***Note: The `spellcheck` DOM attribute is not affected by user preferences that override the `spellcheck` content attribute, and therefore might not reflect the actual spellchecking state.***

On setting, if the new value is true, then the element's `spellcheck` content attribute must be set to the literal string "`true`", otherwise it must be set to the literal string "`false`".

User agents must only consider the following pieces of text as checkable for the purposes of this feature:

- The value of `input` elements to which the `readonly` attribute applies, but that are not *immutable* (i.e. that do not have the `readonly` attribute specified and that are not disabled).
- The value of `textarea` elements that do not have a `readonly` attribute and that are not disabled.
- Text in text nodes that are children of editable elements.
- Text in attributes of editable elements.

For text that is part of a text node, the element with which the text is associated is the element that is the immediate parent of the first character of the word, sentence, or other piece of text. For text in attributes, it is the attribute's element. For the values of `input` and `textarea` elements, it is the element itself.

To determine if a word, sentence, or other piece of text in an applicable element (as defined above) is to have spelling- and/or grammar-checking enabled, the UA must use the following algorithm:

1. If the user has disabled the checking for this text, then the checking is disabled.
2. Otherwise, if the user has forced the checking for this text to always be enabled, then the checking is enabled.
3. Otherwise, if the element with which the text is associated has a `spellcheck` content attribute, then: if that attribute is in the *true* state, then checking is enabled; otherwise, if that attribute is in the *false* state, then checking is disabled.
4. Otherwise, if there is an ancestor element with a `spellcheck` content attribute that is not in the *inherit* state, then: if the nearest such ancestor's `spellcheck` content attribute is in the *true* state, then checking is enabled; otherwise, checking is disabled.
5. Otherwise, if the element's default behavior is true-by-default, then checking is enabled.
6. Otherwise, if the element's default behavior is false-by-default, then checking is

disabled.

7. Otherwise, if the element's parent element has *its* checking enabled, then checking is enabled.

8. Otherwise, checking is disabled.

If the checking is enabled for a word/sentence/text, the user agent should indicate spelling and/or grammar errors in that text. User agents should take into account the other semantics given in the document when suggesting spelling and grammar corrections. User agents may use the language of the element to determine what spelling and grammar rules to use, or may use the user's preferred language settings. UAs should use `input` element attributes such as `pattern` to ensure that the resulting value is valid, where possible.

If checking is disabled, the user agent should not indicate spelling or grammar errors for that text.

> The element with ID "a" in the following example would be the one used to determine if the word "Hello" is checked for spelling errors. In this example, it would not be.
>
> ```
> 
>  Hello!
> 
> ```
>
> The element with ID "b" in the following example would have checking enabled (the leading space character in the attribute's value on the `input` element causes the attribute to be ignored, so the ancestor's value is used instead, regardless of the default).
>
> ```
> 
>  <label>Name: <input spellcheck=" false" id="b"></label>
> 
> ```

6.9 Drag and drop

This section defines an event-based drag-and-drop mechanism.

This specification does not define exactly what a *drag-and-drop operation* actually is.

On a visual medium with a pointing device, a drag operation could be the default action of a `mousedown` event that is followed by a series of `mousemove` events, and the drop could be triggered by the mouse being released.

On media without a pointing device, the user would probably have to explicitly indicate his intention to perform a drag-and-drop operation, stating what he wishes to drag and what he wishes to drop, respectively.

However it is implemented, drag-and-drop operations must have a starting point (e.g. where the mouse was clicked, or the start of the selection or element that was selected for the drag), may have any number of intermediate steps (elements that the mouse moves over during a drag, or elements that the user picks as possible drop points as he cycles through possibilities), and must either have an end point (the element above which the mouse button was released, or the element that was finally selected), or be canceled. The end point must

be the last element selected as a possible drop point before the drop occurs (so if the operation is not canceled, there must be at least one element in the middle step).

6.9.1 Introduction

This section is non-normative.

It's also currently non-existent.

6.9.2 The `DragEvent` and `DataTransfer` interfaces

The drag-and-drop processing model involves several events. They all use the `DragEvent` interface.

```
interface DragEvent : MouseEvent {
  readonly attribute DataTransfer dataTransfer;

  void initDragEvent(in DOMString typeArg, in boolean canBubbleArg, in boolean can
  void initDragEventNS(in DOMString namespaceURIArg, in DOMString typeArg, in bool
};
```

***event*.`dataTransfer`**

Returns the `DataTransfer` object for the event.

The **`initDragEvent()`** and **`initDragEventNS()`** methods must initialize the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS]

Note: The `initDragEvent()` and `initDragEventNS()` methods handle modifier keys differently, much like the equivalent methods on the `MouseEvent` interface.

The **`dataTransfer`** attribute of the `DragEvent` interface represents the context information for the event.

```
interface DataTransfer {
           attribute DOMString dropEffect;
           attribute DOMString effectAllowed;
  readonly attribute DOMStringList types;
  void clearData([Optional] in DOMString format);
  void setData(in DOMString format, in DOMString data);
  DOMString getData(in DOMString format);
  void setDragImage(in Element image, in long x, in long y);
  void addElement(in Element element);
};
```

`DataTransfer` objects can hold pieces of data, each associated with a unique format. Formats are generally given by MIME types, with some values special-cased for legacy reasons. For the purposes of this API, however, the format strings are opaque, case-sensitive, strings, and the empty string is a valid format string.

dataTransfer **.** **`dropEffect`** **[=** ***value*** **]**

Returns the kind of operation that is currently selected. If the kind of operation isn't one of those that is allowed by the `effectAllowed` attribute, then the operation will fail.

Can be set, to change the selected operation.

The possible values are `none`, `copy`, `link`, and `move`.

dataTransfer **.** **`effectAllowed`** **[=** ***value*** **]**

Returns the kinds of operations that are to be allowed.

Can be set, to change the allowed operations.

The possible values are `none`, `copy`, `copyLink`, `copyMove`, `link`, `linkMove`, `move`, `all`, and `uninitialized`,

dataTransfer **.** **`types`**

Returns a `DOMStringList` of the formats available.

dataTransfer **.** **`clearData`****(** **[** ***format*** **]** **)**

Removes the data of the specified formats. Removes all data if the argument is omitted.

dataTransfer **.** **`setData`****(*****format*****,** ***data*****)**

Adds the specified data.

data **=** ***dataTransfer*** **.** **`getData`****(*****format*****)**

Returns the specified data. If there is no such data, returns the empty string.

dataTransfer **.** **`setDragImage`****(*****element*****,** ***x*****,** ***y*****)**

Uses the given element to update the drag feedback, replacing any previously specified feedback.

dataTransfer **.** **`addElement`****(*****element*****)**

Adds the given element to the list of elements used to render the drag feedback.

When a `DataTransfer` object is created, it must be initialized as follows:

- The `DataTransfer` object must initially contain no data, no elements, and have no associated image.
- The `DataTransfer` object's `effectAllowed` attribute must be set to "`uninitialized`".
- The `dropEffect` attribute must be set to "`none`".

The **`dropEffect`** attribute controls the drag-and-drop feedback that the user is given during a drag-and-drop operation.

The attribute must ignore any attempts to set it to a value other than `none`, `copy`, `link`, and `move`. On getting, the attribute must return the last of those four values that it was set to.

The **`effectAllowed`** attribute is used in the drag-and-drop processing model to initialize the `dropEffect` attribute during the `dragenter` and `dragover` events.

The attribute must ignore any attempts to set it to a value other than `none`, `copy`, `copyLink`, `copyMove`, `link`, `linkMove`, `move`, `all`, and `uninitialized`. On getting, the attribute must return the last of those values that it was set to.

The **`types`** attribute must return a live `DOMStringList` that contains the list of formats that are stored in the `DataTransfer` object.

The **`clearData()`** method, when called with no arguments, must clear the `DataTransfer` object of all data (for all formats).

When called with an argument, the `clearData(`*`format`*`)` method must clear the `DataTransfer` object of any data associated with the given *format*. If *format* is the value "`Text`", then it must be treated as "`text/plain`". If the *format* is "`URL`", then it must be treated as "`text/uri-list`".

The **`setData(`*`format, data`*`)`** method must add *data* to the data stored in the `DataTransfer` object, labeled as being of the type *format*. This must replace any previous data that had been set for that format. If *format* is the value "`Text`", then it must be treated as "`text/plain`". If the *format* is "`URL`", then it must be treated as "`text/uri-list`".

The **`getData(`*`format`*`)`** method must return the data that is associated with the type *format*, if any, and must return the empty string otherwise. If *format* is the value "`Text`", then it must be treated as "`text/plain`". If the *format* is "`URL`", then the data associated with the "`text/uri-list`" format must be parsed as appropriate for `text/uri-list` data, and the first URL from the list must be returned. If there is no data with that format, or if there is but it has no URLs, then the method must return the empty string. [RFC2483]

The **`setDragImage(`*`element, x, y`*`)`** method sets which element to use to generate the drag feedback. The *element* argument can be any `Element`; if it is an `img` element, then the user agent should use the element's image (at its intrinsic size) to generate the feedback, otherwise the user agent should base the feedback on the given element (but the exact mechanism for doing so is not specified).

The **`addElement(`*`element`*`)`** method is an alternative way of specifying how the user agent is to render the drag feedback. It adds an element to the `DataTransfer` object.

> ***Note: The difference between `setDragImage()` and `addElement()` is that the latter automatically generates the image based on the current rendering of the elements added, whereas the former uses the exact specified image.***

6.9.3 Events fired during a drag-and-drop action

The following events are involved in the drag-and-drop model. Whenever the processing model described below causes one of these events to be fired, the event fired must use the `DragEvent` interface defined above, must have the bubbling and cancelable behaviors given in the table below, and must have the context information set up as described after the table,

with the `view` attribute set to the view with which the user interacted to trigger the drag-and-drop event, the `detail` attribute set to zero, the mouse and key attributes set according to the state of the input devices as they would be for user interaction events, and the `relatedTarget` attribute set to null.

Event Name	Target	Bubbles?	Cancelable?	`dataTransfer`	`effectAllowed`	`dropEffect`	
`dragstart`	Source node	✓ Bubbles	✓ Cancelable	Contains source node unless a selection is being dragged, in which case it is empty	`uninitialized`	`none`	Ini dra op
`drag`	Source node	✓ Bubbles	✓ Cancelable	Empty	Same as last event	`none`	Cc dra op
`dragenter`	Immediate user selection or the body element	✓ Bubbles	✓ Cancelable	Empty	Same as last event	Based on `effectAllowed` value	Re im us as tar ele
`dragleave`	Previous target element	✓ Bubbles	—	Empty	Same as last event	`none`	Nc
`dragover`	Current target element	✓ Bubbles	✓ Cancelable	Empty	Same as last event	Based on `effectAllowed` value	Re cu op "nc
`drop`	Current target element	✓ Bubbles	✓ Cancelable	`getData()` returns data set in `dragstart` event	Same as last event	Current drag operation	Va
`dragend`	Source node	✓ Bubbles	—	Empty	Same as last event	Current drag operation	Va

The `dataTransfer` object's contents are empty except for `dragstart` events and `drop` events, for which the contents are set as described in the processing model, below.

The `effectAllowed` attribute must be set to "`uninitialized`" for `dragstart` events, and to whatever value the field had after the last drag-and-drop event was fired for all other events (only counting events fired by the user agent for the purposes of the drag-and-drop model described below).

The `dropEffect` attribute must be set to "`none`" for `dragstart`, `drag`, and `dragleave` events (except when stated otherwise in the algorithms given in the sections below), to the value corresponding to the current drag operation for `drop` and `dragend` events, and to a value

based on the `effectAllowed` attribute's value and to the drag-and-drop source, as given by the following table, for the remaining events (`dragenter` and `dragover`):

`effectAllowed`	`dropEffect`
`none`	`none`
`copy`, `copyLink`, `copyMove`, `all`	`copy`
`link`, `linkMove`	`link`
`move`	`move`
`uninitialized` when what is being dragged is a selection from a text field	`move`
`uninitialized` when what is being dragged is a selection	`copy`
`uninitialized` when what is being dragged is an `a` element with an `href` attribute	`link`
Any other case	`copy`

6.9.4 Drag-and-drop processing model

When the user attempts to begin a drag operation, the user agent must first determine what is being dragged. If the drag operation was invoked on a selection, then it is the selection that is being dragged. Otherwise, it is the first element, going up the ancestor chain, starting at the node that the user tried to drag, that has the DOM attribute `draggable` set to true. If there is no such element, then nothing is being dragged, the drag-and-drop operation is never started, and the user agent must not continue with this algorithm.

> ***Note: `img` elements and `a` elements with an `href` attribute have their `draggable` attribute set to true by default.***

If the user agent determines that something can be dragged, a `dragstart` event must then be fired.

If it is a selection that is being dragged, then this event must be fired on the node that the user started the drag on (typically the text node that the user originally clicked). If the user did not specify a particular node, for example if the user just told the user agent to begin a drag of "the selection", then the event must be fired on the deepest node that is a common ancestor of all parts of the selection.

We should look into how browsers do other types (e.g. Firefox apparently also adds text/html for internal drag and drop of a selection).

If it is not a selection that is being dragged, then the event must be fired on the element that is being dragged.

The node on which the event is fired is the **source node**. Multiple events are fired on this node during the course of the drag-and-drop operation.

If it is a selection that is being dragged, the `dataTransfer` member of the event must be created with no nodes. Otherwise, it must be created containing just the source node. Script can use the `addElement()` method to add further elements to the list of what is being dragged.

If it is a selection that is being dragged, the `dataTransfer` member of the event must have the text of the selection added to it as the data associated with the `text/plain` format. Otherwise, if it is an `img` element being dragged, then the value of the element's `src` DOM attribute must be added, associated with the `text/uri-list` format. Otherwise, if it is an `a` element being dragged, then the value of the element's `href` DOM attribute must be added, associated with the `text/uri-list` format. Otherwise, no data is added to the object by the user agent.

If the event is canceled, then the drag-and-drop operation must not occur; the user agent must not continue with this algorithm.

If it is not canceled, then the drag-and-drop operation must be initiated.

> ***Note: Since events with no event handlers registered are, almost by definition, never canceled, drag-and-drop is always available to the user if the author does not specifically prevent it.***

The drag-and-drop feedback must be generated from the first of the following sources that is available:

1. The element specified in the last call to the `setDragImage()` method of the `dataTransfer` object of the `dragstart` event, if the method was called. In visual media, if this is used, the *x* and *y* arguments that were passed to that method should be used as hints for where to put the cursor relative to the resulting image. The values are expressed as distances in CSS pixels from the left side and from the top side of the image respectively. [CSS21]

2. The elements that were added to the `dataTransfer` object, both before the event was fired, and during the handling of the event using the `addElement()` method, if any such elements were indeed added.

3. The selection that the user is dragging.

The user agent must take a note of the data that was placed in the `dataTransfer` object. This data will be made available again when the `drop` event is fired.

From this point until the end of the drag-and-drop operation, device input events (e.g. mouse and keyboard events) must be suppressed. In addition, the user agent must track all DOM changes made during the drag-and-drop operation, and add them to its undo history as one atomic operation once the drag-and-drop operation has ended.

During the drag operation, the element directly indicated by the user as the drop target is called the **immediate user selection**. (Only elements can be selected by the user; other nodes must not be made available as drop targets.) However, the immediate user selection is not necessarily the **current target element**, which is the element currently selected for the drop part of the drag-and-drop operation. The immediate user selection changes as the user selects different elements (either by pointing at them with a pointing device, or by selecting them in some other way). The current target element changes when the immediate user selection changes, based on the results of event handlers in the document, as described below.

Both the current target element and the immediate user selection can be null, which means no target element is selected. They can also both be elements in other (DOM-based) documents, or other (non-Web) programs altogether. (For example, a user could drag text to

a word-processor.) The current target element is initially null.

In addition, there is also a **current drag operation**, which can take on the values "none", "copy", "link", and "move". Initially it has the value "none". It is updated by the user agent as described in the steps below.

User agents must, every 350ms (±200ms), perform the following steps in sequence. (If the user agent is still performing the previous iteration of the sequence when the next iteration becomes due, the user agent must not execute the overdue iteration, effectively "skipping missed frames" of the drag-and-drop operation.)

1. First, the user agent must fire a `drag` event at the source node. If this event is canceled, the user agent must set the current drag operation to none (no drag operation).

2. Next, if the `drag` event was not canceled and the user has not ended the drag-and-drop operation, the user agent must check the state of the drag-and-drop operation, as follows:

    1. First, if the user is indicating a different immediate user selection than during the last iteration (or if this is the first iteration), and if this immediate user selection is not the same as the current target element, then the current target element must be updated, as follows:

        1. If the new immediate user selection is null, or is in a non-DOM document or application, then set the current target element to the same value.

        2. Otherwise, the user agent must fire a `dragenter` event at the immediate user selection.

        3. If the event is canceled, then the current target element must be set to the immediate user selection.

        4. Otherwise, if the current target element is not the body element, the user agent must fire a `dragenter` event at the body element, and the current target element must be set to the body element, regardless of whether that event was canceled or not. (If the body element is null, then the current target element would be set to null too in this case, it wouldn't be set to the `Document` object.)

    2. If the previous step caused the current target element to change, and if the previous target element was not null or a part of a non-DOM document, the user agent must fire a `dragleave` event at the previous target element.

    3. If the current target element is a DOM element, the user agent must fire a `dragover` event at this current target element.

        If the `dragover` event is not canceled, the current drag operation must be reset to "none".

        Otherwise, the current drag operation must be set based on the values the `effectAllowed` and `dropEffect` attributes of the `dataTransfer` object had after the event was handled, as per the following table:

`effectAllowed`	`dropEffect`	Drag operation

effectAllowed	dropEffect	Drag operation
uninitialized, copy, copyLink, copyMove, or all	copy	"copy"
uninitialized, link, copyLink, linkMove, or all	link	"link"
uninitialized, move, copyMove, linkMove, or all	move	"move"
Any other case		"none"

Then, regardless of whether the dragover event was canceled or not, the drag feedback (e.g. the mouse cursor) must be updated to match the current drag operation, as follows:

Drag operation	Feedback
"copy"	Data will be copied if dropped here.
"link"	Data will be linked if dropped here.
"move"	Data will be moved if dropped here.
"none"	No operation allowed, dropping here will cancel the drag-and-drop operation.

4. Otherwise, if the current target element is not a DOM element, the user agent must use platform-specific mechanisms to determine what drag operation is being performed (none, copy, link, or move). This sets the *current drag operation*.

3. Otherwise, if the user ended the drag-and-drop operation (e.g. by releasing the mouse button in a mouse-driven drag-and-drop interface), or if the drag event was canceled, then this will be the last iteration. The user agent must execute the following steps, then stop looping.

    1. If the current drag operation is none (no drag operation), or, if the user ended the drag-and-drop operation by canceling it (e.g. by hitting the Escape key), or if the current target element is null, then the drag operation failed. If the current target element is a DOM element, the user agent must fire a dragleave event at it; otherwise, if it is not null, it must use platform-specific conventions for drag cancellation.

    2. Otherwise, the drag operation was as success. If the current target element is a DOM element, the user agent must fire a drop event at it; otherwise, it must use platform-specific conventions for indicating a drop.

        When the target is a DOM element, the dropEffect attribute of the event's dataTransfer object must be given the value representing the current drag operation (copy, link, or move), and the object must be set up so that the getData() method will return the data that was added during the dragstart event.

        If the event is canceled, the current drag operation must be set to the value of the dropEffect attribute of the event's dataTransfer object as it stood after the event was handled.

        Otherwise, the event is not canceled, and the user agent must perform the event's default action, which depends on the exact target as follows:

↪ **If the current target element is a text field (e.g. `textarea`, or an `input` element whose `type` attribute is in the Text state)**

The user agent must insert the data associated with the `text/plain` format, if any, into the text field in a manner consistent with platform-specific conventions (e.g. inserting it at the current mouse cursor position, or inserting it at the end of the field).

↪ **Otherwise**

Reset the current drag operation to "none".

3. Finally, the user agent must fire a `dragend` event at the source node, with the `dropEffect` attribute of the event's `dataTransfer` object being set to the value corresponding to the current drag operation.

Note: The current drag operation can change during the processing of the `drop` event, if one was fired.

The event is not cancelable. After the event has been handled, the user agent must act as follows:

↪ **If the current target element is a text field (e.g. `textarea`, or an `input` element whose `type` attribute is in the Text state), and a `drop` event was fired in the previous step, and the current drag operation is "move", and the source of the drag-and-drop operation is a selection in the DOM**

The user agent should delete the range representing the dragged selection from the DOM.

↪ **If the current target element is a text field (e.g. `textarea`, or an `input` element whose `type` attribute is in the Text state), and a `drop` event was fired in the previous step, and the current drag operation is "move", and the source of the drag-and-drop operation is a selection in a text field**

The user agent should delete the dragged selection from the relevant text field.

↪ **Otherwise**

The event has no default action.

6.9.4.1 When the drag-and-drop operation starts or ends in another document

The model described above is independent of which `Document` object the nodes involved are from; the events must be fired as described above and the rest of the processing model must be followed as described above, irrespective of how many documents are involved in the operation.

6.9.4.2 When the drag-and-drop operation starts or ends in another application

If the drag is initiated in another application, the source node is not a DOM node, and the user agent must use platform-specific conventions instead when the requirements above involve the source node. User agents in this situation must act as if the dragged data had been added to the `DataTransfer` object when the drag started, even though no `dragstart` event was actually fired; user agents must similarly use platform-specific conventions when deciding on what drag feedback to use.

If a drag is started in a document but ends in another application, then the user agent must instead replace the parts of the processing model relating to handling the *target* according to platform-specific conventions.

In any case, scripts running in the context of the document must not be able to distinguish the case of a drag-and-drop operation being started or ended in another application from the case of a drag-and-drop operation being started or ended in another document from another domain.

6.9.5 The `draggable` attribute

All elements may have the `draggable` content attribute set. The `draggable` attribute is an enumerated attribute. It has three states. The first state is *true* and it has the keyword `true`. The second state is *false* and it has the keyword `false`. The third state is *auto*; it has no keywords but it is the *missing value default*.

The *true* state means the element is draggable; the *false* state means that it is not. The *auto* state uses the default behavior of the user agent.

element **.** **`draggable`** **[= *value*]**

Returns true if the element is draggable; otherwise, returns false.

Can be set, to override the default and set the `draggable` content attribute.

The **`draggable`** DOM attribute, whose value depends on the content attribute's in the way described below, controls whether or not the element is draggable. Generally, only text selections are draggable, but elements whose `draggable` DOM attribute is true become draggable as well.

If an element's `draggable` content attribute has the state *true*, the `draggable` DOM attribute must return true.

Otherwise, if the element's `draggable` content attribute has the state *false*, the `draggable` DOM attribute must return false.

Otherwise, the element's `draggable` content attribute has the state *auto*. If the element is an `img` element, or, if the element is an `a` element with an `href` content attribute, the `draggable` DOM attribute must return true.

Otherwise, the `draggable` DOM must return false.

If the `draggable` DOM attribute is set to the value false, the `draggable` content attribute must be set to the literal value `false`. If the `draggable` DOM attribute is set to the value true, the `draggable` content attribute must be set to the literal value `true`.

6.9.6 Copy and paste

Copy-and-paste is a form of drag-and-drop: the "copy" part is equivalent to dragging content

to another application (the "clipboard"), and the "paste" part is equivalent to dragging content *from* another application.

Select-and-paste (a model used by mouse operations in the X Window System) is equivalent to a drag-and-drop operation where the source is the selection.

6.9.6.1 Copy to clipboard

When the user invokes a copy operation, the user agent must act as if the user had invoked a drag on the current selection. If the drag-and-drop operation initiates, then the user agent must act as if the user had indicated (as the immediate user selection) a hypothetical application representing the clipboard. Then, the user agent must act as if the user had ended the drag-and-drop operation without canceling it. If the drag-and-drop operation didn't get canceled, the user agent should then follow the relevant platform-specific conventions for copy operations (e.g. updating the clipboard).

6.9.6.2 Cut to clipboard

When the user invokes a cut operation, the user agent must act as if the user had invoked a copy operation (see the previous section), followed, if the copy was completed successfully, by a selection delete operation.

6.9.6.3 Paste from clipboard

When the user invokes a clipboard paste operation, the user agent must act as if the user had invoked a drag on a hypothetical application representing the clipboard, setting the data associated with the drag as the content on the clipboard (in whatever formats are available).

Then, the user agent must act as if the user had indicated (as the immediate user selection) the element with the keyboard focus, and then ended the drag-and-drop operation without canceling it.

6.9.6.4 Paste from selection

When the user invokes a selection paste operation, the user agent must act as if the user had invoked a drag on the current selection, then indicated (as the immediate user selection) the element with the keyboard focus, and then ended the drag-and-drop operation without canceling it.

6.9.7 Security risks in the drag-and-drop model

User agents must not make the data added to the `DataTransfer` object during the `dragstart` event available to scripts until the `drop` event, because otherwise, if a user were to drag sensitive information from one document to a second document, crossing a hostile third document in the process, the hostile document could intercept the data.

For the same reason, user agents must consider a drop to be successful only if the user

specifically ended the drag operation — if any scripts end the drag operation, it must be considered unsuccessful (canceled) and the `drop` event must not be fired.

User agents should take care to not start drag-and-drop operations in response to script actions. For example, in a mouse-and-window environment, if a script moves a window while the user has his mouse button depressed, the UA would not consider that to start a drag. This is important because otherwise UAs could cause data to be dragged from sensitive sources and dropped into hostile documents without the user's consent.

6.10 Undo history

There has got to be a better way of doing this, surely.

6.10.1 Introduction

...

6.10.2 Definitions

The user agent must associate an **undo transaction history** with each `HTMLDocument` object.

The undo transaction history is a list of entries. The entries are of two type: DOM changes and undo objects.

Each **DOM changes** entry in the undo transaction history consists of batches of one or more of the following:

- Changes to the content attributes of an `Element` node.
- Changes to the DOM attributes of a `Node`.
- Changes to the DOM hierarchy of nodes that are descendants of the `HTMLDocument` object (`parentNode`, `childNodes`).

Undo object entries consist of objects representing state that scripts running in the document are managing. For example, a Web mail application could use an undo object to keep track of the fact that a user has moved an e-mail to a particular folder, so that the user can undo the action and have the e-mail return to its former location.

Broadly speaking, DOM changes entries are handled by the UA in response to user edits of form controls and editing hosts on the page, and undo object entries are handled by script in response to higher-level user actions (such as interactions with server-side state, or in the implementation of a drawing tool).

6.10.3 The `UndoManager` interface

This API sucks. Seriously. It's a terrible API. Really bad. I hate it. Here are the requirements:

- Has to cope with cases where the server has undo state already when the page is loaded, that can be stuffed into the undo buffer onload.
- Has to support undo/redo.
- Has to cope with the "undo" action being "contact the server and tell it to undo", rather than it being the opposite of the "redo" action.
- Has to cope with some undo states expiring from the undo history (e.g. server can only remember one undelete action) but other states not expiring (e.g. client can undo arbitrary amounts of local edits).

To manage undo object entries in the undo transaction history, the `UndoManager` interface can be used:

```
interface UndoManager {
  readonly attribute unsigned long length;
  [IndexGetter] any item(in unsigned long index);
  readonly attribute unsigned long position;
  unsigned long add(in any data, in DOMString title);
  void remove(in unsigned long index);
  void clearUndo();
  void clearRedo();
};
```

***window* . `undoManager`**

Returns the `UndoManager` object.

***undoManager* . `length`**

Returns the number of entries in the undo history.

***data* = *undoManager* . `item`(*index*)**
***undoManager*[*index*]**

Returns the entry with index *index* in the undo history.

Returns null if *index* is out of range.

***undoManager* . `position`**

Returns the number of the current entry in the undo history. (Entries at and past this point are *redo* entries.)

***undoManager* . `add`(*data*, *title*)**

Adds the specified entry to the undo history.

***undoManager* . `remove`(*index*)**

Removes the specified entry to the undo history.

Throws an `INDEX_SIZE_ERR` exception if the given index is out of range.

***undoManager*`.clearUndo()`**

Removes all entries before the current position in the undo history.

***undoManager*`.clearRedo()`**

Removes all entries at and after the current position in the undo history.

The **`undoManager`** attribute of the `Window` interface must return the object implementing the `UndoManager` interface for that `Window` object's associated `HTMLDocument` object.

`UndoManager` objects represent their document's undo transaction history. Only undo object entries are visible with this API, but this does not mean that DOM changes entries are absent from the undo transaction history.

The **`length`** attribute must return the number of undo object entries in the undo transaction history. This is the *length*.

The object's indices of the supported indexed properties are the numbers in the range zero to *length*-1, unless the *length* is zero, in which case there are no supported indexed properties.

The **`item(`*`n`*`)`** method must return the *n*th undo object entry in the undo transaction history.

The undo transaction history has a **current position**. This is the position between two entries in the undo transaction history's list where the previous entry represents what needs to happen if the user invokes the "undo" command (the "undo" side, lower numbers), and the next entry represents what needs to happen if the user invokes the "redo" command (the "redo" side, higher numbers).

The **`position`** attribute must return the index of the undo object entry nearest to the undo position, on the "redo" side. If there are no undo object entries on the "redo" side, then the attribute must return the same as the `length` attribute. If there are no undo object entries on the "undo" side of the undo position, the `position` attribute returns zero.

Note: Since the undo transaction history contains both undo object entries and DOM changes entries, but the `position` attribute only returns indices relative to undo object entries, it is possible for several "undo" or "redo" actions to be performed without the value of the `position` attribute changing.

The **`add(`*`data, title`*`)`** method's behavior depends on the current state. Normally, it must insert the *data* object passed as an argument into the undo transaction history immediately before the undo position, optionally remembering the given *title* to use in the UI. If the method is called during an undo operation, however, the object must instead be added immediately *after* the undo position.

If the method is called and there is neither an undo operation in progress nor a redo operation in progress then any entries in the undo transaction history after the undo position must be removed (as if `clearRedo()` had been called).

We could fire events when someone adds something to the undo history -- one event per undo object entry before the position (or after, during redo addition), allowing the script to decide if that entry should remain or not. Or something. Would make it potentially easier to expire server-held state when the server limitations come into play.

The **`remove(`*`index`*`)`** method must remove the undo object entry with the specified *index*. If the index is less than zero or greater than or equal to `length` then the method must raise an `INDEX_SIZE_ERR` exception. DOM changes entries are unaffected by this method.

The **`clearUndo()`** method must remove all entries in the undo transaction history before the undo position, be they DOM changes entries or undo object entries.

The **`clearRedo()`** method must remove all entries in the undo transaction history after the undo position, be they DOM changes entries or undo object entries.

Another idea is to have a way for scripts to say "startBatchingDOMChangesForUndo()" and after that the changes to the DOM go in as if the user had done them.

6.10.4 Undo: moving back in the undo transaction history

When the user invokes an undo operation, or when the `execCommand()` method is called with the `undo` command, the user agent must perform an undo operation.

If the undo position is at the start of the undo transaction history, then the user agent must do nothing.

If the entry immediately before the undo position is a DOM changes entry, then the user agent must remove that DOM changes entry, reverse the DOM changes that were listed in that entry, and, if the changes were reversed with no problems, add a new DOM changes entry (consisting of the opposite of those DOM changes) to the undo transaction history on the other side of the undo position.

If the DOM changes cannot be undone (e.g. because the DOM state is no longer consistent with the changes represented in the entry), then the user agent must simply remove the DOM changes entry, without doing anything else.

If the entry immediately before the undo position is an undo object entry, then the user agent must first remove that undo object entry from the undo transaction history, and then must fire an `undo` event on the `Document` object, using the undo object entry's associated undo object as the event's data.

Any calls to `add()` while the event is being handled will be used to populate the redo history, and will then be used if the user invokes the "redo" command to undo his undo.

6.10.5 Redo: moving forward in the undo transaction history

When the user invokes a redo operation, or when the `execCommand()` method is called with the `redo` command, the user agent must perform a redo operation.

This is mostly the opposite of an undo operation, but the full definition is included here for

completeness.

If the undo position is at the end of the undo transaction history, then the user agent must do nothing.

If the entry immediately after the undo position is a DOM changes entry, then the user agent must remove that DOM changes entry, reverse the DOM changes that were listed in that entry, and, if the changes were reversed with no problems, add a new DOM changes entry (consisting of the opposite of those DOM changes) to the undo transaction history on the other side of the undo position.

If the DOM changes cannot be redone (e.g. because the DOM state is no longer consistent with the changes represented in the entry), then the user agent must simply remove the DOM changes entry, without doing anything else.

If the entry immediately after the undo position is an undo object entry, then the user agent must first remove that undo object entry from the undo transaction history, and then must fire a `redo` event on the `Document` object, using the undo object entry's associated undo object as the event's data.

6.10.6 The `UndoManagerEvent` interface and the `undo` and `redo` events

```
interface UndoManagerEvent : Event {
  readonly attribute any data;
  void initUndoManagerEvent(in DOMString typeArg, in boolean canBubbleArg, in bool
  void initUndoManagerEventNS(in DOMString namespaceURIArg, in DOMString typeArg,
};
```

***event*`.data`**

Returns the data that was passed to the `add()` method.

The **`initUndoManagerEvent()`** and **`initUndoManagerEventNS()`** methods must initialize the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS]

The **`data`** attribute represents the undo object for the event.

The **`undo`** and **`redo`** events do not bubble, cannot be canceled, and have no default action. When the user agent fires one of these events it must use the `UndoManagerEvent` interface, with the `data` field containing the relevant undo object.

6.10.7 Implementation notes

How user agents present the above conceptual model to the user is not defined. The undo interface could be a filtered view of the undo transaction history, it could manipulate the undo transaction history in ways not described above, and so forth. For example, it is possible to design a UA that appears to have separate undo transaction histories for each form control; similarly, it is possible to design systems where the user has access to more undo information

than is present in the official (as described above) undo transaction history (such as providing a tree-based approach to document state). Such UI models should be based upon the single undo transaction history described in this section, however, such that to a script there is no detectable difference.

6.11 Command APIs

document **.** **`execCommand`**(***commandId*** **[,** ***showUI*** **[,** ***value*** **]])**

Runs the action specified by the first argument, as described in the list below. The second and third arguments sometimes affect the action. (If they don't they are ignored.)

document **.** **`queryCommandEnabled`**(***commandId***)

Returns whether the given command is enabled, as described in the list below.

document **.** **`queryCommandIndeterm`**(***commandId***)

Returns whether the given command is indeterminate, as described in the list below.

document **.** **`queryCommandState`**(***commandId***)

Returns the state of the command, as described in the list below.

document **.** **`queryCommandSupported`**(***commandId***)

Returns true if the command is supported; otherwise, returns false.

document **.** **`queryCommandValue`**(***commandId***)

Returns teh value of the command, as described in the list below.

The **`execCommand(`*`commandId, showUI, value`*`)`** method on the `HTMLDocument` interface allows scripts to perform actions on the current selection or at the current caret position. Generally, these commands would be used to implement editor UI, for example having a "delete" button on a toolbar.

There are three variants to this method, with one, two, and three arguments respectively. The *showUI* and *value* parameters, even if specified, are ignored unless otherwise stated.

When `execCommand()` is invoked, the user agent must follow the following steps:

1. If the given *commandId* maps to an entry in the list below whose "Enabled When" entry has a condition that is currently false, do nothing; abort these steps.

2. Otherwise, execute the "Action" listed below for the given *commandId*.

A document is **ready for editing host commands** if it has a selection that is entirely within an editing host, or if it has no selection but its caret is inside an editing host.

The **`queryCommandEnabled(`*`commandId`*`)`** method, when invoked, must return true if the condition listed below under "Enabled When" for the given *commandId* is true, and false otherwise.

The **`queryCommandIndeterm(`*`commandId`*`)`** method, when invoked, must return true if the condition listed below under "Indeterminate When" for the given *commandId* is true, and false otherwise.

The **`queryCommandState(`*`commandId`*`)`** method, when invoked, must return the value expressed below under "State" for the given *commandId*.

The **`queryCommandSupported(`*`commandId`*`)`** method, when invoked, must return true if the given *commandId* is in the list below, and false otherwise.

The **`queryCommandValue(`*`commandId`*`)`** method, when invoked, must return the value expressed below under "Value" for the given *commandId*.

The possible values for *commandId*, and their corresponding meanings, are as follows. These values must be compared to the argument in an ASCII case-insensitive manner.

`bold`

Summary: Toggles whether the selection is bold.
Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the *`b`* element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: True if the selection, or the caret, if there is no selection, is, or is contained within, a `b` element. False otherwise.
Value: The string "`true`" if the expression given for the "State" above is true, the string "`false`" otherwise.

`createLink`

Summary: Toggles whether the selection is a link or not. If the second argument is true, and a link is to be added, the user agent will ask the user for the address. Otherwise, the third argument will be used as the address.
Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the *`a`* element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA). If the user agent creates an `a` element or modifies an existing `a` element, then if the *showUI* argument is present and has the value false, then the value of the *value* argument must be used as the URL of the link. Otherwise, the user agent should prompt the user for the URL of the link.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`delete`

Summary: Deletes the selection or the character before the cursor.
Action: The user agent must act as if the user had performed a backspace operation.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

formatBlock

Summary: Wraps the selection in the element given by the second argument. If the second argument doesn't specify an element that is prose content but not phrasing content, does nothing.
Action: The user agent must run the following steps:

1. If the *value* argument wasn't specified, abort these steps without doing anything.

2. If the *value* argument has a leading U+003C LESS-THAN SIGN character ('<') and a trailing U+003E GREATER-THAN SIGN character ('>'), then remove the first and last characters from *value*.

3. If *value* is (now) an ASCII case-insensitive match for the tag name of an element defined by this specification that is defined to be prose content but not phrasing content, then, for every position in the selection, take the furthest flow content ancestor element of that position that contains only phrasing content, and, if that element is editable, and has a content model that allows it to contain prose content other than phrasing content, and has a parent element whose content model allows that parent to contain any prose content, rename the element (as if the `Element.renameNode()` method had been used) to *value*, using the HTML namespace.

   If there is no selection, then, where in the description above refers to the selection, the user agent must act as if the selection was an empty range (with just one position) at the caret position.

Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

forwardDelete

Summary: Deletes the selection or the character after the cursor.
Action: The user agent must act as if the user had performed a forward delete operation.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

insertImage

Summary: Toggles whether the selection is an image or not. If the second argument is true, and an image is to be added, the user agent will ask the user for the address. Otherwise, the third argument will be used as the address.
Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the *`img`* element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA). If the user agent creates an `img` element or modifies an existing `img` element, then if the *showUI* argument is present and has the value false, then the value of the *value* argument must be used as the URL of the image. Otherwise, the user agent should prompt the user for the URL of the image.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.

Value: Always the string "`false`".

`insertHTML`

Summary: Replaces the selection with the value of the third argument parsed as HTML.
Action: The user agent must run the following steps:

1. If the document is an XML document, then throw an `INVALID_ACCESS_ERR` exception and abort these steps.
2. If the *value* argument wasn't specified, abort these steps without doing anything.
3. If there is a selection, act as if the user had requested that the selection be deleted.
4. Invoke the HTML fragment parsing algorithm with an arbitrary orphan `body` element as the *context* element and with the *value* argument as *input*.
5. Insert the nodes returned by the previous step into the document at the location of the caret, without firing any mutation events.

Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`insertLineBreak`

Summary: Inserts a line break.
Action: The user agent must act as if the user had requested a line separator.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`insertOrderedList`

Summary: Toggles whether the selection is an ordered list.
Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `ol` element (or unwrapped, or, if there is no selection, have that semantic inserted or removed — the exact behavior is UA-defined).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`insertUnorderedList`

Summary: Toggles whether the selection is an unordered list.
Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `ul` element (or unwrapped, or, if there is no selection, have that semantic inserted or removed — the exact behavior is UA-defined).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`insertParagraph`

Summary: Inserts a paragraph break.
Action: The user agent must act as if the user had performed a break block editing action.

Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`insertText`

Summary: Inserts the text given in the third parameter.
Action: The user agent must act as if the user had inserted text corresponding to the *value* parameter.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`italic`

Summary: Toggles whether the selection is italic.
Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the *`i`* element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: True if the selection, or the caret, if there is no selection, is, or is contained within, a `i` element. False otherwise.
Value: The string "`true`" if the expression given for the "State" above is true, the string "`false`" otherwise.

`redo`

Summary: Acts as if the user had requested a redo.
Action: The user agent must move forward one step in its undo transaction history, restoring the associated state. If the undo position is at the end of the undo transaction history, the user agent must do nothing. See the undo history.
Enabled When: The undo position is not at the end of the undo transaction history.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`selectAll`

Summary: Selects all the editable content.
Action: The user agent must change the selection so that all the content in the currently focused editing host is selected. If no editing host is focused, then the content of the entire document must be selected.
Enabled When: Always.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`subscript`

Summary: Toggles whether the selection is subscripted.
Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the *`sub`* element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: True if the selection, or the caret, if there is no selection, is, or is contained within, a `sub` element. False otherwise.
Value: The string "`true`" if the expression given for the "State" above is true, the string

"`false`" otherwise.

superscript

Summary: Toggles whether the selection is superscripted.
Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `sup` element (or unwrapped, or, if there is no selection, have that semantic inserted or removed — the exact behavior is UA-defined).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: True if the selection, or the caret, if there is no selection, is, or is contained within, a `sup` element. False otherwise.
Value: The string "`true`" if the expression given for the "State" above is true, the string "`false`" otherwise.

undo

Summary: Acts as if the user had requested an undo.
Action: The user agent must move back one step in its undo transaction history, restoring the associated state. If the undo position is at the start of the undo transaction history, the user agent must do nothing. See the undo history.
Enabled When: The undo position is not at the start of the undo transaction history.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

unlink

Summary: Removes all links from the selection.
Action: The user agent must remove all `a` elements that have `href` attributes and that are partially or completely included in the current selection.
Enabled When: The document has a selection that is entirely within an editing host and that contains (either partially or completely) at least one `a` element that has an `href` attribute.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

unselect

Summary: Unselects everything.
Action: The user agent must change the selection so that nothing is selected.
Enabled When: Always.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

vendorID-customCommandID

Action: User agents may implement vendor-specific extensions to this API. Vendor-specific extensions to the list of commands should use the syntax *`vendorID-customCommandID`* so as to prevent clashes between extensions from different vendors and future additions to this specification.
Enabled When: UA-defined.
Indeterminate When: UA-defined.
State: UA-defined.
Value: UA-defined.

Anything else

Action: User agents must do nothing.
Enabled When: Never.
Indeterminate When: Never.

State: Always false.
Value: Always the string "`false`".

7 Communication

7.1 Event definitions

Messages in server-sent events, Web sockets, cross-document messaging, and channel messaging use the **`message`** event.

The following interface is defined for this event:

```
interface MessageEvent : Event {
  readonly attribute any data;
  readonly attribute DOMString origin;
  readonly attribute DOMString lastEventId;
  readonly attribute WindowProxy source;
  readonly attribute MessagePortArray ports;
  void initMessageEvent(in DOMString typeArg, in boolean canBubbleArg, in boolean
  void initMessageEventNS(in DOMString namespaceURI, in DOMString typeArg, in bool
};
```

***event*`.data`**

Returns the data of the message.

***event*`.origin`**

Returns the origin of the message, for server-sent events and cross-document messaging.

***event*`.lastEventId`**

Returns the last event ID, for server-sent events.

***event*`.source`**

Returns the `WindowProxy` of the source window, for cross-document messaging.

***event*`.ports`**

Returns the `MessagePortArray` sent with the message, for cross-document messaging and channel messaging.

The **`initMessageEvent()`** and **`initMessageEventNS()`** methods must initialize the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS]

The **`data`** attribute represents the message being sent.

The **`origin`** attribute represents, in server-sent events and cross-document messaging, the origin of the document that sent the message (typically the scheme, hostname, and port of

the document, but not its path or fragment identifier).

The **`lastEventId`** attribute represents, in server-sent events, the last event ID string of the event source.

The **`source`** attribute represents, in cross-document messaging, the `WindowProxy` of the browsing context of the `Window` object from which the message came.

The **`ports`** attribute represents, in cross-document messaging and channel messaging the `MessagePortArray` being sent, if any.

Unless otherwise specified, when the user agent creates and dispatches a `message` event in the algorithms described in the following sections, the `lastEventId` attribute must be the empty string, the `origin` attribute must be the empty string, the `source` attribute must be null, and the `ports` attribute must be null.

7.2 Cross-document messaging

Web browsers, for security and privacy reasons, prevent documents in different domains from affecting each other; that is, cross-site scripting is disallowed.

While this is an important security feature, it prevents pages from different domains from communicating even when those pages are not hostile. This section introduces a messaging system that allows documents to communicate with each other regardless of their source domain, in a way designed to not enable cross-site scripting attacks.

The task source for the tasks in cross-document messaging is the **posted message task source**.

7.2.1 Introduction

This section is non-normative.

> For example, if document A contains an `iframe` element that contains document B, and script in document A calls `postMessage()` on the `Window` object of document B, then a message event will be fired on that object, marked as originating from the `Window` of document A. The script in document A might look like:
>
> ```
> var o = document.getElementsByTagName('iframe')[0];
> o.contentWindow.postMessage('Hello world', 'http://b.example.org/');
> ```
>
> To register an event handler for incoming events, the script would use `addEventListener()` (or similar mechanisms). For example, the script in document B might look like:
>
> ```
> window.addEventListener('message', receiver, false);
> function receiver(e) {
>   if (e.origin == 'http://example.com') {
>     if (e.data == 'Hello world') {
>       e.source.postMessage('Hello', e.origin);
>     } else {
>       alert(e.data);
>     }
> ```

```
    }
  }
```

This script first checks the domain is the expected domain, and then looks at the message, which it either displays to the user, or responds to by sending a message back to the document which sent the message in the first place.

7.2.2 Security

7.2.2.1 Authors

?Warning! ***Use of this API requires extra care to protect users from hostile entities abusing a site for their own purposes.***

Authors should check the `origin` attribute to ensure that messages are only accepted from domains that they expect to receive messages from. Otherwise, bugs in the author's message handling code could be exploited by hostile sites.

Authors should not use the wildcard keyword ("*") in the *targetOrigin* argument in messages that contain any confidential information, as otherwise there is no way to guarantee that the message is only delivered to the recipient to which it was intended.

7.2.2.2 User agents

The integrity of this API is based on the inability for scripts of one origin to post arbitrary events (using `dispatchEvent()` or otherwise) to objects in other origins (those that are not the same).

> ***Note: Implementors are urged to take extra care in the implementation of this feature. It allows authors to transmit information from one domain to another domain, which is normally disallowed for security reasons. It also requires that UAs be careful to allow access to certain properties but not others.***

7.2.3 Posting messages

window **.** **`postMessage`**(***message***, [***ports***,] ***targetOrigin***)

Posts a message, optionally with an array of ports, to the given window.

If the origin of the target window doesn't match the given origin, the message is discarded, to avoid information leakage. To send the message to the target regardless of origin, set the target origin to "`*`".

Throws an `INVALID_STATE_ERR` if the *ports* array is not null and it contains either null entries, duplicate ports, or ports that are not entangled.

When a script invokes the **`postMessage(`*`message, targetOrigin`*`)`** method (with only two arguments) on a `Window` object, the user agent must follow these steps:

1. If the value of the *targetOrigin* argument is not a single U+002A ASTERISK character ("*"), and resolving it relative to the first script's base URL either fails or results in a URL with a `<host-specific>` component that is neither empty nor a single U+002F SOLIDUS character (/), then throw a `SYNTAX_ERR` exception and abort the overall set of steps.

2. Let *message clone* be the result of obtaining a structured clone of the *message* argument. If this throws an exception, then throw that exception and abort these steps.

3. Return from the `postMessage()` method, but asynchronously continue running these steps.

4. If the *targetOrigin* argument has a value other than a single literal U+002A ASTERISK character ("*"), and the `Document` of the `Window` object on which the method was invoked does not have the same origin as *targetOrigin*, then abort these steps silently.

5. Create an event that uses the `MessageEvent` interface, with the event name `message`, which does not bubble, is cancelable, and has no default action. The `data` attribute must be set to the value of *message clone*, the `origin` attribute must be set to the Unicode serialization of the origin of the script that invoked the method, and the `source` attribute must be set to the script's global object.

6. Queue a task to dispatch the event created in the previous step at the `Window` object on which the method was invoked. The task source for this task is the posted message task source.

7.2.4 Posting messages with message ports

When a script invokes the **`postMessage(`*`message, ports, targetOrigin`*`)`** method (with three arguments) on a `Window` object, the user agent must follow these steps:

1. If the value of the *targetOrigin* argument is not a single U+002A ASTERISK character ("*"), and resolving it relative to the first script's base URL either fails or results in a URL with a `<host-specific>` component that is neither empty nor a single U+002F SOLIDUS character (/), then throw a `SYNTAX_ERR` exception and abort the overall set of steps.

2. Let *message clone* be the result of obtaining a structured clone of the *message* argument. If this throws an exception, then throw that exception and abort these steps.

3. If the *ports* argument is null, then act as if the method had just been called with two arguments, *message* and *targetOrigin*.

4. If any of the entries in *ports* are null, if any of the entries in *ports* are not entangled `MessagePort` objects, or if any `MessagePort` object is listed in *ports* more than once, then throw an `INVALID_STATE_ERR` exception.

5. Let *new ports* be an empty array.

   For each port in *ports* in turn, obtain a new port by cloning the port with the `Window` object on which the method was invoked as the owner of the clone, and append the

clone to the *new ports* array.

Note: If the original ports array was empty, then the new ports array will also be empty.

6. Return from the `postMessage()` method, but asynchronously continue running these steps.

7. If the *targetOrigin* argument has a value other than a single literal U+002A ASTERISK character ("*"), and the `Document` of the `Window` object on which the method was invoked does not have the same origin as *targetOrigin*, then abort these steps silently.

8. Create an event that uses the `MessageEvent` interface, with the event name `message`, which does not bubble, is cancelable, and has no default action. The `data` attribute must be set to the value of *message clone*, the `origin` attribute must be set to the Unicode serialization of the origin of the script that invoked the method, and the `source` attribute must be set to the script's global object.

9. Let the `ports` attribute of the event be the *new ports* array.

10. Queue a task to dispatch the event created in the previous step at the `Window` object on which the method was invoked. The task source for this task is the posted message task source.

Note: These steps, with the exception of the second and third steps and the penultimate step, are identical to those in the previous section.

7.3 Channel messaging

7.3.1 Introduction

This section is non-normative.

An introduction to the channel and port APIs.

7.3.2 Message channels

```
[Constructor]
interface MessageChannel {
  readonly attribute MessagePort port1;
  readonly attribute MessagePort port2;
};
```

***channel* = new `MessageChannel`()**

Returns a new `MessageChannel` object with two new `MessagePort` objects.

***channel* . `port1`**

Returns the first `MessagePort` object.

***channel* . `port2`**

Returns the second `MessagePort` object.

When the **`MessageChannel()`** constructor is called, it must run the following algorithm:

1. Create a new `MessagePort` object owned by the script's global object, and let *port1* be that object.
2. Create a new `MessagePort` object owned by the script's global object, and let *port2* be that object.
3. Entangle the *port1* and *port2* objects.
4. Instantiate a new `MessageChannel` object, and let *channel* be that object.
5. Let the `port1` attribute of the *channel* object be *port1*.
6. Let the `port2` attribute of the *channel* object be *port2*.
7. Return *channel*.

The **`port1`** and **`port2`** attributes must return the values they were assigned when the `MessageChannel` object was created.

7.3.3 Message ports

Each channel has two message ports. Data sent through one port is received by the other port, and vice versa.

```
typedef sequence<MessagePort> MessagePortArray;

interface MessagePort {
  readonly attribute boolean active;
  void postMessage(in any message, [Optional] in MessagePortArray ports);
  void start();
  void close();

  // event handler attributes
           attribute Function onmessage;
};
```

***port* . `active`**

Returns true if the port is still active; otherwise, returns false.

***port* . `postMessage`(*message* [, *ports*])**

Posts a message through the channel, optionally with the given ports.

Throws an `INVALID_STATE_ERR` if the *ports* array is not null and it contains either null entries, duplicate ports, ports that are not entangled, or the source or target port.

***port*`.start()`**

Begins dispatching messages received on the port.

***port*`.close()`**

Disconnects the port, so that it is no longer active.

Objects implementing the `MessagePort` interface must also implement the `EventTarget` interface.

Each `MessagePort` object can be entangled with another (a symmetric relationship). Each `MessagePort` object also has a task source called the **port message queue**, initial empty. A port message queue can be open or closed, and is initially closed.

When the user agent is to **create a new `MessagePort` object** owned by a script's global object object *owner*, it must instantiate a new `MessagePort` object, and let its owner be *owner*.

When the user agent is to **entangle** two `MessagePort` objects, it must run the following steps:

1. If one of the ports is already entangled, then unentangle it and the port that it was entangled with.

   Note: If those two previously entangled ports were the two ports of a `MessageChannel` object, then that `MessageChannel` object no longer represents an actual channel: the two ports in that object are no longer entangled.

2. Associate the two ports to be entangled, so that they form the two parts of a new channel. (There is no `MessageChannel` object that represents this channel.)

When the user agent is to **clone a port** *original port*, with the clone being owned by *owner*, it must run the following steps, which return either a new `MessagePort` object or an exception for the caller to raise:

1. If the *original port* is not entangled without another port, then return an `INVALID_STATE_ERR` exception and abort all these steps.

2. Let the *remote port* be the port with which the *original port* is entangled.

3. Create a new `MessagePort` object owned by *owner*, and let *new port* be that object.

4. Move all the events in the port message queue of *original port* to the port message queue of *new port*, if any, leaving the *new port*'s port message queue in its initial closed state.

5. Entangle the *remote port* and *new port* objects. The *original port* object will be unentangled by this process.

6. Return *new port*. It is the clone.

The **`active`** attribute must return true if the port is entangled, and false otherwise.

The **`postMessage()`** method, when called on a port *source port*, must cause the user agent to run the following steps:

1. Let *message* be the method's first argument.

2. Let *data port* be the method's second argument, if any.

3. Let *message clone* be the result of obtaining a structured clone of the *message* argument. If this throws an exception, then throw that exception and abort these steps.

4. If the *source port* is not entangled with another port, then return and abort these steps.

5. Let *target port* be the port with which *source port* is entangled.

6. Create an event that uses the `MessageEvent` interface, with the name `message`, which does not bubble, is cancelable, and has no default action.

7. Let the `data` attribute of the event have the value of *message clone*.

8. If the method was called with a second argument *ports* and that argument isn't null, then run the following substeps:

    1. If any of the entries in *ports* are null, if any of the entries in *ports* are not entangled `MessagePort` objects, or if any `MessagePort` object is listed in *ports* more than once, then throw an `INVALID_STATE_ERR` exception.

    2. Let *new ports* be an empty array.

        For each port in *ports* in turn, obtain a new port by cloning the port with the owner of the *target port* as the owner of the clone, and append the clone to the *new ports* array.

        Note: If the original ports array was empty, then the new ports array will also be empty.

    3. Let the `ports` attribute of the event be the *new ports* array.

9. Return from the method, but continue with these steps.

10. Add the event to the port message queue of *target port*.

The **`start()`** method must open its port's port message queue, if it is not already open.

When a port's port message queue is open, the event loop must use it as one of its task

sources.

Note: If the `Document` of the port's event handlers' global object is not fully active, then the messages are lost.

The **`close()`** method, when called on a port *local port* that is entangled with another port, must cause the user agents to unentangle the two ports. If the method is called on a port that is not entangled, then the method must do nothing.

The following are the event handler attributes that must be supported, as DOM attributes, by all objects implementing the `MessagePort` interface:

`onmessage`

Must be invoked whenever a `message` event is targeted at or bubbles through the `MessagePort` object.

The first time a `MessagePort` object's `onmessage` DOM attribute is set, the port's port message queue must be opened, as if the `start()` method had been called.

7.3.3.1 Ports and garbage collection

User agents must act as if `MessagePort` objects have a strong reference to their entangled `MessagePort` object.

Thus, a message port can be received, given an event listener, and then forgotten, and so long as that event listener could receive a message, the channel will be maintained.

Of course, if this was to occur on both sides of the channel, then both ports would be garbage collected, since they would not be reachable from live code, despite having a strong reference to each other.

Furthermore, a `MessagePort` object must not be garbage collected while there exists a message in a task queue that is to be dispatched on that `MessagePort` object, or while the `MessagePort` object's port message queue is open and there exists a `message` event in that queue.

8 The HTML syntax

Note: This section only describes the rules for `text/html` resources. Rules for XML resources are discussed in the section below entitled "The XHTML syntax".

8.1 Writing HTML documents

This section only applies to documents, authoring tools, and markup generators. In particular, it does not apply to conformance checkers; conformance checkers must use the requirements given in the next section ("parsing HTML documents").

Documents must consist of the following parts, in the given order:

1. Optionally, a single U+FEFF BYTE ORDER MARK (BOM) character.
2. Any number of comments and space characters.
3. A DOCTYPE.
4. Any number of comments and space characters.
5. The root element, in the form of an `html` element.
6. Any number of comments and space characters.

The various types of content mentioned above are described in the next few sections.

In addition, there are some restrictions on how character encoding declarations are to be serialized, as discussed in the section on that topic.

Space characters before the root `html` element, and space characters at the start of the `html` element and before the `head` element, will be dropped when the document is parsed; space characters after the root `html` element will be parsed as if they were at the end of the `body` element. Thus, space characters around the root element do not round-trip.

It is suggested that newlines be inserted after the DOCTYPE, after any comments that are before the root element, after the `html` element's start tag (if it is not omitted), and after any comments that are inside the `html` element but before the `head` element.

Many strings in the HTML syntax (e.g. the names of elements and their attributes) are case-insensitive, but only for characters in the ranges U+0041 .. U+005A (LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) and U+0061 .. U+007A (LATIN SMALL LETTER A to LATIN SMALL LETTER Z). For convenience, in this section this is just referred to as

"case-insensitive".

8.1.1 The DOCTYPE

A **DOCTYPE** is a mostly useless, but required, header.

> ***Note: DOCTYPEs are required for legacy reasons. When omitted, browsers tend to use a different rendering mode that is incompatible with some specifications. Including the DOCTYPE in a document ensures that the browser makes a best-effort attempt at following the relevant specifications.***

A DOCTYPE must consist of the following characters, in this order:

1. A U+003C LESS-THAN SIGN (<) character.
2. A U+0021 EXCLAMATION MARK (!) character.
3. A string that is an ASCII case-insensitive match for the string "DOCTYPE".
4. One or more space characters.
5. A string that is an ASCII case-insensitive match for the string "HTML".
6. Optionally, a DOCTYPE legacy string (defined below).
7. Zero or more space characters.
8. A U+003E GREATER-THAN SIGN (>) character.

> ***Note: In other words, `<!DOCTYPE HTML>`, case-insensitively.***

For the purposes of HTML generators that cannot output HTML markup with the short DOCTYPE "<!DOCTYPE HTML>", a **DOCTYPE legacy string** may be inserted into the DOCTYPE (in the position defined above). This string must consist of:

1. One or more space characters.
2. A string that is an ASCII case-insensitive match for the string "SYSTEM".
3. One or more space characters.
4. A U+0022 QUOTATION MARK or U+0027 APOSTROPHE character (the *quote mark*).
5. The literal string "about:legacy-compat".
6. A matching U+0022 QUOTATION MARK or U+0027 APOSTROPHE character (i.e. the same character as in the earlier step marked *quote mark*).

> ***Note: In other words, `<!DOCTYPE HTML SYSTEM "about:legacy-compat">` or `<!DOCTYPE HTML SYSTEM 'about:legacy-compat'>`, case-insensitively except for the bit in quotes.***

The DOCTYPE legacy string should not be used unless the document is generated from a system that cannot output the shorter string.

8.1.2 Elements

There are five different kinds of **elements**: void elements, CDATA elements, RCDATA elements, foreign elements, and normal elements.

Void elements

`base`, `command`, `link`, `meta`, `hr`, `br`, `img`, `embed`, `param`, `area`, `col`, `input`, `source`

CDATA elements

`style`, `script`

RCDATA elements

`title`, `textarea`

Foreign elements

Elements from the MathML namespace and the SVG namespace.

Normal elements

All other allowed HTML elements are normal elements.

Tags are used to delimit the start and end of elements in the markup. CDATA, RCDATA, and normal elements have a start tag to indicate where they begin, and an end tag to indicate where they end. The start and end tags of certain normal elements can be omitted, as described later. Those that cannot be omitted must not be omitted. Void elements only have a start tag; end tags must not be specified for void elements. Foreign elements must either have a start tag and an end tag, or a start tag that is marked as self-closing, in which case they must not have an end tag.

The contents of the element must be placed between just after the start tag (which might be implied, in certain cases) and just before the end tag (which again, might be implied in certain cases). The exact allowed contents of each individual element depends on the content model of that element, as described earlier in this specification. Elements must not contain content that their content model disallows. In addition to the restrictions placed on the contents by those content models, however, the five types of elements have additional *syntactic* requirements.

Void elements can't have any contents (since there's no end tag, no content can be put between the start tag and the end tag).

CDATA elements can have text, though it has restrictions described below.

RCDATA elements can have text and character references, but the text must not contain an ambiguous ampersand. There are also further restrictions described below.

Foreign elements whose start tag is marked as self-closing can't have any contents (since, again, as there's no end tag, no content can be put between the start tag and the end tag). Foreign elements whose start tag is *not* marked as self-closing can have text, character references, CDATA sections, other elements, and comments, but the text must not contain the character U+003C LESS-THAN SIGN (<) or an ambiguous ampersand.

Normal elements can have text, character references, other elements, and comments, but the text must not contain the character U+003C LESS-THAN SIGN (<) or an ambiguous ampersand. Some normal elements also have yet more restrictions on what content they are allowed to hold, beyond the restrictions imposed by the content model and those described in this paragraph. Those restrictions are described below.

Tags contain a **tag name**, giving the element's name. HTML elements all have names that only use characters in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A .. U+007A LATIN SMALL LETTER Z, U+0041 LATIN CAPITAL LETTER A .. U+005A LATIN CAPITAL LETTER Z, and U+002D HYPHEN-MINUS (-). In the HTML syntax, tag names may be written with any mix of lower- and uppercase letters that, when converted to all-lowercase, matches the element's tag name; tag names are case-insensitive.

8.1.2.1 Start tags

Start tags must have the following format:

1. The first character of a start tag must be a U+003C LESS-THAN SIGN (<).
2. The next few characters of a start tag must be the element's tag name.
3. If there are to be any attributes in the next step, there must first be one or more space characters.
4. Then, the start tag may have a number of attributes, the syntax for which is described below. Attributes may be separated from each other by one or more space characters.
5. After the attributes, there may be one or more space characters. (Some attributes are required to be followed by a space. See the attributes section below.)
6. Then, if the element is one of the void elements, or if the element is a foreign element, then there may be a single U+002F SOLIDUS (/) character. This character has no effect on void elements, but on foreign elements it marks the start tag as self-closing.
7. Finally, start tags must be closed by a U+003E GREATER-THAN SIGN (>) character.

8.1.2.2 End tags

End tags must have the following format:

1. The first character of an end tag must be a U+003C LESS-THAN SIGN (<).
2. The second character of an end tag must be a U+002F SOLIDUS (/).
3. The next few characters of an end tag must be the element's tag name.
4. After the tag name, there may be one or more space characters.
5. Finally, end tags must be closed by a U+003E GREATER-THAN SIGN (>) character.

8.1.2.3 Attributes

Attributes for an element are expressed inside the element's start tag.

Attributes have a name and a value. **Attribute names** must consist of one or more characters other than the space characters, U+0000 NULL, U+0022 QUOTATION MARK ("), U+0027 APOSTROPHE ('), U+003E GREATER-THAN SIGN (>), U+002F SOLIDUS (/), and U+003D EQUALS SIGN (=) characters, the control characters, and any characters that are not defined by Unicode. In the HTML syntax, attribute names may be written with any mix of lower- and uppercase letters that are an ASCII case-insensitive match for the attribute's name.

Attribute values are a mixture of text and character references, except with the additional restriction that the text cannot contain an ambiguous ampersand.

Attributes can be specified in four different ways:

Empty attribute syntax

Just the attribute name.

> In the following example, the `disabled` attribute is given with the empty attribute syntax:
>
> ```
> <input disabled>
> ```

If an attribute using the empty attribute syntax is to be followed by another attribute, then there must be a space character separating the two.

Unquoted attribute value syntax

The attribute name, followed by zero or more space characters, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters, followed by the attribute value, which, in addition to the requirements given above for attribute values, must not contain any literal space characters, any U+0022 QUOTATION MARK (") characters, U+0027 APOSTROPHE (') characters, U+003D EQUALS SIGN (=) characters, or U+003E GREATER-THAN SIGN (>) characters, and must not be the empty string.

> In the following example, the `value` attribute is given with the unquoted attribute value syntax:
>
> ```
> <input value=yes>
> ```

If an attribute using the unquoted attribute syntax is to be followed by another attribute or by one of the optional U+002F SOLIDUS (/) characters allowed in step 6 of the start tag syntax above, then there must be a space character separating the two.

Single-quoted attribute value syntax

The attribute name, followed by zero or more space characters, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters, followed by a single U+0027 APOSTROPHE (') character, followed by the attribute value, which, in addition to the requirements given above for attribute values, must not contain any literal U+0027 APOSTROPHE (') characters, and finally followed by a second single U+0027 APOSTROPHE (') character.

> In the following example, the `type` attribute is given with the single-quoted attribute value syntax:
>
> ```
> <input type='checkbox'>
> ```

If an attribute using the single-quoted attribute syntax is to be followed by another attribute, then there must be a space character separating the two.

Double-quoted attribute value syntax

The attribute name, followed by zero or more space characters, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters, followed by a single U+0022 QUOTATION MARK (") character, followed by the attribute value, which, in addition to the requirements given above for attribute values, must not contain any literal U+0022 QUOTATION MARK (") characters, and finally followed by a second single U+0022 QUOTATION MARK (") character.

> In the following example, the name attribute is given with the double-quoted attribute value syntax:
>
> ```
> <input name="be evil">
> ```

If an attribute using the double-quoted attribute syntax is to be followed by another attribute, then there must be a space character separating the two.

There must never be two or more attributes on the same start tag whose names are an ASCII case-insensitive match for each other.

8.1.2.4 Optional tags

Certain tags can be **omitted**.

An html element's start tag may be omitted if the first thing inside the html element is not a comment.

An html element's end tag may be omitted if the html element is not immediately followed by a comment.

A head element's start tag may be omitted if the first thing inside the head element is an element.

A head element's end tag may be omitted if the head element is not immediately followed by a space character or a comment.

A body element's start tag may be omitted if the first thing inside the body element is not a space character or a comment, except if the first thing inside the body element is a script or style element.

A body element's end tag may be omitted if the body element is not immediately followed by a comment.

A li element's end tag may be omitted if the li element is immediately followed by another li element or if there is no more content in the parent element.

A dt element's end tag may be omitted if the dt element is immediately followed by another dt element or a dd element.

A dd element's end tag may be omitted if the dd element is immediately followed by another dd element or a dt element, or if there is no more content in the parent element.

A p element's end tag may be omitted if the p element is immediately followed by an address, article, aside, blockquote, datagrid, dialog, dir, div, dl, fieldset, footer, form, h1, h2, h3, h4, h5, h6, header, hr, menu, nav, ol, p, pre, section, table, or ul, element, or if there is no more content in the parent element and the parent element is not an a element.

An rt element's end tag may be omitted if the rt element is immediately followed by an rt or rp element, or if there is no more content in the parent element.

An rp element's end tag may be omitted if the rp element is immediately followed by an rt or rp element, or if there is no more content in the parent element.

An optgroup element's end tag may be omitted if the optgroup element is immediately followed by another optgroup element, or if there is no more content in the parent element.

An option element's end tag may be omitted if the option element is immediately followed by another option element, or if it is immediately followed by an optgroup element, or if there is no more content in the parent element.

A colgroup element's start tag may be omitted if the first thing inside the colgroup element is a col element, and if the element is not immediately preceded by another colgroup element whose end tag has been omitted.

A colgroup element's end tag may be omitted if the colgroup element is not immediately followed by a space character or a comment.

A thead element's end tag may be omitted if the thead element is immediately followed by a tbody or tfoot element.

A tbody element's start tag may be omitted if the first thing inside the tbody element is a tr element, and if the element is not immediately preceded by a tbody, thead, or tfoot element whose end tag has been omitted.

A tbody element's end tag may be omitted if the tbody element is immediately followed by a tbody or tfoot element, or if there is no more content in the parent element.

A tfoot element's end tag may be omitted if the tfoot element is immediately followed by a tbody element, or if there is no more content in the parent element.

A tr element's end tag may be omitted if the tr element is immediately followed by another tr element, or if there is no more content in the parent element.

A td element's end tag may be omitted if the td element is immediately followed by a td or th element, or if there is no more content in the parent element.

A th element's end tag may be omitted if the th element is immediately followed by a td or th element, or if there is no more content in the parent element.

However, a start tag must never be omitted if it has any attributes.

8.1.2.5 Restrictions on content models

For historical reasons, certain elements have extra restrictions beyond even the restrictions given by their content model.

A table element must not contain tr elements, even though these elements are technically allowed inside table elements according to the content models described in this specification. (If a tr element is put inside a table in the markup, it will in fact imply a tbody start tag before it.)

A single U+000A LINE FEED (LF) character may be placed immediately after the start tag of pre and textarea elements. This does not affect the processing of the element. The otherwise optional U+000A LINE FEED (LF) character *must* be included if the element's contents start with that character (because otherwise the leading newline in the contents would be treated

like the optional newline, and ignored).

> The following two `pre` blocks are equivalent:
>
> ```
> <pre>Hello</pre>
>
> <pre>
> Hello</pre>
> ```

8.1.2.6 Restrictions on the contents of CDATA and RCDATA elements

The text in CDATA and RCDATA elements must not contain any occurrences of the string "`</`" (U+003C LESS-THAN SIGN, U+002F SOLIDUS) followed by characters that case-insensitively match the tag name of the element followed by one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), U+0020 SPACE, U+003E GREATER-THAN SIGN (>), or U+002F SOLIDUS (/), unless that string is part of an escaping text span.

An **escaping text span** is a span of text that starts with an escaping text span start that is not itself in an escaping text span, and ends at the next escaping text span end. There cannot be any character references inside an escaping text span — sequences of characters that would look like character references do not have special meaning.

An **escaping text span start** is a part of text that consists of the four character sequence "`<!--`" (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS).

An **escaping text span end** is a part of text that consists of the three character sequence "`-->`" (U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN) whose U+003E GREATER-THAN SIGN (>).

An escaping text span start may share its U+002D HYPHEN-MINUS characters with its corresponding escaping text span end.

The text in CDATA and RCDATA elements must not have an escaping text span start that is not followed by an escaping text span end.

8.1.3 Text

Text is allowed inside elements, attributes, and comments. Text must consist of Unicode characters. Text must not contain U+0000 characters. Text must not contain permanently undefined Unicode characters. Text must not contain control characters other than space characters. Extra constraints are placed on what is and what is not allowed in text based on where the text is to be put, as described in the other sections.

8.1.3.1 Newlines

Newlines in HTML may be represented either as U+000D CARRIAGE RETURN (CR) characters, U+000A LINE FEED (LF) characters, or pairs of U+000D CARRIAGE RETURN (CR), U+000A LINE FEED (LF) characters in that order.

8.1.4 Character references

In certain cases described in other sections, text may be mixed with **character references**. These can be used to escape characters that couldn't otherwise legally be included in text.

Character references must start with a U+0026 AMPERSAND (`&`). Following this, there are three possible kinds of character references:

Named character references
: The ampersand must be followed by one of the names given in the named character references section, using the same case. The name must be one that is terminated by a U+003B SEMICOLON (`;`) character.

Decimal numeric character reference
: The ampersand must be followed by a U+0023 NUMBER SIGN (`#`) character, followed by one or more digits in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, representing a base-ten integer that itself is a Unicode code point that is not U+0000, U+000D, in the range U+0080 .. U+009F, or in the range 0xD800 .. 0xDFFF (surrogates). The digits must then be followed by a U+003B SEMICOLON character (`;`).

Hexadecimal numeric character reference
: The ampersand must be followed by a U+0023 NUMBER SIGN (`#`) character, which must be followed by either a U+0078 LATIN SMALL LETTER X or a U+0058 LATIN CAPITAL LETTER X character, which must then be followed by one or more digits in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F, and U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F, representing a base-sixteen integer that itself is a Unicode code point that is not U+0000, U+000D, in the range U+0080 .. U+009F, or in the range 0xD800 .. 0xDFFF (surrogates). The digits must then be followed by a U+003B SEMICOLON character (`;`).

An **ambiguous ampersand** is a U+0026 AMPERSAND (`&`) character that is followed by some text other than a space character, a U+003C LESS-THAN SIGN character ('<'), or another U+0026 AMPERSAND (`&`) character.

8.1.5 CDATA sections

CDATA sections must start with the character sequence U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+005B LEFT SQUARE BRACKET, U+0043 LATIN CAPITAL LETTER C, U+0044 LATIN CAPITAL LETTER D, U+0041 LATIN CAPITAL LETTER A, U+0054 LATIN CAPITAL LETTER T, U+0041 LATIN CAPITAL LETTER A, U+005B LEFT SQUARE BRACKET (`<![CDATA[`). Following this sequence, the CDATA section may have text, with the additional restriction that the text must not contain the three character sequence U+005D RIGHT SQUARE BRACKET, U+005D RIGHT SQUARE BRACKET, U+003E GREATER-THAN SIGN (`]]>`). Finally, the CDATA section must be ended by the three character sequence U+005D RIGHT SQUARE BRACKET, U+005D RIGHT SQUARE BRACKET, U+003E GREATER-THAN SIGN (`]]>`).

8.1.6 Comments

Comments must start with the four character sequence U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS (`<!--`). Following this sequence, the comment may have text, with the additional restriction that the text must not start with a single U+003E GREATER-THAN SIGN ('>') character, nor start with a U+002D HYPHEN-MINUS (`-`) character followed by a U+003E GREATER-THAN SIGN ('>') character, nor contain two consecutive U+002D HYPHEN-MINUS (`-`) characters, nor end with a U+002D HYPHEN-MINUS (`-`) character. Finally, the comment must be ended by the three character sequence U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN (`-->`).

8.2 Parsing HTML documents

This section only applies to user agents, data mining tools, and conformance checkers.

The rules for parsing XML documents (and thus XHTML documents) into DOM trees are covered by the next section, entitled "The XHTML syntax".

For HTML documents, user agents must use the parsing rules described in this section to generate the DOM trees. Together, these rules define what is referred to as the **HTML parser**.

> ***While the HTML form of HTML5 bears a close resemblance to SGML and XML, it is a separate language with its own parsing rules.***
>
> ***Some earlier versions of HTML (in particular from HTML2 to HTML4) were based on SGML and used SGML parsing rules. However, few (if any) web browsers ever implemented true SGML parsing for HTML documents; the only user agents to strictly handle HTML as an SGML application have historically been validators. The resulting confusion — with validators claiming documents to have one representation while widely deployed Web browsers interoperably implemented a different representation — has wasted decades of productivity. This version of HTML thus returns to a non-SGML basis.***
>
> ***Authors interested in using SGML tools in their authoring pipeline are encouraged to use XML tools and the XML serialization of HTML5.***

This specification defines the parsing rules for HTML documents, whether they are syntactically correct or not. Certain points in the parsing algorithm are said to be **parse errors**. The error handling for parse errors is well-defined: user agents must either act as described below when encountering such problems, or must abort processing at the first error that they encounter for which they do not wish to apply the rules described below.

Conformance checkers must report at least one parse error condition to the user if one or more parse error conditions exist in the document and must not report parse error conditions if none exist in the document. Conformance checkers may report more than one parse error condition if more than one parse error conditions exist in the document. Conformance checkers are not required to recover from parse errors.

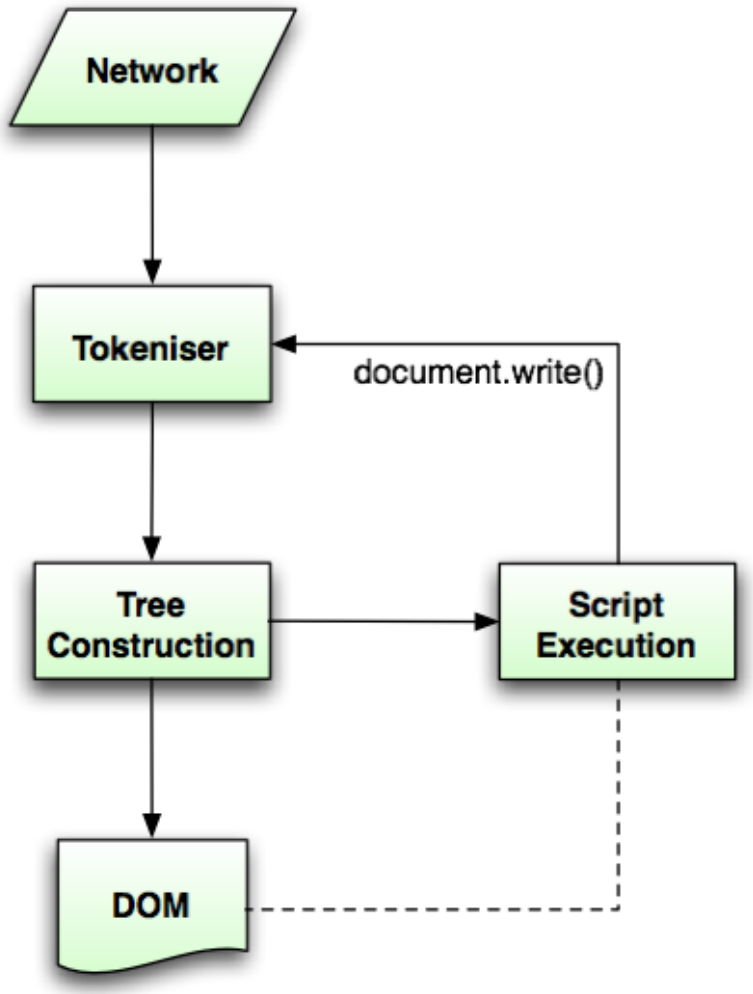
Note: Parse errors are only errors with the syntax of HTML. In addition to checking for parse errors, conformance checkers will also verify that the document obeys all the other conformance requirements described in this specification.

8.2.1 Overview of the parsing model

The input to the HTML parsing process consists of a stream of Unicode characters, which is passed through a tokenization stage (lexical analysis) followed by a tree construction stage (semantic analysis). The output is a `Document` object.

Note: Implementations that do not support scripting do not have to actually create a DOM `Document` object, but the DOM tree in such cases is still used as the model for the rest of the specification.

In the common case, the data handled by the tokenization stage comes from the network, but it can also come from script, e.g. using the `document.write()` API.



There is only one set of state for the tokeniser stage and the tree construction stage, but the tree construction stage is reentrant, meaning that while the tree construction stage is handling one token, the tokeniser might be resumed, causing further tokens to be emitted and processed before the first token's processing is complete.

> In the following example, the tree construction stage will be called upon to handle a "p" start tag token while handling the "script" start tag token:
>
> ```
> ...
> <script>
>  document.write('<p>');
> </script>
> ...
> ```

To handle these cases, parsers have a **script nesting level**, which must be initially set to zero, and a **parser pause flag**, which must be initially set to false.

8.2.2 The input stream

The stream of Unicode characters that consists the input to the tokenization stage will be initially seen by the user agent as a stream of bytes (typically coming over the network or from the local file system). The bytes encode the actual characters according to a particular *character encoding*, which the user agent must use to decode the bytes into characters.

> ***Note: For XML documents, the algorithm user agents must use to determine the character encoding is given by the XML specification. This section does not apply to XML documents. [XML]***

8.2.2.1 Determining the character encoding

In some cases, it might be impractical to unambiguously determine the encoding before parsing the document. Because of this, this specification provides for a two-pass mechanism with an optional pre-scan. Implementations are allowed, as described below, to apply a simplified parsing algorithm to whatever bytes they have available before beginning to parse the document. Then, the real parser is started, using a tentative encoding derived from this pre-parse and other out-of-band metadata. If, while the document is being loaded, the user agent discovers an encoding declaration that conflicts with this information, then the parser can get reinvoked to perform a parse of the document with the real encoding.

User agents must use the following algorithm (the **encoding sniffing algorithm**) to determine the character encoding to use when decoding a document in the first pass. This algorithm takes as input any out-of-band metadata available to the user agent (e.g. the Content-Type metadata of the document) and all the bytes available so far, and returns an encoding and a **confidence**. The confidence is either *tentative*, *certain*, or *irrelevant*. The encoding used, and whether the confidence in that encoding is *tentative* or *confident*, is used during the parsing to determine whether to change the encoding. If no encoding is necessary, e.g. because the parser is operating on a stream of Unicode characters and doesn't have to use an encoding at all, then the confidence is *irrelevant*.

1. If the transport layer specifies an encoding, and it is supported, return that encoding with the confidence *certain*, and abort these steps.

2. The user agent may wait for more bytes of the resource to be available, either in this step or at any later step in this algorithm. For instance, a user agent might wait 500ms or 512 bytes, whichever came first. In general preparsing the source to find the encoding improves performance, as it reduces the need to throw away the data structures used when parsing upon finding the encoding information. However, if the user agent delays too long to obtain data to determine the encoding, then the cost of the delay could outweigh any performance improvements from the preparse.

3. For each of the rows in the following table, starting with the first one and going down, if there are as many or more bytes available than the number of bytes in the first column, and the first bytes of the file match the bytes given in the first column, then return the encoding given in the cell in the second column of that row, with the confidence *certain*, and abort these steps:

Bytes in Hexadecimal	Encoding
FE FF	UTF-16BE
FF FE	UTF-16LE
EF BB BF	UTF-8

Note: This step looks for Unicode Byte Order Marks (BOMs).

4. Otherwise, the user agent will have to search for explicit character encoding information in the file itself. This should proceed as follows:

   Let *position* be a pointer to a byte in the input stream, initially pointing at the first byte. If at any point during these substeps the user agent either runs out of bytes or decides that scanning further bytes would not be efficient, then skip to the next step of the overall character encoding detection algorithm. User agents may decide that scanning *any* bytes is not efficient, in which case these substeps are entirely skipped.

   Now, repeat the following "two" steps until the algorithm aborts (either because user agent aborts, as described above, or because a character encoding is found):

   1. If *position* points to:

      ↪ **A sequence of bytes starting with: 0x3C 0x21 0x2D 0x2D (ASCII '<!--')**
      Advance the *position* pointer so that it points at the first 0x3E byte which is preceded by two 0x2D bytes (i.e. at the end of an ASCII '-->' sequence) and comes after the 0x3C byte that was found. (The two 0x2D bytes can be the same as the those in the '<!--' sequence.)

      ↪ **A sequence of bytes starting with: 0x3C, 0x4D or 0x6D, 0x45 or 0x65, 0x54 or 0x74, 0x41 or 0x61, and finally one of 0x09, 0x0A, 0x0C, 0x0D, 0x20, 0x2F (case-insensitive ASCII '<meta' followed by a space or slash)**

      1. Advance the *position* pointer so that it points at the next 0x09, 0x0A, 0x0C, 0x0D, 0x20, or 0x2F byte (the one in sequence of characters matched above).

      2. Get an attribute and its value. If no attribute was sniffed, then skip this inner set of steps, and jump to the second step in the overall

"two step" algorithm.

3. If the attribute's name is neither "`charset`" nor "`content`", then return to step 2 in these inner steps.

4. If the attribute's name is "`charset`", let *charset* be the attribute's value, interpreted as a character encoding.

5. Otherwise, the attribute's name is "`content`": apply the algorithm for extracting an encoding from a Content-Type, giving the attribute's value as the string to parse. If an encoding is returned, let *charset* be that encoding. Otherwise, return to step 2 in these inner steps.

6. If *charset* is a UTF-16 encoding, change it to UTF-8.

7. If *charset* is a supported character encoding, then return the given encoding, with confidence *tentative*, and abort all these steps.

8. Otherwise, return to step 2 in these inner steps.

↪ **A sequence of bytes starting with a 0x3C byte (ASCII '<'), optionally a 0x2F byte (ASCII '/'), and finally a byte in the range 0x41-0x5A or 0x61-0x7A (an ASCII letter)**

1. Advance the *position* pointer so that it points at the next 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), or 0x3E (ASCII '>') byte.

2. Repeatedly get an attribute until no further attributes can be found, then jump to the second step in the overall "two step" algorithm.

↪ **A sequence of bytes starting with: 0x3C 0x21 (ASCII '<!')**
↪ **A sequence of bytes starting with: 0x3C 0x2F (ASCII '</')**
↪ **A sequence of bytes starting with: 0x3C 0x3F (ASCII '<?')**

Advance the *position* pointer so that it points at the first 0x3E byte (ASCII '>') that comes after the 0x3C byte that was found.

↪ **Any other byte**

Do nothing with that byte.

2. Move *position* so it points at the next byte in the input stream, and return to the first step of this "two step" algorithm.

When the above "two step" algorithm says to **get an attribute**, it means doing this:

1. If the byte at *position* is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), or 0x2F (ASCII '/') then advance *position* to the next byte and redo this substep.

2. If the byte at *position* is 0x3E (ASCII '>'), then abort the "get an attribute" algorithm. There isn't one.

3. Otherwise, the byte at *position* is the start of the attribute name. Let *attribute name*

and *attribute value* be the empty string.

4. *Attribute name*: Process the byte at *position* as follows:

   ↪ **If it is 0x3D (ASCII '='), and the *attribute name* is longer than the empty string**
   Advance *position* to the next byte and jump to the step below labeled *value*.

   ↪ **If it is 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space)**
   Jump to the step below labeled *spaces*.

   ↪ **If it is 0x2F (ASCII '/') or 0x3E (ASCII '>')**
   Abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name*, its value is the empty string.

   ↪ **If it is in the range 0x41 (ASCII 'A') to 0x5A (ASCII 'Z')**
   Append the Unicode character with codepoint *b*+0x20 to *attribute name* (where *b* is the value of the byte at *position*).

   ↪ **Anything else**
   Append the Unicode character with the same codepoint as the value of the byte at *position*) to *attribute name*. (It doesn't actually matter how bytes outside the ASCII range are handled here, since only ASCII characters can contribute to the detection of a character encoding.)

5. Advance *position* to the next byte and return to the previous step.

6. *Spaces.* If the byte at *position* is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space) then advance *position* to the next byte, then, repeat this step.

7. If the byte at *position* is *not* 0x3D (ASCII '='), abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name*, its value is the empty string.

8. Advance *position* past the 0x3D (ASCII '=') byte.

9. *Value.* If the byte at *position* is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space) then advance *position* to the next byte, then, repeat this step.

10. Process the byte at *position* as follows:

    ↪ **If it is 0x22 (ASCII '"') or 0x27 ("'")**

    1. Let *b* be the value of the byte at *position*.

    2. Advance *position* to the next byte.

    3. If the value of the byte at *position* is the value of *b*, then advance *position* to the next byte and abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name*, and its value is the value of *attribute value*.

    4. Otherwise, if the value of the byte at *position* is in the range 0x41 (ASCII 'A') to 0x5A (ASCII 'Z'), then append a Unicode character

to *attribute value* whose codepoint is 0x20 more than the value of the byte at *position*.

5. Otherwise, append a Unicode character to *attribute value* whose codepoint is the same as the value of the byte at *position*.

6. Return to the second step in these substeps.

↪ **If it is 0x3E (ASCII '>')**
Abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name*, its value is the empty string.

↪ **If it is in the range 0x41 (ASCII 'A') to 0x5A (ASCII 'Z')**
Append the Unicode character with codepoint *b*+0x20 to *attribute value* (where *b* is the value of the byte at *position*). Advance *position* to the next byte.

↪ **Anything else**
Append the Unicode character with the same codepoint as the value of the byte at *position*) to *attribute value*. Advance *position* to the next byte.

11. Process the byte at *position* as follows:

↪ **If it is 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), or 0x3E (ASCII '>')**
Abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name* and its value is the value of *attribute value*.

↪ **If it is in the range 0x41 (ASCII 'A') to 0x5A (ASCII 'Z')**
Append the Unicode character with codepoint *b*+0x20 to *attribute value* (where *b* is the value of the byte at *position*).

↪ **Anything else**
Append the Unicode character with the same codepoint as the value of the byte at *position*) to *attribute value*.

12. Advance *position* to the next byte and return to the previous step.

For the sake of interoperability, user agents should not use a pre-scan algorithm that returns different results than the one described above. (But, if you do, please at least let us know, so that we can improve this algorithm and benefit everyone...)

5. If the user agent has information on the likely encoding for this page, e.g. based on the encoding of the page when it was last visited, then return that encoding, with the confidence *tentative*, and abort these steps.

6. The user agent may attempt to autodetect the character encoding from applying frequency analysis or other algorithms to the data stream. If autodetection succeeds in determining a character encoding, then return that encoding, with the confidence *tentative*, and abort these steps. [UNIVCHARDET]

7. Otherwise, return an implementation-defined or user-specified default character encoding, with the confidence *tentative*. In non-legacy environments, the more comprehensive `UTF-8` encoding is recommended. Due to its use in legacy content, `windows-1252` is recommended as a default in predominantly Western demographics instead. Since these encodings can in many cases be distinguished by inspection, a user agent may heuristically decide which to use as a default.

The document's character encoding must immediately be set to the value returned from this algorithm, at the same time as the user agent uses the returned value to select the decoder to use for the input stream.

8.2.2.2 Preprocessing the input stream

Given an encoding, the bytes in the input stream must be converted to Unicode characters for the tokeniser, as described by the rules for that encoding, except that the leading U+FEFF BYTE ORDER MARK character, if any, must not be stripped by the encoding layer (it is stripped by the rule below).

Bytes or sequences of bytes in the original byte stream that could not be converted to Unicode characters must be converted to U+FFFD REPLACEMENT CHARACTER code points.

> ***Note: Bytes or sequences of bytes in the original byte stream that did not conform to the encoding specification (e.g. invalid UTF-8 byte sequences in a UTF-8 input stream) are errors that conformance checkers are expected to report.***

Any byte or sequences of bytes in the original byte stream that is misinterpreted for compatibility is a parse error.

One leading U+FEFF BYTE ORDER MARK character must be ignored if any are present.

All U+0000 NULL characters in the input must be replaced by U+FFFD REPLACEMENT CHARACTERs. Any occurrences of such characters is a parse error.

Any occurrences of any characters in the ranges U+0001 to U+0008, U+000E to U+001F, U+007F to U+009F, U+D800 to U+DFFF, U+FDD0 to U+FDEF, and characters U+000B, U+FFFE, U+FFFF, U+1FFFE, U+1FFFF, U+2FFFE, U+2FFFF, U+3FFFE, U+3FFFF, U+4FFFE, U+4FFFF, U+5FFFE, U+5FFFF, U+6FFFE, U+6FFFF, U+7FFFE, U+7FFFF, U+8FFFE, U+8FFFF, U+9FFFE, U+9FFFF, U+AFFFE, U+AFFFF, U+BFFFE, U+BFFFF, U+CFFFE, U+CFFFF, U+DFFFE, U+DFFFF, U+EFFFE, U+EFFFF, U+FFFFE, U+FFFFF, U+10FFFE, and U+10FFFF are parse errors. (These are all control characters or permanently undefined Unicode characters.)

U+000D CARRIAGE RETURN (CR) characters and U+000A LINE FEED (LF) characters are treated specially. Any CR characters that are followed by LF characters must be removed, and any CR characters not followed by LF characters must be converted to LF characters. Thus, newlines in HTML DOMs are represented by LF characters, and there are never any CR characters in the input to the tokenization stage.

The **next input character** is the first character in the input stream that has not yet been **consumed**. Initially, the *next input character* is the first character in the input. The **current input character** is the last character to have been *consumed*.

The **insertion point** is the position (just before a character or just before the end of the input stream) where content inserted using `document.write()` is actually inserted. The insertion point is relative to the position of the character immediately after it, it is not an absolute offset into the input stream. Initially, the insertion point is uninitialized.

The "EOF" character in the tables below is a conceptual character representing the end of the input stream. If the parser is a script-created parser, then the end of the input stream is reached when an **explicit "EOF" character** (inserted by the `document.close()` method) is consumed. Otherwise, the "EOF" character is not a real character in the stream, but rather the lack of any further characters.

8.2.2.3 Changing the encoding while parsing

When the parser requires the user agent to **change the encoding**, it must run the following steps. This might happen if the encoding sniffing algorithm described above failed to find an encoding, or if it found an encoding that was not the actual encoding of the file.

1. If the new encoding is a UTF-16 encoding, change it to UTF-8.

2. If the new encoding is identical or equivalent to the encoding that is already being used to interpret the input stream, then set the confidence to *confident* and abort these steps. This happens when the encoding information found in the file matches what the encoding sniffing algorithm determined to be the encoding, and in the second pass through the parser if the first pass found that the encoding sniffing algorithm described in the earlier section failed to find the right encoding.

3. If all the bytes up to the last byte converted by the current decoder have the same Unicode interpretations in both the current encoding and the new encoding, and if the user agent supports changing the converter on the fly, then the user agent may change to the new converter for the encoding on the fly. Set the document's character encoding and the encoding used to convert the input stream to the new encoding, set the confidence to *confident*, and abort these steps.

4. Otherwise, navigate to the document again, with replacement enabled, and using the same source browsing context, but this time skip the encoding sniffing algorithm and instead just set the encoding to the new encoding and the confidence to *confident*. Whenever possible, this should be done without actually contacting the network layer (the bytes should be re-parsed from memory), even if, e.g., the document is marked as not being cacheable. If this is not possible and contacting the network layer would involve repeating a request that uses a method other than HTTP GET (or equivalent for non-HTTP URLs), then instead set the confidence to *confident* and ignore the new encoding. The resource will be misinterpreted. User agents may notify the user of the situation, to aid in application development.

8.2.3 Parse state

8.2.3.1 The insertion mode

The **insertion mode** is a flag that controls the primary operation of the tree construction stage.

Initially the insertion mode is "initial". It can change to "before html", "before head", "in head", "in head noscript", "after head", "in body", "in CDATA/RCDATA", "in table", "in caption", "in column group", "in table body", "in row", "in cell", "in select", "in select in table", "in foreign content", "after body", "in frameset", "after frameset", "after after body", and "after after frameset" during the course of the parsing, as described in the tree construction stage. The

insertion mode affects how tokens are processed and whether CDATA sections are supported.

Seven of these modes, namely "in head", "in body", "in CDATA/RCDATA", "in table", "in table body", "in row", "in cell", and "in select", are special, in that the other modes defer to them at various times. When the algorithm below says that the user agent is to do something "**using the rules for** the *m* insertion mode", where *m* is one of these modes, the user agent must use the rules described under the *m* insertion mode's section, but must leave the insertion mode unchanged unless the rules in *m* themselves switch the insertion mode to a new value.

When the insertion mode is switched to "in CDATA/RCDATA", the **original insertion mode** is also set. This is the insertion mode to which the tree construction stage will return when the corresponding end tag is parsed.

When the insertion mode is switched to "in foreign content", the **secondary insertion mode** is also set. This secondary mode is used within the rules for the "in foreign content" mode to handle HTML (i.e. not foreign) content.

When the steps below require the UA to **reset the insertion mode appropriately**, it means the UA must follow these steps:

1. Let *last* be false.
2. Let *node* be the last node in the stack of open elements.
3. If *node* is the first node in the stack of open elements, then set *last* to true and set *node* to the *context* element. (fragment case)
4. If *node* is a `select` element, then switch the insertion mode to "in select" and abort these steps. (fragment case)
5. If *node* is a `td` or `th` element and *last* is false, then switch the insertion mode to "in cell" and abort these steps.
6. If *node* is a `tr` element, then switch the insertion mode to "in row" and abort these steps.
7. If *node* is a `tbody`, `thead`, or `tfoot` element, then switch the insertion mode to "in table body" and abort these steps.
8. If *node* is a `caption` element, then switch the insertion mode to "in caption" and abort these steps.
9. If *node* is a `colgroup` element, then switch the insertion mode to "in column group" and abort these steps. (fragment case)
10. If *node* is a `table` element, then switch the insertion mode to "in table" and abort these steps.
11. If *node* is an element from the MathML namespace or the SVG namespace, then switch the insertion mode to "in foreign content", let the secondary insertion mode be "in body", and abort these steps.
12. If *node* is a `head` element, then switch the insertion mode to "in body" ("in body"! *not "in*

head"!) and abort these steps. (fragment case)

13. If *node* is a `body` element, then switch the insertion mode to "in body" and abort these steps.

14. If *node* is a `frameset` element, then switch the insertion mode to "in frameset" and abort these steps. (fragment case)

15. If *node* is an `html` element, then: if the `head` element pointer is null, switch the insertion mode to "before head", otherwise, switch the insertion mode to "after head". In either case, abort these steps. (fragment case)

16. If *last* is true, then switch the insertion mode to "in body" and abort these steps. (fragment case)

17. Let *node* now be the node before *node* in the stack of open elements.

18. Return to step 3.

8.2.3.2 The stack of open elements

Initially the **stack of open elements** is empty. The stack grows downwards; the topmost node on the stack is the first one added to the stack, and the bottommost node of the stack is the most recently added node in the stack (notwithstanding when the stack is manipulated in a random access fashion as part of the handling for misnested tags).

The "before html" insertion mode creates the `html` root element node, which is then added to the stack.

In the fragment case, the stack of open elements is initialized to contain an `html` element that is created as part of that algorithm. (The fragment case skips the "before html" insertion mode.)

The `html` node, however it is created, is the topmost node of the stack. It never gets popped off the stack.

The **current node** is the bottommost node in this stack.

The **current table** is the last `table` element in the stack of open elements, if there is one. If there is no `table` element in the stack of open elements (fragment case), then the current table is the first element in the stack of open elements (the `html` element).

Elements in the stack fall into the following categories:

Special

The following HTML elements have varying levels of special parsing rules: `address`, `area`, `article`, `aside`, `base`, `basefont`, `bgsound`, `blockquote`, `body`, `br`, `center`, `col`, `colgroup`, `command`, `datagrid`, `dd`, `details`, `dialog`, `dir`, `div`, `dl`, `dt`, `embed`, `fieldset`, `figure`, `footer`, `form`, `frame`, `frameset`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `head`, `header`, `hr`, `iframe`, `img`, `input`, `isindex`, `li`, `link`, `listing`, `menu`, `meta`, `nav`, `noembed`, `noframes`, `noscript`, `ol`, `p`, `param`, `plaintext`, `pre`, `script`, `section`, `select`, `spacer`, `style`, `tbody`, `textarea`, `tfoot`, `thead`, `title`, `tr`, `ul`, and `wbr`.

Scoping

The following HTML elements introduce new scopes for various parts of the parsing: `applet`, `button`, `caption`, `html`, `marquee`, `object`, `table`, `td`, `th`, and SVG's `foreignObject`.

Formatting

The following HTML elements are those that end up in the list of active formatting elements: `a`, `b`, `big`, `code`, `em`, `font`, `i`, `nobr`, `s`, `small`, `strike`, `strong`, `tt`, and `u`.

Phrasing

All other elements found while parsing an HTML document.

The stack of open elements is said to **have an element in scope** when the following algorithm terminates in a match state:

1. Initialize *node* to be the current node (the bottommost node of the stack).
2. If *node* is the target node, terminate in a match state.
3. Otherwise, if *node* is one of the following elements, terminate in a failure state:
    - `applet` in the HTML namespace
    - `caption` in the HTML namespace
    - `html` in the HTML namespace
    - `table` in the HTML namespace
    - `td` in the HTML namespace
    - `th` in the HTML namespace
    - `button` in the HTML namespace
    - `marquee` in the HTML namespace
    - `object` in the HTML namespace
    - `foreignObject` in the SVG namespace
4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2. (This will never fail, since the loop will always terminate in the previous step if the top of the stack — an `html` element — is reached.)

The stack of open elements is said to **have an element in *table scope*** when the following algorithm terminates in a match state:

1. Initialize *node* to be the current node (the bottommost node of the stack).
2. If *node* is the target node, terminate in a match state.
3. Otherwise, if *node* is one of the following elements, terminate in a failure state:
    - `html` in the HTML namespace
    - `table` in the HTML namespace
4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2. (This will never fail, since the loop will always terminate in the previous step if the top of the stack — an `html` element — is reached.)

Nothing happens if at any time any of the elements in the stack of open elements are moved to a new location in, or removed from, the `Document` tree. In particular, the stack is not changed in this situation. This can cause, amongst other strange effects, content to be

appended to nodes that are no longer in the DOM.

Note: In some cases (namely, when closing misnested formatting elements), the stack is manipulated in a random-access fashion.

8.2.3.3 The list of active formatting elements

Initially the **list of active formatting elements** is empty. It is used to handle mis-nested formatting element tags.

The list contains elements in the formatting category, and scope markers. The scope markers are inserted when entering `applet` elements, buttons, `object` elements, marquees, table cells, and table captions, and are used to prevent formatting from "leaking" into `applet` elements, buttons, `object` elements, marquees, and tables.

When the steps below require the UA to **reconstruct the active formatting elements**, the UA must perform the following steps:

1. If there are no entries in the list of active formatting elements, then there is nothing to reconstruct; stop this algorithm.
2. If the last (most recently added) entry in the list of active formatting elements is a marker, or if it is an element that is in the stack of open elements, then there is nothing to reconstruct; stop this algorithm.
3. Let *entry* be the last (most recently added) element in the list of active formatting elements.
4. If there are no entries before *entry* in the list of active formatting elements, then jump to step 8.
5. Let *entry* be the entry one earlier than *entry* in the list of active formatting elements.
6. If *entry* is neither a marker nor an element that is also in the stack of open elements, go to step 4.
7. Let *entry* be the element one later than *entry* in the list of active formatting elements.
8. Perform a shallow clone of the element *entry* to obtain *clone*. [DOM3CORE]
9. Append *clone* to the current node and push it onto the stack of open elements so that it is the new current node.
10. Replace the entry for *entry* in the list with an entry for *clone*.
11. If the entry for *clone* in the list of active formatting elements is not the last entry in the list, return to step 7.

This has the effect of reopening all the formatting elements that were opened in the current body, cell, or caption (whichever is youngest) that haven't been explicitly closed.

Note: The way this specification is written, the list of active formatting elements always consists of elements in chronological order with the least recently added element first and the most recently added element last (except for while steps 8 to 11 of the above algorithm are being executed, of course).

When the steps below require the UA to **clear the list of active formatting elements up to the last marker**, the UA must perform the following steps:

1. Let *entry* be the last (most recently added) entry in the list of active formatting elements.
2. Remove *entry* from the list of active formatting elements.
3. If *entry* was a marker, then stop the algorithm at this point. The list has been cleared up to the last marker.
4. Go to step 1.

8.2.3.4 The element pointers

Initially the **`head` element pointer** and the **`form` element pointer** are both null.

Once a `head` element has been parsed (whether implicitly or explicitly) the `head` element pointer gets set to point to this node.

The `form` element pointer points to the last `form` element that was opened and whose end tag has not yet been seen. It is used to make form controls associate with forms in the face of dramatically bad markup, for historical reasons.

8.2.3.5 Other parsing state flags

The **scripting flag** is set to "enabled" if scripting was enabled for the `Document` with which the parser is associated when the parser was created, and "disabled" otherwise.

The **frameset-ok flag** is set to "ok" when the parser is created. It is set to "not ok" after certain tokens are seen.

8.2.4 Tokenization

Implementations must act as if they used the following state machine to tokenise HTML. The state machine must start in the data state. Most states consume a single character, which may have various side-effects, and either switches the state machine to a new state to *reconsume* the same character, or switches it to a new state (to consume the next character), or repeats the same state (to consume the next character). Some states have more complicated behavior and can consume several characters before switching to another state.

The exact behavior of certain states depends on a **content model flag** that is set after certain tokens are emitted. The flag has several states: *PCDATA*, *RCDATA*, *CDATA*, and *PLAINTEXT*. Initially it must be in the PCDATA state. In the RCDATA and CDATA states, a further **escape flag** is used to control the behavior of the tokeniser. It is either true or false,

and initially must be set to the false state. The insertion mode and the stack of open elements also affects tokenization.

The output of the tokenization step is a series of zero or more of the following tokens: DOCTYPE, start tag, end tag, comment, character, end-of-file. DOCTYPE tokens have a name, a public identifier, a system identifier, and a *force-quirks flag*. When a DOCTYPE token is created, its name, public identifier, and system identifier must be marked as missing (which is a distinct state from the empty string), and the *force-quirks flag* must be set to *off* (its other state is *on*). Start and end tag tokens have a tag name, a *self-closing flag*, and a list of attributes, each of which has a name and a value. When a start or end tag token is created, its *self-closing flag* must be unset (its other state is that it be set), and its attributes list must be empty. Comment and character tokens have data.

When a token is emitted, it must immediately be handled by the tree construction stage. The tree construction stage can affect the state of the content model flag, and can insert additional characters into the stream. (For example, the `script` element can result in scripts executing and using the dynamic markup insertion APIs to insert characters into the stream being tokenised.)

When a start tag token is emitted with its *self-closing flag* set, if the flag is not **acknowledged** when it is processed by the tree construction stage, that is a parse error.

When an end tag token is emitted, the content model flag must be switched to the PCDATA state.

When an end tag token is emitted with attributes, that is a parse error.

When an end tag token is emitted with its *self-closing flag* set, that is a parse error.

Before each step of the tokeniser, the user agent must first check the parser pause flag. If it is true, then the tokeniser must abort the processing of any nested invocations of the tokeniser, yielding control back to the caller. If it is false, then the user agent may then check to see if either one of the scripts in the list of scripts that will execute as soon as possible or the first script in the list of scripts that will execute asynchronously, has completed loading. If one has, then it must be executed and removed from its list.

The tokeniser state machine consists of the states defined in the following subsections.

8.2.4.1 Data state

Consume the next input character:

- **U+0026 AMPERSAND (&)**
  When the content model flag is set to one of the PCDATA or RCDATA states and the escape flag is false: switch to the character reference data state.
  Otherwise: treat it as per the "anything else" entry below.
- **U+002D HYPHEN-MINUS (-)**
  If the content model flag is set to either the RCDATA state or the CDATA state, and the escape flag is false, and there are at least three characters before this one in the input stream, and the last four characters in the input stream, including this one, are U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, and U+002D HYPHEN-MINUS ("<!--"), then set the escape flag

to true.

In any case, emit the input character as a character token. Stay in the data state.

↪ **U+003C LESS-THAN SIGN (<)**
When the content model flag is set to the PCDATA state: switch to the tag open state.
When the content model flag is set to either the RCDATA state or the CDATA state, and the escape flag is false: switch to the tag open state.
Otherwise: treat it as per the "anything else" entry below.

↪ **U+003E GREATER-THAN SIGN (>)**
If the content model flag is set to either the RCDATA state or the CDATA state, and the escape flag is true, and the last three characters in the input stream including this one are U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN ("-->"), set the escape flag to false.

In any case, emit the input character as a character token. Stay in the data state.

↪ **EOF**
Emit an end-of-file token.

↪ **Anything else**
Emit the input character as a character token. Stay in the data state.

8.2.4.2 Character reference data state

(This cannot happen if the content model flag is set to the CDATA state.)

Attempt to consume a character reference, with no additional allowed character.

If nothing is returned, emit a U+0026 AMPERSAND character token.

Otherwise, emit the character token that was returned.

Finally, switch to the data state.

8.2.4.3 Tag open state

The behavior of this state depends on the content model flag.

If the content model flag is set to the RCDATA or CDATA states
Consume the next input character. If it is a U+002F SOLIDUS (/) character, switch to the close tag open state. Otherwise, emit a U+003C LESS-THAN SIGN character token and reconsume the current input character in the data state.

If the content model flag is set to the PCDATA state
Consume the next input character:

↪ **U+0021 EXCLAMATION MARK (!)**
Switch to the markup declaration open state.

↪ **U+002F SOLIDUS (/)**
Switch to the close tag open state.

↪ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

Create a new start tag token, set its tag name to the lowercase version of the input character (add 0x0020 to the character's code point), then switch to the tag name state. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+0061 LATIN SMALL LETTER A through to U+007A LATIN SMALL LETTER Z**

Create a new start tag token, set its tag name to the input character, then switch to the tag name state. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Emit a U+003C LESS-THAN SIGN character token and a U+003E GREATER-THAN SIGN character token. Switch to the data state.

↪ **U+003F QUESTION MARK (?)**

Parse error. Switch to the bogus comment state.

↪ **Anything else**

Parse error. Emit a U+003C LESS-THAN SIGN character token and reconsume the current input character in the data state.

8.2.4.4 Close tag open state

If the content model flag is set to the RCDATA or CDATA states but no start tag token has ever been emitted by this instance of the tokeniser (fragment case), or, if the content model flag is set to the RCDATA or CDATA states and the next few characters do not match the tag name of the last start tag token emitted (compared in an ASCII case-insensitive manner), or if they do but they are not immediately followed by one of the following characters:

- U+0009 CHARACTER TABULATION
- U+000A LINE FEED (LF)
- U+000C FORM FEED (FF)
- U+0020 SPACE
- U+003E GREATER-THAN SIGN (>)
- U+002F SOLIDUS (/)
- EOF

...then emit a U+003C LESS-THAN SIGN character token, a U+002F SOLIDUS character token, and switch to the data state to process the next input character.

Otherwise, if the content model flag is set to the PCDATA state, or if the next few characters *do* match that tag name, consume the next input character:

↪ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

Create a new end tag token, set its tag name to the lowercase version of the input character (add 0x0020 to the character's code point), then switch to the tag name state. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+0061 LATIN SMALL LETTER A through to U+007A LATIN SMALL LETTER Z**

Create a new end tag token, set its tag name to the input character, then switch to the tag name state. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Switch to the data state.

- **EOF**
  Parse error. Emit a U+003C LESS-THAN SIGN character token and a U+002F SOLIDUS character token. Reconsume the EOF character in the data state.
- **Anything else**
  Parse error. Switch to the bogus comment state.

8.2.4.5 Tag name state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Switch to the before attribute name state.
- **U+002F SOLIDUS (/)**
  Switch to the self-closing start tag state.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current tag token. Switch to the data state.
- **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**
  Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the current tag token's tag name. Stay in the tag name state.
- **EOF**
  Parse error. Emit the current tag token. Reconsume the EOF character in the data state.
- **Anything else**
  Append the current input character to the current tag token's tag name. Stay in the tag name state.

8.2.4.6 Before attribute name state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Stay in the before attribute name state.
- **U+002F SOLIDUS (/)**
  Switch to the self-closing start tag state.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current tag token. Switch to the data state.
- **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**
  Start a new attribute in the current tag token. Set that attribute's name to the lowercase version of the current input character (add 0x0020 to the character's code point), and its value to the empty string. Switch to the attribute name state.
- **U+0022 QUOTATION MARK (")**
- **U+0027 APOSTROPHE (')**
- **U+003D EQUALS SIGN (=)**

Parse error. Treat it as per the "anything else" entry below.

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

↪ **Anything else**

Start a new attribute in the current tag token. Set that attribute's name to the current input character, and its value to the empty string. Switch to the attribute name state.

8.2.4.7 Attribute name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000C FORM FEED (FF)**
↪ **U+0020 SPACE**

Switch to the after attribute name state.

↪ **U+002F SOLIDUS (/)**

Switch to the self-closing start tag state.

↪ **U+003D EQUALS SIGN (=)**

Switch to the before attribute value state.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state.

↪ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the current attribute's name. Stay in the attribute name state.

↪ **U+0022 QUOTATION MARK (")**
↪ **U+0027 APOSTROPHE (')**

Parse error. Treat it as per the "anything else" entry below.

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

↪ **Anything else**

Append the current input character to the current attribute's name. Stay in the attribute name state.

When the user agent leaves the attribute name state (and before emitting the tag token, if appropriate), the complete attribute's name must be compared to the other attributes on the same token; if there is already an attribute on the token with the exact same name, then this is a parse error and the new attribute must be dropped, along with the value that gets associated with it (if any).

8.2.4.8 After attribute name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000C FORM FEED (FF)**

- **U+0020 SPACE**
  Stay in the after attribute name state.
- **U+002F SOLIDUS (/)**
  Switch to the self-closing start tag state.
- **U+003D EQUALS SIGN (=)**
  Switch to the before attribute value state.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current tag token. Switch to the data state.
- **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**
  Start a new attribute in the current tag token. Set that attribute's name to the lowercase version of the current input character (add 0x0020 to the character's code point), and its value to the empty string. Switch to the attribute name state.
- **U+0022 QUOTATION MARK (")**
- **U+0027 APOSTROPHE (')**
  Parse error. Treat it as per the "anything else" entry below.
- **EOF**
  Parse error. Emit the current tag token. Reconsume the EOF character in the data state.
- **Anything else**
  Start a new attribute in the current tag token. Set that attribute's name to the current input character, and its value to the empty string. Switch to the attribute name state.

8.2.4.9 Before attribute value state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Stay in the before attribute value state.
- **U+0022 QUOTATION MARK (")**
  Switch to the attribute value (double-quoted) state.
- **U+0026 AMPERSAND (&)**
  Switch to the attribute value (unquoted) state and reconsume this input character.
- **U+0027 APOSTROPHE (')**
  Switch to the attribute value (single-quoted) state.
- **U+003E GREATER-THAN SIGN (>)**
  Parse error. Emit the current tag token. Switch to the data state.
- **U+003D EQUALS SIGN (=)**
  Parse error. Treat it as per the "anything else" entry below.
- **EOF**
  Parse error. Emit the current tag token. Reconsume the character in the data state.
- **Anything else**
  Append the current input character to the current attribute's value. Switch to the attribute value (unquoted) state.

8.2.4.10 Attribute value (double-quoted) state

Consume the next input character:

- **U+0022 QUOTATION MARK (")**
  Switch to the after attribute value (quoted) state.
- **U+0026 AMPERSAND (&)**
  Switch to the character reference in attribute value state, with the additional allowed character being U+0022 QUOTATION MARK (").
- **EOF**
  Parse error. Emit the current tag token. Reconsume the character in the data state.
- **Anything else**
  Append the current input character to the current attribute's value. Stay in the attribute value (double-quoted) state.

8.2.4.11 Attribute value (single-quoted) state

Consume the next input character:

- **U+0027 APOSTROPHE (')**
  Switch to the after attribute value (quoted) state.
- **U+0026 AMPERSAND (&)**
  Switch to the character reference in attribute value state, with the additional allowed character being U+0027 APOSTROPHE (').
- **EOF**
  Parse error. Emit the current tag token. Reconsume the character in the data state.
- **Anything else**
  Append the current input character to the current attribute's value. Stay in the attribute value (single-quoted) state.

8.2.4.12 Attribute value (unquoted) state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Switch to the before attribute name state.
- **U+0026 AMPERSAND (&)**
  Switch to the character reference in attribute value state, with no additional allowed character.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current tag token. Switch to the data state.
- **U+0022 QUOTATION MARK (")**
- **U+0027 APOSTROPHE (')**
- **U+003D EQUALS SIGN (=)**
  Parse error. Treat it as per the "anything else" entry below.
- **EOF**
  Parse error. Emit the current tag token. Reconsume the character in the data state.
- **Anything else**
  Append the current input character to the current attribute's value. Stay in the attribute value (unquoted) state.

8.2.4.13 Character reference in attribute value state

Attempt to consume a character reference.

If nothing is returned, append a U+0026 AMPERSAND character to the current attribute's value.

Otherwise, append the returned character token to the current attribute's value.

Finally, switch back to the attribute value state that you were in when were switched into this state.

8.2.4.14 After attribute value (quoted) state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Switch to the before attribute name state.
- **U+002F SOLIDUS (/)**
  Switch to the self-closing start tag state.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current tag token. Switch to the data state.
- **EOF**
  Parse error. Emit the current tag token. Reconsume the EOF character in the data state.
- **Anything else**
  Parse error. Reconsume the character in the before attribute name state.

8.2.4.15 Self-closing start tag state

Consume the next input character:

- **U+003E GREATER-THAN SIGN (>)**
  Set the *self-closing flag* of the current tag token. Emit the current tag token. Switch to the data state.
- **EOF**
  Parse error. Emit the current tag token. Reconsume the EOF character in the data state.
- **Anything else**
  Parse error. Reconsume the character in the before attribute name state.

8.2.4.16 Bogus comment state

(This can only happen if the content model flag is set to the PCDATA state.)

Consume every character up to and including the first U+003E GREATER-THAN SIGN character (>) or the end of the file (EOF), whichever comes first. Emit a comment token

whose data is the concatenation of all the characters starting from and including the character that caused the state machine to switch into the bogus comment state, up to and including the character immediately before the last consumed character (i.e. up to the character just before the U+003E or EOF character). (If the comment was started by the end of the file (EOF), the token is empty.)

Switch to the data state.

If the end of the file was reached, reconsume the EOF character.

8.2.4.17 Markup declaration open state

(This can only happen if the content model flag is set to the PCDATA state.)

If the next two characters are both U+002D HYPHEN-MINUS (-) characters, consume those two characters, create a comment token whose data is the empty string, and switch to the comment start state.

Otherwise, if the next seven characters are an ASCII case-insensitive match for the word "DOCTYPE", then consume those characters and switch to the DOCTYPE state.

Otherwise, if the insertion mode is "in foreign content" and the current node is not an element in the HTML namespace and the next seven characters are an ASCII case-sensitive match for the string "[CDATA[" (the five uppercase letters "CDATA" with a U+005B LEFT SQUARE BRACKET character before and after), then consume those characters and switch to the CDATA section state (which is unrelated to the content model flag's CDATA state).

Otherwise, this is a parse error. Switch to the bogus comment state. The next character that is consumed, if any, is the first character that will be in the comment.

8.2.4.18 Comment start state

Consume the next input character:

- **U+002D HYPHEN-MINUS (-)**
  Switch to the comment start dash state.
- **U+003E GREATER-THAN SIGN (>)**
  Parse error. Emit the comment token. Switch to the data state.
- **EOF**
  Parse error. Emit the comment token. Reconsume the EOF character in the data state.
- **Anything else**
  Append the input character to the comment token's data. Switch to the comment state.

8.2.4.19 Comment start dash state

Consume the next input character:

- **U+002D HYPHEN-MINUS (-)**

Switch to the comment end state

↪ **U+003E GREATER-THAN SIGN (>)**
Parse error. Emit the comment token. Switch to the data state.

↪ **EOF**
Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**
Append a U+002D HYPHEN-MINUS (-) character and the input character to the comment token's data. Switch to the comment state.

8.2.4.20 Comment state

Consume the next input character:

↪ **U+002D HYPHEN-MINUS (-)**
Switch to the comment end dash state

↪ **EOF**
Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**
Append the input character to the comment token's data. Stay in the comment state.

8.2.4.21 Comment end dash state

Consume the next input character:

↪ **U+002D HYPHEN-MINUS (-)**
Switch to the comment end state

↪ **EOF**
Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**
Append a U+002D HYPHEN-MINUS (-) character and the input character to the comment token's data. Switch to the comment state.

8.2.4.22 Comment end state

Consume the next input character:

↪ **U+003E GREATER-THAN SIGN (>)**
Emit the comment token. Switch to the data state.

↪ **U+002D HYPHEN-MINUS (-)**
Parse error. Append a U+002D HYPHEN-MINUS (-) character to the comment token's data. Stay in the comment end state.

↪ **EOF**
Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Append two U+002D HYPHEN-MINUS (-) characters and the input character to the comment token's data. Switch to the comment state.

8.2.4.23 DOCTYPE state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Switch to the before DOCTYPE name state.
- **Anything else**
  Parse error. Reconsume the current character in the before DOCTYPE name state.

8.2.4.24 Before DOCTYPE name state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Stay in the before DOCTYPE name state.
- **U+003E GREATER-THAN SIGN (>)**
  Parse error. Create a new DOCTYPE token. Set its *force-quirks flag* to *on*. Emit the token. Switch to the data state.
- **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**
  Create a new DOCTYPE token. Set the token's name to the lowercase version of the input character (add 0x0020 to the character's code point). Switch to the DOCTYPE name state.
- **EOF**
  Parse error. Create a new DOCTYPE token. Set its *force-quirks flag* to *on*. Emit the token. Reconsume the EOF character in the data state.
- **Anything else**
  Create a new DOCTYPE token. Set the token's name to the current input character. Switch to the DOCTYPE name state.

8.2.4.25 DOCTYPE name state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Switch to the after DOCTYPE name state.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current DOCTYPE token. Switch to the data state.

- **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**
  Append the lowercase version of the input character (add 0x0020 to the character's code point) to the current DOCTYPE token's name. Stay in the DOCTYPE name state.
- **EOF**
  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.
- **Anything else**
  Append the current input character to the current DOCTYPE token's name. Stay in the DOCTYPE name state.

8.2.4.26 After DOCTYPE name state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Stay in the after DOCTYPE name state.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current DOCTYPE token. Switch to the data state.
- **EOF**
  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.
- **Anything else**
  If the six characters starting from the current input character are an ASCII case-insensitive match for the word "PUBLIC", then consume those characters and switch to the before DOCTYPE public identifier state.

  Otherwise, if the six characters starting from the current input character are an ASCII case-insensitive match for the word "SYSTEM", then consume those characters and switch to the before DOCTYPE system identifier state.

  Otherwise, this is the parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Switch to the bogus DOCTYPE state.

8.2.4.27 Before DOCTYPE public identifier state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Stay in the before DOCTYPE public identifier state.
- **U+0022 QUOTATION MARK (")**
  Set the DOCTYPE token's public identifier to the empty string (not missing), then switch to the DOCTYPE public identifier (double-quoted) state.
- **U+0027 APOSTROPHE (')**

Set the DOCTYPE token's public identifier to the empty string (not missing), then switch to the DOCTYPE public identifier (single-quoted) state.

↪ **U+003E GREATER-THAN SIGN (>)**
Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**
Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**
Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Switch to the bogus DOCTYPE state.

8.2.4.28 DOCTYPE public identifier (double-quoted) state

Consume the next input character:

↪ **U+0022 QUOTATION MARK (")**
Switch to the after DOCTYPE public identifier state.

↪ **U+003E GREATER-THAN SIGN (>)**
Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**
Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**
Append the current input character to the current DOCTYPE token's public identifier. Stay in the DOCTYPE public identifier (double-quoted) state.

8.2.4.29 DOCTYPE public identifier (single-quoted) state

Consume the next input character:

↪ **U+0027 APOSTROPHE (')**
Switch to the after DOCTYPE public identifier state.

↪ **U+003E GREATER-THAN SIGN (>)**
Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**
Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**
Append the current input character to the current DOCTYPE token's public identifier. Stay in the DOCTYPE public identifier (single-quoted) state.

8.2.4.30 After DOCTYPE public identifier state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Stay in the after DOCTYPE public identifier state.
- **U+0022 QUOTATION MARK (")**

  Set the DOCTYPE token's system identifier to the empty string (not missing), then switch to the DOCTYPE system identifier (double-quoted) state.
- **U+0027 APOSTROPHE (')**

  Set the DOCTYPE token's system identifier to the empty string (not missing), then switch to the DOCTYPE system identifier (single-quoted) state.
- **U+003E GREATER-THAN SIGN (>)**

  Emit the current DOCTYPE token. Switch to the data state.
- **EOF**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.
- **Anything else**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Switch to the bogus DOCTYPE state.

8.2.4.31 Before DOCTYPE system identifier state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Stay in the before DOCTYPE system identifier state.
- **U+0022 QUOTATION MARK (")**

  Set the DOCTYPE token's system identifier to the empty string (not missing), then switch to the DOCTYPE system identifier (double-quoted) state.
- **U+0027 APOSTROPHE (')**

  Set the DOCTYPE token's system identifier to the empty string (not missing), then switch to the DOCTYPE system identifier (single-quoted) state.
- **U+003E GREATER-THAN SIGN (>)**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.
- **EOF**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.
- **Anything else**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Switch to the bogus DOCTYPE state.

8.2.4.32 DOCTYPE system identifier (double-quoted) state

Consume the next input character:

- **U+0022 QUOTATION MARK (")**

  Switch to the after DOCTYPE system identifier state.

- **U+003E GREATER-THAN SIGN (>)**
  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.
- **EOF**
  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.
- **Anything else**
  Append the current input character to the current DOCTYPE token's system identifier. Stay in the DOCTYPE system identifier (double-quoted) state.

8.2.4.33 DOCTYPE system identifier (single-quoted) state

Consume the next input character:

- **U+0027 APOSTROPHE (')**
  Switch to the after DOCTYPE system identifier state.
- **U+003E GREATER-THAN SIGN (>)**
  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.
- **EOF**
  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.
- **Anything else**
  Append the current input character to the current DOCTYPE token's system identifier. Stay in the DOCTYPE system identifier (single-quoted) state.

8.2.4.34 After DOCTYPE system identifier state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Stay in the after DOCTYPE system identifier state.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current DOCTYPE token. Switch to the data state.
- **EOF**
  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.
- **Anything else**
  Parse error. Switch to the bogus DOCTYPE state. (This does *not* set the DOCTYPE token's *force-quirks flag* to *on*.)

8.2.4.35 Bogus DOCTYPE state

Consume the next input character:

- **U+003E GREATER-THAN SIGN (>)**

Emit the DOCTYPE token. Switch to the data state.

- **EOF**
  Emit the DOCTYPE token. Reconsume the EOF character in the data state.
- **Anything else**
  Stay in the bogus DOCTYPE state.

8.2.4.36 CDATA section state

(This can only happen if the content model flag is set to the PCDATA state, and is unrelated to the content model flag's CDATA state.)

Consume every character up to the next occurrence of the three character sequence U+005D RIGHT SQUARE BRACKET U+005D RIGHT SQUARE BRACKET U+003E GREATER-THAN SIGN (`]]>`), or the end of the file (EOF), whichever comes first. Emit a series of character tokens consisting of all the characters consumed except the matching three character sequence at the end (if one was found before the end of the file).

Switch to the data state.

If the end of the file was reached, reconsume the EOF character.

8.2.4.37 Tokenizing character references

This section defines how to **consume a character reference**. This definition is used when parsing character references in text and in attributes.

The behavior depends on the identity of the next character (the one immediately after the U+0026 AMPERSAND character):

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
- **U+003C LESS-THAN SIGN**
- **U+0026 AMPERSAND**
- **EOF**
- **The *additional allowed character*, if there is one**
  Not a character reference. No characters are consumed, and nothing is returned. (This is not an error, either.)
- **U+0023 NUMBER SIGN (#)**
  Consume the U+0023 NUMBER SIGN.

  The behavior further depends on the character after the U+0023 NUMBER SIGN:

  - **U+0078 LATIN SMALL LETTER X**
  - **U+0058 LATIN CAPITAL LETTER X**
    Consume the X.

    Follow the steps below, but using the range of characters U+0030 DIGIT ZERO through to U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER

A through to U+0066 LATIN SMALL LETTER F, and U+0041 LATIN CAPITAL LETTER A, through to U+0046 LATIN CAPITAL LETTER F (in other words, 0-9, A-F, a-f).

When it comes to interpreting the number, interpret it as a hexadecimal number.

↪ **Anything else**

Follow the steps below, but using the range of characters U+0030 DIGIT ZERO through to U+0039 DIGIT NINE (i.e. just 0-9).

When it comes to interpreting the number, interpret it as a decimal number.

Consume as many characters as match the range of characters given above.

If no characters match the range, then don't consume any characters (and unconsume the U+0023 NUMBER SIGN character and, if appropriate, the X character). This is a parse error; nothing is returned.

Otherwise, if the next character is a U+003B SEMICOLON, consume that too. If it isn't, there is a parse error.

If one or more characters match the range, then take them all and interpret the string of characters as a number (either hexadecimal or decimal as appropriate).

If that number is one of the numbers in the first column of the following table, then this is a parse error. Find the row with that number in the first column, and return a character token for the Unicode character given in the second column of that row.

Number	Unicode character	
0x0D	U+000A	LINE FEED (LF)
0x80	U+20AC	EURO SIGN ('€')
0x81	U+FFFD	REPLACEMENT CHARACTER
0x82	U+201A	SINGLE LOW-9 QUOTATION MARK (',')
0x83	U+0192	LATIN SMALL LETTER F WITH HOOK ('ƒ')
0x84	U+201E	DOUBLE LOW-9 QUOTATION MARK ('„')
0x85	U+2026	HORIZONTAL ELLIPSIS ('…')
0x86	U+2020	DAGGER ('†')
0x87	U+2021	DOUBLE DAGGER ('‡')
0x88	U+02C6	MODIFIER LETTER CIRCUMFLEX ACCENT ('ˆ')
0x89	U+2030	PER MILLE SIGN ('‰')
0x8A	U+0160	LATIN CAPITAL LETTER S WITH CARON ('Š')
0x8B	U+2039	SINGLE LEFT-POINTING ANGLE QUOTATION MARK ('‹')
0x8C	U+0152	LATIN CAPITAL LIGATURE OE ('Œ')
0x8D	U+FFFD	REPLACEMENT CHARACTER
0x8E	U+017D	LATIN CAPITAL LETTER Z WITH CARON ('Ž')

Number		Unicode character
0x8F	U+FFFD	REPLACEMENT CHARACTER
0x90	U+FFFD	REPLACEMENT CHARACTER
0x91	U+2018	LEFT SINGLE QUOTATION MARK ('‘')
0x92	U+2019	RIGHT SINGLE QUOTATION MARK ('’')
0x93	U+201C	LEFT DOUBLE QUOTATION MARK ('“')
0x94	U+201D	RIGHT DOUBLE QUOTATION MARK ('”')
0x95	U+2022	BULLET ('•')
0x96	U+2013	EN DASH ('–')
0x97	U+2014	EM DASH ('—')
0x98	U+02DC	SMALL TILDE ('˜')
0x99	U+2122	TRADE MARK SIGN ('™')
0x9A	U+0161	LATIN SMALL LETTER S WITH CARON ('š')
0x9B	U+203A	SINGLE RIGHT-POINTING ANGLE QUOTATION MARK ('›')
0x9C	U+0153	LATIN SMALL LIGATURE OE ('œ')
0x9D	U+FFFD	REPLACEMENT CHARACTER
0x9E	U+017E	LATIN SMALL LETTER Z WITH CARON ('ž')
0x9F	U+0178	LATIN CAPITAL LETTER Y WITH DIAERESIS ('Ÿ')

Otherwise, if the number is in the range 0x0000 to 0x0008, 0x000E to 0x001F, 0x007F to 0x009F, 0xD800 to 0xDFFF, 0xFDD0 to 0xFDEF, or is one of 0x000B, 0xFFFE, 0xFFFF, 0x1FFFE, 0x1FFFF, 0x2FFFE, 0x2FFFF, 0x3FFFE, 0x3FFFF, 0x4FFFE, 0x4FFFF, 0x5FFFE, 0x5FFFF, 0x6FFFE, 0x6FFFF, 0x7FFFE, 0x7FFFF, 0x8FFFE, 0x8FFFF, 0x9FFFE, 0x9FFFF, 0xAFFFE, 0xAFFFF, 0xBFFFE, 0xBFFFF, 0xCFFFE, 0xCFFFF, 0xDFFFE, 0xDFFFF, 0xEFFFE, 0xEFFFF, 0xFFFFE, 0xFFFFF, 0x10FFFE, or 0x10FFFF, or is higher than 0x10FFFF, then this is a parse error; return a character token for the U+FFFD REPLACEMENT CHARACTER character instead.

Otherwise, return a character token for the Unicode character whose code point is that number.

↪ **Anything else**

Consume the maximum number of characters possible, with the consumed characters matching one of the identifiers in the first column of the named character references table (in a case-sensitive manner).

If no match can be made, then this is a parse error. No characters are consumed, and nothing is returned.

If the last character matched is not a U+003B SEMICOLON (;), there is a parse error.

If the character reference is being consumed as part of an attribute, and the last character matched is not a U+003B SEMICOLON (;), and the next character is in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE, U+0041 LATIN CAPITAL

LETTER A to U+005A LATIN CAPITAL LETTER Z, or U+0061 LATIN SMALL LETTER A to U+007A LATIN SMALL LETTER Z, then, for historical reasons, all the characters that were matched after the U+0026 AMPERSAND (&) must be unconsumed, and nothing is returned.

Otherwise, return a character token for the character corresponding to the character reference name (as given by the second column of the named character references table).

> If the markup contains `I'm &notit; I tell you`, the character reference is parsed as "not", as in, `I'm ¬it; I tell you`. But if the markup was `I'm ∉ I tell you`, the character reference would be parsed as "notin;", resulting in `I'm ∉ I tell you`.

8.2.5 Tree construction

The input to the tree construction stage is a sequence of tokens from the tokenization stage. The tree construction stage is associated with a DOM `Document` object when a parser is created. The "output" of this stage consists of dynamically modifying or extending that document's DOM tree.

This specification does not define when an interactive user agent has to render the `Document` so that it is available to the user, or when it has to begin accepting user input.

As each token is emitted from the tokeniser, the user agent must process the token according to the rules given in the section corresponding to the current insertion mode.

When the steps below require the UA to **insert a character** into a node, if that node has a child immediately before where the character is to be inserted, and that child is a `Text` node, and that `Text` node was the last node that the parser inserted into the document, then the character must be appended to that `Text` node; otherwise, a new `Text` node whose data is just that character must be inserted in the appropriate place.

DOM mutation events must not fire for changes caused by the UA parsing the document. (Conceptually, the parser is not mutating the DOM, it is constructing it.) This includes the parsing of any content inserted using `document.write()` and `document.writeln()` calls. [DOM3EVENTS]

> ***Note: Not all of the tag names mentioned below are conformant tag names in this specification; many are included to handle legacy content. They still form part of the algorithm that implementations are required to implement to claim conformance.***

> ***Note: The algorithm described below places no limit on the depth of the DOM tree generated, or on the length of tag names, attribute names, attribute values, text nodes, etc. While implementors are encouraged to avoid arbitrary limits, it is recognized that practical concerns will likely force user agents to impose nesting depths.***

8.2.5.1 Creating and inserting elements

When the steps below require the UA to **create an element for a token** in a particular namespace, the UA must create a node implementing the interface appropriate for the element type corresponding to the tag name of the token in the given namespace (as given in the specification that defines that element, e.g. for an `a` element in the HTML namespace, this specification defines it to be the `HTMLAnchorElement` interface), with the tag name being the name of that element, with the node being in the given namespace, and with the attributes on the node being those given in the given token.

The interface appropriate for an element in the HTML namespace that is not defined in this specification is `HTMLElement`. The interface appropriate for an element in another namespace that is not defined by that namespace's specification is `Element`.

When a resettable element is created in this manner, its reset algorithm must be invoked once the attributes are set. (This initializes the element's value and checkedness based on the element's attributes.)

When the steps below require the UA to **insert an HTML element** for a token, the UA must first create an element for the token in the HTML namespace, and then append this node to the current node, and push it onto the stack of open elements so that it is the new current node.

The steps below may also require that the UA insert an HTML element in a particular place, in which case the UA must follow the same steps except that it must insert or append the new node in the location specified instead of appending it to the current node. (This happens in particular during the parsing of tables with invalid content.)

If an element created by the insert an HTML element algorithm is a form-associated element, and the `form` element pointer is not null, and the newly created element doesn't have a `form` attribute, the user agent must associate the newly created element with the `form` element pointed to by the `form` element pointer before inserting it wherever it is to be inserted.

When the steps below require the UA to **insert a foreign element** for a token, the UA must first create an element for the token in the given namespace, and then append this node to the current node, and push it onto the stack of open elements so that it is the new current node. If the newly created element has an `xmlns` attribute in the XMLNS namespace whose value is not exactly the same as the element's namespace, that is a parse error. Similarly, if the newly created element has an `xmlns:xlink` attribute in the XMLNS namespace whose value is not the XLink Namespace, that is a parse error.

When the steps below require the user agent to **adjust MathML attributes** for a token, then, if the token has an attribute named `definitionurl`, change its name to `definitionURL` (note the case difference).

When the steps below require the user agent to **adjust SVG attributes** for a token, then, for each attribute on the token whose attribute name is one of the ones in the first column of the following table, change the attribute's name to the name given in the corresponding cell in the second column. (This fixes the case of SVG attributes that are not all lowercase.)

Attribute name on token	Attribute name on element

Attribute name on token	Attribute name on element
attributename	attributeName
attributetype	attributeType
basefrequency	baseFrequency
baseprofile	baseProfile
calcmode	calcMode
clippathunits	clipPathUnits
contentscripttype	contentScriptType
contentstyletype	contentStyleType
diffuseconstant	diffuseConstant
edgemode	edgeMode
externalresourcesrequired	externalResourcesRequired
filterres	filterRes
filterunits	filterUnits
glyphref	glyphRef
gradienttransform	gradientTransform
gradientunits	gradientUnits
kernelmatrix	kernelMatrix
kernelunitlength	kernelUnitLength
keypoints	keyPoints
keysplines	keySplines
keytimes	keyTimes
lengthadjust	lengthAdjust
limitingconeangle	limitingConeAngle
markerheight	markerHeight
markerunits	markerUnits
markerwidth	markerWidth
maskcontentunits	maskContentUnits
maskunits	maskUnits
numoctaves	numOctaves
pathlength	pathLength
patterncontentunits	patternContentUnits
patterntransform	patternTransform
patternunits	patternUnits
pointsatx	pointsAtX
pointsaty	pointsAtY
pointsatz	pointsAtZ
preservealpha	preserveAlpha

Attribute name on token	Attribute name on element
preserveaspectratio	preserveAspectRatio
primitiveunits	primitiveUnits
refx	refX
refy	refY
repeatcount	repeatCount
repeatdur	repeatDur
requiredextensions	requiredExtensions
requiredfeatures	requiredFeatures
specularconstant	specularConstant
specularexponent	specularExponent
spreadmethod	spreadMethod
startoffset	startOffset
stddeviation	stdDeviation
stitchtiles	stitchTiles
surfacescale	surfaceScale
systemlanguage	systemLanguage
tablevalues	tableValues
targetx	targetX
targety	targetY
textlength	textLength
viewbox	viewBox
viewtarget	viewTarget
xchannelselector	xChannelSelector
ychannelselector	yChannelSelector
zoomandpan	zoomAndPan

When the steps below require the user agent to **adjust foreign attributes** for a token, then, if any of the attributes on the token match the strings given in the first column of the following table, let the attribute be a namespaced attribute, with the prefix being the string given in the corresponding cell in the second column, the local name being the string given in the corresponding cell in the third column, and the namespace being the namespace given in the corresponding cell in the fourth column. (This fixes the use of namespaced attributes, in particular `xml:lang`.)

Attribute name	Prefix	Local name	Namespace
xlink:actuate	xlink	actuate	XLink namespace
xlink:arcrole	xlink	arcrole	XLink namespace
xlink:href	xlink	href	XLink namespace
xlink:role	xlink	role	XLink namespace

Attribute name	Prefix	Local name	Namespace
xlink:show	xlink	show	XLink namespace
xlink:title	xlink	title	XLink namespace
xlink:type	xlink	type	XLink namespace
xml:base	xml	base	XML namespace
xml:lang	xml	lang	XML namespace
xml:space	xml	space	XML namespace
xmlns	(none)	xmlns	XMLNS namespace
xmlns:xlink	xmlns	xlink	XMLNS namespace

The **generic CDATA element parsing algorithm** and the **generic RCDATA element parsing algorithm** consist of the following steps. These algorithms are always invoked in response to a start tag token.

1. Insert an HTML element for the token.
2. If the algorithm that was invoked is the generic CDATA element parsing algorithm, switch the tokeniser's content model flag to the CDATA state; otherwise the algorithm invoked was the generic RCDATA element parsing algorithm, switch the tokeniser's content model flag to the RCDATA state.
3. Let the original insertion mode be the current insertion mode.
4. Then, switch the insertion mode to "in CDATA/RCDATA".

8.2.5.2 Closing elements that have implied end tags

When the steps below require the UA to **generate implied end tags**, then, while the current node is a `dd` element, a `dt` element, an `li` element, an `option` element, an `optgroup` element, a `p` element, an `rp` element, or an `rt` element, the UA must pop the current node off the stack of open elements.

If a step requires the UA to generate implied end tags but lists an element to exclude from the process, then the UA must perform the above steps as if that element was not in the above list.

8.2.5.3 Foster parenting

Foster parenting happens when content is misnested in tables.

When a node *node* is to be **foster parented**, the node *node* must be inserted into the *foster parent element*, and the current table must be marked as **tainted**. (Once the current table has been tainted, whitespace characters are inserted into the *foster parent element* instead of the current node.)

The **foster parent element** is the parent element of the last `table` element in the stack of open elements, if there is a `table` element and it has such a parent element. If there is no

`table` element in the stack of open elements (fragment case), then the *foster parent element* is the first element in the stack of open elements (the `html` element). Otherwise, if there is a `table` element in the stack of open elements, but the last `table` element in the stack of open elements has no parent, or its parent node is not an element, then the *foster parent element* is the element before the last `table` element in the stack of open elements.

If the *foster parent element* is the parent element of the last `table` element in the stack of open elements, then *node* must be inserted immediately *before* the last `table` element in the stack of open elements in the *foster parent element*; otherwise, *node* must be *appended* to the *foster parent element*.

8.2.5.4 The "initial" insertion mode

When the insertion mode is "initial", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Ignore the token.

↪ **A comment token**

Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

If the DOCTYPE token's `name` is not a case-sensitive match for the string "`html`", or if the token's public identifier is not missing, or if the token's system identifier is neither missing nor a case-sensitive match for the string "`about:legacy-compat`", then there is a parse error (this is the **DOCTYPE parse error**). Conformance checkers may, instead of reporting this error, switch to a conformance checking mode for another language (e.g. based on the DOCTYPE token a conformance checker could recognize that the document is an HTML4-era document, and defer to an HTML4 conformance checker.)

Append a `DocumentType` node to the `Document` node, with the `name` attribute set to the name given in the DOCTYPE token, or the empty string if the name was missing; the `publicId` attribute set to the public identifier given in the DOCTYPE token, or the empty string if the public identifier was missing; the `systemId` attribute set to the system identifier given in the DOCTYPE token, or the empty string if the system identifier was missing; and the other attributes specific to `DocumentType` objects set to null and empty lists as appropriate. Associate the `DocumentType` node with the `Document` object so that it is returned as the value of the `doctype` attribute of the `Document` object.

Then, if the DOCTYPE token matches one of the conditions in the following list, then set the document to quirks mode:

- The *force-quirks flag* is set to *on*.
- The name is set to anything other than "`HTML`".
- The public identifier starts with: "`+//Silmaril//dtd html Pro v0r11 19970101//`"
- The public identifier starts with: "`-//AdvaSoft Ltd//DTD HTML 3.0 asWedit + extensions//`"

- The public identifier starts with: "-//AS//DTD HTML 3.0 asWedit + extensions//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Strict Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Strict Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Strict//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0//"
- The public identifier starts with: "-//IETF//DTD HTML 2.1E//"
- The public identifier starts with: "-//IETF//DTD HTML 3.0//"
- The public identifier starts with: "-//IETF//DTD HTML 3.2 Final//"
- The public identifier starts with: "-//IETF//DTD HTML 3.2//"
- The public identifier starts with: "-//IETF//DTD HTML 3//"
- The public identifier starts with: "-//IETF//DTD HTML Level 0//"
- The public identifier starts with: "-//IETF//DTD HTML Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML Level 3//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 0//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 3//"
- The public identifier starts with: "-//IETF//DTD HTML Strict//"
- The public identifier starts with: "-//IETF//DTD HTML//"
- The public identifier starts with: "-//Metrius//DTD Metrius Presentational//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 HTML Strict//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 HTML//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 Tables//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 HTML Strict//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 HTML//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 Tables//"
- The public identifier starts with: "-//Netscape Comm. Corp.//DTD HTML//"
- The public identifier starts with: "-//Netscape Comm. Corp.//DTD Strict HTML//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML 2.0//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML Extended 1.0//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML Extended Relaxed 1.0//"
- The public identifier starts with: "-//SoftQuad Software//DTD HoTMetaL PRO 6.0::19990601::extensions to HTML 4.0//"
- The public identifier starts with: "-//SoftQuad//DTD HoTMetaL PRO 4.0::19971010::extensions to HTML 4.0//"
- The public identifier starts with: "-//Spyglass//DTD HTML 2.0 Extended//"
- The public identifier starts with: "-//SQ//DTD HTML 2.0 HoTMetaL + extensions//"
- The public identifier starts with: "-//Sun Microsystems Corp.//DTD HotJava

HTML//"
- The public identifier starts with: "-//Sun Microsystems Corp.//DTD HotJava Strict HTML//"
- The public identifier starts with: "-//W3C//DTD HTML 3 1995-03-24//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2 Draft//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2 Final//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2S Draft//"
- The public identifier starts with: "-//W3C//DTD HTML 4.0 Frameset//"
- The public identifier starts with: "-//W3C//DTD HTML 4.0 Transitional//"
- The public identifier starts with: "-//W3C//DTD HTML Experimental 19960712//"
- The public identifier starts with: "-//W3C//DTD HTML Experimental 970421//"
- The public identifier starts with: "-//W3C//DTD W3 HTML//"
- The public identifier starts with: "-//W3O//DTD W3 HTML 3.0//"
- The public identifier is set to: "-//W3O//DTD W3 HTML Strict 3.0//EN//"
- The public identifier starts with: "-//WebTechs//DTD Mozilla HTML 2.0//"
- The public identifier starts with: "-//WebTechs//DTD Mozilla HTML//"
- The public identifier is set to: "-/W3C/DTD HTML 4.0 Transitional/EN"
- The public identifier is set to: "HTML"
- The system identifier is set to: "http://www.ibm.com/data/dtd/v11/ibmxhtml1-transitional.dtd"
- The system identifier is missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Frameset//"
- The system identifier is missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Transitional//"

Otherwise, if the DOCTYPE token matches one of the conditions in the following list, then set the document to limited quirks mode:

- The public identifier starts with: "-//W3C//DTD XHTML 1.0 Frameset//"
- The public identifier starts with: "-//W3C//DTD XHTML 1.0 Transitional//"
- The system identifier is not missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Frameset//"
- The system identifier is not missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Transitional//"

The name, system identifier, and public identifier strings must be compared to the values given in the lists above in an ASCII case-insensitive manner. A system identifier whose value is the empty string is not considered missing for the purposes of the conditions above.

Then, switch the insertion mode to "before html".

↪ **Anything else**

Parse error.

Set the document to quirks mode.

Switch the insertion mode to "before html", then reprocess the current token.

8.2.5.5 The "before html" insertion mode

When the insertion mode is "before html", tokens must be handled as follows:

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A comment token**

Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Ignore the token.

↪ **A start tag whose tag name is "html"**

Create an element for the token in the HTML namespace. Append it to the `Document` object. Put this element in the stack of open elements.

If the `Document` is being loaded as part of navigation of a browsing context, then: if the newly created element has a `manifest` attribute, then resolve the value of that attribute to an absolute URL, relative to the newly created element, and if that is successful, run the application cache selection algorithm with the resulting absolute URL; otherwise, if there is no such attribute or resolving it fails, run the application cache selection algorithm with no manifest. The algorithm must be passed the `Document` object.

Switch the insertion mode to "before head".

↪ **Anything else**

Create an `html` element. Append it to the `Document` object. Put this element in the stack of open elements.

If the `Document` is being loaded as part of navigation of a browsing context, then: run the application cache selection algorithm with no manifest, passing it the `Document` object.

Switch the insertion mode to "before head", then reprocess the current token.

Should probably make end tags be ignored, so that "</head><!-- --><html>" puts the comment before the root node (or should we?)

The root element can end up being removed from the `Document` object, e.g. by scripts; nothing in particular happens in such cases, content continues being appended to the nodes as described in the next section.

8.2.5.6 The "before head" insertion mode

When the insertion mode is "before head", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Ignore the token.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data

given in the comment token.

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "head"**
Insert an HTML element for the token.

Set the `head` element pointer to the newly created `head` element.

Switch the insertion mode to "in head".

↪ **An end tag whose tag name is one of: "head", "body", "html", "br"**
Act as if a start tag token with the tag name "head" and no attributes had been seen, then reprocess the current token.

↪ **Any other end tag**
Parse error. Ignore the token.

↪ **Anything else**
Act as if a start tag token with the tag name "head" and no attributes had been seen, then reprocess the current token.

Note: This will result in an empty `head` element being generated, with the current token being reprocessed in the "after head" insertion mode.

8.2.5.7 The "in head" insertion mode

When the insertion mode is "in head", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
Insert the character into the current node.

↪ **A comment token**
Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is one of: "base", "command", "link"**
Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "meta"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

If the element has a `charset` attribute, and its value is a supported encoding, and the confidence is currently *tentative*, then change the encoding to the encoding given by the value of the `charset` attribute.

Otherwise, if the element has a `content` attribute, and applying the algorithm for extracting an encoding from a Content-Type to its value returns a supported encoding *encoding*, and the confidence is currently *tentative*, then change the encoding to the encoding *encoding*.

↪ **A start tag whose tag name is "title"**

Follow the generic RCDATA element parsing algorithm.

↪ **A start tag whose tag name is "noscript", if the scripting flag is enabled**

↪ **A start tag whose tag name is one of: "noframes", "style"**

Follow the generic CDATA element parsing algorithm.

↪ **A start tag whose tag name is "noscript", if the scripting flag is disabled**

Insert an HTML element for the token.

Switch the insertion mode to "in head noscript".

↪ **A start tag whose tag name is "script"**

1. Create an element for the token in the HTML namespace.
2. Mark the element as being "parser-inserted".

   Note: This ensures that, if the script is external, any `document.write()` calls in the script will execute in-line, instead of blowing the document away, as would happen in most other cases. It also prevents the script from executing until the end tag is seen.

3. If the parser was originally created for the HTML fragment parsing algorithm, then mark the `script` element as "already executed". (fragment case)
4. Append the new element to the current node and push it onto the stack of open elements.
5. Switch the tokeniser's content model flag to the CDATA state.
6. Let the original insertion mode be the current insertion mode.
7. Switch the insertion mode to "in CDATA/RCDATA".

↪ **An end tag whose tag name is "head"**
Pop the current node (which will be the `head` element) off the stack of open elements.

Switch the insertion mode to "after head".

↪ **An end tag whose tag name is one of: "body", "html", "br"**
Act as described in the "anything else" entry below.

↪ **A start tag whose tag name is "head"**
↪ **Any other end tag**
Parse error. Ignore the token.

↪ **Anything else**
Act as if an end tag token with the tag name "head" had been seen, and reprocess the current token.

In certain UAs, some elements don't trigger the "in body" mode straight away, but instead get put into the head. Do we want to copy that?

8.2.5.8 The "in head noscript" insertion mode

When the insertion mode is "in head noscript", tokens must be handled as follows:

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **An end tag whose tag name is "noscript"**
Pop the current node (which will be a `noscript` element) from the stack of open elements; the new current node will be a `head` element.

Switch the insertion mode to "in head".

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
↪ **A comment token**
↪ **A start tag whose tag name is one of: "link", "meta", "noframes", "style"**
Process the token using the rules for the "in head" insertion mode.

↪ **An end tag whose tag name is "br"**
Act as described in the "anything else" entry below.

↪ **A start tag whose tag name is one of: "head", "noscript"**
↪ **Any other end tag**
Parse error. Ignore the token.

↪ **Anything else**
Parse error. Act as if an end tag with the tag name "noscript" had been seen and

reprocess the current token.

8.2.5.9 The "after head" insertion mode

When the insertion mode is "after head", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
Insert the character into the current node.

↪ **A comment token**
Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "body"**
Insert an HTML element for the token.

Switch the insertion mode to "in body".

↪ **A start tag whose tag name is "frameset"**
Insert an HTML element for the token.

Switch the insertion mode to "in frameset".

↪ **A start tag token whose tag name is one of: "base", "link", "meta", "noframes", "script", "style", "title"**
Parse error.

Push the node pointed to by the `head` element pointer onto the stack of open elements.

Process the token using the rules for the "in head" insertion mode.

Remove the node pointed to by the `head` element pointer from the stack of open elements.

↪ **An end tag whose tag name is one of: "body", "html", "br"**
Act as described in the "anything else" entry below.

↪ **A start tag whose tag name is "head"**
↪ **Any other end tag**
Parse error. Ignore the token.

↪ **Anything else**
Act as if a start tag token with the tag name "body" and no attributes had been seen, and then reprocess the current token.

8.2.5.10 The "in body" insertion mode

When the insertion mode is "in body", tokens must be handled as follows:

↪ **A character token**

Reconstruct the active formatting elements, if any.

Insert the token's character into the current node.

If the token is not one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE, then set the frameset-ok flag to "not ok".

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Parse error. For each attribute on the token, check to see if the attribute is already present on the top element of the stack of open elements. If it is not, add the attribute and its corresponding value to that element.

↪ **A start tag token whose tag name is one of: "base", "command", "link", "meta", "noframes", "script", "style", "title"**

Process the token using the rules for the "in head" insertion mode.

↪ **A start tag whose tag name is "body"**

Parse error.

If the second element on the stack of open elements is not a `body` element, or, if the stack of open elements has only one node on it, then ignore the token. (fragment case)

Otherwise, for each attribute on the token, check to see if the attribute is already present on the `body` element (the second element) on the stack of open elements. If it is not, add the attribute and its corresponding value to that element.

↪ **A start tag whose tag name is "frameset"**

Parse error.

If the second element on the stack of open elements is not a `body` element, or, if the stack of open elements has only one node on it, then ignore the token. (fragment case)

If the frameset-ok flag is set to "not ok", ignore the token.

Otherwise, run the following steps:

1. Remove the second element on the stack of open elements from its parent node, if it has one.

2. Pop all the nodes from the bottom of the stack of open elements, from the current node up to the root `html` element.

3. Insert an HTML element for the token.

4. Switch the insertion mode to "in frameset".

↪ **An end-of-file token**

If there is a node in the stack of open elements that is not either a `dd` element, a `dt` element, an `li` element, a `p` element, a `tbody` element, a `td` element, a `tfoot` element, a `th` element, a `thead` element, a `tr` element, the `body` element, or the `html` element, then this is a parse error.

Stop parsing.

↪ **An end tag whose tag name is "body"**

If the stack of open elements does not have a `body` element in scope, this is a parse error; ignore the token.

Otherwise, if there is a node in the stack of open elements that is not either a `dd` element, a `dt` element, an `li` element, an `optgroup` element, an `option` element, a `p` element, an `rp` element, an `rt` element, a `tbody` element, a `td` element, a `tfoot` element, a `th` element, a `thead` element, a `tr` element, the `body` element, or the `html` element, then this is a parse error.

Switch the insertion mode to "after body".

↪ **An end tag whose tag name is "html"**

Act as if an end tag with tag name "body" had been seen, then, if that token wasn't ignored, reprocess the current token.

> ***Note: The fake end tag token here can only be ignored in the fragment case.***

↪ **A start tag whose tag name is one of: "address", "article", "aside", "blockquote", "center", "datagrid", "details", "dialog", "dir", "div", "dl", "fieldset", "figure", "footer", "header", "menu", "nav", "ol", "p", "section", "ul"**

If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token.

↪ **A start tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

If the current node is an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then this is a parse error; pop the current node off the stack of open elements.

Insert an HTML element for the token.

↪ **A start tag whose tag name is one of: "pre", "listing"**

If the stack of open elements has a p element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token.

If the next token is a U+000A LINE FEED (LF) character token, then ignore that token and move on to the next one. (Newlines at the start of pre blocks are ignored as an authoring convenience.)

Set the frameset-ok flag to "not ok".

↪ **A start tag whose tag name is "form"**

If the form element pointer is not null, then this is a parse error; ignore the token.

Otherwise:

If the stack of open elements has a p element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token, and set the form element pointer to point to the element created.

↪ **A start tag whose tag name is "li"**

Run the following algorithm:

1. Set the frameset-ok flag to "not ok".
2. Initialize *node* to be the current node (the bottommost node of the stack).
3. If *node* is an li element, then act as if an end tag with the tag name "li" had been seen, then jump to the last step.
4. If *node* is not in the formatting category, and is not in the phrasing category, and is not an address, div, or p element, then jump to the last step.
5. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2.
6. This is the last step.

   If the stack of open elements has a p element in scope, then act as if an end tag with the tag name "p" had been seen.

   Finally, insert an HTML element for the token.

↪ **A start tag whose tag name is one of: "dd", "dt"**

Run the following algorithm:

1. Set the frameset-ok flag to "not ok".
2. Initialize *node* to be the current node (the bottommost node of the stack).
3. If *node* is a dd or dt element, then act as if an end tag with the same tag name as *node* had been seen, then jump to the last step.

4. If *node* is not in the formatting category, and is not in the phrasing category, and is not an `address`, `div`, or `p` element, then jump to the last step.

5. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2.

6. This is the last step.

   If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

   Finally, insert an HTML element for the token.

↪ **A start tag whose tag name is "plaintext"**

If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token.

Switch the content model flag to the PLAINTEXT state.

> ***Note: Once a start tag with the tag name "plaintext" has been seen, that will be the last token ever seen other than character tokens (and the end-of-file token), because there is no way to switch the content model flag out of the PLAINTEXT state.***

↪ **An end tag whose tag name is one of: "address", "article", "aside", "blockquote", "center", "datagrid", "details", "dialog", "dir", "div", "dl", "fieldset", "figure", "footer", "header", "listing", "menu", "nav", "ol", "pre", "section", "ul"**

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.

2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.

3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.

↪ **An end tag whose tag name is "form"**

Let *node* be the element that the `form` element pointer is set to.

Set the `form` element pointer to null.

If *node* is null or the stack of open elements does not have *node* in scope, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.

2. If the current node is not *node*, then this is a parse error.

3. Remove *node* from the stack of open elements.

↪ **An end tag whose tag name is "p"**

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; act as if a start tag with the tag name "p" had been seen, then reprocess the current token.

Otherwise, run these steps:

1. Generate implied end tags, except for elements with the same tag name as the token.

2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.

3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.

↪ **An end tag whose tag name is one of: "dd", "dt", "li"**

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags, except for elements with the same tag name as the token.

2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.

3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.

↪ **An end tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

If the stack of open elements does not have an element in scope whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.

2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.

3. Pop elements from the stack of open elements until an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6" has been popped from the stack.

↪ **An end tag whose tag name is "sarcasm"**

Take a deep breath, then act as described in the "any other end tag" entry below.

↪ **A start tag whose tag name is "a"**

If the list of active formatting elements contains an element whose tag name is "a" between the end of the list and the last marker on the list (or the start of the list if there is no marker on the list), then this is a parse error; act as if an end tag with the tag name "a" had been seen, then remove that element from the list of active formatting elements and the stack of open elements if the end tag didn't already remove it (it might not have if the element is not in table scope).

> In the non-conforming stream `<a href="a">a<table><a href="b">b</table>x`, the first `a` element would be closed upon seeing the second one, and the "x" character would be inside a link to "b", not to "a". This is despite the fact that the outer `a` element is not in table scope (meaning that a regular `</a>` end tag at the start of the table wouldn't close the outer `a` element).

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Add that element to the list of active formatting elements.

↪ **A start tag whose tag name is one of: "b", "big", "code", "em", "font", "i", "s", "small", "strike", "strong", "tt", "u"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Add that element to the list of active formatting elements.

↪ **A start tag whose tag name is "nobr"**

Reconstruct the active formatting elements, if any.

If the stack of open elements has a `nobr` element in scope, then this is a parse error; act as if an end tag with the tag name "nobr" had been seen, then once again reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Add that element to the list of active formatting elements.

↪ **An end tag whose tag name is one of: "a", "b", "big", "code", "em", "font", "i", "nobr", "s", "small", "strike", "strong", "tt", "u"**

Follow these steps:

1. Let the *formatting element* be the last element in the list of active formatting elements that:
   - is between the end of the list and the last scope marker in the list, if any, or the start of the list otherwise, and
   - has the same tag name as the token.

   If there is no such node, or, if that node is also in the stack of open elements but the element is not in scope, then this is a parse error; ignore the token, and abort these steps.

   Otherwise, if there is such a node, but that node is not in the stack of open

elements, then this is a parse error; remove the element from the list, and abort these steps.

Otherwise, there is a *formatting element* and that element is in the stack and is in scope. If the element is not the current node, this is a parse error. In any case, proceed with the algorithm as written in the following steps.

2. Let the *furthest block* be the topmost node in the stack of open elements that is lower in the stack than the *formatting element*, and is not an element in the phrasing or formatting categories. There might not be one.

3. If there is no *furthest block*, then the UA must skip the subsequent steps and instead just pop all the nodes from the bottom of the stack of open elements, from the current node up to and including the *formatting element*, and remove the *formatting element* from the list of active formatting elements.

4. Let the *common ancestor* be the element immediately above the *formatting element* in the stack of open elements.

5. Let a bookmark note the position of the *formatting element* in the list of active formatting elements relative to the elements on either side of it in the list.

6. Let *node* and *last node* be the *furthest block*. Follow these steps:

    1. Let *node* be the element immediately above *node* in the stack of open elements.

    2. If *node* is not in the list of active formatting elements, then remove *node* from the stack of open elements and then go back to step 1.

    3. Otherwise, if *node* is the *formatting element*, then go to the next step in the overall algorithm.

    4. Otherwise, if *last node* is the *furthest block*, then move the aforementioned bookmark to be immediately after the *node* in the list of active formatting elements.

    5. Perform a shallow clone of *node*, replace the entry for *node* in the list of active formatting elements with an entry for the clone, replace the entry for *node* in the stack of open elements with an entry for the clone, and let *node* be the clone.

    6. Insert *last node* into *node*, first removing it from its previous parent node if any.

    7. Let *last node* be *node*.

    8. Return to step 1 of this inner set of steps.

7. If the *common ancestor* node is a `table`, `tbody`, `tfoot`, `thead`, or `tr` element, then, foster parent whatever *last node* ended up being in the previous step, first removing it from its previous parent node if any.

    Otherwise, append whatever *last node* ended up being in the previous step to

the *common ancestor* node, first removing it from its previous parent node if any.

8. Perform a shallow clone of the *formatting element*.

9. Take all of the child nodes of the *furthest block* and append them to the clone created in the last step.

10. Append that clone to the *furthest block*.

11. Remove the *formatting element* from the list of active formatting elements, and insert the clone into the list of active formatting elements at the position of the aforementioned bookmark.

12. Remove the *formatting element* from the stack of open elements, and insert the clone into the stack of open elements immediately below the position of the *furthest block* in that stack.

13. Jump back to step 1 in this series of steps.

Note: The way these steps are defined, only elements in the formatting category ever get cloned by this algorithm.

Note: Because of the way this algorithm causes elements to change parents, it has been dubbed the "adoption agency algorithm" (in contrast with other possibly algorithms for dealing with misnested content, which included the "incest algorithm", the "secret affair algorithm", and the "Heisenberg algorithm").

↪ **A start tag whose tag name is "button"**

If the stack of open elements has a `button` element in scope, then this is a parse error; act as if an end tag with the tag name "button" had been seen, then reprocess the token.

Otherwise:

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Insert a marker at the end of the list of active formatting elements.

Set the frameset-ok flag to "not ok".

↪ **A start tag token whose tag name is one of: "applet", "marquee", "object"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Insert a marker at the end of the list of active formatting elements.

Set the frameset-ok flag to "not ok".

↪ **An end tag token whose tag name is one of: "applet", "button", "marquee", "object"**

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.
2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.
3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.
4. Clear the list of active formatting elements up to the last marker.

↪ **A start tag whose tag name is "table"**

If the stack of open elements has a p element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token.

Set the frameset-ok flag to "not ok".

Switch the insertion mode to "in table".

↪ **A start tag whose tag name is one of: "area", "basefont", "bgsound", "br", "embed", "img", "input", "spacer", "wbr"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

Set the frameset-ok flag to "not ok".

↪ **A start tag whose tag name is one of: "param", "source"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "hr"**

If the stack of open elements has a p element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

Set the frameset-ok flag to "not ok".

↪ **A start tag whose tag name is "image"**

Parse error. Change the token's tag name to "img" and reprocess it. (Don't ask.)

↪ **A start tag whose tag name is "isindex"**

Parse error.

If the `form` element pointer is not null, then ignore the token.

Otherwise:

Acknowledge the token's *self-closing flag*, if it is set.

Act as if a start tag token with the tag name "form" had been seen.

If the token has an attribute called "action", set the `action` attribute on the resulting `form` element to the value of the "action" attribute of the token.

Act as if a start tag token with the tag name "hr" had been seen.

Act as if a start tag token with the tag name "p" had been seen.

Act as if a start tag token with the tag name "label" had been seen.

Act as if a stream of character tokens had been seen (see below for what they should say).

Act as if a start tag token with the tag name "input" had been seen, with all the attributes from the "isindex" token except "name", "action", and "prompt". Set the `name` attribute of the resulting `input` element to the value "`isindex`".

Act as if a stream of character tokens had been seen (see below for what they should say).

Act as if an end tag token with the tag name "label" had been seen.

Act as if an end tag token with the tag name "p" had been seen.

Act as if a start tag token with the tag name "hr" had been seen.

Act as if an end tag token with the tag name "form" had been seen.

If the token has an attribute with the name "prompt", then the first stream of characters must be the same string as given in that attribute, and the second stream of characters must be empty. Otherwise, the two streams of character tokens together should, together with the `input` element, express the equivalent of "This is a searchable index. Insert your search keywords here: (input field)" in the user's preferred language.

↪ **A start tag whose tag name is "textarea"**

1. Insert an HTML element for the token.

2. If the next token is a U+000A LINE FEED (LF) character token, then ignore that token and move on to the next one. (Newlines at the start of `textarea` elements are ignored as an authoring convenience.)

3. Switch the tokeniser's content model flag to the RCDATA state.

4. Let the original insertion mode be the current insertion mode.

5. Set the frameset-ok flag to "not ok".

6. Switch the insertion mode to "in CDATA/RCDATA".

↪ **A start tag whose tag name is "xmp"**

Reconstruct the active formatting elements, if any.

Set the frameset-ok flag to "not ok".

Follow the generic CDATA element parsing algorithm.

↪ **A start tag whose tag name is "iframe"**

Set the frameset-ok flag to "not ok".

Follow the generic CDATA element parsing algorithm.

↪ **A start tag whose tag name is "noembed"**

↪ **A start tag whose tag name is "noscript", if the scripting flag is enabled**

Follow the generic CDATA element parsing algorithm.

↪ **A start tag whose tag name is "select"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Set the frameset-ok flag to "not ok".

If the insertion mode is one of in table", "in caption", "in column group", "in table body", "in row", or "in cell", then switch the insertion mode to "in select in table". Otherwise, switch the insertion mode to "in select".

↪ **A start tag whose tag name is one of: "optgroup", "option"**

If the stack of open elements has an `option` element in scope, then act as if an end tag with the tag name "option" had been seen.

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

↪ **A start tag whose tag name is one of: "rp", "rt"**

If the stack of open elements has a `ruby` element in scope, then generate implied end tags. If the current node is not then a `ruby` element, this is a parse error; pop all the nodes from the current node up to the node immediately before the bottommost `ruby` element on the stack of open elements.

Insert an HTML element for the token.

↪ **An end tag whose tag name is "br"**

Parse error. Act as if a start tag token with the tag name "br" had been seen. Ignore the end tag token.

↪ **A start tag whose tag name is "math"**

Reconstruct the active formatting elements, if any.

Adjust MathML attributes for the token. (This fixes the case of MathML attributes that are not all lowercase.)

Adjust foreign attributes for the token. (This fixes the use of namespaced attributes, in particular XLink.)

Insert a foreign element for the token, in the MathML namespace.

If the token has its *self-closing flag* set, pop the current node off the stack of open elements and acknowledge the token's *self-closing flag*.

Otherwise, let the secondary insertion mode be the current insertion mode, and then switch the insertion mode to "in foreign content".

↪ **A start tag whose tag name is "svg"**

Reconstruct the active formatting elements, if any.

Adjust SVG attributes for the token. (This fixes the case of SVG attributes that are not all lowercase.)

Adjust foreign attributes for the token. (This fixes the use of namespaced attributes, in particular XLink in SVG.)

Insert a foreign element for the token, in the SVG namespace.

If the token has its *self-closing flag* set, pop the current node off the stack of open elements and acknowledge the token's *self-closing flag*.

Otherwise, let the secondary insertion mode be the current insertion mode, and then switch the insertion mode to "in foreign content".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "frame", "head", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↪ **Any other start tag**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Note: This element will be a phrasing element.

↪ **Any other end tag**

Run the following steps:

1. Initialize *node* to be the current node (the bottommost node of the stack).
2. If *node* has the same tag name as the end tag token, then:
    1. Generate implied end tags.

2. If the tag name of the end tag token does not match the tag name of the current node, this is a parse error.

3. Pop all the nodes from the current node up to *node*, including *node*, then stop these steps.

3. Otherwise, if *node* is in neither the formatting category nor the phrasing category, then this is a parse error; ignore the token, and abort these steps.

4. Set *node* to the previous entry in the stack of open elements.

5. Return to step 2.

8.2.5.11 The "in CDATA/RCDATA" insertion mode

When the insertion mode is "in CDATA/RCDATA", tokens must be handled as follows:

↪ **A character token**

Insert the token's character into the current node.

↪ **An end-of-file token**

Parse error.

If the current node is a `script` element, mark the `script` element as "already executed".

Pop the current node off the stack of open elements.

Switch the insertion mode to the original insertion mode and reprocess the current token.

↪ **An end tag whose tag name is "script"**

Let *script* be the current node (which will be a `script` element).

Pop the current node off the stack of open elements.

Switch the insertion mode to the original insertion mode.

Let the *old insertion point* have the same value as the current insertion point. Let the insertion point be just before the next input character.

Increment the parser's script nesting level by one.

Run the *script*. This might cause some script to execute, which might cause new characters to be inserted into the tokeniser, and might cause the tokeniser to output more tokens, resulting in a reentrant invocation of the parser.

Decrement the parser's script nesting level by one. If the parser's script nesting level is zero, then set the parser pause flag to false.

Let the insertion point have the value of the *old insertion point*. (In other words, restore the insertion point to its previous value. This value might be the "undefined" value.)

At this stage, if there is a pending external script, then:

↪ **If the tree construction stage is being called reentrantly, say from a call to `document.write()`:**
Set the parser pause flag to true, and abort the processing of any nested invocations of the tokeniser, yielding control back to the caller. (Tokenization will resume when the caller returns to the "outer" tree construction stage.)

↪ **Otherwise:**
Follow these steps:

1. Let *the script* be the pending external script. There is no longer a pending external script.
2. Pause until the script has completed loading.
3. Let the insertion point be just before the next input character.
4. Execute the script.
5. Let the insertion point be undefined again.
6. If there is once again a pending external script, then repeat these steps from step 1.

↪ **Any other end tag**
Pop the current node off the stack of open elements.

Switch the insertion mode to the original insertion mode.

8.2.5.12 The "in table" insertion mode

When the insertion mode is "in table", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
If the current table is tainted, then act as described in the "anything else" entry below.

Otherwise, insert the character into the current node.

↪ **A comment token**
Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "caption"**
Clear the stack back to a table context. (See below.)

Insert a marker at the end of the list of active formatting elements.

Insert an HTML element for the token, then switch the insertion mode to "in caption".

↪ **A start tag whose tag name is "colgroup"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in column group".

↪ **A start tag whose tag name is "col"**

Act as if a start tag token with the tag name "colgroup" had been seen, then reprocess the current token.

↪ **A start tag whose tag name is one of: "tbody", "tfoot", "thead"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in table body".

↪ **A start tag whose tag name is one of: "td", "th", "tr"**

Act as if a start tag token with the tag name "tbody" had been seen, then reprocess the current token.

↪ **A start tag whose tag name is "table"**

Parse error. Act as if an end tag token with the tag name "table" had been seen, then, if that token wasn't ignored, reprocess the current token.

Note: The fake end tag token here can only be ignored in the fragment case.

↪ **An end tag whose tag name is "table"**

If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Pop elements from this stack until a `table` element has been popped from the stack.

Reset the insertion mode appropriately.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↪ **A start tag whose tag name is one of: "style", "script"**

If the current table is tainted then act as described in the "anything else" entry below.

Otherwise, process the token using the rules for the "in head" insertion mode.

↪ **A start tag whose tag name is "input"**

If the token does not have an attribute with the name "type", or if it does, but that attribute's value is not an ASCII case-insensitive match for the string "`hidden`", or, if the current table is tainted, then: act as described in the "anything else" entry below.

Otherwise:

Parse error.

Insert an HTML element for the token.

Pop that `input` element off the stack of open elements.

↪ **An end-of-file token**

If the current node is not the root `html` element, then this is a parse error.

Note: It can only be the current node in the fragment case.

Stop parsing.

↪ **Anything else**

Parse error. Process the token using the rules for the "in body" insertion mode, except that if the current node is a `table`, `tbody`, `tfoot`, `thead`, or `tr` element, then, whenever a node would be inserted into the current node, it must instead be foster parented.

When the steps above require the UA to **clear the stack back to a table context**, it means that the UA must, while the current node is not a `table` element or an `html` element, pop elements from the stack of open elements.

Note: The current node being an `html` element after this process is a fragment case.

8.2.5.13 The "in caption" insertion mode

When the insertion mode is "in caption", tokens must be handled as follows:

↪ **An end tag whose tag name is "caption"**

If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Generate implied end tags.

Now, if the current node is not a `caption` element, then this is a parse error.

Pop elements from this stack until a `caption` element has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in table".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

↪ **An end tag whose tag name is "table"**

Parse error. Act as if an end tag with the tag name "caption" had been seen, then, if that token wasn't ignored, reprocess the current token.

Note: The fake end tag token here can only be ignored in the fragment case.

↪ **An end tag whose tag name is one of: "body", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↪ **Anything else**

Process the token using the rules for the "in body" insertion mode.

8.2.5.14 The "in column group" insertion mode

When the insertion mode is "in column group", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Insert the character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "col"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **An end tag whose tag name is "colgroup"**

If the current node is the root `html` element, then this is a parse error; ignore the token. (fragment case)

Otherwise, pop the current node (which will be a `colgroup` element) from the stack of open elements. Switch the insertion mode to "in table".

↪ **An end tag whose tag name is "col"**

Parse error. Ignore the token.

↪ **An end-of-file token**

If the current node is the root `html` element, then stop parsing. (fragment case)

Otherwise, act as described in the "anything else" entry below.

↪ **Anything else**

Act as if an end tag with the tag name "colgroup" had been seen, and then, if that token wasn't ignored, reprocess the current token.

Note: The fake end tag token here can only be ignored in the fragment case.

8.2.5.15 The "in table body" insertion mode

When the insertion mode is "in table body", tokens must be handled as follows:

↪ **A start tag whose tag name is "tr"**

Clear the stack back to a table body context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in row".

↪ **A start tag whose tag name is one of: "th", "td"**

Parse error. Act as if a start tag with the tag name "tr" had been seen, then reprocess the current token.

↪ **An end tag whose tag name is one of: "tbody", "tfoot", "thead"**

If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token.

Otherwise:

Clear the stack back to a table body context. (See below.)

Pop the current node from the stack of open elements. Switch the insertion mode to "in table".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "tfoot", "thead"**

↪ **An end tag whose tag name is "table"**

If the stack of open elements does not have a `tbody`, `thead`, or `tfoot` element in table scope, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Clear the stack back to a table body context. (See below.)

Act as if an end tag with the same tag name as the current node ("tbody", "tfoot", or "thead") had been seen, then reprocess the current token.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html",**

"td", "th", "tr"
Parse error. Ignore the token.

↪ **Anything else**
Process the token using the rules for the "in table" insertion mode.

When the steps above require the UA to **clear the stack back to a table body context**, it means that the UA must, while the current node is not a `tbody`, `tfoot`, `thead`, or `html` element, pop elements from the stack of open elements.

Note: The current node being an `html` element after this process is a fragment case.

8.2.5.16 The "in row" insertion mode

When the insertion mode is "in row", tokens must be handled as follows:

↪ **A start tag whose tag name is one of: "th", "td"**
Clear the stack back to a table row context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in cell".

Insert a marker at the end of the list of active formatting elements.

↪ **An end tag whose tag name is "tr"**
If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Clear the stack back to a table row context. (See below.)

Pop the current node (which will be a `tr` element) from the stack of open elements. Switch the insertion mode to "in table body".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "tfoot", "thead", "tr"**
↪ **An end tag whose tag name is "table"**
Act as if an end tag with the tag name "tr" had been seen, then, if that token wasn't ignored, reprocess the current token.

Note: The fake end tag token here can only be ignored in the fragment case.

↪ **An end tag whose tag name is one of: "tbody", "tfoot", "thead"**
If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token.

Otherwise, act as if an end tag with the tag name "tr" had been seen, then reprocess the current token.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "td", "th"**

Parse error. Ignore the token.

↪ **Anything else**

Process the token using the rules for the "in table" insertion mode.

When the steps above require the UA to **clear the stack back to a table row context**, it means that the UA must, while the current node is not a `tr` element or an `html` element, pop elements from the stack of open elements.

Note: The current node being an `html` element after this process is a fragment case.

8.2.5.17 The "in cell" insertion mode

When the insertion mode is "in cell", tokens must be handled as follows:

↪ **An end tag whose tag name is one of: "td", "th"**

If the stack of open elements does not have an element in table scope with the same tag name as that of the token, then this is a parse error and the token must be ignored.

Otherwise:

Generate implied end tags.

Now, if the current node is not an element with the same tag name as the token, then this is a parse error.

Pop elements from this stack until an element with the same tag name as the token has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in row". (The current node will be a `tr` element at this point.)

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

If the stack of open elements does *not* have a `td` or `th` element in table scope, then this is a parse error; ignore the token. (fragment case)

Otherwise, close the cell (see below) and reprocess the current token.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html"**

Parse error. Ignore the token.

↪ **An end tag whose tag name is one of: "table", "tbody", "tfoot", "thead", "tr"**

If the stack of open elements does not have an element in table scope with the same tag name as that of the token (which can only happen for "tbody", "tfoot" and "thead", or, in the fragment case), then this is a parse error and the token must be

ignored.

Otherwise, close the cell (see below) and reprocess the current token.

↪ **Anything else**
Process the token using the rules for the "in body" insertion mode.

Where the steps above say to **close the cell**, they mean to run the following algorithm:

1. If the stack of open elements has a `td` element in table scope, then act as if an end tag token with the tag name "td" had been seen.
2. Otherwise, the stack of open elements will have a `th` element in table scope; act as if an end tag token with the tag name "th" had been seen.

Note: The stack of open elements cannot have both a `td` and a `th` element in table scope at the same time, nor can it have neither when the insertion mode is "in cell".

8.2.5.18 The "in select" insertion mode

When the insertion mode is "in select", tokens must be handled as follows:

↪ **A character token**
Insert the token's character into the current node.

↪ **A comment token**
Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "option"**
If the current node is an `option` element, act as if an end tag with the tag name "option" had been seen.

Insert an HTML element for the token.

↪ **A start tag whose tag name is "optgroup"**
If the current node is an `option` element, act as if an end tag with the tag name "option" had been seen.

If the current node is an `optgroup` element, act as if an end tag with the tag name "optgroup" had been seen.

Insert an HTML element for the token.

↪ **An end tag whose tag name is "optgroup"**

First, if the current node is an `option` element, and the node immediately before it in the stack of open elements is an `optgroup` element, then act as if an end tag with the tag name "option" had been seen.

If the current node is an `optgroup` element, then pop that node from the stack of open elements. Otherwise, this is a parse error; ignore the token.

↪ **An end tag whose tag name is "option"**
If the current node is an `option` element, then pop that node from the stack of open elements. Otherwise, this is a parse error; ignore the token.

↪ **An end tag whose tag name is "select"**
If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Pop elements from the stack of open elements until a `select` element has been popped from the stack.

Reset the insertion mode appropriately.

↪ **A start tag whose tag name is "select"**
Parse error. Act as if the token had been an end tag with the tag name "select" instead.

↪ **A start tag whose tag name is one of: "input", "textarea"**
Parse error. Act as if an end tag with the tag name "select" had been seen, and reprocess the token.

↪ **A start tag token whose tag name is "script"**
Process the token using the rules for the "in head" insertion mode.

↪ **An end-of-file token**
If the current node is not the root `html` element, then this is a parse error.

Note: It can only be the current node in the fragment case.

Stop parsing.

↪ **Anything else**
Parse error. Ignore the token.

8.2.5.19 The "in select in table" insertion mode

When the insertion mode is "in select in table", tokens must be handled as follows:

↪ **A start tag whose tag name is one of: "caption", "table", "tbody", "tfoot", "thead", "tr", "td", "th"**
Parse error. Act as if an end tag with the tag name "select" had been seen, and reprocess the token.

↪ **An end tag whose tag name is one of: "caption", "table", "tbody", "tfoot", "thead", "tr", "td", "th"**

Parse error.

If the stack of open elements has an element in table scope with the same tag name as that of the token, then act as if an end tag with the tag name "select" had been seen, and reprocess the token. Otherwise, ignore the token.

↪ **Anything else**

Process the token using the rules for the "in select" insertion mode.

8.2.5.20 The "in foreign content" insertion mode

When the insertion mode is "in foreign content", tokens must be handled as follows:

↪ **A character token**

Insert the token's character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **An end tag whose tag name is "script", if the current node is a `script` element in the SVG namespace.**

Pop the current node off the stack of open elements.

Let the *old insertion point* have the same value as the current insertion point. Let the insertion point be just before the next input character.

Increment the parser's script nesting level by one. Set the parser pause flag to true.

Process the `script` element according to the SVG rules. [SVG]

Note: Even if this causes new characters to be inserted into the tokeniser, the parser will not be executed reentrantly, since the parser pause flag is true.

Decrement the parser's script nesting level by one. If the parser's script nesting level is zero, then set the parser pause flag to false.

Let the insertion point have the value of the *old insertion point*. (In other words, restore the insertion point to its previous value. This value might be the "undefined" value.)

↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `mi` element in the MathML namespace.**

↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `mo` element in the MathML namespace.**

↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `mn` element in the MathML namespace.**
↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `ms` element in the MathML namespace.**
↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `mtext` element in the MathML namespace.**
↪ **A start tag whose tag name is "svg", if the current node is an `annotation-xml` element in the MathML namespace.**
↪ **A start tag, if the current node is a `foreignObject` element in the SVG namespace.**
↪ **A start tag, if the current node is a `desc` element in the SVG namespace.**
↪ **A start tag, if the current node is a `title` element in the SVG namespace.**
↪ **A start tag, if the current node is an element in the HTML namespace.**
↪ **An end tag**

Process the token using the rules for the secondary insertion mode.

If, after doing so, the insertion mode is still "in foreign content", but there is no element in scope that has a namespace other than the HTML namespace, switch the insertion mode to the secondary insertion mode.

↪ **A start tag whose tag name is one of: "b", "big", "blockquote", "body", "br", "center", "code", "dd", "div", "dl", "dt", "em", "embed", "h1", "h2", "h3", "h4", "h5", "h6", "head", "hr", "i", "img", "li", "listing", "menu", "meta", "nobr", "ol", "p", "pre", "ruby", "s", "small", "span", "strong", "strike", "sub", "sup", "table", "tt", "u", "ul", "var"**
↪ **A start tag whose tag name is "font", if the token has any attributes named "color", "face", or "size"**
↪ **An end-of-file token**

Parse error.

Pop elements from the stack of open elements until the current node is in the HTML namespace.

Switch the insertion mode to the secondary insertion mode, and reprocess the token.

↪ **Any other start tag**

If the current node is an element in the MathML namespace, adjust MathML attributes for the token. (This fixes the case of MathML attributes that are not all lowercase.)

If the current node is an element in the SVG namespace, and the token's tag name is one of the ones in the first column of the following table, change the tag name to the name given in the corresponding cell in the second column. (This fixes the case of SVG elements that are not all lowercase.)

Tag name	Element name
`altglyph`	`altGlyph`
`altglyphdef`	`altGlyphDef`
`altglyphitem`	`altGlyphItem`
`animatecolor`	`animateColor`

Tag name	Element name
animatemotion	animateMotion
animatetransform	animateTransform
clippath	clipPath
feblend	feBlend
fecolormatrix	feColorMatrix
fecomponenttransfer	feComponentTransfer
fecomposite	feComposite
feconvolvematrix	feConvolveMatrix
fediffuselighting	feDiffuseLighting
fedisplacementmap	feDisplacementMap
fedistantlight	feDistantLight
feflood	feFlood
fefunca	feFuncA
fefuncb	feFuncB
fefuncg	feFuncG
fefuncr	feFuncR
fegaussianblur	feGaussianBlur
feimage	feImage
femerge	feMerge
femergenode	feMergeNode
femorphology	feMorphology
feoffset	feOffset
fepointlight	fePointLight
fespecularlighting	feSpecularLighting
fespotlight	feSpotLight
fetile	feTile
feturbulence	feTurbulence
foreignobject	foreignObject
glyphref	glyphRef
lineargradient	linearGradient
radialgradient	radialGradient
textpath	textPath

If the current node is an element in the SVG namespace, adjust SVG attributes for the token. (This fixes the case of SVG attributes that are not all lowercase.)

Adjust foreign attributes for the token. (This fixes the use of namespaced attributes, in particular XLink in SVG.)

Insert a foreign element for the token, in the same namespace as the current node.

If the token has its *self-closing flag* set, pop the current node off the stack of open elements and acknowledge the token's *self-closing flag*.

8.2.5.21 The "after body" insertion mode

When the insertion mode is "after body", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
Process the token using the rules for the "in body" insertion mode.

↪ **A comment token**
Append a `Comment` node to the first element in the stack of open elements (the `html` element), with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **An end tag whose tag name is "html"**
If the parser was originally created as part of the HTML fragment parsing algorithm, this is a parse error; ignore the token. (fragment case)

Otherwise, switch the insertion mode to "after after body".

↪ **An end-of-file token**
Stop parsing.

↪ **Anything else**
Parse error. Switch the insertion mode to "in body" and reprocess the token.

8.2.5.22 The "in frameset" insertion mode

When the insertion mode is "in frameset", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
Insert the character into the current node.

↪ **A comment token**
Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "frameset"**
Insert an HTML element for the token.

↪ **An end tag whose tag name is "frameset"**
If the current node is the root `html` element, then this is a parse error; ignore the token. (fragment case)

Otherwise, pop the current node from the stack of open elements.

If the parser was *not* originally created as part of the HTML fragment parsing algorithm (fragment case), and the current node is no longer a `frameset` element, then switch the insertion mode to "after frameset".

↪ **A start tag whose tag name is "frame"**
Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "noframes"**
Process the token using the rules for the "in head" insertion mode.

↪ **An end-of-file token**
If the current node is not the root `html` element, then this is a parse error.

Note: It can only be the current node in the fragment case.

Stop parsing.

↪ **Anything else**
Parse error. Ignore the token.

8.2.5.23 The "after frameset" insertion mode

When the insertion mode is "after frameset", tokens must be handled as follows:

↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
Insert the character into the current node.

↪ **A comment token**
Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **An end tag whose tag name is "html"**
Switch the insertion mode to "after after frameset".

↪ **A start tag whose tag name is "noframes"**
Process the token using the rules for the "in head" insertion mode.

↪ **An end-of-file token**
Stop parsing.

↪ **Anything else**
Parse error. Ignore the token.

This doesn't handle UAs that don't support frames, or that do support frames but want to show the NOFRAMES content. Supporting the former is easy; supporting the latter is harder.

8.2.5.24 The "after after body" insertion mode

When the insertion mode is "after after body", tokens must be handled as follows:

↪ **A comment token**
Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **An end-of-file token**
Stop parsing.

↪ **Anything else**
Parse error. Switch the insertion mode to "in body" and reprocess the token.

8.2.5.25 The "after after frameset" insertion mode

When the insertion mode is "after after frameset", tokens must be handled as follows:

↪ **A comment token**
Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**
↪ **A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**
↪ **A start tag whose tag name is "html"**
Process the token using the rules for the "in body" insertion mode.

↪ **An end-of-file token**
Stop parsing.

↪ **A start tag whose tag name is "noframes"**
Process the token using the rules for the "in head" insertion mode.

↪ **Anything else**
Parse error. Ignore the token.

8.2.6 The end

Once the user agent **stops parsing** the document, the user agent must follow the steps in this section.

First, the current document readiness must be set to "interactive".

Then, the rules for when a script completes loading start applying (script execution is no longer managed by the parser).

If any of the scripts in the list of scripts that will execute as soon as possible have completed loading, or if the list of scripts that will execute asynchronously is not empty and the first script in that list has completed loading, then the user agent must act as if those scripts just completed loading, following the rules given for that in the `script` element definition.

Then, if the list of scripts that will execute when the document has finished parsing is not empty, and the first item in this list has already completed loading, then the user agent must act as if that script just finished loading.

By this point, there will be no scripts that have loaded but have not yet been executed.

The user agent must then fire a simple event called `DOMContentLoaded` at the `Document`.

Once everything that **delays the load event** has completed, the user agent must run the following steps:

1. Queue a task to set the current document readiness to "complete".

2. If the `Document` is in a browsing context, then queue a task to fire a `load` event at the `Document`'s `Window` object.

3. If the `Document` has a pending state object, then queue a task to fire a `popstate` event in no namespace on the `Document`'s `Window` object using the `PopStateEvent` interface, with the `state` attribute set to the current value of the pending state object. This event must bubble but not be cancelable and has no default action.

The task source for these tasks is the DOM manipulation task source.

delaying the load event for things like image loads allows for intranet port scans (even without javascript!). Should we really encode that into the spec?

8.2.7 Coercing an HTML DOM into an infoset

When an application uses an HTML parser in conjunction with an XML pipeline, it is possible that the constructed DOM is not compatible with the XML tool chain in certain subtle ways. For example, an XML toolchain might not be able to represent attributes with the name `xmlns`, since they conflict with the Namespaces in XML syntax. There is also some data that the HTML parser generates that isn't included in the DOM itself. This section specifies some rules for handling these issues.

If the XML API being used doesn't support DOCTYPEs, the tool may drop DOCTYPEs altogether.

If the XML API doesn't support attributes in no namespace that are named "`xmlns`", attributes whose names start with "`xmlns:`", or attributes in the XMLNS namespace, then the tool may drop such attributes.

The tool may annotate the output with any namespace declarations required for proper operation.

If the XML API being used restricts the allowable characters in the local names of elements and attributes, then the tool may map all element and attribute local names that the API wouldn't support to a set of names that *are* allowed, by replacing any character that isn't supported with the uppercase letter U and the six digits of the character's Unicode codepoint when expressed in hexadecimal, using digits 0-9 and capital letters A-F as the symbols, in increasing numeric order.

> For example, the element name `foo<bar`, which can be output by the HTML parser, though it is neither a legal HTML element name nor a well-formed XML element name, would be converted into `fooU00003Cbar`, which *is* a well-formed XML element name (though it's still not legal in HTML by any means).

> As another example, consider the attribute `xlink:href`. Used on a MathML element, it becomes, after being adjusted, an attribute with a prefix "`xlink`" and a local name "`href`". However, used on an HTML element, it becomes an attribute with no prefix and the local name "`xlink:href`", which is not a valid NCName, and thus might not be accepted by an XML API. It could thus get converted, becoming "`xlinkU00003Ahref`".

Note: The resulting names from this conversion conveniently can't clash with any attribute generated by the HTML parser, since those are all either lowercase or those listed in the adjust foreign attributes algorithm's table.

If the XML API restricts comments from having two consecutive U+002D HYPHEN-MINUS characters (--), the tool may insert a single U+0020 SPACE character between any such offending characters.

If the XML API restricts comments from ending in a U+002D HYPHEN-MINUS character (-), the tool may insert a single U+0020 SPACE character at the end of such comments.

If the XML API restricts allowed characters in character data, the tool may replace any U+000C FORM FEED (FF) character with a U+0020 SPACE character, and any other literal non-XML character with a U+FFFD REPLACEMENT CHARACTER.

If the tool has no way to convey out-of-band information, then the tool may drop the following information:

- Whether the document is set to *no quirks mode*, *limited quirks mode*, or *quirks mode*
- The association between form controls and forms that aren't their nearest `form` element ancestor (use of the `form` element pointer in the parser)

Note: The mutations allowed by this section apply after the HTML parser's rules have been applied. For example, a `<a::>` start tag will be closed by a `</a::>` end tag, and never by a `</aU00003AU00003A>` end tag, even if the user agent is using the rules above to then generate an actual element in the DOM with the name `aU00003AU00003A` for that start tag.

8.3 Namespaces

The **HTML namespace** is: `http://www.w3.org/1999/xhtml`

The **MathML namespace** is: `http://www.w3.org/1998/Math/MathML`

The **SVG namespace** is: `http://www.w3.org/2000/svg`

The **XLink namespace** is: `http://www.w3.org/1999/xlink`

The **XML namespace** is: `http://www.w3.org/XML/1998/namespace`

The **XMLNS namespace** is: `http://www.w3.org/2000/xmlns/`

Data mining tools and other user agents that perform operations on `text/html` content without running scripts, evaluating CSS or XPath expressions, or otherwise exposing the resulting DOM to arbitrary content, may "support namespaces" by just asserting that their DOM node analogues are in certain namespaces, without actually exposing the above strings.

8.4 Serializing HTML fragments

The following steps form the **HTML fragment serialization algorithm**. The algorithm takes as input a DOM `Element` or `Document`, referred to as *the node*, and either returns a string or raises an exception.

Note: This algorithm serializes the children of the node being serialized, not the node itself.

1. Let *s* be a string, and initialize it to the empty string.
2. For each child node of *the node*, in tree order, run the following steps:
    1. Let *current node* be the child node being processed.
    2. Append the appropriate string from the following list to *s*:

↪ **If *current node* is an `Element`**

Append a U+003C LESS-THAN SIGN (<) character, followed by the element's tag name. (For nodes created by the HTML parser, `Document.createElement()`, or `Document.renameNode()`, the tag name will be lowercase.)

For each attribute that the element has, append a U+0020 SPACE character, the attribute's name (which, for attributes set by the HTML parser or by `Element.setAttributeNode()` or `Element.setAttribute()`, will be lowercase), a U+003D EQUALS SIGN (=) character, a U+0022 QUOTATION MARK (") character, the attribute's value, escaped as described below in *attribute mode*, and a second U+0022 QUOTATION MARK (") character.

While the exact order of attributes is UA-defined, and may depend on factors such as the order that the attributes were given in the original markup, the sort order must be stable, such that consecutive invocations of this algorithm serialize an element's attributes in the same order.

Append a U+003E GREATER-THAN SIGN (>) character.

If *current node* is an `area`, `base`, `basefont`, `bgsound`, `br`, `col`, `embed`, `frame`, `hr`, `img`, `input`, `link`, `meta`, `param`, `spacer`, or `wbr` element, then continue on to the next child node at this point.

If *current node* is a `pre`, `textarea`, or `listing` element, append a U+000A LINE FEED (LF) character.

Append the value of running the HTML fragment serialization algorithm on the *current node* element (thus recursing into this algorithm for that element), followed by a U+003C LESS-THAN SIGN (<) character, a U+002F SOLIDUS (/) character, the element's tag name again, and finally a U+003E GREATER-THAN SIGN (>) character.

↪ **If *current node* is a `Text` or `CDATASection` node**

If one of the ancestors of *current node* is a `style`, `script`, `xmp`, `iframe`, `noembed`, `noframes`, `noscript`, or `plaintext` element, then append the value of *current node*'s `data` DOM attribute literally.

Otherwise, append the value of *current node*'s `data` DOM attribute, escaped as described below.

↪ **If *current node* is a `Comment`**

Append the literal string `<!--` (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS), followed by the value of *current node*'s `data` DOM attribute, followed by the literal string `-->` (U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN).

↪ **If *current node* is a `ProcessingInstruction`**

Append the literal string `<?` (U+003C LESS-THAN SIGN, U+003F QUESTION MARK), followed by the value of *current node*'s `target`

DOM attribute, followed by a single U+0020 SPACE character, followed by the value of *current node*'s `data` DOM attribute, followed by a single U+003E GREATER-THAN SIGN character ('>').

↪ **If *current node* is a `DocumentType`**

Append the literal string `<!DOCTYPE` (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+0044 LATIN CAPITAL LETTER D, U+004F LATIN CAPITAL LETTER O, U+0043 LATIN CAPITAL LETTER C, U+0054 LATIN CAPITAL LETTER T, U+0059 LATIN CAPITAL LETTER Y, U+0050 LATIN CAPITAL LETTER P, U+0045 LATIN CAPITAL LETTER E), followed by a space (U+0020 SPACE), followed by the value of *current node*'s `name` DOM attribute, followed by the literal string `>` (U+003E GREATER-THAN SIGN).

Other node types (e.g. `Attr`) cannot occur as children of elements. If, despite this, they somehow do occur, this algorithm must raise an `INVALID_STATE_ERR` exception.

3. The result of the algorithm is the string *s*.

Escaping a string (for the purposes of the algorithm above) consists of replacing any occurrences of the "`&`" character by the string "`&`", any occurrences of the U+00A0 NO-BREAK SPACE character by the string "` `", and, if the algorithm was invoked in the *attribute mode*, any occurrences of the "`"`" character by the string "`"`", or if it was not, any occurrences of the "`<`" character by the string "`<`", any occurrences of the "`>`" character by the string "`>`".

Note: Entity reference nodes are assumed to be expanded by the user agent, and are therefore not covered in the algorithm above.

Note: It is possible that the output of this algorithm, if parsed with an HTML parser, will not return the original tree structure. For instance, if a `textarea` element to which a `Comment` node has been appended is serialized and the output is then reparsed, the comment will end up being displayed in the text field. Similarly, if, as a result of DOM manipulation, an element contains a comment that contains the literal string "`-->`", then when the result of serializing the element is parsed, the comment will be truncated at that point and the rest of the comment will be interpreted as markup. More examples would be making a `script` element contain a text node with the text string "`</script>`", or having a `p` element that contains a `ul` element (as the `ul` element's start tag would imply the end tag for the `p`).

8.5 Parsing HTML fragments

The following steps form the **HTML fragment parsing algorithm**. The algorithm optionally takes as input an `Element` node, referred to as the *context* element, which gives the context for the parser, as well as *input*, a string to parse, and returns a list of zero or more nodes.

Note: Parts marked* fragment case *in algorithms in the parser section are parts that only occur if the parser was created for the purposes of this algorithm (and with a context element). The algorithms have been annotated with such markings for informational purposes only; such markings have no normative weight. If it is possible for a condition described as a fragment case to occur even when the parser wasn't created for the purposes of handling this algorithm, then that is an error in the specification.

1. Create a new `Document` node, and mark it as being an HTML document.

2. Create a new HTML parser, and associate it with the just created `Document` node.

3. If there is a *context* element, run these substeps:

    1. Set the HTML parser's tokenization stage's content model flag according to the *context* element, as follows:

        ↪ **If it is a `title` or `textarea` element**
        Set the content model flag to the RCDATA state.

        ↪ **If it is a `style`, `script`, `xmp`, `iframe`, `noembed`, or `noframes` element**
        Set the content model flag to the CDATA state.

        ↪ **If it is a `noscript` element**
        If the scripting flag is enabled, set the content model flag to the CDATA state. Otherwise, set the content model flag to the PCDATA state.

        ↪ **If it is a `plaintext` element**
        Set the content model flag to PLAINTEXT.

        ↪ **Otherwise**
        Leave the content model flag in the PCDATA state.

    2. Let *root* be a new `html` element with no attributes.

    3. Append the element *root* to the `Document` node created above.

    4. Set up the parser's stack of open elements so that it contains just the single element *root*.

    5. Reset the parser's insertion mode appropriately.

        Note: The parser will reference the context element as part of that algorithm.

    6. Set the parser's `form` element pointer to the nearest node to the *context* element that is a `form` element (going straight up the ancestor chain, and including the element itself, if it is a `form` element), or, if there is no such `form` element, to null.

4. Place into the input stream for the HTML parser just created the *input*. The encoding confidence is *irrelevant*.

5. Start the parser and let it run until it has consumed all the characters just inserted into the input stream.

6. If there is a *context* element, return the child nodes of *root*, in tree order.

Otherwise, return the children of the `Document` object, in tree order.

8.6 Named character references

This table lists the character reference names that are supported by HTML, and the code points to which they refer. It is referenced by the previous sections.

Name	Character	Name	Character
AElig;	U+000C6	Cconint;	U+02230
AElig	U+000C6	Cdot;	U+0010A
AMP;	U+00026	Cedilla;	U+000B8
AMP	U+00026	CenterDot;	U+000B7
Aacute;	U+000C1	Cfr;	U+0212D
Aacute	U+000C1	Chi;	U+003A7
Abreve;	U+00102	CircleDot;	U+02299
Acirc;	U+000C2	CircleMinus;	U+02296
Acirc	U+000C2	CirclePlus;	U+02295
Acy;	U+00410	CircleTimes;	U+02297
Afr;	U+1D504	ClockwiseContourIntegral;	U+02232
Agrave;	U+000C0	CloseCurlyDoubleQuote;	U+0201D
Agrave	U+000C0	CloseCurlyQuote;	U+02019
Alpha;	U+00391	Colon;	U+02237
Amacr;	U+00100	Colone;	U+02A74
And;	U+02A53	Congruent;	U+02261
Aogon;	U+00104	Conint;	U+0222F
Aopf;	U+1D538	ContourIntegral;	U+0222E
ApplyFunction;	U+02061	Copf;	U+02102
Aring;	U+000C5	Coproduct;	U+02210
Aring	U+000C5	CounterClockwiseContourIntegral;	U+02233
Ascr;	U+1D49C	Cross;	U+02A2F
Assign;	U+02254	Cscr;	U+1D49E
Atilde;	U+000C3	Cup;	U+022D3
Atilde	U+000C3	CupCap;	U+0224D
Auml;	U+000C4	DD;	U+02145
Auml	U+000C4	DDotrahd;	U+02911
Backslash;	U+02216	DJcy;	U+00402
Barv;	U+02AE7	DScy;	U+00405
Barwed;	U+02306	DZcy;	U+0040F
Bcy;	U+00411	Dagger;	U+02021
Because;	U+02235	Darr;	U+021A1
Bernoullis;	U+0212C	Dashv;	U+02AE4
Beta;	U+00392	Dcaron;	U+0010E
Bfr;	U+1D505	Dcy;	U+00414
Bopf;	U+1D539	Del;	U+02207
Breve;	U+002D8	Delta;	U+00394
Bscr;	U+0212C	Dfr;	U+1D507
Bumpeq;	U+0224E	DiacriticalAcute;	U+000B4
CHcy;	U+00427	DiacriticalDot;	U+002D9
COPY;	U+000A9	DiacriticalDoubleAcute;	U+002DD
COPY	U+000A9	DiacriticalGrave;	U+00060
Cacute;	U+00106	DiacriticalTilde;	U+002DC
Cap;	U+022D2	Diamond;	U+022C4
CapitalDifferentialD;	U+02145	DifferentialD;	U+02146
Cayleys;	U+0212D	Dopf;	U+1D53B
Ccaron;	U+0010C	Dot;	U+000A8
Ccedil;	U+000C7	DotDot;	U+020DC
Ccedil	U+000C7	DotEqual;	U+02250
Ccirc;	U+00108	DoubleContourIntegral;	U+0222F

Name	Character
DoubleDot;	U+000A8
DoubleDownArrow;	U+021D3
DoubleLeftArrow;	U+021D0
DoubleLeftRightArrow;	U+021D4
DoubleLeftTee;	U+02AE4
DoubleLongLeftArrow;	U+027F8
DoubleLongLeftRightArrow;	U+027FA
DoubleLongRightArrow;	U+027F9
DoubleRightArrow;	U+021D2
DoubleRightTee;	U+022A8
DoubleUpArrow;	U+021D1
DoubleUpDownArrow;	U+021D5
DoubleVerticalBar;	U+02225
DownArrow;	U+02193
DownArrowBar;	U+02913
DownArrowUpArrow;	U+021F5
DownBreve;	U+00311
DownLeftRightVector;	U+02950
DownLeftTeeVector;	U+0295E
DownLeftVector;	U+021BD
DownLeftVectorBar;	U+02956
DownRightTeeVector;	U+0295F
DownRightVector;	U+021C1
DownRightVectorBar;	U+02957
DownTee;	U+022A4
DownTeeArrow;	U+021A7
Downarrow;	U+021D3
Dscr;	U+1D49F
Dstrok;	U+00110
ENG;	U+0014A
ETH;	U+000D0
ETH	U+000D0
Eacute;	U+000C9
Eacute	U+000C9
Ecaron;	U+0011A
Ecirc;	U+000CA
Ecirc	U+000CA
Ecy;	U+0042D
Edot;	U+00116
Efr;	U+1D508
Egrave;	U+000C8
Egrave	U+000C8
Element;	U+02208
Emacr;	U+00112
EmptySmallSquare;	U+025FB
EmptyVerySmallSquare;	U+025AB
Eogon;	U+00118
Eopf;	U+1D53C
Epsilon;	U+00395
Equal;	U+02A75
EqualTilde;	U+02242
Equilibrium;	U+021CC
Escr;	U+02130
Esim;	U+02A73
Eta;	U+00397
Euml;	U+000CB
Euml	U+000CB
Exists;	U+02203
ExponentialE;	U+02147
Fcy;	U+00424
Ffr;	U+1D509

Name	Character
FilledSmallSquare;	U+025FC
FilledVerySmallSquare;	U+025AA
Fopf;	U+1D53D
ForAll;	U+02200
Fouriertrf;	U+02131
Fscr;	U+02131
GJcy;	U+00403
GT;	U+0003E
GT	U+0003E
Gamma;	U+00393
Gammad;	U+003DC
Gbreve;	U+0011E
Gcedil;	U+00122
Gcirc;	U+0011C
Gcy;	U+00413
Gdot;	U+00120
Gfr;	U+1D50A
Gg;	U+022D9
Gopf;	U+1D53E
GreaterEqual;	U+02265
GreaterEqualLess;	U+022DB
GreaterFullEqual;	U+02267
GreaterGreater;	U+02AA2
GreaterLess;	U+02277
GreaterSlantEqual;	U+02A7E
GreaterTilde;	U+02273
Gscr;	U+1D4A2
Gt;	U+0226B
HARDcy;	U+0042A
Hacek;	U+002C7
Hat;	U+0005E
Hcirc;	U+00124
Hfr;	U+0210C
HilbertSpace;	U+0210B
Hopf;	U+0210D
HorizontalLine;	U+02500
Hscr;	U+0210B
Hstrok;	U+00126
HumpDownHump;	U+0224E
HumpEqual;	U+0224F
IEcy;	U+00415
IJlig;	U+00132
IOcy;	U+00401
Iacute;	U+000CD
Iacute	U+000CD
Icirc;	U+000CE
Icirc	U+000CE
Icy;	U+00418
Idot;	U+00130
Ifr;	U+02111
Igrave;	U+000CC
Igrave	U+000CC
Im;	U+02111
Imacr;	U+0012A
ImaginaryI;	U+02148
Implies;	U+021D2
Int;	U+0222C
Integral;	U+0222B
Intersection;	U+022C2
InvisibleComma;	U+02063
InvisibleTimes;	U+02062

Name	Character
Iogon;	U+0012E
Iopf;	U+1D540
Iota;	U+00399
Iscr;	U+02110
Itilde;	U+00128
Iukcy;	U+00406
Iuml;	U+000CF
Iuml	U+000CF
Jcirc;	U+00134
Jcy;	U+00419
Jfr;	U+1D50D
Jopf;	U+1D541
Jscr;	U+1D4A5
Jsercy;	U+00408
Jukcy;	U+00404
KHcy;	U+00425
KJcy;	U+0040C
Kappa;	U+0039A
Kcedil;	U+00136
Kcy;	U+0041A
Kfr;	U+1D50E
Kopf;	U+1D542
Kscr;	U+1D4A6
LJcy;	U+00409
LT;	U+0003C
LT	U+0003C
Lacute;	U+00139
Lambda;	U+0039B
Lang;	U+027EA
Laplacetrf;	U+02112
Larr;	U+0219E
Lcaron;	U+0013D
Lcedil;	U+0013B
Lcy;	U+0041B
LeftAngleBracket;	U+027E8
LeftArrow;	U+02190
LeftArrowBar;	U+021E4
LeftArrowRightArrow;	U+021C6
LeftCeiling;	U+02308
LeftDoubleBracket;	U+027E6
LeftDownTeeVector;	U+02961
LeftDownVector;	U+021C3
LeftDownVectorBar;	U+02959
LeftFloor;	U+0230A
LeftRightArrow;	U+02194
LeftRightVector;	U+0294E
LeftTee;	U+022A3
LeftTeeArrow;	U+021A4
LeftTeeVector;	U+0295A
LeftTriangle;	U+022B2
LeftTriangleBar;	U+029CF
LeftTriangleEqual;	U+022B4
LeftUpDownVector;	U+02951
LeftUpTeeVector;	U+02960
LeftUpVector;	U+021BF
LeftUpVectorBar;	U+02958
LeftVector;	U+021BC
LeftVectorBar;	U+02952
Leftarrow;	U+021D0
Leftrightarrow;	U+021D4
LessEqualGreater;	U+022DA

Name	Character
LessFullEqual;	U+02266
LessGreater;	U+02276
LessLess;	U+02AA1
LessSlantEqual;	U+02A7D
LessTilde;	U+02272
Lfr;	U+1D50F
Ll;	U+022D8
Lleftarrow;	U+021DA
Lmidot;	U+0013F
LongLeftArrow;	U+027F5
LongLeftRightArrow;	U+027F7
LongRightArrow;	U+027F6
Longleftarrow;	U+027F8
Longleftrightarrow;	U+027FA
Longrightarrow;	U+027F9
Lopf;	U+1D543
LowerLeftArrow;	U+02199
LowerRightArrow;	U+02198
Lscr;	U+02112
Lsh;	U+021B0
Lstrok;	U+00141
Lt;	U+0226A
Map;	U+02905
Mcy;	U+0041C
MediumSpace;	U+0205F
Mellintrf;	U+02133
Mfr;	U+1D510
MinusPlus;	U+02213
Mopf;	U+1D544
Mscr;	U+02133
Mu;	U+0039C
NJcy;	U+0040A
Nacute;	U+00143
Ncaron;	U+00147
Ncedil;	U+00145
Ncy;	U+0041D
NegativeMediumSpace;	U+0200B
NegativeThickSpace;	U+0200B
NegativeThinSpace;	U+0200B
NegativeVeryThinSpace;	U+0200B
NestedGreaterGreater;	U+0226B
NestedLessLess;	U+0226A
NewLine;	U+0000A
Nfr;	U+1D511
NoBreak;	U+02060
NonBreakingSpace;	U+000A0
Nopf;	U+02115
Not;	U+02AEC
NotCongruent;	U+02262
NotCupCap;	U+0226D
NotDoubleVerticalBar;	U+02226
NotElement;	U+02209
NotEqual;	U+02260
NotExists;	U+02204
NotGreater;	U+0226F
NotGreaterEqual;	U+02271
NotGreaterLess;	U+02279
NotGreaterTilde;	U+02275
NotLeftTriangle;	U+022EA
NotLeftTriangleEqual;	U+022EC
NotLess;	U+0226E

Name	Character
NotLessEqual;	U+02270
NotLessGreater;	U+02278
NotLessTilde;	U+02274
NotPrecedes;	U+02280
NotPrecedesSlantEqual;	U+022E0
NotReverseElement;	U+0220C
NotRightTriangle;	U+022EB
NotRightTriangleEqual;	U+022ED
NotSquareSubsetEqual;	U+022E2
NotSquareSupersetEqual;	U+022E3
NotSubsetEqual;	U+02288
NotSucceeds;	U+02281
NotSucceedsSlantEqual;	U+022E1
NotSupersetEqual;	U+02289
NotTilde;	U+02241
NotTildeEqual;	U+02244
NotTildeFullEqual;	U+02247
NotTildeTilde;	U+02249
NotVerticalBar;	U+02224
Nscr;	U+1D4A9
Ntilde;	U+000D1
Ntilde	U+000D1
Nu;	U+0039D
OElig;	U+00152
Oacute;	U+000D3
Oacute	U+000D3
Ocirc;	U+000D4
Ocirc	U+000D4
Ocy;	U+0041E
Odblac;	U+00150
Ofr;	U+1D512
Ograve;	U+000D2
Ograve	U+000D2
Omacr;	U+0014C
Omega;	U+003A9
Omicron;	U+0039F
Oopf;	U+1D546
OpenCurlyDoubleQuote;	U+0201C
OpenCurlyQuote;	U+02018
Or;	U+02A54
Oscr;	U+1D4AA
Oslash;	U+000D8
Oslash	U+000D8
Otilde;	U+000D5
Otilde	U+000D5
Otimes;	U+02A37
Ouml;	U+000D6
Ouml	U+000D6
OverBar;	U+000AF
OverBrace;	U+023DE
OverBracket;	U+023B4
OverParenthesis;	U+023DC
PartialD;	U+02202
Pcy;	U+0041F
Pfr;	U+1D513
Phi;	U+003A6
Pi;	U+003A0
PlusMinus;	U+000B1
Poincareplane;	U+0210C
Popf;	U+02119
Pr;	U+02ABB

Name	Character
Precedes;	U+0227A
PrecedesEqual;	U+02AAF
PrecedesSlantEqual;	U+0227C
PrecedesTilde;	U+0227E
Prime;	U+02033
Product;	U+0220F
Proportion;	U+02237
Proportional;	U+0221D
Pscr;	U+1D4AB
Psi;	U+003A8
QUOT;	U+00022
QUOT	U+00022
Qfr;	U+1D514
Qopf;	U+0211A
Qscr;	U+1D4AC
RBarr;	U+02910
REG;	U+000AE
REG	U+000AE
Racute;	U+00154
Rang;	U+027EB
Rarr;	U+021A0
Rarrtl;	U+02916
Rcaron;	U+00158
Rcedil;	U+00156
Rcy;	U+00420
Re;	U+0211C
ReverseElement;	U+0220B
ReverseEquilibrium;	U+021CB
ReverseUpEquilibrium;	U+0296F
Rfr;	U+0211C
Rho;	U+003A1
RightAngleBracket;	U+027E9
RightArrow;	U+02192
RightArrowBar;	U+021E5
RightArrowLeftArrow;	U+021C4
RightCeiling;	U+02309
RightDoubleBracket;	U+027E7
RightDownTeeVector;	U+0295D
RightDownVector;	U+021C2
RightDownVectorBar;	U+02955
RightFloor;	U+0230B
RightTee;	U+022A2
RightTeeArrow;	U+021A6
RightTeeVector;	U+0295B
RightTriangle;	U+022B3
RightTriangleBar;	U+029D0
RightTriangleEqual;	U+022B5
RightUpDownVector;	U+0294F
RightUpTeeVector;	U+0295C
RightUpVector;	U+021BE
RightUpVectorBar;	U+02954
RightVector;	U+021C0
RightVectorBar;	U+02953
Rightarrow;	U+021D2
Ropf;	U+0211D
RoundImplies;	U+02970
Rrightarrow;	U+021DB
Rscr;	U+0211B
Rsh;	U+021B1
RuleDelayed;	U+029F4
SHCHcy;	U+00429

Name	Character
SHcy;	U+00428
SOFTcy;	U+0042C
Sacute;	U+0015A
Sc;	U+02ABC
Scaron;	U+00160
Scedil;	U+0015E
Scirc;	U+0015C
Scy;	U+00421
Sfr;	U+1D516
ShortDownArrow;	U+02193
ShortLeftArrow;	U+02190
ShortRightArrow;	U+02192
ShortUpArrow;	U+02191
Sigma;	U+003A3
SmallCircle;	U+02218
Sopf;	U+1D54A
Sqrt;	U+0221A
Square;	U+025A1
SquareIntersection;	U+02293
SquareSubset;	U+0228F
SquareSubsetEqual;	U+02291
SquareSuperset;	U+02290
SquareSupersetEqual;	U+02292
SquareUnion;	U+02294
Sscr;	U+1D4AE
Star;	U+022C6
Sub;	U+022D0
Subset;	U+022D0
SubsetEqual;	U+02286
Succeeds;	U+0227B
SucceedsEqual;	U+02AB0
SucceedsSlantEqual;	U+0227D
SucceedsTilde;	U+0227F
SuchThat;	U+0220B
Sum;	U+02211
Sup;	U+022D1
Superset;	U+02283
SupersetEqual;	U+02287
Supset;	U+022D1
THORN;	U+000DE
THORN	U+000DE
TRADE;	U+02122
TSHcy;	U+0040B
TScy;	U+00426
Tab;	U+00009
Tau;	U+003A4
Tcaron;	U+00164
Tcedil;	U+00162
Tcy;	U+00422
Tfr;	U+1D517
Therefore;	U+02234
Theta;	U+00398
ThinSpace;	U+02009
Tilde;	U+0223C
TildeEqual;	U+02243
TildeFullEqual;	U+02245
TildeTilde;	U+02248
Topf;	U+1D54B
TripleDot;	U+020DB
Tscr;	U+1D4AF
Tstrok;	U+00166

Name	Character
Uacute;	U+000DA
Uacute	U+000DA
Uarr;	U+0219F
Uarrocir;	U+02949
Ubrcy;	U+0040E
Ubreve;	U+0016C
Ucirc;	U+000DB
Ucirc	U+000DB
Ucy;	U+00423
Udblac;	U+00170
Ufr;	U+1D518
Ugrave;	U+000D9
Ugrave	U+000D9
Umacr;	U+0016A
UnderBar;	U+00332
UnderBrace;	U+023DF
UnderBracket;	U+023B5
UnderParenthesis;	U+023DD
Union;	U+022C3
UnionPlus;	U+0228E
Uogon;	U+00172
Uopf;	U+1D54C
UpArrow;	U+02191
UpArrowBar;	U+02912
UpArrowDownArrow;	U+021C5
UpDownArrow;	U+02195
UpEquilibrium;	U+0296E
UpTee;	U+022A5
UpTeeArrow;	U+021A5
Uparrow;	U+021D1
Updownarrow;	U+021D5
UpperLeftArrow;	U+02196
UpperRightArrow;	U+02197
Upsi;	U+003D2
Upsilon;	U+003A5
Uring;	U+0016E
Uscr;	U+1D4B0
Utilde;	U+00168
Uuml;	U+000DC
Uuml	U+000DC
VDash;	U+022AB
Vbar;	U+02AEB
Vcy;	U+00412
Vdash;	U+022A9
Vdashl;	U+02AE6
Vee;	U+022C1
Verbar;	U+02016
Vert;	U+02016
VerticalBar;	U+02223
VerticalLine;	U+0007C
VerticalSeparator;	U+02758
VerticalTilde;	U+02240
VeryThinSpace;	U+0200A
Vfr;	U+1D519
Vopf;	U+1D54D
Vscr;	U+1D4B1
Vvdash;	U+022AA
Wcirc;	U+00174
Wedge;	U+022C0
Wfr;	U+1D51A
Wopf;	U+1D54E

Name	Character
Wscr;	U+1D4B2
Xfr;	U+1D51B
Xi;	U+0039E
Xopf;	U+1D54F
Xscr;	U+1D4B3
YAcy;	U+0042F
YIcy;	U+00407
YUcy;	U+0042E
Yacute;	U+000DD
Yacute	U+000DD
Ycirc;	U+00176
Ycy;	U+0042B
Yfr;	U+1D51C
Yopf;	U+1D550
Yscr;	U+1D4B4
Yuml;	U+00178
ZHcy;	U+00416
Zacute;	U+00179
Zcaron;	U+0017D
Zcy;	U+00417
Zdot;	U+0017B
ZeroWidthSpace;	U+0200B
Zeta;	U+00396
Zfr;	U+02128
Zopf;	U+02124
Zscr;	U+1D4B5
aacute;	U+000E1
aacute	U+000E1
abreve;	U+00103
ac;	U+0223E
acd;	U+0223F
acirc;	U+000E2
acirc	U+000E2
acute;	U+000B4
acute	U+000B4
acy;	U+00430
aelig;	U+000E6
aelig	U+000E6
af;	U+02061
afr;	U+1D51E
agrave;	U+000E0
agrave	U+000E0
alefsym;	U+02135
aleph;	U+02135
alpha;	U+003B1
amacr;	U+00101
amalg;	U+02A3F
amp;	U+00026
amp	U+00026
and;	U+02227
andand;	U+02A55
andd;	U+02A5C
andslope;	U+02A58
andv;	U+02A5A
ang;	U+02220
ange;	U+029A4
angle;	U+02220
angmsd;	U+02221
angmsdaa;	U+029A8
angmsdab;	U+029A9
angmsdac;	U+029AA

Name	Character
angmsdad;	U+029AB
angmsdae;	U+029AC
angmsdaf;	U+029AD
angmsdag;	U+029AE
angmsdah;	U+029AF
angrt;	U+0221F
angrtvb;	U+022BE
angrtvbd;	U+0299D
angsph;	U+02222
angst;	U+0212B
angzarr;	U+0237C
aogon;	U+00105
aopf;	U+1D552
ap;	U+02248
apE;	U+02A70
apacir;	U+02A6F
ape;	U+0224A
apid;	U+0224B
apos;	U+00027
approx;	U+02248
approxeq;	U+0224A
aring;	U+000E5
aring	U+000E5
ascr;	U+1D4B6
ast;	U+0002A
asymp;	U+02248
asympeq;	U+0224D
atilde;	U+000E3
atilde	U+000E3
auml;	U+000E4
auml	U+000E4
awconint;	U+02233
awint;	U+02A11
bNot;	U+02AED
backcong;	U+0224C
backepsilon;	U+003F6
backprime;	U+02035
backsim;	U+0223D
backsimeq;	U+022CD
barvee;	U+022BD
barwed;	U+02305
barwedge;	U+02305
bbrk;	U+023B5
bbrktbrk;	U+023B6
bcong;	U+0224C
bcy;	U+00431
bdquo;	U+0201E
becaus;	U+02235
because;	U+02235
bemptyv;	U+029B0
bepsi;	U+003F6
bernou;	U+0212C
beta;	U+003B2
beth;	U+02136
between;	U+0226C
bfr;	U+1D51F
bigcap;	U+022C2
bigcirc;	U+025EF
bigcup;	U+022C3
bigodot;	U+02A00
bigoplus;	U+02A01

Name	Character
bigotimes;	U+02A02
bigsqcup;	U+02A06
bigstar;	U+02605
bigtriangledown;	U+025BD
bigtriangleup;	U+025B3
biguplus;	U+02A04
bigvee;	U+022C1
bigwedge;	U+022C0
bkarow;	U+0290D
blacklozenge;	U+029EB
blacksquare;	U+025AA
blacktriangle;	U+025B4
blacktriangledown;	U+025BE
blacktriangleleft;	U+025C2
blacktriangleright;	U+025B8
blank;	U+02423
blk12;	U+02592
blk14;	U+02591
blk34;	U+02593
block;	U+02588
bnot;	U+02310
bopf;	U+1D553
bot;	U+022A5
bottom;	U+022A5
bowtie;	U+022C8
boxDL;	U+02557
boxDR;	U+02554
boxDl;	U+02556
boxDr;	U+02553
boxH;	U+02550
boxHD;	U+02566
boxHU;	U+02569
boxHd;	U+02564
boxHu;	U+02567
boxUL;	U+0255D
boxUR;	U+0255A
boxUl;	U+0255C
boxUr;	U+02559
boxV;	U+02551
boxVH;	U+0256C
boxVL;	U+02563
boxVR;	U+02560
boxVh;	U+0256B
boxVl;	U+02562
boxVr;	U+0255F
boxbox;	U+029C9
boxdL;	U+02555
boxdR;	U+02552
boxdl;	U+02510
boxdr;	U+0250C
boxh;	U+02500
boxhD;	U+02565
boxhU;	U+02568
boxhd;	U+0252C
boxhu;	U+02534
boxminus;	U+0229F
boxplus;	U+0229E
boxtimes;	U+022A0
boxuL;	U+0255B
boxuR;	U+02558
boxul;	U+02518

Name	Character
boxur;	U+02514
boxv;	U+02502
boxvH;	U+0256A
boxvL;	U+02561
boxvR;	U+0255E
boxvh;	U+0253C
boxvl;	U+02524
boxvr;	U+0251C
bprime;	U+02035
breve;	U+002D8
brvbar;	U+000A6
brvbar	U+000A6
bscr;	U+1D4B7
bsemi;	U+0204F
bsim;	U+0223D
bsime;	U+022CD
bsol;	U+0005C
bsolb;	U+029C5
bull;	U+02022
bullet;	U+02022
bump;	U+0224E
bumpE;	U+02AAE
bumpe;	U+0224F
bumpeq;	U+0224F
cacute;	U+00107
cap;	U+02229
capand;	U+02A44
capbrcup;	U+02A49
capcap;	U+02A4B
capcup;	U+02A47
capdot;	U+02A40
caret;	U+02041
caron;	U+002C7
ccaps;	U+02A4D
ccaron;	U+0010D
ccedil;	U+000E7
ccedil	U+000E7
ccirc;	U+00109
ccups;	U+02A4C
ccupssm;	U+02A50
cdot;	U+0010B
cedil;	U+000B8
cedil	U+000B8
cemptyv;	U+029B2
cent;	U+000A2
cent	U+000A2
centerdot;	U+000B7
cfr;	U+1D520
chcy;	U+00447
check;	U+02713
checkmark;	U+02713
chi;	U+003C7
cir;	U+025CB
cirE;	U+029C3
circ;	U+002C6
circeq;	U+02257
circlearrowleft;	U+021BA
circlearrowright;	U+021BB
circledR;	U+000AE
circledS;	U+024C8
circledast;	U+0229B

Name	Character
circledcirc;	U+0229A
circleddash;	U+0229D
cire;	U+02257
cirfnint;	U+02A10
cirmid;	U+02AEF
cirscir;	U+029C2
clubs;	U+02663
clubsuit;	U+02663
colon;	U+0003A
colone;	U+02254
coloneq;	U+02254
comma;	U+0002C
commat;	U+00040
comp;	U+02201
compfn;	U+02218
complement;	U+02201
complexes;	U+02102
cong;	U+02245
congdot;	U+02A6D
conint;	U+0222E
copf;	U+1D554
coprod;	U+02210
copy;	U+000A9
copy	U+000A9
copysr;	U+02117
crarr;	U+021B5
cross;	U+02717
cscr;	U+1D4B8
csub;	U+02ACF
csube;	U+02AD1
csup;	U+02AD0
csupe;	U+02AD2
ctdot;	U+022EF
cudarrl;	U+02938
cudarrr;	U+02935
cuepr;	U+022DE
cuesc;	U+022DF
cularr;	U+021B6
cularrp;	U+0293D
cup;	U+0222A
cupbrcap;	U+02A48
cupcap;	U+02A46
cupcup;	U+02A4A
cupdot;	U+0228D
cupor;	U+02A45
curarr;	U+021B7
curarrm;	U+0293C
curlyeqprec;	U+022DE
curlyeqsucc;	U+022DF
curlyvee;	U+022CE
curlywedge;	U+022CF
curren;	U+000A4
curren	U+000A4
curvearrowleft;	U+021B6
curvearrowright;	U+021B7
cuvee;	U+022CE
cuwed;	U+022CF
cwconint;	U+02232
cwint;	U+02231
cylcty;	U+0232D
dArr;	U+021D3
dHar;	U+02965
dagger;	U+02020
daleth;	U+02138
darr;	U+02193
dash;	U+02010
dashv;	U+022A3
dbkarow;	U+0290F
dblac;	U+002DD
dcaron;	U+0010F
dcy;	U+00434
dd;	U+02146
ddagger;	U+02021
ddarr;	U+021CA
ddotseq;	U+02A77
deg;	U+000B0
deg	U+000B0
delta;	U+003B4
demptyv;	U+029B1
dfisht;	U+0297F
dfr;	U+1D521
dharl;	U+021C3
dharr;	U+021C2
diam;	U+022C4
diamond;	U+022C4
diamondsuit;	U+02666
diams;	U+02666
die;	U+000A8
digamma;	U+003DD
disin;	U+022F2
div;	U+000F7
divide;	U+000F7
divide	U+000F7
divideontimes;	U+022C7
divonx;	U+022C7
djcy;	U+00452
dlcorn;	U+0231E
dlcrop;	U+0230D
dollar;	U+00024
dopf;	U+1D555
dot;	U+002D9
doteq;	U+02250
doteqdot;	U+02251
dotminus;	U+02238
dotplus;	U+02214
dotsquare;	U+022A1
doublebarwedge;	U+02306
downarrow;	U+02193
downdownarrows;	U+021CA
downharpoonleft;	U+021C3
downharpoonright;	U+021C2
drbkarow;	U+02910
drcorn;	U+0231F
drcrop;	U+0230C
dscr;	U+1D4B9
dscy;	U+00455
dsol;	U+029F6
dstrok;	U+00111
dtdot;	U+022F1
dtri;	U+025BF
dtrif;	U+025BE
duarr;	U+021F5

Name	Character
duhar;	U+0296F
dwangle;	U+029A6
dzcy;	U+0045F
dzigrarr;	U+027FF
eDDot;	U+02A77
eDot;	U+02251
eacute;	U+000E9
eacute	U+000E9
easter;	U+02A6E
ecaron;	U+0011B
ecir;	U+02256
ecirc;	U+000EA
ecirc	U+000EA
ecolon;	U+02255
ecy;	U+0044D
edot;	U+00117
ee;	U+02147
efDot;	U+02252
efr;	U+1D522
eg;	U+02A9A
egrave;	U+000E8
egrave	U+000E8
egs;	U+02A96
egsdot;	U+02A98
el;	U+02A99
elinters;	U+023E7
ell;	U+02113
els;	U+02A95
elsdot;	U+02A97
emacr;	U+00113
empty;	U+02205
emptyset;	U+02205
emptyv;	U+02205
emsp13;	U+02004
emsp14;	U+02005
emsp;	U+02003
eng;	U+0014B
ensp;	U+02002
eogon;	U+00119
eopf;	U+1D556
epar;	U+022D5
eparsl;	U+029E3
eplus;	U+02A71
epsi;	U+003F5
epsilon;	U+003B5
epsiv;	U+003B5
eqcirc;	U+02256
eqcolon;	U+02255
eqsim;	U+02242
eqslantgtr;	U+02A96
eqslantless;	U+02A95
equals;	U+0003D
equest;	U+0225F
equiv;	U+02261
equivDD;	U+02A78
eqvparsl;	U+029E5
erDot;	U+02253
erarr;	U+02971
escr;	U+0212F
esdot;	U+02250
esim;	U+02242

Name	Character
eta;	U+003B7
eth;	U+000F0
eth	U+000F0
euml;	U+000EB
euml	U+000EB
euro;	U+020AC
excl;	U+00021
exist;	U+02203
expectation;	U+02130
exponentiale;	U+02147
fallingdotseq;	U+02252
fcy;	U+00444
female;	U+02640
ffilig;	U+0FB03
fflig;	U+0FB00
ffllig;	U+0FB04
ffr;	U+1D523
filig;	U+0FB01
flat;	U+0266D
fllig;	U+0FB02
fltns;	U+025B1
fnof;	U+00192
fopf;	U+1D557
forall;	U+02200
fork;	U+022D4
forkv;	U+02AD9
fpartint;	U+02A0D
frac12;	U+000BD
frac12	U+000BD
frac13;	U+02153
frac14;	U+000BC
frac14	U+000BC
frac15;	U+02155
frac16;	U+02159
frac18;	U+0215B
frac23;	U+02154
frac25;	U+02156
frac34;	U+000BE
frac34	U+000BE
frac35;	U+02157
frac38;	U+0215C
frac45;	U+02158
frac56;	U+0215A
frac58;	U+0215D
frac78;	U+0215E
frasl;	U+02044
frown;	U+02322
fscr;	U+1D4BB
gE;	U+02267
gEl;	U+02A8C
gacute;	U+001F5
gamma;	U+003B3
gammad;	U+003DD
gap;	U+02A86
gbreve;	U+0011F
gcirc;	U+0011D
gcy;	U+00433
gdot;	U+00121
ge;	U+02265
gel;	U+022DB
geq;	U+02265

Name	Character
geqq;	U+02267
geqslant;	U+02A7E
ges;	U+02A7E
gescc;	U+02AA9
gesdot;	U+02A80
gesdoto;	U+02A82
gesdotol;	U+02A84
gesles;	U+02A94
gfr;	U+1D524
gg;	U+0226B
ggg;	U+022D9
gimel;	U+02137
gjcy;	U+00453
gl;	U+02277
glE;	U+02A92
gla;	U+02AA5
glj;	U+02AA4
gnE;	U+02269
gnap;	U+02A8A
gnapprox;	U+02A8A
gne;	U+02A88
gneq;	U+02A88
gneqq;	U+02269
gnsim;	U+022E7
gopf;	U+1D558
grave;	U+00060
gscr;	U+0210A
gsim;	U+02273
gsime;	U+02A8E
gsiml;	U+02A90
gt;	U+0003E
gt	U+0003E
gtcc;	U+02AA7
gtcir;	U+02A7A
gtdot;	U+022D7
gtlPar;	U+02995
gtquest;	U+02A7C
gtrapprox;	U+02A86
gtrarr;	U+02978
gtrdot;	U+022D7
gtreqless;	U+022DB
gtreqqless;	U+02A8C
gtrless;	U+02277
gtrsim;	U+02273
hArr;	U+021D4
hairsp;	U+0200A
half;	U+000BD
hamilt;	U+0210B
hardcy;	U+0044A
harr;	U+02194
harrcir;	U+02948
harrw;	U+021AD
hbar;	U+0210F
hcirc;	U+00125
hearts;	U+02665
heartsuit;	U+02665
hellip;	U+02026
hercon;	U+022B9
hfr;	U+1D525
hksearow;	U+02925
hkswarow;	U+02926

Name	Character
hoarr;	U+021FF
homtht;	U+0223B
hookleftarrow;	U+021A9
hookrightarrow;	U+021AA
hopf;	U+1D559
horbar;	U+02015
hscr;	U+1D4BD
hslash;	U+0210F
hstrok;	U+00127
hybull;	U+02043
hyphen;	U+02010
iacute;	U+000ED
iacute	U+000ED
ic;	U+02063
icirc;	U+000EE
icirc	U+000EE
icy;	U+00438
iecy;	U+00435
iexcl;	U+000A1
iexcl	U+000A1
iff;	U+021D4
ifr;	U+1D526
igrave;	U+000EC
igrave	U+000EC
ii;	U+02148
iiiint;	U+02A0C
iiint;	U+0222D
iinfin;	U+029DC
iiota;	U+02129
ijlig;	U+00133
imacr;	U+0012B
image;	U+02111
imagline;	U+02110
imagpart;	U+02111
imath;	U+00131
imof;	U+022B7
imped;	U+001B5
in;	U+02208
incare;	U+02105
infin;	U+0221E
infintie;	U+029DD
inodot;	U+00131
int;	U+0222B
intcal;	U+022BA
integers;	U+02124
intercal;	U+022BA
intlarhk;	U+02A17
intprod;	U+02A3C
iocy;	U+00451
iogon;	U+0012F
iopf;	U+1D55A
iota;	U+003B9
iprod;	U+02A3C
iquest;	U+000BF
iquest	U+000BF
iscr;	U+1D4BE
isin;	U+02208
isinE;	U+022F9
isindot;	U+022F5
isins;	U+022F4
isinsv;	U+022F3

Name	Character
isinv;	U+02208
it;	U+02062
itilde;	U+00129
iukcy;	U+00456
iuml;	U+000EF
iuml	U+000EF
jcirc;	U+00135
jcy;	U+00439
jfr;	U+1D527
jmath;	U+00237
jopf;	U+1D55B
jscr;	U+1D4BF
jsercy;	U+00458
jukcy;	U+00454
kappa;	U+003BA
kappav;	U+003F0
kcedil;	U+00137
kcy;	U+0043A
kfr;	U+1D528
kgreen;	U+00138
khcy;	U+00445
kjcy;	U+0045C
kopf;	U+1D55C
kscr;	U+1D4C0
lAarr;	U+021DA
lArr;	U+021D0
lAtail;	U+0291B
lBarr;	U+0290E
lE;	U+02266
lEg;	U+02A8B
lHar;	U+02962
lacute;	U+0013A
laemptyv;	U+029B4
lagran;	U+02112
lambda;	U+003BB
lang;	U+027E8
langd;	U+02991
langle;	U+027E8
lap;	U+02A85
laquo;	U+000AB
laquo	U+000AB
larr;	U+02190
larrb;	U+021E4
larrbfs;	U+0291F
larrfs;	U+0291D
larrhk;	U+021A9
larrlp;	U+021AB
larrpl;	U+02939
larrsim;	U+02973
larrtl;	U+021A2
lat;	U+02AAB
latail;	U+02919
late;	U+02AAD
lbarr;	U+0290C
lbbrk;	U+02772
lbrace;	U+0007B
lbrack;	U+0005B
lbrke;	U+0298B
lbrksld;	U+0298F
lbrkslu;	U+0298D
lcaron;	U+0013E

Name	Character
lcedil;	U+0013C
lceil;	U+02308
lcub;	U+0007B
lcy;	U+0043B
ldca;	U+02936
ldquo;	U+0201C
ldquor;	U+0201E
ldrdhar;	U+02967
ldrushar;	U+0294B
ldsh;	U+021B2
le;	U+02264
leftarrow;	U+02190
leftarrowtail;	U+021A2
leftharpoondown;	U+021BD
leftharpoonup;	U+021BC
leftleftarrows;	U+021C7
leftrightarrow;	U+02194
leftrightarrows;	U+021C6
leftrightharpoons;	U+021CB
leftrightsquigarrow;	U+021AD
leftthreetimes;	U+022CB
leg;	U+022DA
leq;	U+02264
leqq;	U+02266
leqslant;	U+02A7D
les;	U+02A7D
lescc;	U+02AA8
lesdot;	U+02A7F
lesdoto;	U+02A81
lesdotor;	U+02A83
lesges;	U+02A93
lessapprox;	U+02A85
lessdot;	U+022D6
lesseqgtr;	U+022DA
lesseqqgtr;	U+02A8B
lessgtr;	U+02276
lesssim;	U+02272
lfisht;	U+0297C
lfloor;	U+0230A
lfr;	U+1D529
lg;	U+02276
lgE;	U+02A91
lhard;	U+021BD
lharu;	U+021BC
lharul;	U+0296A
lhblk;	U+02584
ljcy;	U+00459
ll;	U+0226A
llarr;	U+021C7
llcorner;	U+0231E
llhard;	U+0296B
lltri;	U+025FA
lmidot;	U+00140
lmoust;	U+023B0
lmoustache;	U+023B0
lnE;	U+02268
lnap;	U+02A89
lnapprox;	U+02A89
lne;	U+02A87
lneq;	U+02A87
lneqq;	U+02268

Name	Character
lnsim;	U+022E6
loang;	U+027EC
loarr;	U+021FD
lobrk;	U+027E6
longleftarrow;	U+027F5
longleftrightarrow;	U+027F7
longmapsto;	U+027FC
longrightarrow;	U+027F6
looparrowleft;	U+021AB
looparrowright;	U+021AC
lopar;	U+02985
lopf;	U+1D55D
loplus;	U+02A2D
lotimes;	U+02A34
lowast;	U+02217
lowbar;	U+0005F
loz;	U+025CA
lozenge;	U+025CA
lozf;	U+029EB
lpar;	U+00028
lparlt;	U+02993
lrarr;	U+021C6
lrcorner;	U+0231F
lrhar;	U+021CB
lrhard;	U+0296D
lrm;	U+0200E
lrtri;	U+022BF
lsaquo;	U+02039
lscr;	U+1D4C1
lsh;	U+021B0
lsim;	U+02272
lsime;	U+02A8D
lsimg;	U+02A8F
lsqb;	U+0005B
lsquo;	U+02018
lsquor;	U+0201A
lstrok;	U+00142
lt;	U+0003C
lt	U+0003C
ltcc;	U+02AA6
ltcir;	U+02A79
ltdot;	U+022D6
lthree;	U+022CB
ltimes;	U+022C9
ltlarr;	U+02976
ltquest;	U+02A7B
ltrPar;	U+02996
ltri;	U+025C3
ltrie;	U+022B4
ltrif;	U+025C2
lurdshar;	U+0294A
luruhar;	U+02966
mDDot;	U+0223A
macr;	U+000AF
macr	U+000AF
male;	U+02642
malt;	U+02720
maltese;	U+02720
map;	U+021A6
mapsto;	U+021A6
mapstodown;	U+021A7

Name	Character
mapstoleft;	U+021A4
mapstoup;	U+021A5
marker;	U+025AE
mcomma;	U+02A29
mcy;	U+0043C
mdash;	U+02014
measuredangle;	U+02221
mfr;	U+1D52A
mho;	U+02127
micro;	U+000B5
micro	U+000B5
mid;	U+02223
midast;	U+0002A
midcir;	U+02AF0
middot;	U+000B7
middot	U+000B7
minus;	U+02212
minusb;	U+0229F
minusd;	U+02238
minusdu;	U+02A2A
mlcp;	U+02ADB
mldr;	U+02026
mnplus;	U+02213
models;	U+022A7
mopf;	U+1D55E
mp;	U+02213
mscr;	U+1D4C2
mstpos;	U+0223E
mu;	U+003BC
multimap;	U+022B8
mumap;	U+022B8
nLeftarrow;	U+021CD
nLeftrightarrow;	U+021CE
nRightarrow;	U+021CF
nVDash;	U+022AF
nVdash;	U+022AE
nabla;	U+02207
nacute;	U+00144
nap;	U+02249
napos;	U+00149
napprox;	U+02249
natur;	U+0266E
natural;	U+0266E
naturals;	U+02115
nbsp;	U+000A0
nbsp	U+000A0
ncap;	U+02A43
ncaron;	U+00148
ncedil;	U+00146
ncong;	U+02247
ncup;	U+02A42
ncy;	U+0043D
ndash;	U+02013
ne;	U+02260
neArr;	U+021D7
nearhk;	U+02924
nearr;	U+02197
nearrow;	U+02197
nequiv;	U+02262
nesear;	U+02928
nexist;	U+02204

Name	Character
nexists;	U+02204
nfr;	U+1D52B
nge;	U+02271
ngeq;	U+02271
ngsim;	U+02275
ngt;	U+0226F
ngtr;	U+0226F
nhArr;	U+021CE
nharr;	U+021AE
nhpar;	U+02AF2
ni;	U+0220B
nis;	U+022FC
nisd;	U+022FA
niv;	U+0220B
njcy;	U+0045A
nlArr;	U+021CD
nlarr;	U+0219A
nldr;	U+02025
nle;	U+02270
nleftarrow;	U+0219A
nleftrightarrow;	U+021AE
nleq;	U+02270
nless;	U+0226E
nlsim;	U+02274
nlt;	U+0226E
nltri;	U+022EA
nltrie;	U+022EC
nmid;	U+02224
nopf;	U+1D55F
not;	U+000AC
not	U+000AC
notin;	U+02209
notinva;	U+02209
notinvb;	U+022F7
notinvc;	U+022F6
notni;	U+0220C
notniva;	U+0220C
notnivb;	U+022FE
notnivc;	U+022FD
npar;	U+02226
nparallel;	U+02226
npolint;	U+02A14
npr;	U+02280
nprcue;	U+022E0
nprec;	U+02280
nrArr;	U+021CF
nrarr;	U+0219B
nrightarrow;	U+0219B
nrtri;	U+022EB
nrtrie;	U+022ED
nsc;	U+02281
nsccue;	U+022E1
nscr;	U+1D4C3
nshortmid;	U+02224
nshortparallel;	U+02226
nsim;	U+02241
nsime;	U+02244
nsimeq;	U+02244
nsmid;	U+02224
nspar;	U+02226
nsqsube;	U+022E2

Name	Character
nsqsupe;	U+022E3
nsub;	U+02284
nsube;	U+02288
nsubseteq;	U+02288
nsucc;	U+02281
nsup;	U+02285
nsupe;	U+02289
nsupseteq;	U+02289
ntgl;	U+02279
ntilde;	U+000F1
ntilde	U+000F1
ntlg;	U+02278
ntriangleleft;	U+022EA
ntrianglelefteq;	U+022EC
ntriangleright;	U+022EB
ntrianglerighteq;	U+022ED
nu;	U+003BD
num;	U+00023
numero;	U+02116
numsp;	U+02007
nvDash;	U+022AD
nvHarr;	U+02904
nvdash;	U+022AC
nvinfin;	U+029DE
nvlArr;	U+02902
nvrArr;	U+02903
nwArr;	U+021D6
nwarhk;	U+02923
nwarr;	U+02196
nwarrow;	U+02196
nwnear;	U+02927
oS;	U+024C8
oacute;	U+000F3
oacute	U+000F3
oast;	U+0229B
ocir;	U+0229A
ocirc;	U+000F4
ocirc	U+000F4
ocy;	U+0043E
odash;	U+0229D
odblac;	U+00151
odiv;	U+02A38
odot;	U+02299
odsold;	U+029BC
oelig;	U+00153
ofcir;	U+029BF
ofr;	U+1D52C
ogon;	U+002DB
ograve;	U+000F2
ograve	U+000F2
ogt;	U+029C1
ohbar;	U+029B5
ohm;	U+02126
oint;	U+0222E
olarr;	U+021BA
olcir;	U+029BE
olcross;	U+029BB
oline;	U+0203E
olt;	U+029C0
omacr;	U+0014D
omega;	U+003C9

Name	Character
omicron;	U+003BF
omid;	U+029B6
ominus;	U+02296
oopf;	U+1D560
opar;	U+029B7
operp;	U+029B9
oplus;	U+02295
or;	U+02228
orarr;	U+021BB
ord;	U+02A5D
order;	U+02134
orderof;	U+02134
ordf;	U+000AA
ordf	U+000AA
ordm;	U+000BA
ordm	U+000BA
origof;	U+022B6
oror;	U+02A56
orslope;	U+02A57
orv;	U+02A5B
oscr;	U+02134
oslash;	U+000F8
oslash	U+000F8
osol;	U+02298
otilde;	U+000F5
otilde	U+000F5
otimes;	U+02297
otimesas;	U+02A36
ouml;	U+000F6
ouml	U+000F6
ovbar;	U+0233D
par;	U+02225
para;	U+000B6
para	U+000B6
parallel;	U+02225
parsim;	U+02AF3
parsl;	U+02AFD
part;	U+02202
pcy;	U+0043F
percnt;	U+00025
period;	U+0002E
permil;	U+02030
perp;	U+022A5
pertenk;	U+02031
pfr;	U+1D52D
phi;	U+003C6
phiv;	U+003C6
phmmat;	U+02133
phone;	U+0260E
pi;	U+003C0
pitchfork;	U+022D4
piv;	U+003D6
planck;	U+0210F
planckh;	U+0210E
plankv;	U+0210F
plus;	U+0002B
plusacir;	U+02A23
plusb;	U+0229E
pluscir;	U+02A22
plusdo;	U+02214
plusdu;	U+02A25

Name	Character
pluse;	U+02A72
plusmn;	U+000B1
plusmn	U+000B1
plussim;	U+02A26
plustwo;	U+02A27
pm;	U+000B1
pointint;	U+02A15
popf;	U+1D561
pound;	U+000A3
pound	U+000A3
pr;	U+0227A
prE;	U+02AB3
prap;	U+02AB7
prcue;	U+0227C
pre;	U+02AAF
prec;	U+0227A
precapprox;	U+02AB7
preccurlyeq;	U+0227C
preceq;	U+02AAF
precnapprox;	U+02AB9
precneqq;	U+02AB5
precnsim;	U+022E8
precsim;	U+0227E
prime;	U+02032
primes;	U+02119
prnE;	U+02AB5
prnap;	U+02AB9
prnsim;	U+022E8
prod;	U+0220F
profalar;	U+0232E
profline;	U+02312
profsurf;	U+02313
prop;	U+0221D
propto;	U+0221D
prsim;	U+0227E
prurel;	U+022B0
pscr;	U+1D4C5
psi;	U+003C8
puncsp;	U+02008
qfr;	U+1D52E
qint;	U+02A0C
qopf;	U+1D562
qprime;	U+02057
qscr;	U+1D4C6
quaternions;	U+0210D
quatint;	U+02A16
quest;	U+0003F
questeq;	U+0225F
quot;	U+00022
quot	U+00022
rAarr;	U+021DB
rArr;	U+021D2
rAtail;	U+0291C
rBarr;	U+0290F
rHar;	U+02964
race;	U+029DA
racute;	U+00155
radic;	U+0221A
raemptyv;	U+029B3
rang;	U+027E9
rangd;	U+02992

Name	Character
range;	U+029A5
rangle;	U+027E9
raquo;	U+000BB
raquo	U+000BB
rarr;	U+02192
rarrap;	U+02975
rarrb;	U+021E5
rarrbfs;	U+02920
rarrc;	U+02933
rarrfs;	U+0291E
rarrhk;	U+021AA
rarrlp;	U+021AC
rarrpl;	U+02945
rarrsim;	U+02974
rarrtl;	U+021A3
rarrw;	U+0219D
ratail;	U+0291A
ratio;	U+02236
rationals;	U+0211A
rbarr;	U+0290D
rbbrk;	U+02773
rbrace;	U+0007D
rbrack;	U+0005D
rbrke;	U+0298C
rbrksld;	U+0298E
rbrkslu;	U+02990
rcaron;	U+00159
rcedil;	U+00157
rceil;	U+02309
rcub;	U+0007D
rcy;	U+00440
rdca;	U+02937
rdldhar;	U+02969
rdquo;	U+0201D
rdquor;	U+0201D
rdsh;	U+021B3
real;	U+0211C
realine;	U+0211B
realpart;	U+0211C
reals;	U+0211D
rect;	U+025AD
reg;	U+000AE
reg	U+000AE
rfisht;	U+0297D
rfloor;	U+0230B
rfr;	U+1D52F
rhard;	U+021C1
rharu;	U+021C0
rharul;	U+0296C
rho;	U+003C1
rhov;	U+003F1
rightarrow;	U+02192
rightarrowtail;	U+021A3
rightharpoondown;	U+021C1
rightharpoonup;	U+021C0
rightleftarrows;	U+021C4
rightleftharpoons;	U+021CC
rightrightarrows;	U+021C9
rightsquigarrow;	U+0219D
rightthreetimes;	U+022CC
ring;	U+002DA

Name	Character
risingdotseq;	U+02253
rlarr;	U+021C4
rlhar;	U+021CC
rlm;	U+0200F
rmoust;	U+023B1
rmoustache;	U+023B1
rnmid;	U+02AEE
roang;	U+027ED
roarr;	U+021FE
robrk;	U+027E7
ropar;	U+02986
ropf;	U+1D563
roplus;	U+02A2E
rotimes;	U+02A35
rpar;	U+00029
rpargt;	U+02994
rppolint;	U+02A12
rrarr;	U+021C9
rsaquo;	U+0203A
rscr;	U+1D4C7
rsh;	U+021B1
rsqb;	U+0005D
rsquo;	U+02019
rsquor;	U+02019
rthree;	U+022CC
rtimes;	U+022CA
rtri;	U+025B9
rtrie;	U+022B5
rtrif;	U+025B8
rtriltri;	U+029CE
ruluhar;	U+02968
rx;	U+0211E
sacute;	U+0015B
sbquo;	U+0201A
sc;	U+0227B
scE;	U+02AB4
scap;	U+02AB8
scaron;	U+00161
sccue;	U+0227D
sce;	U+02AB0
scedil;	U+0015F
scirc;	U+0015D
scnE;	U+02AB6
scnap;	U+02ABA
scnsim;	U+022E9
scpolint;	U+02A13
scsim;	U+0227F
scy;	U+00441
sdot;	U+022C5
sdotb;	U+022A1
sdote;	U+02A66
seArr;	U+021D8
searhk;	U+02925
searr;	U+02198
searrow;	U+02198
sect;	U+000A7
sect	U+000A7
semi;	U+0003B
seswar;	U+02929
setminus;	U+02216
setmn;	U+02216

Name	Character
sext;	U+02736
sfr;	U+1D530
sfrown;	U+02322
sharp;	U+0266F
shchcy;	U+00449
shcy;	U+00448
shortmid;	U+02223
shortparallel;	U+02225
shy;	U+000AD
shy	U+000AD
sigma;	U+003C3
sigmaf;	U+003C2
sigmav;	U+003C2
sim;	U+0223C
simdot;	U+02A6A
sime;	U+02243
simeq;	U+02243
simg;	U+02A9E
simgE;	U+02AA0
siml;	U+02A9D
simlE;	U+02A9F
simne;	U+02246
simplus;	U+02A24
simrarr;	U+02972
slarr;	U+02190
smallsetminus;	U+02216
smashp;	U+02A33
smeparsl;	U+029E4
smid;	U+02223
smile;	U+02323
smt;	U+02AAA
smte;	U+02AAC
softcy;	U+0044C
sol;	U+0002F
solb;	U+029C4
solbar;	U+0233F
sopf;	U+1D564
spades;	U+02660
spadesuit;	U+02660
spar;	U+02225
sqcap;	U+02293
sqcup;	U+02294
sqsub;	U+0228F
sqsube;	U+02291
sqsubset;	U+0228F
sqsubseteq;	U+02291
sqsup;	U+02290
sqsupe;	U+02292
sqsupset;	U+02290
sqsupseteq;	U+02292
squ;	U+025A1
square;	U+025A1
squarf;	U+025AA
squf;	U+025AA
srarr;	U+02192
sscr;	U+1D4C8
ssetmn;	U+02216
ssmile;	U+02323
sstarf;	U+022C6
star;	U+02606
starf;	U+02605

Name	Character
straightepsilon;	U+003F5
straightphi;	U+003D5
strns;	U+000AF
sub;	U+02282
subE;	U+02AC5
subdot;	U+02ABD
sube;	U+02286
subedot;	U+02AC3
submult;	U+02AC1
subnE;	U+02ACB
subne;	U+0228A
subplus;	U+02ABF
subrarr;	U+02979
subset;	U+02282
subseteq;	U+02286
subseteqq;	U+02AC5
subsetneq;	U+0228A
subsetneqq;	U+02ACB
subsim;	U+02AC7
subsub;	U+02AD5
subsup;	U+02AD3
succ;	U+0227B
succapprox;	U+02AB8
succcurlyeq;	U+0227D
succeq;	U+02AB0
succnapprox;	U+02ABA
succneqq;	U+02AB6
succnsim;	U+022E9
succsim;	U+0227F
sum;	U+02211
sung;	U+0266A
sup1;	U+000B9
sup1	U+000B9
sup2;	U+000B2
sup2	U+000B2
sup3;	U+000B3
sup3	U+000B3
sup;	U+02283
supE;	U+02AC6
supdot;	U+02ABE
supdsub;	U+02AD8
supe;	U+02287
supedot;	U+02AC4
suphsub;	U+02AD7
suplarr;	U+0297B
supmult;	U+02AC2
supnE;	U+02ACC
supne;	U+0228B
supplus;	U+02AC0
supset;	U+02283
supseteq;	U+02287
supseteqq;	U+02AC6
supsetneq;	U+0228B
supsetneqq;	U+02ACC
supsim;	U+02AC8
supsub;	U+02AD4
supsup;	U+02AD6
swArr;	U+021D9
swarhk;	U+02926
swarr;	U+02199
swarrow;	U+02199

Name	Character
swnwar;	U+0292A
szlig;	U+000DF
szlig	U+000DF
target;	U+02316
tau;	U+003C4
tbrk;	U+023B4
tcaron;	U+00165
tcedil;	U+00163
tcy;	U+00442
tdot;	U+020DB
telrec;	U+02315
tfr;	U+1D531
there4;	U+02234
therefore;	U+02234
theta;	U+003B8
thetasym;	U+003D1
thetav;	U+003D1
thickapprox;	U+02248
thicksim;	U+0223C
thinsp;	U+02009
thkap;	U+02248
thksim;	U+0223C
thorn;	U+000FE
thorn	U+000FE
tilde;	U+002DC
times;	U+000D7
times	U+000D7
timesb;	U+022A0
timesbar;	U+02A31
timesd;	U+02A30
tint;	U+0222D
toea;	U+02928
top;	U+022A4
topbot;	U+02336
topcir;	U+02AF1
topf;	U+1D565
topfork;	U+02ADA
tosa;	U+02929
tprime;	U+02034
trade;	U+02122
triangle;	U+025B5
triangledown;	U+025BF
triangleleft;	U+025C3
trianglelefteq;	U+022B4
triangleq;	U+0225C
triangleright;	U+025B9
trianglerighteq;	U+022B5
tridot;	U+025EC
trie;	U+0225C
triminus;	U+02A3A
triplus;	U+02A39
trisb;	U+029CD
tritime;	U+02A3B
trpezium;	U+023E2
tscr;	U+1D4C9
tscy;	U+00446
tshcy;	U+0045B
tstrok;	U+00167
twixt;	U+0226C
twoheadleftarrow;	U+0219E
twoheadrightarrow;	U+021A0

Name	Character
uArr;	U+021D1
uHar;	U+02963
uacute;	U+000FA
uacute	U+000FA
uarr;	U+02191
ubrcy;	U+0045E
ubreve;	U+0016D
ucirc;	U+000FB
ucirc	U+000FB
ucy;	U+00443
udarr;	U+021C5
udblac;	U+00171
udhar;	U+0296E
ufisht;	U+0297E
ufr;	U+1D532
ugrave;	U+000F9
ugrave	U+000F9
uharl;	U+021BF
uharr;	U+021BE
uhblk;	U+02580
ulcorn;	U+0231C
ulcorner;	U+0231C
ulcrop;	U+0230F
ultri;	U+025F8
umacr;	U+0016B
uml;	U+000A8
uml	U+000A8
uogon;	U+00173
uopf;	U+1D566
uparrow;	U+02191
updownarrow;	U+02195
upharpoonleft;	U+021BF
upharpoonright;	U+021BE
uplus;	U+0228E
upsi;	U+003C5
upsih;	U+003D2
upsilon;	U+003C5
upuparrows;	U+021C8
urcorn;	U+0231D
urcorner;	U+0231D
urcrop;	U+0230E
uring;	U+0016F
urtri;	U+025F9
uscr;	U+1D4CA
utdot;	U+022F0
utilde;	U+00169
utri;	U+025B5
utrif;	U+025B4
uuarr;	U+021C8
uuml;	U+000FC
uuml	U+000FC
uwangle;	U+029A7
vArr;	U+021D5
vBar;	U+02AE8
vBarv;	U+02AE9
vDash;	U+022A8
vangrt;	U+0299C
varepsilon;	U+003B5
varkappa;	U+003F0
varnothing;	U+02205
varphi;	U+003C6

Name	Character
varpi;	U+003D6
varpropto;	U+0221D
varr;	U+02195
varrho;	U+003F1
varsigma;	U+003C2
vartheta;	U+003D1
vartriangleleft;	U+022B2
vartriangleright;	U+022B3
vcy;	U+00432
vdash;	U+022A2
vee;	U+02228
veebar;	U+022BB
veeeq;	U+0225A
vellip;	U+022EE
verbar;	U+0007C
vert;	U+0007C
vfr;	U+1D533
vltri;	U+022B2
vopf;	U+1D567
vprop;	U+0221D
vrtri;	U+022B3
vscr;	U+1D4CB
vzigzag;	U+0299A
wcirc;	U+00175
wedbar;	U+02A5F
wedge;	U+02227
wedgeq;	U+02259
weierp;	U+02118
wfr;	U+1D534
wopf;	U+1D568
wp;	U+02118
wr;	U+02240
wreath;	U+02240
wscr;	U+1D4CC
xcap;	U+022C2
xcirc;	U+025EF
xcup;	U+022C3
xdtri;	U+025BD
xfr;	U+1D535
xhArr;	U+027FA
xharr;	U+027F7
xi;	U+003BE
xlArr;	U+027F8
xlarr;	U+027F5
xmap;	U+027FC
xnis;	U+022FB
xodot;	U+02A00
xopf;	U+1D569
xoplus;	U+02A01
xotime;	U+02A02
xrArr;	U+027F9
xrarr;	U+027F6
xscr;	U+1D4CD
xsqcup;	U+02A06
xuplus;	U+02A04
xutri;	U+025B3
xvee;	U+022C1
xwedge;	U+022C0
yacute;	U+000FD
yacute	U+000FD
yacy;	U+0044F
ycirc;	U+00177
ycy;	U+0044B
yen;	U+000A5
yen	U+000A5
yfr;	U+1D536
yicy;	U+00457
yopf;	U+1D56A
yscr;	U+1D4CE
yucy;	U+0044E
yuml;	U+000FF
yuml	U+000FF
zacute;	U+0017A
zcaron;	U+0017E
zcy;	U+00437
zdot;	U+0017C
zeetrf;	U+02128
zeta;	U+003B6
zfr;	U+1D537
zhcy;	U+00436
zigrarr;	U+021DD
zopf;	U+1D56B
zscr;	U+1D4CF
zwj;	U+0200D
zwnj;	U+0200C

Name	Character

9 The XHTML syntax

Note: This section only describes the rules for XML resources. Rules for `text/html` resources are discussed in the section above entitled "The HTML syntax".

9.1 Writing XHTML documents

The syntax for using HTML with XML, whether in XHTML documents or embedded in other XML documents, is defined in the XML and Namespaces in XML specifications. [XML] [XMLNS]

This specification does not define any syntax-level requirements beyond those defined for XML proper.

XML documents may contain a `DOCTYPE` if desired, but this is not required to conform to this specification. This specification does not define a public or system identifier, nor provide a format DTD.

Note: According to the XML specification, XML processors are not guaranteed to process the external DTD subset referenced in the DOCTYPE. This means, for example, that using entity references for characters in XHTML documents is unsafe if they are defined in an external file (except for `<`, `>`, `&`, `"` and `'`).

9.2 Parsing XHTML documents

This section describes the relationship between XML and the DOM, with a particular emphasis on how this interacts with HTML.

An **XML parser**, for the purposes of this specification, is a construct that follows the rules given in the XML specification to map a string of bytes or characters into a `Document` object.

An XML parser is either associated with a `Document` object when it is created, or creates one implicitly.

This `Document` must then be populated with DOM nodes that represent the tree structure of the input passed to the parser, as defined by the XML specification, the Namespaces in XML specification, and the DOM Core specification. DOM mutation events must not fire for the operations that the XML parser performs on the `Document`'s tree, but the user agent must act as if elements and attributes were individually appended and set respectively so as to trigger rules in this specification regarding what happens when an element in inserted into a document or has its attributes set. [XML] [XMLNS] [DOMCORE] [DOMEVENTS]

Certain algorithms in this specification **spoon-feed the parser** characters one string at a time. In such cases, the XML parser must act as it would have if faced with a single string consisting of the concatenation of all those characters.

When an XML parser creates a `script` element, it must be marked as being "parser-inserted". If the parser was originally created for the XML fragment parsing algorithm, then the element must be marked as "already executed" also. When the element's end tag is parsed, the user agent must run the `script` element. If this causes there to be a pending external script, then the user agent must pause until that script has completed loading, and then execute it.

> ***Note: Since the `document.write()` API is not available for XML documents, much of the complexity in the HTML parser is not needed in the XML parser.***

When an XML parser reaches the end of its input, it must stop parsing, following the same rules as the HTML parser.

9.3 Serializing XHTML fragments

The **XML fragment serialization algorithm** for a `Document` or `Element` node either returns a fragment of XML that represents that node or raises an exception.

For `Document`s, the algorithm must return a string in the form of a document entity, if none of the error cases below apply.

For `Element`s, the algorithm must return a string in the form of an internal general parsed entity, if none of the error cases below apply.

In both cases, the string returned must be XML namespace-well-formed and must be an isomorphic serialization of all of that node's child nodes, in tree order. User agents may adjust prefixes and namespace declarations in the serialization (and indeed might be forced to do so in some cases to obtain namespace-well-formed XML).

For `Element`s, if any of the elements in the serialization are in no namespace, the default namespace in scope for those elements must be explicitly declared as the empty string. (This doesn't apply in the `Document` case.) [XML] [XMLNS]

If any of the following error cases are found in the DOM subtree being serialized, then the algorithm raises an `INVALID_STATE_ERR` exception instead of returning a string:

- A `Document` node with no child element nodes.
- A `DocumentType` node that has an external subset public identifier that contains characters that are not matched by the XML `PubidChar` production. [XML]
- A `DocumentType` node that has an external subset system identifier that contains both a U+0022 QUOTATION MARK (") and a U+0027 APOSTROPHE (').
- A node with a prefix or local name containing a U+003A COLON (":").
- An `Attr` node, `Text` node, `CDATASection` node, `Comment` node, or `ProcessingInstruction` node whose data contains characters that are not matched by the XML `Char` production.

[XML]

- A `CDATASection` node whose data contains the string "`]]>`".
- A `Comment` node whose data contains two adjacent U+002D HYPHEN-MINUS (-) characters or ends with such a character.
- A `ProcessingInstruction` node whose target name is an ASCII case-insensitive match for the string "`xml`".
- A `ProcessingInstruction` node whose target name contains a U+003A COLON (":").
- A `ProcessingInstruction` node whose data contains the string "`?>`".

Note: These are the only ways to make a DOM unserializable. The DOM enforces all the other XML constraints; for example, trying to set an attribute with a name that contains an equals sign (=) will raised an `INVALID_CHARACTER_ERR` exception.

9.4 Parsing XHTML fragments

The **XML fragment parsing algorithm** for either returns a `Document` or raises a `SYNTAX_ERR` exception. Given a string *input* and an optional context element *context*, the algorithm is as follows:

1. Create a new XML parser.
2. If there is a *context* element, feed the parser just created the string corresponding to the start tag of that element, declaring all the namespace prefixes that are in scope on that element in the DOM, as well as declaring the default namespace (if any) that is in scope on that element in the DOM.

   A namespace prefix is in scope if the DOM Core `lookupNamespaceURI()` method on the element would return a non-null value for that prefix.

   The default namespace is the namespace for which the DOM Core `isDefaultNamespace()` method on the element would return true.
3. Feed the parser just created the string *input*.
4. If there is a *context* element, feed the parser just created the string corresponding to the end tag of that element.
5. If there is an XML well-formedness or XML namespace well-formedness error, then raise a `SYNTAX_ERR` exception and abort these steps.
6. If there is a *context* element, then return the child nodes of the root element of the resulting `Document`, in tree order.

   Otherwise, return the children of the `Document` object, in tree order.

10 Rendering

User agents are not required present HTML documents in any particular way. However, this section provides a set of suggestions for rendering HTML documents that, if followed, are likely to lead to a user experience that closely resembles the experience intended by the documents' authors. So as to avoid confusion regarding the normativity of this section, RFC2119 terms have not been used. Instead, the term "expected" is used to indicate behavior that will lead to this experience.

10.1 Introduction

In general, user agents are expected to support CSS, and many of the suggestions in this section are expressed in CSS terms. User agents that use other presentation mechanisms can derive their expected behavior by translating from the CSS rules given in this section.

In the absence of style-layer rules to the contrary (e.g. author style sheets), user agents are expected to render an element so that it conveys to the user the meaning that the element **represents**, as described by this specification.

The suggestions in this section generally assume a visual output medium with a resolution of 96dpi or greater, but HTML is intended to apply to multiple media (it is a *media-independent* language). User agents are encouraged to adapt the suggestions in this section to their target media.

10.2 The CSS user agent style sheet and presentational hints

10.2.1 Introduction

The CSS rules given in these subsections are, unless otherwise specified, expected to be used as part of the user-agent level style sheet defaults for all documents that contain HTML elements.

Some rules are intended for the author-level zero-specificity presentational hints part of the CSS cascade; these are explicitly called out as **presentational hints**.

Some of the rules regarding left and right margins are given here as appropriate for elements whose 'direction' property is 'ltr', and are expected to be flipped around on elements whose 'direction' property is 'rtl'. These are marked "**LTR-specific**".

When the text below says that an attribute *attribute* on an element *element* **maps to the pixel length property** (or properties) *properties*, it means that if *element* has an attribute *attribute* set, and parsing that attribute's value using the rules for parsing non-negative integers doesn't generate an error, then the user agent is expected to use the parsed value as a pixel length for a presentational hint for *properties*.

When the text below says that an attribute *attribute* on an element *element* **maps to the dimension property** (or properties) *properties*, it means that if *element* has an attribute *attribute* set, and parsing that attribute's value using the rules for parsing dimension values

doesn't generate an error, then the user agent is expected to use the parsed dimension as the value for a presentational hint for *properties*, with the value given as a pixel length if the dimension was an integer, and with the value given as a percentage if the dimension was a percentage.

10.2.2 Display types

```
@namespace url(http://www.w3.org/1999/xhtml);

[hidden], area, audio:not([controls]), base, basefont, command,
datalist, head, input[type=hidden], link, menu[type=context], meta,
noembed, noframes, param, script, source, style, title {
  display: none;
}

address, article, aside, blockquote, body, center, dd, dialog, dir,
div, dl, dt, figure, footer, form, h1, h2, h3, h4, h5, h6, header, hr,
html, legend, listing, menu, nav, ol, p, plaintext, pre, rp, section,
ul, xmp { display: block; }

table { display: table; }
caption { display: table-caption; }
colgroup { display: table-column-group; }
col { display: table-column; }
thead { display: table-header-group; }
tbody { display: table-row-group; }
tfoot { display: table-footer-group; }
tr { display: table-row; }
td, th { display: table-cell; }

li { display: list-item; }

ruby { display: ruby; }
rt { display: ruby-text; }
```

For the purposes of the CSS table model, the `col` element is to be treated as if it was present as many times as its `span` attribute specifies.

For the purposes of the CSS table model, the `colgroup` element, if it contains no `col` element, is to be treated as if it had as many such children as its `span` attribute specifies.

For the purposes of the CSS table model, the `colspan` and `rowspan` attributes on `td` and `th` elements are expected to provide the *special knowledge* regarding cells spanning rows and columns.

For the purposes of the CSS ruby model, runs of descendants of `ruby` elements that are not `rt` or `rp` elements are expected to be wrapped in anonymous boxes whose 'display' property has the value 'ruby-base'.

User agents that do not support correct ruby rendering are expected to render parentheses around the text of `rt` elements in the absence of `rp` elements.

The `br` element is expected to render as if its contents were a single U+000A LINE FEED (LF) character and its 'white-space' property was 'pre'.

The user agent is expected to hide `noscript` elements for whom scripting is disabled, irrespective of CSS rules.

10.2.3 Margins and padding

```
@namespace url(http://www.w3.org/1999/xhtml);

article, aside, blockquote, dir, dl, figure, listing, menu, nav, ol,
p, plaintext, pre, section, ul, xmp {
  margin-top: 1em; margin-bottom: 1em;
}

dir dir, dir dl, dir menu, dir ol, dir ul,
dl dir, dl dl, dl menu, dl ol, dl ul,
menu dir, menu dl, menu menu, menu ol, menu ul,
ol dir, ol dl, ol menu, ol ol, ol ul,
ul dir, ul dl, ul menu, ul ol, ul ul {
  margin-top: 0; margin-bottom: 0;
}

h1 { margin-top: 0.67em; margin-bottom; 0.67em; }
h2 { margin-top: 0.83em; margin-bottom; 0.83em; }
h3 { margin-top: 1.00em; margin-bottom; 1.00em; }
h4 { margin-top: 1.33em; margin-bottom; 1.33em; }
h5 { margin-top: 1.67em; margin-bottom; 1.67em; }
h6 { margin-top: 2.33em; margin-bottom; 2.33em; }

dd { margin-left: 40px; } /* LTR-specific: use 'margin-right' for rtl elements */
dir, menu, ol, ul { padding-left: 40px; } /* LTR-specific: use 'padding-right' for
blockquote, figure { margin-left: 40px; margin-right: 40px; }

table { border-spacing: 2px; border-collapse: separate; }
td, th { padding: 1px; }
```

For each property in the table below, given a body element, the first attribute that exists maps to the pixel length property on the body element. If none of the attributes for a property are found, or if the value of the attribute that was found cannot be parsed successfully, then a default value of 8px is expected to be used for that property instead.

Property	Source
'margin-top'	body element's marginheight attribute
	The body element's container frame element's marginheight attribute
	body element's topmargin attribute
'margin-right'	body element's marginwidth attribute
	The body element's container frame element's marginwidth attribute
	body element's rightmargin attribute
'margin-bottom'	body element's marginheight attribute
	The body element's container frame element's marginheight attribute
	body element's topmargin attribute
'margin-left'	body element's marginwidth attribute
	The body element's container frame element's marginwidth attribute
	body element's rightmargin attribute

If the body element's Document's browsing context is a nested browsing context, and the browsing context container of that nested browsing context is a frame or iframe element, then the the **container frame element** of the body element is that frame or iframe element. Otherwise, there is no container frame element.

If the `Document` has a root element, and the `Document`'s browsing context is a nested browsing context, and the browsing context container of that nested browsing context is a `frame` or `iframe` element, and that element has a `scrolling` attribute, then the user agent is expected to compare the value of the attribute in an ASCII case-insensitive manner to the values in the first column of the following table, and if one of them matches, then the user agent is expected to treat that attribute as a presentational hint for the aforementioned root element's 'overflow' property, setting it to the value given in the corresponding cell on the same row in the second column:

Attribute value	'overflow' value
`on`	'scroll'
`scroll`	'scroll'
`yes`	'scroll'
`off`	'hidden'
`noscroll`	'hidden'
`no`	'hidden'
`auto`	'auto'

The `table` element's `cellspacing` attribute maps to the pixel length property 'border-spacing' on the element.

The `table` element's `cellpadding` attribute maps to the pixel length properties 'padding-top', 'padding-right', 'padding-bottom', and 'padding-left' of any `td` and `th` elements that have corresponding cells in the table corresponding to the `table` element.

The `table` element's `hspace` attribute maps to the dimension properties 'margin-left' and 'margin-right' on the `table` element.

The `table` element's `vspace` attribute maps to the dimension properties 'margin-top' and 'margin-bottom' on the `table` element.

The `table` element's `height` attribute maps to the dimension property 'height' on the `table` element.

The `table` element's `width` attribute maps to the dimension property 'width' on the `table` element.

The `col` element's `width` attribute maps to the dimension property 'width' on the `col` element.

The `tr` element's `height` attribute maps to the dimension property 'height' on the `tr` element.

The `td` and `th` elements' `height` attributes map to the dimension property 'height' on the element.

The `td` and `th` elements' `width` attributes map to the dimension property 'width' on the element.

In quirks mode, the following rules are also expected to apply:

```
@namespace url(http://www.w3.org/1999/xhtml);

form { margin-bottom: 1em; }
```

When a `Document` is in quirks mode, margins on HTML elements that collapse with the top or bottom of the initial containing block, or the top of bottom of `td` or `th` elements, are expected to be collapsed to zero.

10.2.4 Alignment

```
@namespace url(http://www.w3.org/1999/xhtml);

thead, tbody, tfoot, table > tr { vertical-align: middle; }
tr, td, th { vertical-align: inherit; }
sub { vertical-align: sub; }
sup { vertical-align: super; }
th { text-align: center; }
```

The following rules are also expected to apply, as presentational hints:

```
@namespace url(http://www.w3.org/1999/xhtml);

table[align=left] { float: left; }
table[align=right] { float: right; }
table[align=center], table[align=abscenter],
table[align=abdmiddle], table[align=middle] {
  margin-left: auto; margin-right: auto;
}

caption[align=bottom] { caption-side: bottom; }
p[align=left], h1[align=left], h2[align=left], h3[align=left],
h4[align=left], h5[align=left], h6[align=left] {
  text-align: left;
}
p[align=right], h1[align=right], h2[align=right], h3[align=right],
h4[align=right], h5[align=right], h6[align=right] {
  text-align: right;
}
p[align=center], h1[align=center], h2[align=center], h3[align=center],
h4[align=center], h5[align=center], h6[align=center] {
  text-align: center;
}
p[align=justify], h1[align=justify], h2[align=justify], h3[align=justify],
h4[align=justify], h5[align=justify], h6[align=justify] {
  text-align: justify;
}
col[valign=top], thead[valign=top], tbody[valign=top],
tfoot[valign=top], tr[valign=top], td[valign=top], th[valign=top] {
  vertial-align: top;
}
col[valign=middle], thead[valign=middle], tbody[valign=middle],
tfoot[valign=middle], tr[valign=middle], td[valign=middle], th[valign=middle] {
  vertial-align: middle;
}
col[valign=bottom], thead[valign=bottom], tbody[valign=bottom],
tfoot[valign=bottom], tr[valign=bottom], td[valign=bottom], th[valign=bottom] {
  vertial-align: bottom;
}
col[valign=baseline], thead[valign=baseline], tbody[valign=baseline],
tfoot[valign=baseline], tr[valign=baseline], td[valign=baseline], th[valign=baseli
  vertial-align: baseline;
```

```
}
```

The center element, the caption element unless specified otherwise below, and the div element when its align attribute's value is an ASCII case-insensitive match for the string "center", are expected to center text within themselves, as if they had their 'text-align' property set to 'center' in a presentational hint, and to align descendants to the center.

The div, caption, thead, tbody, tfoot, tr, td, and th elements, when they have an align attribute whose value is an ASCII case-insensitive match for the string "left", are expected to left-align text within themselves, as if they had their 'text-align' property set to 'left' in a presentational hint, and to align descendants to the left.

The div, caption, thead, tbody, tfoot, tr, td, and th elements, when they have an align attribute whose value is an ASCII case-insensitive match for the string "right", are expected to right-align text within themselves, as if they had their 'text-align' property set to 'right' in a presentational hint, and to align descendants to the right.

The div, caption, thead, tbody, tfoot, tr, td, and th elements, when they have an align attribute whose value is an ASCII case-insensitive match for the string "justify", are expected to full-justify text within themselves, as if they had their 'text-align' property set to 'justify' in a presentational hint, and to align descendants to the left.

When a user agent is to **align descendants** of a node, the user agent is expected to align only those descendants that have both their 'margin-left' and 'margin-right' properties computing to a value other than 'auto', that are over-constrained and that have one of those two margins with a used value forced to a greater value, and that do not themselves have an applicable align attribute.

10.2.5 Fonts and colors

```
@namespace url(http://www.w3.org/1999/xhtml);

address, cite, dfn, em, i, var { font-style: italic; }
b, strong, th { font-weight: bold; }
code, kbd, listing, plaintext, pre, samp, tt, xmp { font-family: monospace; }
h1 { font-size: 2.00em; font-weight: bold; }
h2 { font-size: 1.50em; font-weight: bold; }
h3 { font-size: 1.17em; font-weight: bold; }
h4 { font-size: 1.00em; font-weight: bold; }
h5 { font-size: 0.83em; font-weight: bold; }
h6 { font-size: 0.67em; font-weight: bold; }
big { font-size: larger; }
small, sub, sup { font-size: smaller; }
sub, sup { line-height: normal; }

:link { color: blue; }
:visited { color: purple; }
mark { background: yellow; color: black; }

table, td, th { border-color: gray; }
thead, tbody, tfoot, tr { border-color: inherit; }
table[rules=none], table[rules=groups], table[rules=rows],
table[rules=cols], table[rules=all], table[frames=void],
table[frames=above], table[frames=below], table[frames=hsides],
table[frames=lhs], table[frames=rhs], table[frames=vsides],
table[frames=box], table[frames=border],
table[rules=none] > tr > td, table[rules=none] > tr > th,
table[rules=groups] > tr > td, table[rules=groups] > tr > th,
```

```
table[rules=rows] > tr > td, table[rules=rows] > tr > th,
table[rules=cols] > tr > td, table[rules=cols] > tr > th,
table[rules=all] > tr > td, table[rules=all] > tr > th,
table[frames=void] > tr > td, table[frames=void] > tr > th,
table[frames=above] > tr > td, table[frames=above] > tr > th,
table[frames=below] td, table[frames=below] > tr > th,
table[frames=hsides] > tr > td, table[frames=hsides] > tr > th,
table[frames=lhs] > tr > td, table[frames=lhs] > tr > th,
table[frames=rhs] > tr > td, table[frames=rhs] > tr > th,
table[frames=vsides] > tr > td, table[frames=vsides] > tr > th,
table[frames=box] > tr > td, table[frames=box] > tr > th,
table[frames=border] > tr > td, table[frames=border] > tr > th,
table[rules=none] > thead > tr > td, table[rules=none] > thead > tr > th,
table[rules=groups] > thead > tr > td, table[rules=groups] > thead > tr > th,
table[rules=rows] > thead > tr > td, table[rules=rows] > thead > tr > th,
table[rules=cols] > thead > tr > td, table[rules=cols] > thead > tr > th,
table[rules=all] > thead > tr > td, table[rules=all] > thead > tr > th,
table[frames=void] > thead > tr > td, table[frames=void] > thead > tr > th,
table[frames=above] > thead > tr > td, table[frames=above] > thead > tr > th,
table[frames=below] td, table[frames=below] > thead > tr > th,
table[frames=hsides] > thead > tr > td, table[frames=hsides] > thead > tr > th,
table[frames=lhs] > thead > tr > td, table[frames=lhs] > thead > tr > th,
table[frames=rhs] > thead > tr > td, table[frames=rhs] > thead > tr > th,
table[frames=vsides] > thead > tr > td, table[frames=vsides] > thead > tr > th,
table[frames=box] > thead > tr > td, table[frames=box] > thead > tr > th,
table[frames=border] > thead > tr > td, table[frames=border] > thead > tr > th,
table[rules=none] > tbody > tr > td, table[rules=none] > tbody > tr > th,
table[rules=groups] > tbody > tr > td, table[rules=groups] > tbody > tr > th,
table[rules=rows] > tbody > tr > td, table[rules=rows] > tbody > tr > th,
table[rules=cols] > tbody > tr > td, table[rules=cols] > tbody > tr > th,
table[rules=all] > tbody > tr > td, table[rules=all] > tbody > tr > th,
table[frames=void] > tbody > tr > td, table[frames=void] > tbody > tr > th,
table[frames=above] > tbody > tr > td, table[frames=above] > tbody > tr > th,
table[frames=below] td, table[frames=below] > tbody > tr > th,
table[frames=hsides] > tbody > tr > td, table[frames=hsides] > tbody > tr > th,
table[frames=lhs] > tbody > tr > td, table[frames=lhs] > tbody > tr > th,
table[frames=rhs] > tbody > tr > td, table[frames=rhs] > tbody > tr > th,
table[frames=vsides] > tbody > tr > td, table[frames=vsides] > tbody > tr > th,
table[frames=box] > tbody > tr > td, table[frames=box] > tbody > tr > th,
table[frames=border] > tbody > tr > td, table[frames=border] > tbody > tr > th,
table[rules=none] > tfoot > tr > td, table[rules=none] > tfoot > tr > th,
table[rules=groups] > tfoot > tr > td, table[rules=groups] > tfoot > tr > th,
table[rules=rows] > tfoot > tr > td, table[rules=rows] > tfoot > tr > th,
table[rules=cols] > tfoot > tr > td, table[rules=cols] > tfoot > tr > th,
table[rules=all] > tfoot > tr > td, table[rules=all] > tfoot > tr > th,
table[frames=void] > tfoot > tr > td, table[frames=void] > tfoot > tr > th,
table[frames=above] > tfoot > tr > td, table[frames=above] > tfoot > tr > th,
table[frames=below] td, table[frames=below] > tfoot > tr > th,
table[frames=hsides] > tfoot > tr > td, table[frames=hsides] > tfoot > tr > th,
table[frames=lhs] > tfoot > tr > td, table[frames=lhs] > tfoot > tr > th,
table[frames=rhs] > tfoot > tr > td, table[frames=rhs] > tfoot > tr > th,
table[frames=vsides] > tfoot > tr > td, table[frames=vsides] > tfoot > tr > th,
table[frames=box] > tfoot > tr > td, table[frames=box] > tfoot > tr > th,
table[frames=border] > tfoot > tr > td, table[frames=border] > tfoot > tr > th {
  border-color: black;
}
```

The initial value for the 'color' property is expected to be black. The initial value for the 'background-color' property is expected to be 'transparent'. The canvas's background is expected to be white.

The `article`, `aside`, `nav`, and `section` elements are expected to affect the styling of `h1` elements, based on the nesting depth. If *x* is a selector that matches elements that are either

article, aside, nav, or section elements, then the following rules capture what is expected:

```
@namespace url(http://www.w3.org/1999/xhtml);

x h1 { font-size: 1.50em; }
x x h1 { font-size: 1.17em; }
x x x h1 { font-size: 1.00em; }
x x x x h1 { font-size: 0.83em; }
x x x x x h1 { font-size: 0.67em; }
```

When a body, table, thead, tbody, tfoot, tr, td, or th element has a background attribute set to a non-empty value, the new value is expected to be resolved relative to the element, and if this is successful, the user agent is expected to treat the attribute as a presentational hint setting the element's 'background-image' property to the resulting absolute URL.

When a body, table, thead, tbody, tfoot, tr, td, or th element has a bgcolor attribute set, the new value is expected to be parsed using the rules for parsing a legacy color value, and the user agent is expected to treat the attribute as a presentational hint setting the element's 'background-color' property to the resulting color.

When a body element has a text attribute, its value is expected to be parsed using the rules for parsing a legacy color value, and the user agent is expected to treat the attribute as a presentational hint setting the element's 'color' property to the resulting color.

When a body element has a link attribute, its value is expected to be parsed using the rules for parsing a legacy color value, and the user agent is expected to treat the attribute as a presentational hint setting the 'color' property of any element in the Document matching the ':link' pseudo-class to the resulting color.

When a body element has a vlink attribute, its value is expected to be parsed using the rules for parsing a legacy color value, and the user agent is expected to treat the attribute as a presentational hint setting the 'color' property of any element in the Document matching the ':visited' pseudo-class to the resulting color.

When a body element has a alink attribute, its value is expected to be parsed using the rules for parsing a legacy color value, and the user agent is expected to treat the attribute as a presentational hint setting the 'color' property of any element in the Document matching the ':active' pseudo-class and either the ':link' pseudo-class or the ':visited' pseudo-class to the resulting color.

When a table element has a bordercolor attribute, its value is expected to be parsed using the rules for parsing a legacy color value, and the user agent is expected to treat the attribute as a presentational hint setting the element's 'border-top-color', 'border-right-color', 'border-bottom-color', and 'border-right-color' properties to the resulting color.

When a font element has a color attribute, its value is expected to be parsed using the rules for parsing a legacy color value, and the user agent is expected to treat the attribute as a presentational hint setting the element's 'color' property to the resulting color.

When a font element has a face attribute, the user agent is expected to treat the attribute as a presentational hint setting the element's 'font-family' property to the attribute's value.

When a font element has a pointsize attribute, the user agent is expected to parse that

attribute's value using the rules for parsing non-negative integers, and if this doesn't generate an error, then the user agent is expected to use the parsed value as a *point* length for a presentational hint for the 'font-size' property on the element.

When a `font` element has a `size` attribute, the user agent is expected to use the following steps to treat the attribute as a presentational hint setting the element's 'font-size' property:

1. Let *input* be the attribute's value.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Skip whitespace.
4. If *position* is past the end of *input*, there is no presentational hint. Abort these steps.
5. If the character at *position* is a U+002B PLUS SIGN character (+), then let *mode* be *relative-plus*, and advance *position* to the next character. Otherwise, if the character at *position* is a U+002D HYPHEN-MINUS character (-), then let *mode* be *relative-minus*, and advance *position* to the next character. Otherwise, let *mode* be *absolute*.
6. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), and let the resulting sequence be *digits*.
7. If *digits* is the empty string, there is no presentational hint. Abort these steps.
8. Interpret *digits* as a base-ten integer. Let *size* be the resulting number.
9. If *mode* is is *relative-plus*, then increment *value* by 3. If *mode* is is *relative-minus*, then decrement *value* by 3.
10. If *value* is greater than 7, let it be 7.
11. If *value* is less than 1, let it be 1.
12. Set 'font-size' to the keyword corresponding to the value of *value* according to the following table:

value	'font-size' keyword	Notes
1	xx-small	
2	small	
3	medium	
4	large	
5	x-large	
6	xx-large	
7	xxx-large	*see below*

The 'xxx-large' value is a non-CSS value used here to indicate a font size one "step" larger than 'xx-large'.

10.2.6 Punctuation and decorations

```
@namespace url(http://www.w3.org/1999/xhtml);

:link, :visited, ins, u { text-decoration: underline; }
abbr[title], acronym[title] { text-decoration: dotted underline; }
del, s, strike { text-decoration: line-through; }
blink { text-decoration: blink; }

q:before { content: open-quote; }
q:after { content: close-quote; }

nobr { white-space: nowrap; }
listing, plaintext, pre, xmp { white-space: pre; }

ol { list-style-type: decimal; }

dir, menu, ul {
  list-style-type: disc;
}

dir dl, dir menu, dir ul,
menu dl, menu menu, menu ul,
ol dl, ol menu, ol ul,
ul dl, ul menu, ul ul {
  list-style-type: circle;
}

dir dir dl, dir dir menu, dir dir ul,
dir menu dl, dir menu menu, dir menu ul,
dir ol dl, dir ol menu, dir ol ul,
dir ul dl, dir ul menu, dir ul ul,
menu dir dl, menu dir menu, menu dir ul,
menu menu dl, menu menu menu, menu menu ul,
menu ol dl, menu ol menu, menu ol ul,
menu ul dl, menu ul menu, menu ul ul,
ol dir dl, ol dir menu, ol dir ul,
ol menu dl, ol menu menu, ol menu ul,
ol ol dl, ol ol menu, ol ol ul,
ol ul dl, ol ul menu, ol ul ul,
ul dir dl, ul dir menu, ul dir ul,
ul menu dl, ul menu menu, ul menu ul,
ul ol dl, ul ol menu, ul ol ul,
ul ul dl, ul ul menu, ul ul ul {
  list-style-type: square;
}

table { border-style: outset; }
td, th { border-style: inset; }

[dir=ltr] { direction: ltr; unicode-bidi: embed; }
[dir=rtl] { direction: rtl; unicode-bidi: embed; }
bdo[dir=ltr], bdo[dir=rtl] { unicode-bidi: bidi-override; }
```

In addition, rules setting the 'quotes' property appropriately for the locales and languages understood by the user are expected to be present.

The following rules are also expected to apply, as presentational hints:

```
@namespace url(http://www.w3.org/1999/xhtml);

td[nowrap], th[nowrap] { white-space: nowrap; }
pre[wrap] { white-space: pre-wrap; }

br[clear=left] { clear: left; }
br[clear=right] { clear: right; }
br[clear=all], br[clear=both] { clear: both; }
```

```
ol[type=1], li[type=1] { list-style-type: decimal; }
ol[type=a], li[type=a] { list-style-type: lower-alpha; }
ol[type=A], li[type=A] { list-style-type: upper-alpha; }
ol[type=i], li[type=i] { list-style-type: lower-roman; }
ol[type=I], li[type=I] { list-style-type: upper-roman; }
ul[type=disc], li[type=disc] { list-style-type: disc; }
ul[type=circle], li[type=circle] { list-style-type: circle; }
ul[type=square], li[type=square] { list-style-type: square; }

table[rules=none], table[rules=groups], table[rules=rows],
table[rules=cols], table[rules=all] {
  border-style: none;
  border-collapse: collapse;
}

table[frames=void] { border-style: hidden hidden hidden hidden; }
table[frames=above] { border-style: solid hidden hidden hidden; }
table[frames=below] { border-style: hidden hidden solid hidden; }
table[frames=hsides] { border-style: solid hidden solid hidden; }
table[frames=lhs] { border-style: hidden hidden hidden solid; }
table[frames=rhs] { border-style: hidden solid hidden hidden; }
table[frames=vsides] { border-style: hidden solid hidden solid; }
table[frames=box],
table[frames=border] { border-style: solid solid solid solid; }

table[frames=void] > tr > td, table[frames=void] > tr > th,
table[frames=above] > tr > td, table[frames=above] > tr > th,
table[frames=below] > tr > td, table[frames=below] > tr > th,
table[frames=hsides] > tr > td, table[frames=hsides] > tr > th,
table[frames=lhs] > tr > td, table[frames=lhs] > tr > th,
table[frames=rhs] > tr > td, table[frames=rhs] > tr > th,
table[frames=vsides] > tr > td, table[frames=vsides] > tr > th,
table[frames=box] > tr > td, table[frames=box] > tr > th,
table[frames=border] > tr > td, table[frames=border] > tr > th,
table[frames=void] > thead > tr > td, table[frames=void] > thead > tr > th,
table[frames=above] > thead > tr > td, table[frames=above] > thead > tr > th,
table[frames=below] > thead > tr > td, table[frames=below] > thead > tr > th,
table[frames=hsides] > thead > tr > td, table[frames=hsides] > thead > tr > th,
table[frames=lhs] > thead > tr > td, table[frames=lhs] > thead > tr > th,
table[frames=rhs] > thead > tr > td, table[frames=rhs] > thead > tr > th,
table[frames=vsides] > thead > tr > td, table[frames=vsides] > thead > tr > th,
table[frames=box] > thead > tr > td, table[frames=box] > thead > tr > th,
table[frames=border] > thead > tr > td, table[frames=border] > thead > tr > th,
table[frames=void] > tbody > tr > td, table[frames=void] > tbody > tr > th,
table[frames=above] > tbody > tr > td, table[frames=above] > tbody > tr > th,
table[frames=below] > tbody > tr > td, table[frames=below] > tbody > tr > th,
table[frames=hsides] > tbody > tr > td, table[frames=hsides] > tbody > tr > th,
table[frames=lhs] > tbody > tr > td, table[frames=lhs] > tbody > tr > th,
table[frames=rhs] > tbody > tr > td, table[frames=rhs] > tbody > tr > th,
table[frames=vsides] > tbody > tr > td, table[frames=vsides] > tbody > tr > th,
table[frames=box] > tbody > tr > td, table[frames=box] > tbody > tr > th,
table[frames=border] > tbody > tr > td, table[frames=border] > tbody > tr > th,
table[frames=void] > tfoot > tr > td, table[frames=void] > tfoot > tr > th,
table[frames=above] > tfoot > tr > td, table[frames=above] > tfoot > tr > th,
table[frames=below] > tfoot > tr > td, table[frames=below] > tfoot > tr > th,
table[frames=hsides] > tfoot > tr > td, table[frames=hsides] > tfoot > tr > th,
table[frames=lhs] > tfoot > tr > td, table[frames=lhs] > tfoot > tr > th,
table[frames=rhs] > tfoot > tr > td, table[frames=rhs] > tfoot > tr > th,
table[frames=vsides] > tfoot > tr > td, table[frames=vsides] > tfoot > tr > th,
table[frames=box] > tfoot > tr > td, table[frames=box] > tfoot > tr > th,
table[frames=border] > tfoot > tr > td, table[frames=border] > tfoot > tr > th {
  border-style: solid;
}

table[rules=none] > tr > td, table[rules=none] > tr > th,
table[rules=none] > thead > tr > td, table[rules=none] > thead > tr > th,
table[rules=none] > tbody > tr > td, table[rules=none] > tbody > tr > th,
table[rules=none] > tfoot > tr > td, table[rules=none] > tfoot > tr > th,
```

```
table[rules=groups] > tr > td, table[rules=groups] > tr > th,
table[rules=groups] > thead > tr > td, table[rules=groups] > thead > tr > th,
table[rules=groups] > tbody > tr > td, table[rules=groups] > tbody > tr > th,
table[rules=groups] > tfoot > tr > td, table[rules=groups] > tfoot > tr > th,
table[rules=rows] > tr > td, table[rules=rows] > tr > th,
table[rules=rows] > thead > tr > td, table[rules=rows] > thead > tr > th,
table[rules=rows] > tbody > tr > td, table[rules=rows] > tbody > tr > th,
table[rules=rows] > tfoot > tr > td, table[rules=rows] > tfoot > tr > th {
  border-style: none;
}

table[rules=groups] > colgroup, table[rules=groups] > thead,
table[rules=groups] > tbody, table[rules=groups] > tfoot {
  border-style: solid;
}

table[rules=rows] > tr, table[rules=rows] > thead > tr,
table[rules=rows] > tbody > tr, table[rules=rows] > tfoot > tr {
  border-style: solid;
}

table[rules=cols] > tr > td, table[rules=cols] > tr > th,
table[rules=cols] > thead > tr > td, table[rules=cols] > thead > tr > th,
table[rules=cols] > tbody > tr > td, table[rules=cols] > tbody > tr > th,
table[rules=cols] > tfoot > tr > td, table[rules=cols] > tfoot > tr > th {
  border-style: none solid none solid;
}

table[rules=all] > tr > td, table[rules=all] > tr > th,
table[rules=all] > thead > tr > td, table[rules=all] > thead > tr > th,
table[rules=all] > tbody > tr > td, table[rules=all] > tbody > tr > th,
table[rules=all] > tfoot > tr > td, table[rules=all] > tfoot > tr > th {
  border-style: solid;
}
```

When rendering `li` elements, user agents are expected to use the ordinal value of the `li` element to render the counter in the list item marker.

The `table` element's `border` attribute maps to the pixel length properties 'border-top-width', 'border-right-width', 'border-bottom-width', 'border-left-width' on the element. If the attribute is present but its value cannot be parsed successfully, a default value of 1px is expected to be used for that property instead.

10.2.7 Resetting rules for inherited properties

The following rules are also expected to be in play, resetting certain properties to block inheritance by default.

```
@namespace url(http://www.w3.org/1999/xhtml);

table, input, select, option, optgroup, button {
  text-indent: initial;
}
```

In quirks mode, the following rules are also expected to apply:

```
@namespace url(http://www.w3.org/1999/xhtml);

table {
  font-weight: initial;
  font-style: initial;
```

```
  font-variant: initial;
  font-size: initial;
  line-height: initial;
  white-space: initial;
  text-align: initial;
}

input { box-sizing: border-box; }
```

10.2.8 The hr element

```
@namespace url(http://www.w3.org/1999/xhtml);

hr { color: gray; border-style: inset; border-width: 1px; }
```

The following rules are also expected to apply, as presentational hints:

```
@namespace url(http://www.w3.org/1999/xhtml);

hr[align=left] { margin-left: 0; margin-right: auto; }
hr[align=right] { margin-left: auto; margin-right: 0; }
hr[align=center] { margin-left: auto; margin-right: auto; }
hr[color], hr[noshade] { border-style: solid; }
```

If an hr element has either a color attribute or a noshade attribute, and furthermore also has a size attribute, and parsing that attribute's value using the rules for parsing non-negative integers doesn't generate an error, then the user agent is expected to use the parsed value divided by two as a pixel length for presentational hints for the properties 'border-top-width', 'border-right-width', 'border-bottom-width', and 'border-left-width' on the element.

Otherwise, if an hr element has neither a color attribute nor a noshade attribute, but does have a size attribute, and parsing that attribute's value using the rules for parsing non-negative integers doesn't generate an error, then: if the parsed value is one, then the user agent is expected to use the attribute as a presentational hint setting the element's 'border-bottom-width' to 0; otherwise, if the parsed value is greater than one, then the user agent is expected to use the parsed value minus two as a pixel length for presentational hints for the 'height' property on the element.

The width attribute on an hr element maps to the dimension property 'width' on the element.

When an hr element has a color attribute, its value is expected to be parsed using the rules for parsing a legacy color value, and the user agent is expected to treat the attribute as a presentational hint setting the element's 'color' property to the resulting color.

10.2.9 The fieldset element

```
@namespace url(http://www.w3.org/1999/xhtml);

fieldset {
  margin-left: 2px; margin-right: 2px;
  border: groove 2px ThreeDFace;
  padding: 0.35em 0.625em 0.75em;
}
```

The first legend element child of a fieldset element, if any, is expected to be rendered over

the top border edge of the `fieldset` element. If the `legend` element in question has an `align` attribute, and its value is an ASCII case-insensitive match for one of the strings in the first column of the following table, then the `legend` is expected to be rendered horizontally aligned over the border edge in the position given in the corresponding cell on the same row in the second column. If the attribute is absent or has a value that doesn't match any of the cases in the table, then the position is expected to be on the right if the 'direction' property on this element has a computed value of 'rtl', and on the left otherwise.

Attribute value	Alignment position
`left`	On the left
`right`	On the right
`center`	In the middle

10.3 Replaced elements

10.3.1 Embedded content

The `applet`, `canvas`, `embed`, `iframe`, and `video` elements are expected to be treated as replaced elements.

An `object` element that represents an image, plugin, or nested browsing context is expected to be treated as a replaced element. Other `object` elements are expected to be treated as ordinary elements in the rendering model.

The `audio` element, when it has a `controls` attribute, is expected to be treated as a replaced element about one line high, as wide as is necessary to expose the user agent's user interface features.

The `video` element's `controls` attribute is not expected to affect the size of the rendering; controls are expected to be overlaid with the page content without causing any layout changes, and are expected to disappear when the user does not need them.

> ***Note: Resizing `video` and `canvas` elements does not interrupt video playback or clear the canvas.***

The following CSS rules are expected to apply:

```
@namespace url(http://www.w3.org/1999/xhtml);

iframe { border: 2px inset; }
```

10.3.2 Images

When an `img` element or an `input` element when its `type` attribute is in the Image Button state represents an image, it is expected to be treated as a replaced element.

When an `img` element or an `input` element when its `type` attribute is in the Image Button state

does not represent an image, but the element already has instrinsic dimensions (e.g. from the dimension attributes or CSS rules), and either the user agent has reason to believe that the image will become *available* and be rendered in due course or the `Document` is in quirks mode, the element is expected to be treated as a replaced element whose content is the text that the element represents, if any, optionally alongside an icon indicating that the image is being obtained. For `input` elements, the text is expected to appear button-like to indicate that the element is a button.

When an `img` element represents some text and the user agent does not expect this to change, the element is expected to be treated as an inline element whose content is the text, optionally with an icon indicating that an image is missing.

When an `img` element represents nothing and the user agent does not expect this to change, the element is expected to not be rendered at all.

When an `img` element might be a key part of the content, but neither the image nor any kind of alternative text is available, and the user agent does not expect this to change, the element is expected to be treated as an inline element whose content is an icon indicating that an image is missing.

When an `input` element whose `type` attribute is in the Image Button state does not represent an image and the user agent does not expect this to change, the element is expected to be treated as a replaced element consisting of a button whose content is the element's alternative text. The intrinsic dimensions of the button are expected to be about one line in height and whatever width is necessary to render the text on one line.

The icons mentioned above are expected to be relatively small so as not to disrupt most text but be easily clickable. In a visual environment, for instance, icons could be 16 pixels by 16 pixels square, or 1em by 1em if the images are scalable. In an audio environment, the icon could be a short bleep. The icons are intended to indicate to the user that they can be used to get to whatever options the UA provides for images, and, where appropriate, are expected to provide access to the context menu that would have come up if the user interacted with the actual image.

The following CSS rules are expected to apply when the `Document` is in quirks mode:

```
@namespace url(http://www.w3.org/1999/xhtml);

img[align=left] { margin-right: 3px; }
img[align=right] { margin-left: 3px; }
```

10.3.3 Attributes for embedded content and images

The following CSS rules are expected to apply as presentational hints:

```
@namespace url(http://www.w3.org/1999/xhtml);

iframe[frameborder=0], iframe[frameborder=no] { border: none; }

applet[align=left], embed[align=left], iframe[align=left],
img[align=left], input[type=image][align=left], object[align=left] {
  float: left;
}
```

```
applet[align=right], embed[align=right], iframe[align=right],
img[align=right], input[type=image][align=right], object[align=right] {
  float: right;
}

applet[align=top], embed[align=top], iframe[align=top],
img[align=top], input[type=image][align=top], object[align=top] {
  vertical-align: top;
}

applet[align=bottom], embed[align=bottom], iframe[align=bottom],
img[align=bottom], input[type=image][align=bottom], object[align=bottom],
applet[align=baseline], embed[align=baseline], iframe[align=baseline],
img[align=baseline], input[type=image][align=baseline], object[align=baseline] {
  vertical-align: baseline;
}

applet[align=texttop], embed[align=texttop], iframe[align=texttop],
img[align=texttop], input[type=image][align=texttop], object[align=texttop] {
  vertical-align: text-top;
}

applet[align=absmiddle], embed[align=absmiddle], iframe[align=absmiddle],
img[align=absmiddle], input[type=image][align=absmiddle], object[align=absmiddle],
applet[align=abscenter], embed[align=abscenter], iframe[align=abscenter],
img[align=abscenter], input[type=image][align=abscenter], object[align=abscenter]
  vertical-align: middle;
}

applet[align=bottom], embed[align=bottom], iframe[align=bottom],
img[align=bottom], input[type=image][align=bottom],
object[align=bottom] {
  vertical-align: bottom;
}
```

When an `applet`, `embed`, `iframe`, `img`, or `object` element, or an `input` element whose `type` attribute is in the Image Button state, has an `align` attribute whose value is an ASCII case-insensitive match for the string "`center`" or the string "`middle`", the user agent is expected to act as if the element's 'vertical-align' property was set to a value that aligns the vertical middle of the element with the parent element's baseline.

The `hspace` attribute of `applet`, `embed`, `iframe`, `img`, or `object` elements, and `input` elements with a `type` attribute in the Image Button state, maps to the dimension properties 'margin-left' and 'margin-right' on the element.

The `vspace` attribute of `applet`, `embed`, `iframe`, `img`, or `object` elements, and `input` elements with a `type` attribute in the Image Button state, maps to the dimension properties 'margin-top' and 'margin-bottom' on the element.

When an `img` element, `object` element, or `input` element with a `type` attribute in the Image Button state is contained within a hyperlink and has a `border` attribute whose value, when parsed using the rules for parsing non-negative integers, is found to be a number greater than zero, the user agent is expected to use the parsed value for eight presentational hints: four setting the parsed value as a pixel length for the element's 'border-top-width', 'border-right-width', 'border-bottom-width', and 'border-left-width' properties, and four setting the element's 'border-top-style', 'border-right-style', 'border-bottom-style', and 'border-left-style' properties to the value 'solid'.

The `width` and `height` attributes on `applet`, `embed`, `iframe`, `img`, `object` or `video` elements, and `input` elements with a `type` attribute in the Image Button state, map to the dimension

properties 'width' and 'height' on the element respectively.

10.3.4 Image maps

Shapes on an image map are expected to act, for the purpose of the CSS cascade, as elements independent of the original `area` element that happen to match the same style rules but inherit from the `img` or `object` element.

For the purposes of the rendering, only the 'cursor' property is expected to have any effect on the shape.

> Thus, for example, if an `area` element has a `style` attribute that sets the 'cursor' property to 'help', then when the user designates that shape, the cursor would change to a Help cursor.

> Similarly, if an `area` element had a CSS rule that set its 'cursor' property to 'inherit' (or if no rule setting the 'cursor' property matched the element at all), the shape's cursor would be inherited from the `img` or `object` element of the image map, not from the parent of the `area` element.

10.3.5 Tool bars

When a `menu` element's `type` attribute is in the tool bar state, the element is expected to be treated as a replaced element with a height about two lines high and a width derived from the contents of the element.

The element is expected to have, by default, the appearance of a tool bar on the user agent's platform. It is expected to contain the menu that is built from the element.

> ...example with screenshot...

10.4 Bindings

10.4.1 Introduction

A number of elements have their rendering defined in terms of the 'binding' property. [BECSS]

The CSS snippets below set the 'binding' property to a user-agent-defined value, represented below by keywords like *bb*. The rules then described for these bindings are only expected to apply if the element's 'binding' property has not been overriden (e.g. by the author) to have another value.

Exactly how the bindings are implemented is not specified by this specification. User agents are encouraged to make their bindings set the 'appearance' CSS property appropriately to achieve platform-native appearances for widgets, and are expected to implement any relevant animations, etc, that are appropriate for the platform. [CSSUI]

The **converting a character width to pixels** algorithm, used by some of the bindings below,

returns (*size-1*)×*avg* + *max*, where *size* is the character width to convert, *avg* is the average character width of the primary font for the element for which the algorithm is being run, in pixels, and *max* is the maximum character width of that same font, also in pixels. (The element's 'letter-spacing' property does not affect the result.)

10.4.2 The `bb` element

```
@namespace url(http://www.w3.org/1999/xhtml);

bb:empty { binding: bb; }
```

When the *bb* binding applies to a `bb` element, the element is expected to render as an 'inline-block' box rendered as a button, about one line high, containing text derived from the element's `type` attribute in a user-agent-defined (and probably locale-specific) fashion.

10.4.3 The `button` element

```
@namespace url(http://www.w3.org/1999/xhtml);

button { binding: button; }
```

When the *button* binding applies to a `button` element, the element is expected to render as an 'inline-block' box rendered as a button whose contents are the contents of the element.

10.4.4 The `datagrid` element

This section will probably include details on how to render DATAGRID (including its pseudo-elements), drag-and-drop, etc, in a visual medium, in concert with CSS. Implementation experience is desired before this section is filled in.

10.4.5 The `details` element

```
@namespace url(http://www.w3.org/1999/xhtml);

details { binding: details; }
```

When the *details* binding applies to a `details` element, the element is expected to render as a 'block' box with its 'padding-left' property set to '40px'. The element's shadow tree is expected to take a child element that matches the selector `:bound-element > legend:first-child` and place it in a first 'block' box container, and then take the remaining child nodes and place them in a later 'block' box container.

The first container is expected to contain at least one line box, and that line box is expected to contain a triangle widget, horizontally positioned within the left padding of the `details` element. That widget is expected to allow the user to request that the details be shown or hidden.

The later container is expected to have its 'overflow' property set to 'hidden'. When the

`details` element has an `open` attribute, the later container is expected to have its 'height' set to 'auto'; when it does not, the later container is expected to have its 'height' set to 0.

10.4.6 The `input` element as a text entry widget

```
@namespace url(http://www.w3.org/1999/xhtml);

input { binding: input-textfield; }
input[type=password] { binding: input-password; }
/* later rules override this for other values of type="" */
```

When the *input-textfield* binding applies to an `input` element whose `type` attribute is in the Text, Search, URL, or E-mail state, the element is expected to render as an 'inline-block' box rendered as a text field.

When the *input-password* binding applies, to an `input` element whose `type` attribute is in the Password state, the element is expected to render as an 'inline-block' box rendered as a text field whose contents are obscured.

If an `input` element whose `type` attribute is in one of the above states has a `size` attribute, and parsing that attribute's value using the rules for parsing non-negative integers doesn't generate an error, then the user agent is expected to use the attribute as a presentational hint for the 'width' property on the element, with the value obtained from applying the converting a character width to pixels algorithm to the value of the attribute.

If an `input` element whose `type` attribute is in one of the above states does *not* have a `size` attribute, then the user agent is expected to act as if it had a user-agent-level style sheet rule setting the 'width' property on the element to the value obtained from applying the converting a character width to pixels algorithm to the number 20.

10.4.7 The `input` element as domain-specific widgets

```
@namespace url(http://www.w3.org/1999/xhtml);

input[type=datetime] { binding: input-datetime; }
input[type=date] { binding: input-date; }
input[type=month] { binding: input-month; }
input[type=week] { binding: input-week; }
input[type=time] { binding: input-time; }
input[type=datetime-local] { binding: input-datetime-local; }
input[type=number] { binding: input-number; }
```

When the *input-datetime* binding applies to an `input` element whose `type` attribute is in the Date and Time state, the element is expected to render as an 'inline-block' box depicting a Date and Time control.

When the *input-date* binding applies to an `input` element whose `type` attribute is in the Date state, the element is expected to render as an 'inline-block' box depicting a Date control.

When the *input-month* binding applies to an `input` element whose `type` attribute is in the Month state, the element is expected to render as an 'inline-block' box depicting a Month control.

When the *input-week* binding applies to an `input` element whose `type` attribute is in the Week state, the element is expected to render as an 'inline-block' box depicting a Week control.

When the *input-time* binding applies to an `input` element whose `type` attribute is in the Time state, the element is expected to render as an 'inline-block' box depicting a Time control.

When the *input-datetime-local* binding applies to an `input` element whose `type` attribute is in the Local Date and Time state, the element is expected to render as an 'inline-block' box depicting a Local Date and Time control.

When the *input-number* binding applies to an `input` element whose `type` attribute is in the Number state, the element is expected to render as an 'inline-block' box depicting a Number control.

These controls are all expected to be about one line high, and about as wide as necessary to show the widest possible value.

10.4.8 The `input` element as a range control

```
@namespace url(http://www.w3.org/1999/xhtml);

input[type=range] { binding: input-range; }
```

When the *input-range* binding applies to an `input` element whose `type` attribute is in the Range state, the element is expected to render as an 'inline-block' box depicting a slider control.

When the control is wider than it is tall (or square), the control is expected to be a horizontal slider, with the lowest value on the right if the 'direction' property on this element has a computed value of 'rtl', and on the left otherwise. When the control is taller than it is wide, it is expected to be a vertical slider, with the lowest value on the bottom.

Predefined suggested values (provided by the `list` attribute) are expected to be shown as tick marks on the slider, which the slider can snap to.

10.4.9 The `input` element as a color well

```
@namespace url(http://www.w3.org/1999/xhtml);

input[type=color] { binding: input-color; }
```

When the *input-color* binding applies to an `input` element whose `type` attribute is in the Color state, the element is expected to render as an 'inline-block' box depicting a color well, which, when activated, provides the user with a color picker (e.g. a color wheel or color palette) from which the color can be changed.

Predefined suggested values (provided by the `list` attribute) are expected to be shown in the color picker interface, not on the color well itself.

10.4.10 The `input` element as a check box and radio button widgets

```
@namespace url(http://www.w3.org/1999/xhtml);

input[type=checkbox] { binding: input-checkbox; }
input[type=radio] { binding: input-radio; }
```

When the *input-checkbox* binding applies to an `input` element whose `type` attribute is in the Checkbox state, the element is expected to render as an 'inline-block' box containing a single check box control, with no label.

When the *input-radio* binding applies to an `input` element whose `type` attribute is in the Radio Button state, the element is expected to render as an 'inline-block' box containing a single radio button control, with no label.

10.4.11 The `input` element as a file upload control

```
@namespace url(http://www.w3.org/1999/xhtml);

input[type=file] { binding: input-file; }
```

When the *input-file* binding applies to an `input` element whose `type` attribute is in the File Upload state, the element is expected to render as an 'inline-block' box containing a span of text giving the filename(s) of the selected files, if any, followed by a button that, when activated, provides the user with a file picker from which the selection can be changed.

10.4.12 The `input` element as a button

```
@namespace url(http://www.w3.org/1999/xhtml);

input[type=submit], input[type=reset], input[type=button] {
  binding: input-button;
}
```

When the *input-button* binding applies to an `input` element whose `type` attribute is in the Submit Button, Reset Button, or Button state, the element is expected to render as an 'inline-block' box rendered as a button, about one line high, containing the contents of the element's `value` attribute, if any, or text derived from the element's `type` attribute in a user-agent-defined (and probably locale-specific) fashion, if not.

10.4.13 The `marquee` element

...(Waiting til I've specced the DOM side of this)...

10.4.14 The `meter` element

```
@namespace url(http://www.w3.org/1999/xhtml);

meter {
  binding: meter;
}
```

When the *meter* binding applies to a `meter` element, the element is expected to render as an 'inline-block' box with a 'height' of '1em' and a 'width' of '5em', a 'vertical-align' of '-0.2em', and with its contents depicting a gauge.

When the element is wider than it is tall (or square), the depiction is expected to be of a horizontal gauge, with the minimum value on the right if the 'direction' property on this element has a computed value of 'rtl', and on the left otherwise. When the element is taller than it is wide, it is expected to depict a vertical gauge, with the minimum value on the bottom.

User agents are expected to use a presentation consistent with platform conventions for gauges, if any.

Note: Requirements for what must be depicted in the gauge are included in the definition of the `meter` element.

10.4.15 The `progress` element

```
@namespace url(http://www.w3.org/1999/xhtml);

progress {
  binding: progress;
}
```

When the *progress* binding applies to a `progress` element, the element is expected to render as an 'inline-block' box with a 'height' of '1em' and a 'width' of '10em', a 'vertical-align' of '-0.2em', and with its contents depicting a horizontal progress bar, with the start on the right and the end on the left if the 'direction' property on this element has a computed value of 'rtl', and with the start on the left and the end on the right otherwise.

User agents are expected to use a presentation consistent with platform conventions for progress bars. In particular, user agents are expected to use different presentations for determinate and indeterminate progress bars. User agents are also expected to vary the presentation based on the dimensions of the element.

> For example, on some platforms for showing indeterminate progress there is an asychronous progress indicator with square dimensions, which could be used when the element is square, and an indeterminate progress bar, which could be used when the element is wide.

Note: Requirements for how to determine if the progress bar is determinate or indeterminate, and what progress a determinate progress bar is to show, are included in the definition of the `progress` element.

10.4.16 The `select` element

```
@namespace url(http://www.w3.org/1999/xhtml);

select {
  binding: select;
}
```

When the *select* binding applies to a `select` element whose `multiple` attribute is present, the element is expected to render as a multi-select list box.

When the *select* binding applies to a `select` element whose `multiple` attribute is absent, and the element's `size` attribute specifies a value greater than 1, the element is expected to render as a single-select list box.

When the element renders as a list box, it is expected to render as an 'inline-block' box whose 'height' is the height necessary to contain as many rows for items as specified by the element's `size` attribute, or four rows if the attribute is absent, and whose 'width' is the width of the `select`'s labels plus the width of a scrollbar.

When the *select* binding applies to a `select` element whose `multiple` attribute is absent, and the element's `size` attribute is either absent or specifies either no value (an error), or a value less than or equal to 1, the element is expected to render as a one-line drop down box whose width is the width of the `select`'s labels.

In either case (list box or drop-down box), the element's items are expected to be the element's list of options, with the element's `optgroup` element children providing headers for groups of options where applicable.

The **width of the `select`'s labels** is the wider of the width necessary to render the widest `optgroup`, and the width necessary to render the widest `option` element in the element's list of options (including its indent, if any).

An `optgroup` element is expected to be rendered by displaying the element's `label` attribute.

An `option` element is expected to be rendered by displaying the element's `label`, indented under its `optgroup` element if it has one.

10.4.17 The `textarea` element

```
@namespace url(http://www.w3.org/1999/xhtml);

textarea { binding: textarea; }
```

When the *textarea* binding applies to a `textarea` element, the element is expected to render as an 'inline-block' box rendered as a multiline text field.

If the element has a `cols` attribute, and parsing that attribute's value using the rules for parsing non-negative integers doesn't generate an error, then the user agent is expected to use the attribute as a presentational hint for the 'width' property on the element, with the value obtained from applying the converting a character width to pixels algorithm to the value of the attribute and then adding the width of a scroll bar.

If the element has a `rows` attribute, and parsing that attribute's value using the rules for parsing non-negative integers doesn't generate an error, then the user agent is expected to use the attribute as a presentational hint for the 'height' property on the element, with the value being the specified number of lines, plus the height of a scrollbar.

10.5 Frames and framesets

When an `html` element's second child element is a `frameset` element, the user agent is expected to render the `frameset` element as described below across the surface of the view, instead of applying the usual CSS rendering rules.

When rendering a `frameset` on a surface, the user agent is expected to use the following layout algorithm:

1. The *cols* and *rows* variables are lists of zero or more pairs consisting of a number and a unit, the unit being one of *percentage*, *relative*, and *absolute*.

   Use the rules for parsing a list of dimensions to parse the value of the element's `cols` attribute, if there is one. Let *cols* be the result, or an empty list if there is no such attribute.

   Use the rules for parsing a list of dimensions to parse the value of the element's `rows` attribute, if there is one. Let *rows* be the result, or an empty list if there is no such attribute.

2. For any of the entries in *cols* or *rows* that have the number zero and the unit *relative*, change the entry's number to one.

3. If *cols* has no entries, then add a single entry consisting of the value 1 and the unit *relative* to *cols*.

   If *rows* has no entries, then add a single entry consisting of the value 1 and the unit *relative* to *rows*.

4. Invoke the algorithm defined below to convert a list of dimensions to a list of pixel values using *cols* as the input list, and the width of the surface that the `frameset` is being rendered into, in CSS pixels, as the input dimension. Let *sized cols* be the resulting list.

   Invoke the algorithm defined below to convert a list of dimensions to a list of pixel values using *rows* as the input list, and the height of the surface that the `frameset` is being rendered into, in CSS pixels, as the input dimension. Let *sized rows* be the resulting list.

5. Split the surface into a grid of *w*×*h* rectangles, where *w* is the number of entries in *sized cols* and *h* is the number of entries in *sized rows*.

   Size the columns so that each column in the grid is as many CSS pixels wide as the corresponding entry in the *sized cols* list.

   Size the rows so that each row in the grid is as many CSS pixels high as the corresponding entry in the *sized rows* list.

6. Let *children* be the list of `frame` and `frameset` elements that are children of the `frameset` element for which the algorithm was invoked.

7. For each row of the grid of rectangles created in the previous step, from top to bottom, run these substeps:

   1. For each rectangle in the row, from left to right, run these substeps:

      1. If there are any elements left in *children*, take the first element in the list, and assign it to the rectangle.

If this is a `frameset` element, then recurse the entire `frameset` layout algorithm for that `frameset` element, with the rectangle as the surface.

Otherwise, it is a `frame` element; create a nested browsing context sized to fit the rectangle.

2. If there are any elements left in *children*, remove the first element from *children*.

8. If the `frameset` element has a border, draw an outer set of borders around the rectangles, using the element's frame border color.

   For each rectangle, if there is an element assigned to that rectangle, and that element has a border, draw an inner set of borders around that rectangle, using the element's frame border color.

   For each (visible) border that does not abut a rectangle that is assigned a `frame` element with a `noresize` attribute (including rectangles in further nested `frameset` elements), the user agent is expected to allow the user to move the border, resizing the rectangles within, keeping the proportions of any nested `frameset` grids.

   A `frameset` or `frame` element **has a border** if the following algorithm returns true:

   1. If the element has a `frameborder` attribute whose value is not the empty string and whose first character is either a U+0031 DIGIT ONE (1), a U+0079 LATIN SMALL LETTER Y, or a U+0059 LATIN CAPITAL LETTER Y, then return true.

   2. Otherwise, if the element has a `frameborder` attribute, return false.

   3. Otherwise, if the element has a parent element that is a `frameset` element, then return true if *that* element has a border, and false if it does not.

   4. Otherwise, return true.

   The **frame border color** of a `frameset` or `frame` element is the color obtained from the following algorithm:

   1. If the element has a `bordercolor` attribute, then return the color obtained from applying the rules for parsing a legacy color value to that attribute's value.

   2. Otherwise, if the element has a parent element that is a `frameset` element, then the frame border color of that element.

   3. Otherwise, return gray.

The algorithm to **convert a list of dimensions to a list of pixel values** consists of the following steps:

1. Let *input list* be the list of numbers and units passed to the algorithm.

   Let *output list* be a list of numbers the same length as *input list*, all zero.

   Entries in *output list* correspond to the entries in *input list* that have the same position.

2. Let *input dimension* be the size passed to the algorithm.

3. Let *count percentage* be the number of entries in *input list* whose unit is *percentage*.

   Let *total percentage* be the sum of all the numbers in *input list* whose unit is *percentage*.

   Let *count relative* be the number of entries in *input list* whose unit is *relative*.

   Let *total relative* be the sum of all the numbers in *input list* whose unit is *relative*.

   Let *count absolute* be the number of entries in *input list* whose unit is *absolute*.

   Let *total absolute* be the sum of all the numbers in *input list* whose unit is *absolute*.

   Let *remaining space* be the value of *input dimension*.

4. If *total absolute* is greater than *remaining space*, then for each entry in *input list* whose unit is *absolute*, set the corresponding value in *output list* to the number of the entry in *input list* multiplied by *remaining space* and divided by *total absolute*. Then, set *remaining space* to zero.

   Otherwise, for each entry in *input list* whose unit is *absolute*, set the corresponding value in *output list* to the number of the entry in *input list*. Then, decrement *remaining space* by *total absolute*.

5. If *total percentage* multiplied by the *input dimension* and divided by 100 is greater than *remaining space*, then for each entry in *input list* whose unit is *percentage*, set the corresponding value in *output list* to the number of the entry in *input list* multiplied by *remaining space* and divided by *total percentage*. Then, set *remaining space* to zero.

   Otherwise, for each entry in *input list* whose unit is *percentage*, set the corresponding value in *output list* to the number of the entry in *input list* multiplied by the *input dimension* and divided by 100. Then, decrement *remaining space* by *total percentage* multiplied by the *input dimension* and divided by 100.

6. For each entry in *input list* whose unit is *relative*, set the corresponding value in *output list* to the number of the entry in *input list* multiplied by *remaining space* and divided by *total relative*.

7. Return *output list*.

User agents working with integer values for frame widths (as opposed to user agents that can lay frames out with subpixel accuracy) are expected to distribute the remainder first the last entry whose unit is *relative*, then equally (not proportionally) to each entry whose unit is *percentage*, then equally (not proportionally) to each entry whose unit is *absolute*, and finally, failing all else, to the last entry.

10.6 Interactive media

10.6.1 Links, forms, and navigation

User agents are expected to allow the user to control aspects of hyperlink activation and form submission, such as which browsing context is to be used for the subsequent navigation.

User agents are expected to allow users to discover the destination of hyperlinks and of forms before triggering their navigation.

User agents are expected to inform the user of whether a hyperlink includes hyperlink auditing, and to let them know at a minimum which domains will be contacted as part of such auditing.

User agents are expected to allow users to navigate browsing contexts to the resources indicated by the `cite` attributes on `q`, `blockquote`, `ins`, and `del` elements.

User agents are expected to surface hyperlinks created by `link` elements in their user interface.

> ***Note: While `link` elements that create hyperlinks will match the ':link' or ':visited' pseudo-classes, will react to clicks if visible, and so forth, this does not extend to any browser interface constructs that expose those same links. Activating a link through the browser's interface, rather than in the page itself, does not trigger `click` events and the like.***

10.6.2 The `mark` element

User agents are expected to allow the user to cycle through all the `mark` elements in a `Document`. User agents are also expected to bring their existence to the user's attention, even when they are off-screen, e.g. by highlighting portions of the scroll bar that represent portions of the document that contain `mark` elements.

10.6.3 The `title` attribute

Given an element (e.g. the element designated by the mouse cursor), if the element, or one of its ancestors, has a `title` attribute, and the nearest such attribute has a value that is not the empty string, it is expected that the user agent will expose the contents of that attribute as a tooltip.

U+000A LINE FEED (LF) characters are expected to cause line breaks in the tooltip.

10.7 Print media

User agents are expected to allow the user to request the opportunity to **obtain a physical form** (or a representation of a physical form) of a `Document`. For example, selecting the option to print a page or convert it to PDF format.

When the user actually obtains a physical form (or a representation of a physical form) of a `Document`, the user agent is expected to create a new view with the print media, render the result, and the discard the view.

10.8 Interaction with CSS

Must define that in CSS, tag and attribute names in HTML documents, and class names in quirks mode documents, are case-insensitive, as well as saying which attribute values must be compared case-insensitively.

11 Obsolete features

Authors and documents must not use the features listed in this section. They are documented to enable user agents to support legacy content in an interoperable fashion.

11.1 Self-contained features

11.1.1 The `applet` element

The `applet` element is a Java-specific variant of the `embed` element. In HTML5 the `applet` element is obsoleted so that all extension frameworks (Java, .NET, Flash, etc) are handled in a consistent manner.

If the sandboxed plugins browsing context flag is set on the browsing context for which the `applet` element's document is the active document, then the element must be ignored (it represents nothing).

Otherwise, define how the element works, if supported.

```
[XXX] interface HTMLDocument {
  readonly attribute HTMLCollection applets;
};
```

The **`applets`** attribute must return an `HTMLCollection` rooted at the `Document` node, whose filter matches only `applet` elements.

11.1.2 The `marquee` element

...

11.2 Other elements and attributes

The following elements are obsolete and either have no meaning whatsoever or have no requirements beyond those described elsewhere in this specification:

- **`center`**

The following attributes are obsolete and either have no meaning whatsoever or have no requirements beyond those described elsewhere in this specification:

- **`name`** on `a` elements
- **`alink`** on `body` elements

- **bgcolor** on body elements
- **link** on body elements
- **text** on body elements
- **vlink** on body elements

11.3 Other DOM APIs

These APIs expose obsolete content attributes.

The [XXX] below is for some annotation meaning "this is just another part of the named interface, and should be treated as if it had been part of the main interface definition".

```
[XXX] interface HTMLBodyElement {
           attribute DOMString text;
           attribute DOMString bgColor;
           attribute DOMString background;
           attribute DOMString link;
           attribute DOMString vLink;
           attribute DOMString aLink;
};
```

The **text** DOM attribute of the body element must reflect the element's text content attribute.

The **bgColor** DOM attribute of the body element must reflect the element's bgcolor content attribute.

The **background** DOM attribute of the body element must reflect the element's background content attribute.

The **link** DOM attribute of the body element must reflect the element's link content attribute.

The **aLink** DOM attribute of the body element must reflect the element's alink content attribute.

The **vLink** DOM attribute of the body element must reflect the element's vlink content attribute.

```
[XXX] interface HTMLDocument {
           attribute DOMString fgColor;
           attribute DOMString bgColor;
           attribute DOMString linkColor;
           attribute DOMString vlinkColor;
           attribute DOMString alinkColor;
};
```

The **fgColor** attribute on the Document object must reflect the text attribute on the body element.

The **bgColor** attribute on the Document object must reflect the bgcolor attribute on the body element.

The **`linkColor`** attribute on the `Document` object must reflect the `link` attribute on the body element.

The **`vLinkColor`** attribute on the `Document` object must reflect the `vlink` attribute on the body element.

The **`aLinkColor`** attribute on the `Document` object must reflect the `alink` attribute on the body element.

11.4 Conformance checkers

To ease the transition from HTML4 Transitional documents to the language defined in *this* specification, conformance checkers are encouraged to categorise errors that represent usage of old obsolete features that generally have no effect (as defined below) into a separate part of their report, to allow authors to distinguish between likely mistakes and mere vestigial markup.

The following errors may be categorised as described above:

- The DOCTYPE parse error, if the DOCTYPE token's `name` is an ASCII case-insensitive match for the string "`HTML`", and either:
    - the token's public identifier is the case-sensitive string "`-//W3C//DTD HTML 4.0//EN`" and the token's system identifier is either missing or the case-sensitive string "`http://www.w3.org/TR/REC-html40/strict.dtd`", or
    - the token's public identifier is the case-sensitive string "`-//W3C//DTD HTML 4.01//EN`" and the token's system identifier is either missing or the case-sensitive string "`http://www.w3.org/TR/html4/strict.dtd`", or
    - the token's public identifier is the case-sensitive string "`-//W3C//DTD XHTML 1.0 Strict//EN`" and the token's system identifier is either missing or the case-sensitive string "`http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd`", or
    - the token's public identifier is the case-sensitive string "`-//W3C//DTD XHTML 1.1//EN`" and the token's system identifier is either missing or the case-sensitive string "`http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd`".
- The presence of a `profile` attribute on the `head` element, if its value is an unordered set of unique space-separated tokens where the words are all valid URLs.
- The presence of a `meta` element with an `http-equiv` attribute in the Content Language state.
- The presence of a `border` attribute on an `img` element if its value is the string "`0`".
- The presence of a `longdesc` attribute on an `img` element, if its value is a valid URL.
- The presence of a `language` attribute on a `script` element if its value is an ASCII case-insensitive match for the string "`JavaScript`".

- The presence of a `name` attribute on an `a` element, if its value is not the empty string.
- The presence of a `summary` attribute on a `table` element.
- The presence of an `abbr` attribute on a `td` or `th` element.

12 Things that you can't do with this specification because they are better handled using other technologies that are further described herein

This section is non-normative.

There are certain features that are not handled by this specification because a client side markup language is not the right level for them, or because the features exist in other languages that can be integrated into this one. This section covers some of the more common requests.

12.1 Localization

If you wish to create localized versions of an HTML application, the best solution is to preprocess the files on the server, and then use HTTP content negotiation to serve the appropriate language.

12.2 Declarative 3D scenes

Embedding 3D imagery into XHTML documents is the domain of X3D, or technologies based on X3D that are namespace-aware.

12.3 Timers

This section is expected to be moved to its own specification in due course. It needs a lot of work to actually make it into a semi-decent spec.

Objects that implement the `Window` interface must also implement the `WindowTimers` interface:

```
[NoInterfaceObject, ImplementedOn=Window] interface WindowTimers {
  // timers
  long setTimeout(in TimeoutHandler handler, in long timeout);
  long setTimeout(in TimeoutHandler handler, in long timeout, arguments...);
  long setTimeout(in DOMString code, in long timeout);
  long setTimeout(in DOMString code, in long timeout, in DOMString language);
  void clearTimeout(in long handle);
  long setInterval(in TimeoutHandler handler, in long timeout);
  long setInterval(in TimeoutHandler handler, in long timeout, arguments...);
  long setInterval(in DOMString code, in long timeout);
  long setInterval(in DOMString code, in long timeout, in DOMString language);
  void clearInterval(in long handle);
};

[Callback=FunctionOnly, NoInterfaceObject]
```

```
interface TimeoutHandler {
  void handleEvent([Variadic] in any args);
};
```

The `setTimeout` and `setInterval` methods allow authors to schedule timer-based events.

The **`setTimeout(`*`handler`*`, `*`timeout`*`[, `*`arguments...`*`])`** method takes a reference to a `TimeoutHandler` object and a length of time in milliseconds. It must return a handle to the timeout created, and then asynchronously wait *timeout* milliseconds and then queue a task to invoke `handleEvent()` on the *handler* object. If any *arguments...* were provided, they must be passed to the *handler* as arguments to the `handleEvent()` function.

Alternatively, **`setTimeout(`*`code`*`, `*`timeout`*`[, `*`language`*`])`** may be used. This variant takes a string instead of a `TimeoutHandler` object. define the actual requirements for this method, as with the previous one. That string must be parsed using the specified *language* (defaulting to ECMAScript if the third argument is omitted) and executed in the scope of the browsing context associated with the `Window` object on which the `setTimeout()` method was invoked.

Need to define *language* values; need to define that the script corresponding to the *code* argument is created before the timer is set up, so that the rule on pausing the ticker, below, makes sense.

The **`setInterval(...)`** variants must work in the same way as the `setTimeout` variants except that if *timeout* is a value greater than zero, the task that invokes the *handler* or `code` must be queued again every *timeout* milliseconds, not just the once.

The **`clearTimeout()`** and **`clearInterval()`** methods take one integer (the value returned by `setTimeout()` and `setInterval()` respectively) and must cancel the specified timeout. When called with a value that does not correspond to an active timeout or interval, the methods must return without doing anything.

For both `setTimeout()` and `setInterval()`, the clock upon which the timers are based must only tick while the `Document` of the global object of their callbacks is fully active.

12.4 Rendering and the DOM

This section is expected to be moved to its own specification in due course. It needs a lot of work to actually make it into a semi-decent spec.

Any object implement the `AbstractView` interface must also implement the `MediaModeAbstractView` interface.

```
[NoInterfaceObject, ImplementedOn=AbstractView] interface MediaModeAbstractView {
  readonly attribute DOMString mediaMode;
};
```

The **`mediaMode`** attribute on objects implementing the `MediaModeAbstractView` interface must return the string that represents the canvas' current rendering mode (`screen`, `print`, etc). This is a lowercase string, as defined by the CSS specification. [CSS21]

Some user agents may support multiple media, in which case there will exist multiple objects implementing the `AbstractView` interface. Only the default view implements the `Window` interface. The other views can be reached using the `view` attribute of the `UIEvent` interface, during event propagation. There is no way currently to enumerate all the views.

Index

This section is non-normative.

List of elements

List of attributes

List of **reflecting DOM attributes** and their corresponding content attributes

List of interfaces

List of events

References

This section will be written in a future draft.

Acknowledgements

Thanks to Aankhen, Aaron Boodman, Aaron Leventhal, Adam Barth, Adam Roben, Addison Phillips, Adele Peterson, Adrian Sutton, Agustín Fernández, Ajai Tirumali, Alastair Campbell, Alex Nicolaou, Alexey Feldgendler, Алексей Проскуряков (Alexey Proskuryakov), Anders Carlsson, Andrew Gove, Andrew Sidwell, Anne van Kesteren, Anthony Hickson, Anthony Ricaud, Antti Koivisto, Arphen Lin, Asbjørn Ulsberg, Ashley Sheridan, Aurelien Levy, Ben Boyle, Ben Godfrey, Ben Meadowcroft, Ben Millard, Benjamin Hawkes-Lewis, Bert Bos, Bijan Parsia, Bill Mason, Billy Wong, Bjoern Hoehrmann, Boris Zbarsky, Brad Fults, Brad Neuberg, Brady Eidson, Brendan Eich, Brett Wilson, Brian Campbell, Brian Smith, Bruce Lawson, Bruce Miller, Cameron McCormack, Cao Yipeng, Carlos Perelló Marín, Chao Cai, 윤석찬 (Channy Yun), Charl van Niekerk, Charles Iliya Krempeaux, Charles McCathieNevile, Chris Pearce, Christian Biesinger, Christian Johansen, Christian Schmidt, Chriswa, Cole Robison, Colin Fine, Collin Jackson, Daniel Barclay, Daniel Brumbaugh Keeney, Daniel Glazman, Daniel Peng, Daniel Spång, Daniel Steinberg, Danny Sullivan, Darin Adler, Darin Fisher, Dave Camp, Dave Singer, Dave Townsend, David Baron, David Bloom, David Carlisle, David Flanagan, David Håsäther, David Hyatt, David Smith, David Woolley, Dean Edridge, Debi Orton, Derek Featherstone, DeWitt Clinton, Dimitri Glazkov, dolphinling, Doron Rosenberg, Doug Kramer, Drew Wilson, Edward O'Connor, Edward Z. Yang, Eira Monstad, Elliotte Harold, Eric Carlson, Eric Law, Erik Arvidsson, Evan Martin, Evan Prodromou, fantasai, Felix Sasaki, Franck 'Shift' Quélain, Garrett Smith, Geoffrey Garen, Geoffrey Sneddon, George Lund, Greg Houston, Håkon Wium Lie, Hans S. Tømmerhalt, Henri Sivonen, Henrik Lied, Henry Mason, Hugh Winkler, Ignacio Javier, Ivo Emanuel Gonçalves, J. King, Jacques Distler, James Craig, James Graham, James Justin Harrell, James M Snell, James Perrett, Jan-Klaas Kollhof, Jason White, Jasper Bryant-Greene, Jed Hartman, Jeff Cutsinger, Jeff Schiller, Jeff Walden, Jens Bannmann, Jens Fendler, Jeroen van der Meer, Jim Jewett, Jim Meehan, Joe Clark, John Fallows, Joseph Kesselman, Jjgod Jiang, Joel Spolsky, Johan Herland, John Boyer, John Bussjaeger, John Harding, Johnny Stenback, Jon Gibbins, Jon Perlow, Jonathan Worent, Jorgen Horstink, Josh Levenberg, Joshua Randall, Jukka K. Korpela, Jules Clément-Ripoche, Julian Reschke, Kai Hendry, Kartikaya Gupta, Kornel Lesinski, 黒澤剛志 (KUROSAWA Takeshi), Kristof Zelechovski, Kyle Hofmann, Lachlan Hunt, Larry Page, Lars Gunther, Laura L. Carlson, Laura Wisewell, Laurens Holst, Lee Kowalkowski, Leif Halvard Silli, Lenny Domnitser, Léonard Bouchet, Leons Petrazickis, Logan, Loune, Maciej Stachowiak, Magnus Kristiansen, Maik Merten, Malcolm Rowe, Mark Nottingham, Mark Rowe, Mark Schenk, Martijn Wargers, Martin Atkins, Martin Dürst, Martin Honnen, Masataka Yakura, Mathieu Henri, Matthew Gregan, Matthew Mastracci, Matthew Raymond, Matthew Thomas, Mattias Waldau, Max Romantschuk, Michael 'Ratt' Iannarelli, Michael A. Nachbaur, Michael A. Puls II, Michael Carter, Michael Gratton, Michael Nordman, Michael Powers, Michael(tm) Smith, Michel Fortin, Michiel van der Blonk, Mihai Şucan, Mike Brown, Mike Dierken, Mike Dixon, Mike Schinkel, Mike Shaver, Mikko Rantalainen, Neil Deakin, Neil Soiffer, Nicholas Shanks, Ojan Vafai, Olaf Hoffmann, Olav Junker Kjær, Oliver Hunt, Patrick H. Lauke, Peter Karlsson, Peter Kasting, Philip Jägenstedt, Philip Taylor, Philip TAYLOR, Rachid Finge, Rajas Moonka, Ralf Stoltze, Ralph Giles, Raphael Champeimont, Rene Saarsoo, Richard Ishida, Rikkert Koppes, Rimantas Liubertas, Robert Blaut, Robert O'Callahan, Robert Sayre, Roman Ivanov, Ryan King, S. Mike Dierken, Sam Kuper, Sam Ruby, Sam Weinig, Scott Hess, Sean Knapp, Shaun Inman, Shiki Okasaka, Silvia Pfeiffer, Simon Pieters, Stefan Haustein, Steffen Meschkat, Stephen Ma, Steve Faulkner, Steve Runyon, Steven Garrity, Stewart Brodie, Stuart Parmenter, Sunava Dutta, Tantek Çelik,

Terrence Wood, Thomas Broyer, Thomas O'Connor, Tim Altman, Tim Johansson, Tommy Thorsen, Travis Leithead, Tyler Close, Vladimir Vukićević, Wakaba, Wayne Pollock, Wellington Fernando de Macedo, William Swanson, Wladimir Palant, Wolfram Kriesing, Yi-An Huang, Zhenbin Xu, and Øistein E. Andersen, for their useful and substantial comments.

Thanks also to everyone who has ever posted about HTML5 to their blogs, public mailing lists, or forums, including the W3C public-html list and the various WHATWG lists.

Special thanks to Richard Williamson for creating the first implementation of `canvas` in Safari, from which the canvas feature was designed.

Special thanks also to the Microsoft employees who first implemented the event-based drag-and-drop mechanism, `contenteditable`, and other features first widely deployed by the Windows Internet Explorer browser.

Special thanks and $10,000 to David Hyatt who came up with a broken implementation of the adoption agency algorithm that the editor had to reverse engineer and fix before using it in the parsing section.

Thanks to the many sources that provided inspiration for the examples used in the specification.

Thanks also to the Microsoft blogging community for some ideas, to the attendees of the W3C Workshop on Web Applications and Compound Documents for inspiration, to the #mrt crew, the #mrt.no crew, and the #whatwg crew, and to Pillar and Hedral for their ideas and support.