connections to be set up back to an originating IP address, but we could, if the subdomain is the empty string.

Web Applications 1.0
Working Draft — 4 January 2007

You can take part in this work. Join the working group's discussion list.

This version:

http://www.whatwg.org/specs/web-apps/current-work/

Latest version:

http://www.whatwg.org/specs/web-apps/current-work/

Previous versions:

http://www.whatwg.org/specs/web-apps/2006-01-01/
http://www.whatwg.org/specs/web-apps/2005-09-01/
Version history from 2006-03-01 available by interactive Web interface at: http://html5.org/tools/specification-diff
Version history from 2006-03-01 available by Subversion interface at: http://svn.whatwg.org/

Editor:

Ian Hickson, Google, ian@hixie.ch

© Copyright 2004-2006 Apple Computer, Inc., Mozilla Foundation, and Opera Software ASA.
You are granted a license to use, reproduce and create derivative works of this document.

Abstract

This specification introduces features to HTML and the DOM that ease the authoring of Web-based applications. Additions include the context menus, a direct-mode graphics canvas, inline popup windows, and server-sent events.

Status of this document

This is a work in progress! This document is changing on a daily if not hourly basis in response to comments and as a general part of its development process. Comments are very welcome, please send them to whatwg@whatwg.org. Thank you.

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 call for implementations should join the WHATWG mailing list and take part in the discussions.

This draft may contain namespaces that use the `uuid:` URI scheme. These are temporary and will be changed before those parts of the specification are ready to be implemented in shipping

products.

To find the latest version of this working draft, please follow the "Latest version" link above.

This specification is intended to replace (be the new version of) what was previously the HTML4, XHTML 1.x, and DOM2 HTML specifications.

Stability

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

Known issues are usually marked like this. There are some spec-wide issues that have not yet been addressed: first, case-sensitivity is a very poorly handled topic right now; and second, there's a ghost of a **DocumentUI** interface going around.

Table of contents

1. Introduction
    1.1. Scope
    1.2. Structure of this specification
        1.2.1. How to read this specification
    1.3. Conformance requirements
        1.3.1. Dependencies
        1.3.2. Features defined in other specifications
        1.3.3. Relationship to HTML 4.01, XHTML 1.1, DOM2 HTML
        1.3.4. Relationship to XHTML2
        1.3.5. Relationship to XUL, WPF/XAML, and other proprietary UI languages
    1.4. Terminology
        1.4.1. HTML vs XHTML

2. The Document Object Model
    2.1. Documents
        2.1.1. Resource metadata management
    2.2. Elements
        2.2.1. Reflecting content attributes in DOM attributes
    2.3. Common DOM interfaces
        2.3.1. Collections
            2.3.1.1. HTMLCollection
            2.3.1.2. HTMLFormControlsCollection
            2.3.1.3. HTMLOptionsCollection
        2.3.2. DOMTokenString
        2.3.3. DOM feature strings
    2.4. DOM tree accessors
    2.5. Dynamic markup insertion
        2.5.1. Controlling the input stream
        2.5.2. Dynamic markup insertion in HTML
        2.5.3. Dynamic markup insertion in XML
    2.6. APIs in HTML documents

3. Semantics and structure of HTML elements
3.1. Introduction
3.2. Common microsyntaxes
3.2.1. Common parser idioms
3.2.2. Numbers
3.2.2.1. Unsigned integers
3.2.2.2. Signed integers
3.2.2.3. Real numbers
3.2.2.4. Ratios
3.2.2.5. Percentages and dimensions
3.2.3. Dates and times
3.2.3.1. Specific moments in time
3.2.3.2. Vaguer moments in time
3.2.4. Tokens
3.3. Documents and document fragments
3.3.1. Semantics
3.3.2. Structure
3.3.3. Kinds of elements
3.3.3.1. Block-level elements
3.3.3.2. Inline-level content
3.3.3.3. Determining if a particular element contains block-level elements or inline-level content
3.3.3.4. Interactive elements
3.3.3.5. Paragraphs
3.4. Global attributes
3.4.1. The `id` attribute
3.4.2. The `title` attribute
3.4.3. The `lang` (HTML only) and `xml:lang` (XML only) attributes
3.4.4. The `dir` attribute
3.4.5. Classes
3.4.5.1. Predefined class names
3.4.5.1.1. Class name "`copyright`"
3.4.5.1.2. Class name "`error`"
3.4.5.1.3. Class name "`example`"
3.4.5.1.4. Class name "`issue`"
3.4.5.1.5. Class name "`note`"
3.4.5.1.6. Class name "`search`"
3.4.5.1.7. Class name "`warning`"
3.4.5.2. Other classes
3.5. Interaction
3.5.1. Activation
3.5.2. Focus
3.5.2.1. Focus management
3.5.2.2. Sequential focus navigation
3.6. The root element
3.6.1. The `html` element
3.7. Document metadata
3.7.1. The `head` element
3.7.2. The `title` element
3.7.3. The `base` element
3.7.4. The `link` element

3.7.5. The meta element
3.7.5.1. Standard metadata names
3.7.5.2. Other metadata names
3.7.5.3. Specifying and establishing the document's character encoding
3.7.6. The style element
3.8. Sections
3.8.1. The body element
3.8.2. The section element
3.8.3. The nav element
3.8.4. The article element
3.8.5. The blockquote element
3.8.6. The aside element
3.8.7. The h1, h2, h3, h4, h5, and h6 elements
3.8.8. The header element
3.8.9. The footer element
3.8.10. The address element
3.8.11. Headings and sections
3.8.11.1. Creating an outline
3.8.11.2. Determining which heading and section applies to a particular node
3.9. Prose
3.9.1. The p element
3.9.2. The hr element
3.9.3. The br element
3.9.4. The dialog element
3.10. Preformatted text
3.10.1. The pre element
3.11. Lists
3.11.1. The ol element
3.11.2. The ul element
3.11.3. The li element
3.11.4. The dl element
3.11.5. The dt element
3.11.6. The dd element
3.12. Phrase elements
3.12.1. The a element
3.12.2. The q element
3.12.3. The cite element
3.12.4. The em element
3.12.5. The strong element
3.12.6. The small element
3.12.7. The m element
3.12.8. The dfn element
3.12.9. The abbr element
3.12.10. The time element
3.12.11. The meter element
3.12.12. The progress element
3.12.13. The code element
3.12.14. The var element
3.12.15. The samp element

3.12.16. The kbd element
3.12.17. The sup and sub elements
3.12.18. The span element
3.12.19. The i element
3.12.20. The b element
3.12.21. The bdo element
3.13. Edits
3.13.1. The ins element
3.13.2. The del element
3.13.3. Attributes common to ins and del elements
3.14. Embedded content
3.14.1. The img element
3.14.2. The iframe element
3.14.3. The embed element
3.14.4. The object element
3.14.5. The param element
3.14.6. The canvas element
3.14.6.1. The 2D context
3.14.6.1.1. The canvas state
3.14.6.1.2. Transformations
3.14.6.1.3. Compositing
3.14.6.1.4. Colors and styles
3.14.6.1.5. Line styles
3.14.6.1.6. Shadows
3.14.6.1.7. Simple shapes (rectangles)
3.14.6.1.8. Complex shapes (paths)
3.14.6.1.9. Images
3.14.6.1.10. Pixel manipulation
3.14.6.1.11. Drawing model
3.14.7. The map element
3.14.8. The area element
3.14.9. The figure element
3.15. Tabular data
3.15.1. The table element
3.15.2. The caption element
3.15.3. The colgroup element
3.15.4. The col element
3.15.5. The tbody element
3.15.6. The thead element
3.15.7. The tfoot element
3.15.8. The tr element
3.15.9. The td element
3.15.10. The th element
3.15.11. Processing model
3.15.11.1. Forming a table
3.15.11.2. Forming relationships between data cells and header cells
3.16. Forms
3.16.1. The form element
3.16.2. The fieldset element
3.16.3. The input element

3.16.4. The button element
3.16.5. The label element
3.16.6. The select element
3.16.7. The datalist element
3.16.8. The optgroup element
3.16.9. The option element
3.16.10. The textarea element
3.16.11. The output element
3.16.12. Processing model
3.16.12.1. Form submission
3.17. Scripting
3.17.1. The script element
3.17.1.1. Script languages
3.17.2. The noscript element
3.18. Interactive elements
3.18.1. The details element
3.18.2. The datagrid element
3.18.2.1. The datagrid data model
3.18.2.2. How rows are identified
3.18.2.3. The data provider interface
3.18.2.4. The default data provider
3.18.2.4.1. Common default data provider method definitions for cells
3.18.2.5. Populating the datagrid element
3.18.2.6. Updating the datagrid
3.18.2.7. Requirements for interactive user agents
3.18.2.8. The selection
3.18.2.9. Columns and captions
3.18.3. The command element
3.18.4. The menu element
3.19. Miscellaneous elements
3.19.1. The legend element
3.19.2. The div element

4. Processing models
4.1. Navigating across documents
4.1.1. Determining the type of a new resource in a browsing context
4.1.2. Page load processing model for HTML files
4.1.3. Page load processing model for XML files
4.1.4. Page load processing model for text files
4.1.5. Page load processing model for syndication feeds
4.1.6. Content-Type
4.2. Scripting
4.2.1. Running executable code
4.2.2. Scripting contexts
4.2.3. Threads
4.2.4. The security model
4.2.5. javascript: URIs
4.2.6. Runtime script errors
4.2.7. Events
4.2.7.1. Event handler attributes

4.2.7.2. Event firing
4.2.7.3. Event handling
4.3. Links
4.3.1. Hyperlink elements
4.3.2. Following hyperlinks
4.3.3. Link types
4.3.3.1. Link type "alternate"
4.3.3.2. Link type "archives"
4.3.3.3. Link type "author"
4.3.3.4. Link type "bookmark"
4.3.3.5. Link type "contact"
4.3.3.6. Link type "external"
4.3.3.7. Link type "feed"
4.3.3.8. Link type "help"
4.3.3.9. Link type "icon"
4.3.3.10. Link type "license"
4.3.3.11. Link type "nofollow"
4.3.3.12. Link type "pingback"
4.3.3.13. Link type "prefetch"
4.3.3.14. Link type "search"
4.3.3.15. Link type "stylesheet"
4.3.3.16. Link type "sidebar"
4.3.3.17. Link type "tag"
4.3.3.18. Hierarchical link types
4.3.3.18.1. Link type "first"
4.3.3.18.2. Link type "index"
4.3.3.18.3. Link type "last"
4.3.3.18.4. Link type "next"
4.3.3.18.5. Link type "prev"
4.3.3.18.6. Link type "up"
4.3.3.19. Other link types
4.3.4. Image maps
4.4. Commands
4.4.1. Using the a element to define a command
4.4.2. Using the button element to define a command
4.4.3. Using the input element to define a command
4.4.4. Using the option element to define a command
4.4.5. Using the command element to define a command
4.5. Menus
4.5.1. Introduction
4.5.2. Building menus
4.5.3. Context menus
4.5.4. Toolbars
4.6. Repetition templates
5. The browser environment
5.1. The global scope
5.2. Session history and navigation
5.2.1. The session history of browsing contexts
5.2.2. The History interface
5.2.3. Activating state objects

5.2.4. The `Location` interface
5.2.5. Implementation notes for session history
5.3. Browser state
5.3.1. Offline Web applications
5.3.2. Custom protocol and content handlers
5.3.2.1. Security and privacy
5.3.2.2. Sample user interface
5.4. Client-side session and persistent storage
5.4.1. Introduction
5.4.2. The `Storage` interface
5.4.3. The `StorageItem` interface
5.4.4. The `sessionStorage` attribute
5.4.5. The `globalStorage` attribute
5.4.6. The `storage` event
5.4.7. Miscellaneous implementation requirements for storage areas
5.4.7.1. Disk space
5.4.7.2. Threads
5.4.8. Security and privacy
5.4.8.1. User tracking
5.4.8.2. Cookie resurrection
5.4.8.3. Integrity of "public" storage areas
5.4.8.4. Cross-protocol and cross-port attacks
5.4.8.5. DNS spoofing attacks
5.4.8.6. Cross-directory attacks
5.4.8.7. Public storage areas corresponding to hosts
5.4.8.8. Storage areas in the face of untrusted higher-level domains that do not correspond to public storage areas
5.4.8.9. Storage areas in the face of untrusted subdomains
5.4.8.10. Implementation risks
5.5. Sound

6. Editing
6.1. Introduction
6.2. The `contenteditable` attribute
6.2.1. User editing actions
6.2.2. Making entire documents editable
6.3. Drag and drop
6.3.1. The `DragEvent` and `DataTransfer` interfaces
6.3.2. Events fired during a drag-and-drop action
6.3.3. Drag-and-drop processing model
6.3.3.1. When the drag-and-drop operation starts or ends in another document
6.3.3.2. When the drag-and-drop operation starts or ends in another application
6.3.4. The `draggable` attribute
6.3.5. Security risks in the drag-and-drop model
6.4. Undo history
6.4.1. The `UndoManager` interface
6.4.2. Undo: moving back in the undo transaction history
6.4.3. Redo: moving forward in the undo transaction history
6.4.4. The `UndoManagerEvent` interface and the `undo` and `redo` events
6.4.5. Implementation notes

6.5. Command 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

7. Communication
7.1. Server-sent DOM events
7.1.1. The `event-source` element
7.1.2. The `RemoteEventTarget` interface
7.1.3. Processing model
7.1.4. The event stream format
7.1.5. Event stream interpretation
7.1.6. The `RemoteEvent` interface
7.1.7. Example
7.2. Network connections
7.2.1. Introduction
7.2.2. The `Connection` interface
7.2.3. Connection Events
7.2.4. TCP connections
7.2.5. Broadcast connections
7.2.5.1. Broadcasting over TCP/IP
7.2.5.2. Broadcasting over Bluetooth
7.2.5.3. Broadcasting over IrDA
7.2.6. Peer-to-peer connections
7.2.6.1. Peer-to-peer connections over TCP/IP
7.2.6.2. Peer-to-peer connections over Bluetooth
7.2.6.3. Peer-to-peer connections over IrDA
7.2.7. The common protocol for TCP-based connections
7.2.7.1. Clients connecting over TCP
7.2.7.2. Servers accepting connections over TCP
7.2.7.3. Sending and receiving data over TCP
7.2.8. Security
7.2.9. Relationship to other standards
7.3. Cross-document messaging
7.3.1. Processing model
7.3.2. Event definitions

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
8.1.3. Text
8.1.3.1. Newlines
8.1.4. Character entity references
8.1.5. Comments
8.2. Parsing HTML documents
8.2.1. Overview of the parsing model
8.2.2. The input stream

8.2.3. Tokenisation
8.2.3.1. Tokenising entities
8.2.4. Tree construction
8.2.4.1. The initial phase
8.2.4.2. The root element phase
8.2.4.3. The main phase
8.2.4.3.1. The stack of open elements
8.2.4.3.2. The list of active formatting elements
8.2.4.3.3. Creating and inserting HTML elements
8.2.4.3.4. Closing elements that have implied end tags
8.2.4.3.5. The element pointers
8.2.4.3.6. The insertion mode
8.2.4.3.7. How to handle tokens in the main phase
8.2.4.4. The trailing end phase
8.3. Namespaces
8.4. Entities

9. Rendering
9.1. Rendering and the DOM

10. Things that you can't do with this specification because they are better handled using other technologies that are further described herein
10.1. Localisation
10.2. Declarative 2D vector graphics and animation
10.3. Declarative 3D scenes
10.4. Alternate style sheets: the `DocumentStyle` interface
10.4.1. Dynamically adding new style sheets
10.4.1.1. Adding style sheets
10.4.1.2. Changing the preferred style sheet set
10.4.1.3. Examples
10.4.2. Interaction with the User Interface
10.4.2.1. Persisting the selected style sheet set
10.4.3. Future compatibility
10.5. Timers

References

Acknowledgements

---

1. Introduction

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.

1.1. 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 addressing presentation concerns (although default rendering rules for Web browsers are included at the end of this specification).

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 targetted 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.

For sophisticated cross-platform applications, there already exist several proprietary solutions (such as Mozilla's XUL and Macromedia's Flash). These solutions are evolving faster than any standards process could follow, and the requirements are evolving even faster. These systems are also significantly more complicated to specify, and are orders of magnitude more difficult to achieve interoperability with, than the solutions described in this document. Platform-specific solutions for such sophisticated applications (for example the MacOS X Core APIs) are even further ahead.

1.2. Structure of this specification

This section is non-normative.

This specification is divided into five key parts:

The DOM

The DOM, or Document Object Model, provides a base for the rest of the specification.

The Semantics

Documents are built from elements. These elements form a tree using the DOM. Each element also has a predefined meaning, which is explained in this section. User agent requirements for how to handle each element are also given, along with rules for authors on how to use the element.

The Processing Models

In addition to the rules for each element, there are rules for handling features that cover multiple elements. These are defined in the processing models section, so that the whole feature can be defined in one place instead of being split across the definitions for multiple elements.

The High-Level APIs

Certain extensions to the DOM provide APIs for interacting with the user. Three sections,

covering the browser enviroment, editing, and communication, define these high-level APIs.

The Language Syntax

All of these features would be for naught if they couldn't be represented in a serialised 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.

1.2.1. 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.

1.3. 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", "SHALL", "SHALL NOT", "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]

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 an XSLT transformation sheet (assuming the UA 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 labelled as `text/html` as described in this specification, so that users can interact with them.

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) 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. 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 do not have to check that `blockquote` elements only contain quoted material).

Conformance checkers must check that the input document conforms when scripting is disabled, and should also check that the input document conforms when scripting is enabled. (This is only a "SHOULD" and not a "MUST" requirement because it has been proven to be impossible. [HALTINGPROBLEM])

The term "validation" specifically refers to a subset of conformance checking that only verifies that a document complies with the requirements given by an SGML or XML DTD. Conformance checkers that only perform validation are non-conforming, as there are many conformance requirements described in this specification that cannot be checked by SGML or XML DTDs.

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.

This needs expanding (see source).

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 behaviour. 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]

Such XML documents may contain a `DOCTYPE` if desired, but this is not required to conform to this specification.

HTML documents, if they are served over the wire (e.g. by HTTP) must be labelled 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.

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.

1.3.1. 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 serialisation with namespaces. [XML] [XMLNAMES]

XML Base

User agents must follow the rules given by XML Base to resolve relative URIs in HTML and XHTML fragments, because that is the mechanism used in this specification for resolving relative URIs in DOM trees. [XMLBASE]

> ***Note: It is possible for `xml:base` attributes to be present even in HTML fragments, as such attributes can be added dynamically using script.***

DOM

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]

Implementations must support some version of the Window Object, because this specification extends this interface to provide some of its features. [WINDOW]

ECMAScript

Implementations that use ECMAScript to implement the APIs defined in this specification must implement them in a manner consistent with the ECMAScript Bindings for DOM Specifications specification, as this specification uses that specification's terminology. [EBFD]

This specification does not require support of any particular network transport protocols, image formats, audio formats, video formats, 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.

1.3.2. Features defined in other specifications

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

Should textContent be defined differently for dir="" and <bdo>? Should we come up with an alternative to textContent that handles those and other things, like alt=""?

The terms **browsing context** and **top-level browsing context** are used as defined in the Window Object specification. [WINDOW]

1.3.3. Relationship to HTML 4.01, XHTML 1.1, DOM2 HTML

This section is non-normative.

This specification represents a new version of HTML4 and XHTML1, along with a new version of the associated DOM2 HTML API. Migration from HTML4 or XHTML1 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.

This specification will eventually supplant Web Forms 2.0 as well. [WF2]

1.3.4. Relationship to XHTML2

This section is non-normative.

XHTML2 [XHTML2] defines a new HTML vocabulary with better 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.

However, it lacks elements to express the semantics of many of the non-document types of content often seen on the Web. For instance, forum sites, auction sites, search engines, online shops, and the like, do not fit the document metaphor well, and are not covered by XHTML2.

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

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

1.3.5. Relationship to XUL, WPF/XAML, and other proprietary UI languages

This section is non-normative.

This specification is independent of the various proprietary UI languages that various vendors provide.

1.4. 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.

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).

The term HTML documents is sometimes used in contrast with XML documents to mean specifically 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 readability, the term URI is used to refer to both ASCII URIs and Unicode IRIs, as those terms are defined by [RFC3986] and [RFC3987] respectively. On the rare occasions where IRIs are not allowed but ASCII URIs are, this is called out explicitly.

The term **root element**, when not qualified to explicitly refer to the document's root element, means the furthest ancestor element node of whatever node is being discussed, or the node itself is there is none. 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.

An element is said to have been **inserted into a document** when its root element changes and is now the document's root element.

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.

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`

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.

Various DOM interfaces are defined in this specification using pseudo-IDL. This looks like OMG IDL but isn't. For instance, method overloading is used, and types from the W3C DOM specifications are used without qualification. Language-specific bindings for these abstract interface definitions must be derived in the way consistent with W3C DOM specifications. Some interface-specific binding information for ECMAScript is included in this specification.

The current situation with IDL blocks is pitiful. IDL is totally inadequate to properly represent what objects have to look like in JS; IDL can't say if a member is enumerable, what the indexing behaviour is, what the stringification behaviour is, what behaviour setting a member whose type is a particular interface should be (e.g. setting of document.location or element.className), what constructor an object implementing an interface should claim to have, how overloads work, etc. I think we should make the IDL blocks non-normative, and/or replace them with something else that is better for JS while still being clear on how it applies to other languages. However, we do

need to have something that says what types the methods take as arguments, since we have to raise exceptions if they are wrong.

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]

The term **text node** refers to any `Text` node, including `CDATASection` nodes (any `Node` with node type 3 or 4).

Some of the algorithms in this specification, for historical reasons, require the user agent to **pause** until some condition has been met. While a user agent is paused, it must ensure that no scripts execute (e.g. no event handlers, no timers, etc). User agents should remain responsive to user input while paused, however.

1.4.1. 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. Generally speaking, authors are discouraged from trying to use XML on the Web, because XML has much stricter syntax rules than the "HTML5" variant described above, and is relatively newer and therefore less mature.

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.

2. The Document Object Model

The Document Object Model (DOM) is a representation — a model — of a document and its content. [DOM3CORE] 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.

This specification defines the language represented in the DOM by features together called DOM5 HTML. DOM5 HTML consists of DOM Core `Document` nodes and DOM Core `Element` nodes, along with text nodes and other content.

Elements in the DOM represent things; that is, they have intrinsic *meaning*, also known as semantics.

> For example, a `p` element represents a paragraph.

In addition, documents and elements in the DOM host APIs that extend the DOM Core APIs, providing new features to application developers using DOM5 HTML.

2.1. Documents

Every XML and HTML document in an HTML UA is represented by a `Document` object. [DOM3CORE]

`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 behaviour 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.***

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 must implement `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`.***

```
interface HTMLDocument {
  // Resource metadata management
  readonly attribute DOMString URL;
           attribute DOMString domain;
  readonly attribute DOMString referrer;
           attribute DOMString cookie;

  // DOM tree accessors
           attribute DOMString title;
```

```
          attribute HTMLElement body;
  readonly attribute HTMLCollection images;
  readonly attribute HTMLCollection links;
  readonly attribute HTMLCollection forms;
  readonly attribute HTMLCollection anchors;
  NodeList getElementsByName(in DOMString elementName);
  NodeList getElementsByClassName(in DOMString[] classNames);

  // Dynamic markup insertion
          attribute DOMString innerHTML;
  void open();
  void open(in DOMString type);
  void open(in DOMString type, in DOMString replace);
  void open(in DOMString url, in DOMString name, in DOMString features);
  void open(in DOMString url, in DOMString name, in DOMString features,
  void close();
  void write(in DOMString text);
  void writeln(in DOMString text);

  // Interaction
  readonly attribute Element activeElement;
  readonly attribute boolean hasFocus;

  // Commands
  readonly attribute HTMLCollection commands;

  // Editing
          attribute boolean designMode;
  boolean execCommand(in DOMString commandID);
  boolean execCommand(in DOMString commandID, in boolean doShowUI);
  boolean execCommand(in DOMString commandID, in boolean doShowUI, in DO
  Selection getSelection();

  // Cross-document messaging
  void postMessage(in DOMString message);

  // lots of other stuff to come
};
```

Note: This specification requires that implementations also implement some version of the Window object specification, so all `HTMLDocument` objects also implement the `DocumentWindow` object and thus the `DocumentView` object.

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.

2.1.1. Resource metadata management

The **`URL`** attribute must return the the document's address.

The **`domain`** attribute must be initialised to the document's domain upon the creation of the `Document` object. On getting, the attribute must return its current value. On setting, if the new value

is an allowed value (as defined below), the attribute's value must be changed to the new value. If the new value is not an allowed value, then a security exception must be raised instead.

A new value is an allowed value for the `document.domain` attribute if it is equal to the attribute's current value, or if the new value, prefixed by a U+002E FULL STOP ("."), exactly matches the end of the current value.

> ***Note: The `domain` attribute is used to enable pages on different hosts of a domain to access each others' DOMs.***

The **`referrer`** attribute must return either the URI of the page which navigated the browsing context to the current document (if any), or the empty string (if there is no such originating page, or if the UA has been configured not to report referrers).

> ***Note: In the case of HTTP, the `referrer` DOM attribute will match the `Referer` (sic) header that was sent when fetching the current page.***

The **`cookie`** attribute must, on getting, 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. [RFC2109]

On setting, the `cookie` attribute must cause the user agent to 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. [RFC2109]

2.2. Elements

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 XHTML elements in XML documents, even when those documents are in another context (e.g. inside an XSLT transform).

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;

  // Metadata attributes
           attribute DOMString id;
           attribute DOMString title;
           attribute DOMString lang;
           attribute DOMString dir;
           attribute DOMTokenString className;

  // Interaction
```

```
           attribute long tabindex;
  void click();
  void focus();
  void blur();

  // Commands
           attribute HTMLMenuElement contextMenu;

  // Editing
           attribute boolean draggable;
           attribute DOMString contenteditable;

  // event handler DOM attributes
           attribute EventListener onclick;
           ...more events...
};
```

As with the `HTMLDocument` interface, 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.

2.2.1. 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.

If a reflecting DOM attribute is a `DOMString` attribute defined to contain a URI, then on getting, the DOM attribute must return the value of the content attribute, resolved to an absolute URI, 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 that is not defined to contain a URI, then the getting and setting must be done in a transparent, case-sensitive manner, except if the content attribute is defined to only allow a specific set of values. In this latter case, the attribute's value must first be converted to lowercase before being returned. 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 boolean attribute, then 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.

If a reflecting DOM attribute is a signed integer type (`long`) then the content attribute must be parsed according to the rules for parsing signed integers first. If that fails, 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 a 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 the content attribute must be parsed according to the rules for parsing unsigned integers first. If that fails, 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 a 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 of the type `DOMTokenString`, then on getting it must return the `DOMTokenString` object that represents the element's corresponding content attribute, and on setting, the *string representation of the new value* must replace the value of the element's `class` content attribute, adding such an attribute if the element doesn't have one. The setter is unusual in that it expects a string, and does not affect the `DOMTokenString` object that is returned on getting. The same `DOMTokenString` object must be returned every time. When the attribute is absent, then the string represented by the `DOMTokenString` object is the empty string.

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.

2.3. Common DOM interfaces

2.3.1. 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.

2.3.1.1. HTMLCollection

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

```
interface HTMLCollection {
```

```
  readonly attribute unsigned long length;
  Element item(in unsigned long index);
  Element namedItem(in DOMString name);
};
```

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

- It is an a, applet, area, form, img, or object element with a name attribute equal to *key*, or,
- It is an HTML element of any kind with an id attribute equal to *key*. (Non-HTML elements, even if they have IDs, are not searched for the purposes of namedItem().)

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

In the ECMAScript DOM binding, objects implementing the HTMLCollection interface must support being dereferenced using the square bracket notation, such that dereferencing with an integer index is equivalent to invoking the item() method with that index, and such that dereferencing with a string index is equivalent to invoking the namedItem() method with that index.

2.3.1.2. HTMLFormControlsCollection

The HTMLFormControlsCollection interface represents a collection of form controls.

```
interface HTMLFormControlsCollection {
  readonly attribute unsigned long length;
  HTMLElement item(in unsigned long index);
  Object namedItem(in DOMString name);
};
```

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 **namedItem(*key*)** 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 *key*, 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 *key*, 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 *key*. The nodes in the NodeList object must be sorted in tree order.
4. Return that NodeList object.

In the ECMAScript DOM binding, objects implementing the `HTMLFormControlsCollection` interface must support being dereferenced using the square bracket notation, such that dereferencing with an integer index is equivalent to invoking the `item()` method with that index, and such that dereferencing with a string index is equivalent to invoking the `namedItem()` method with that index.

2.3.1.3. HTMLOptionsCollection

The `HTMLOptionsCollection` interface represents a list of `option` elements.

```
interface HTMLOptionsCollection {
           attribute unsigned long length;
  HTMLOptionElement item(in unsigned long index);
  Object namedItem(in DOMString name);
};
```

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

On setting, the behaviour 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 **`namedItem(`*`key`*`)`** 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 *key*, 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 *key*, 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 *key*. The nodes in the `NodeList` object must be sorted in tree order.
4. Return that `NodeList` object.

In the ECMAScript DOM binding, objects implementing the `HTMLOptionsCollection` interface must support being dereferenced using the square bracket notation, such that dereferencing with an integer index is equivalent to invoking the `item()` method with that index, and such that dereferencing with a string index is equivalent to invoking the `namedItem()` method with that index.

We may want to add `add()` and `remove()` methods here too because IE implements HTMLSelectElement and HTMLOptionsCollection on the same object, and so people use them almost interchangeably in the wild.

2.3.2. DOMTokenString

The `DOMTokenString` interface represents a string that consists of an unordered set of space-separated tokens.

```
interface DOMTokenString : DOMString {
  boolean has(in DOMString token);
  void add(in DOMString token);
  void remove(in DOMString token);
};
```

The **`has(token)`** method must run the following algorithm:

1. If the *token* argument contains any spaces, 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 spaces, 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 `DOMTokenString` object's underlying string then stop the algorithm.
4. Otherwise, if the last character of the `DOMTokenString` object's underlying 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 `DOMTokenString` object's underlying string.

The **`remove(token)`** method must run the following algorithm:

1. If the *token* argument contains any spaces, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.
2. Otherwise, remove the given *token* from the underlying string.

In the ECMAScript DOM binding, objects implementing the `DOMTokenString` interface must stringify to the object's underlying string representation.

`DOMTokenString` inherits from `DOMString`, so in bindings where strings have attributes or

methods, those attributes and methods will also operate on `DOMTokenString` objects.

2.3.3. 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 guarenteed that an implementation that supports "`HTML`" "`5.0`" also supports "`HTML`" "`2.0`".***

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.

The `title` element of a document is the first `title` element that is a child of the `head` element, if there is one, or null otherwise.

The **`title`** attribute must, on getting, 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 `head` element is null, then the attribute must do nothing. Stop the algorithm here.
2. If the `title` element is null, then a new `title` element must be created and appended to the `head` element.
3. The children of the `title` element (if any) must all be removed.
4. A single `Text` node whose data is the new value being assigned must be appended to the `title` element.

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 followed:

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.

The **`images`** attribute must return an `HTMLCollection` rooted at the `Document` node, whose filter matches only `img` elements.

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 **`getElementsByName(`*`name`*`)`** method a string *name*, and must return a live `NodeList` containing all the `a`, `applet`, `button`, `form`, `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.

The **`getElementsByClassName(`*`classNames`*`)`** method takes an array of strings representing classes. When called, the method must return a live `NodeList` object containing all the elements in the document that have all the classes specified in that array. If the array is empty, then the method must return an empty `NodeList`.

HTML, XHTML, 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 labelled as being in specific classes. UAs must not assume that all attributes of the name `class` for elements in any namespace work in this way, however, and must not assume that such attributes, when used as global attributes, label other elements 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('ccc bbb')` 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.
>
> A call to `getElementsByClassName([''])` would also return no nodes, since none of the nodes are in the "" class (indeed, in HTML, it is impossible to specify that an element is in the "" class).

The **`getElementsByClassName()`** method on the `HTMLElement` interface must return the nodes that the `HTMLDocument` `getElementsByClassName()` method would return, excluding any elements that are not descendants of the `HTMLElement` object on which the method was invoked.

2.5. Dynamic markup insertion

The `document.write()` family of methods and the `innerHTML` family of DOM attributes enable script authors to dynamically insert markup into the document.

Because these APIs interact with the parser, their behaviour varies depending on whether they are used with HTML documents (and the HTML parser) or XHTML in XML documents (and the XML parser). The following table cross-references the various versions of these APIs.

	`document.write()`	**`innerHTML`**
For documents that are HTML documents	`document.write()` in HTML	`innerHTML` in HTML
For documents that are XML documents	`document.write()` in XML	`innerHTML` in XML

Regardless of the parsing mode, the **`document.writeln(`*`s`*`)`** method must call the `document.write()` method with the same argument *s*, and then call the `document.write()` method with, as its argument, a string consisting of a single newline character (U+000A).

2.5.1. Controlling the input stream

The **`open()`** method comes in several variants with different numbers of arguments.

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 has 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.

> ***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. Otherwise, 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?
5. Remove all child nodes of the document.
6. 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).
7. Mark the document as being an HTML document (it might already be so-marked).
8. If *type* does not have the value "`text/html`", then act as if the tokeniser had emitted a `pre` element start tag, then set the HTML parser's tokenisation stage's content model flag to *PLAINTEXT*.
9. If *replace* is false, then: remove all the entries in the browsing context's session history after the current entry in its `DocumentUI`'s `History` object, add a new entry whose address is the same as the current entry's at the end of the list, and then advance to that page as if the `history.forward()` method had been invoked.
10. Finally, set the insertion point to point at just before the end of the input stream (which at this point will be empty).

We shouldn't hard-code `text/plain` there. We should do it some other way, e.g. hand off to the section on content-sniffing and handling of incoming data streams, the part that defines how this all works when stuff comes over the network.

When called with three or more arguments, the `open()` method on the `HTMLDocument` object must call the `open()` method on the `WindowHTML` interface of the object returned by the `defaultView` attribute of the `DocumentView` interface of the `HTMLDocument` object, with the same arguments as the original call to the `open()` method. If the `defaultView` attribute of the `DocumentView` interface of the `HTMLDocument` object is null, 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 insertion point of the parser's input stream.

2.5.2. Dynamic markup insertion in HTML

In HTML, the **`document.write(`*`s`*`)`** method must act as follows:

1. If the insertion point is undefined, the `open()` method must be called (with no arguments) on the `document` object. The insertion point will point at just before the end of the (empty) input stream.
2. The string *s* must be inserted into the input stream just before the insertion point.
3. If there is a script that will execute as soon as the parser resumes, then the method must now return without further processing of the input stream.

4. 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` start 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.

5. Finally, the method must return.

In HTML, the **`innerHTML`** DOM attribute of all `HTMLElement` and `HTMLDocument` nodes returns a serialisation of the node's children using the HTML syntax. On setting, it replaces the node's children with new nodes that result from parsing the given value. The formal definitions follow.

On getting, the `innerHTML` DOM attribute must return the result of running the following algorithm:

1. Let *s* be a string, and initialise it to the empty string.
2. For each child node *child*, in tree order, append the appropriate string from the following list to *s*:

   ↪ **If the child node is an `Element`**

   Append a U+003C LESS-THAN SIGN (<) character, followed by the element's tag name (which is all lowercase).

   For each attribute that the element has, append a U+0020 SPACE character, the attribute's name (which again will be all lowercase), a U+003D EQUALS SIGN (=) character, a U+0022 QUOTATION MARK (") character, the attribute's value, escaped as described below, 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 calls to `innerHTML` serialise an element's attributes in the same order.

   Append a U+003E GREATER-THAN SIGN (>) character.

   If the child node is an `Element` with a tag name that is one of `area`, `base`, `basefont`, `bgsound`, `br`, `col`, `embed`, `frame`, `hr`, `img`, `input`, `link`, `meta`, `param`, `spacer`, or `wbr`, then continue on to the next child node at this point.

   Otherwise, append the value of the *child* element's `innerHTML` DOM attribute (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 (which is again all lowercase), and finally a U+003E GREATER-THAN SIGN (>) character.

   ↪ **If the child node is a `Text` or `CDATASection` node**

   If one of the ancestors of the child node is a `style`, `script`, `xmp`, `iframe`, `noembed`, `noframes`, or `noscript` element, then append the value of the *child* node's `data` DOM attribute literally.

Otherwise, append the value of the *child* node's `data` DOM attribute, escaped as described below.

↪ **If the child 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 the *child* node's `data` DOM attribute, followed by the literal string `-->` (U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN).

↪ **If the child 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 the *child* node's `name` DOM attribute, followed by the literal string `>` (U+003E GREATER-THAN SIGN).

Other nodes types (e.g. `Attr`) cannot occur as children of elements. If they do, the `innerHTML` attribute 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 occurances of the "`&`" character by the string "`&`", any occurances of the "`<`" character by the string "`<`", any occurances of the "`>`" character by the string "`>`", and any occurances 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: If the element's contents are not conformant, it is possible that the roundtripping through `innerHTML` will not work. For instance, if the element is a `textarea` element to which a `Comment` node has been appended, then assigning `innerHTML` to itself will result in the comment being displayed in the text field. Similarly, if, as a result of DOM manipulation, the element contains a comment that contains the literal string "-->", then when the result of serialising the element is parsed, the comment will be truncated at that point and the rest of the comment will be interpreted as markup. Another example would be making a `script` element contain a text node with the text string "</script>".

On setting, if the node is a document, the `innerHTML` DOM attribute must run the following algorithm:

1. Otherwise, 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?

2. The user agent must remove the children nodes of the `Document` whose `innerHTML`

attribute is being set.

3. The user agent must create a new HTML parser, in its initial state, and associate it with the `Document` node.

4. The user agent must place into the input stream for the HTML parser just created the string being assigned into the `innerHTML` attribute.

5. The user agent must start the parser and let it run until it has consumed all the characters just inserted into the input stream. (The `Document` node will have been populated with elements and a `load` event will have fired on its body element.)

Otherwise, if the node is an element, then setting the `innerHTML` DOM attribute must cause the following algorithm to run instead:

1. The user agent must create a new `Document` node, and mark it as being an HTML document.

   The user agent must create a new HTML parser, and associate it with the just created `Document` node.

   > ***Note: Parts marked `innerHTML` case in algorithms in the parser section are parts that only occur if the parser was created for the purposes of handling the setting of an element's `innerHTML` attribute. 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 an `innerHTML` case to occur even when the parser wasn't created for the purposes of handling an element's `innerHTML` attribute, then that is an error in the specification.***

2. The user agent must set the HTML parser's tokenisation stage's content model flag according to the name of the element whose `innerHTML` attribute is being set, as follows:

   ↪ **If it is a `title` or `textarea` element**

   Set the content model flag to *RCDATA*.

   ↪ **If it is a `style`, `script`, `xmp`, `iframe`, `noembed`, `noframes`, or `noscript` element**

   Set the content model flag to *CDATA*.

   ↪ **If it is a `plaintext` element**

   Set the content model flag to *PLAINTEXT*.

   ↪ **Otherwise**

   Set the content model flag to *PCDATA*.

3. The user agent must switch the HTML parser's tree construction stage to the main phase.

4. Let *root* be a new `html` element with no attributes.

5. The user agent must append the element *root* to the `Document` node created above.

6. The user agent must set up the parser's stack of open elements so that it contains just the single element *root*.

7. The user agent must reset the parser's insertion mode appropriately.

8. The user agent must set the parser's `form` element pointer to the nearest node to the element whose `innerHTML` attribute is being set 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.

9. The user agent must place into the input stream for the HTML parser just created the string being assigned into the `innerHTML` attribute.

10. The user agent must start the parser and let it run until it has consumed all the characters just inserted into the input stream.

11. The user agent must remove the children of the element whose `innerHTML` attribute is being set.

12. The user agent must move all the child nodes of the *root* element to the element whose `innerHTML` attribute is being set, preserving their order.

2.5.3. Dynamic markup insertion in XML

In an XML context, the **`document.write(s)`** method must raise an `INVALID_ACCESS_ERR` exception.

The **`innerHTML`** attributes, on the other hand, in an XML context, are usable.

On getting, the `innerHTML` DOM attribute on `HTMLElement`s and `HTMLDocument`s, in an XML context, must return a namespace-well-formed XML representation of the element or document. User agents may adjust prefixes and namespace declarations in the serialisation (and indeed might be forced to do so in some cases to obtain namespace-well-formed XML). [XML] [XMLNS]

On setting, if the node is a document, the `innerHTML` DOM attribute on must run the following algorithm:

1. The user agent must remove the children nodes of the `Document` whose `innerHTML` attribute is being set.

2. The user agent must create a new XML parser.

3. If the `innerHTML` attribute is being set on an element, the user agent must 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.

4. The user agent must feed the parser just created the string being assigned into the `innerHTML` attribute.

5. If the `innerHTML` attribute is being set on an element, the user agent must feed the parser the string corresponding to the end tag of that element.

6. If the parser found a well-formedness error, the attribute's setter must raise a `SYNTAX_ERR` exception and abort these steps.

7. Otherwise, the user agent must take the children of the document, if the attribute is being set on a `Document` node, or of the document's root element, if the attribute is being set on an `Element` node, and append them to the node whose `innerHTML` attribute is being set, preserving their order.

2.6. 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 summarised 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`, `Node.nodeName`, and `Node.localName`

These attributes return tag names in all 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 HTML element, but doesn't support its interfaces, because it really has another tag name not accessible from the DOM APIs.***

`Element.setAttributeNode()`

When an `Attr` node is set on an HTML element, it must have its name lowercased before the element is affected.

> ***Note: This doesn't apply to `Document.setAttributeNodeNS()`.***

`Element.setAttribute()`

When an attribute is set on an HTML element, the name argument must be lowercased before the element is affected.

> ***Note: This doesn't apply to `Document.setAttributeNS()`.***

`Document.getElementsByTagName()` and `Element.getElementsByTagName()`

These methods (but not their namespaced counterparts) must compare the given argument case-insensitively when looking at HTML elements, and case-sensitively 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()`

If the new namespace is the HTML namespace, then the new qualified name must be lowercased before the rename takes place.

3. Semantics and structure of HTML elements

3.1. Introduction

This section is non-normative.

An introduction to marking up a document.

3.2. 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.

Somewhere we need to define case-sensitivity of attribute values. e.g. contenteditable="TrUe" in HTML vs XHTML. Same with leading/trailing whitespace.

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.

3.2.1. 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+000B LINE TABULATION, U+000C FORM FEED (FF), and U+000D CARRIAGE RETURN (CR).

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 Zs characters** means that the user agent must collect a sequence of characters that are in the Unicode character class Zs. In both cases, the collected characters are not used. [UNICODE]

Similarly, when the algorithms below say to

3.2.2. Numbers

3.2.2.1. Unsigned 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).

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 not one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), then return an error.
7. 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).
8. Return *value*.

3.2.2.2. 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.

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*.

3.2.2.3. Real numbers

A string is a **valid floating point number** if it consists of one of more characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), optionally with a single U+002E FULL STOP (".") character somewhere (either before these numbers, in between two numbers, or after the numbers), all optionally prefixed with a U+002D HYPHEN-MINUS ("-") character.

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 a value. 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 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) or U+002E FULL STOP ("."), then return an error.

9. If the next character is U+002E FULL STOP ("."), but either that is the last character or the character after that one is not one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9), then return an error.

10. 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 past the end of *input*, then if *sign* is "positive", return *value*, otherwise return 0-*value*.

    5. Otherwise return to the top of step 10 in the overall algorithm (that's the step within which these substeps find themselves).

11. Otherwise, if the next character is not a U+002E FULL STOP ("."), then if *sign* is "positive", return *value*, otherwise return 0-*value*.

12. The next character is a U+002E FULL STOP ("."). Advance *position* to the character after that.

13. Let *divisor* be 1.

14. If the next character is one of U+0030 DIGIT ZERO (0) .. U+0039 DIGIT NINE (9):

    1. Multiply *divisor* by ten.

    2. Add the value of the current character (0..9) divided by *divisor*, to *value*.

    3. Advance *position* to the next character.

    4. If *position* is past the end of *input*, then if *sign* is "positive", return *value*, otherwise return 0-*value*.

    5. Otherwise return to the top of step 14 in the overall algorithm (that's the step within which these substeps find themselves).

15. Otherwise, if *sign* is "positive", return *value*, otherwise return 0-*value*.

3.2.2.4. 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

Denominator Punctuation Character		Value
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 any characters in the string that are in the Unicode character class Zs (this might match zero characters). [UNICODE]
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 nothing and abort these steps.
11. Set *number2* to the number returned by the sub-algorithm in step 9.
12. 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.
13. 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.
14. 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* 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 guarenteed to be a valid floating point number).

6. Return *number*.

3.2.2.5. Percentages and dimensions

valid non-negative percentages, **rules for parsing dimension values**

3.2.3. 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]

3.2.3.1. Specific moments in time

A string is a **valid datetime** if it has four digits (representing the year), a literal hyphen, two digits (representing the month), a literal hyphen, two digits (representing the day), optionally some spaces, either a literal T or a space, optionally some more spaces, two digits (for the hour), a colon, two digits (the minutes), another colon, two digits (the integer part of the seconds), optionally a decimal point followed by one or more digits (for the fractional part of the seconds), optionally some spaces, and finally either a literal Z (indicating the time zone is UTC), or, a plus sign or a minus sign followed by two digits, a colon, and two digits (for the sign, the hours and minutes of the timezone offset respectively); with the month-day combination being a valid date in the given year according to the Gregorian calendar, the hour values (*h*) being in the range $0 \leq h \leq 23$, the minute values (*m*) in the range $0 \leq m \leq 59$, and the second value (*s*) being in the range $0 \leq h < 60$. [GREGORIAN]

The digits must be characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), the hyphens must be a U+002D HYPHEN-MINUS characters, the T must be a U+0054 LATIN CAPITAL LETTER T, the colons must be U+003A COLON characters, the decimal point must be a U+002E FULL STOP, the Z must be a U+005A LATIN CAPITAL LETTER Z, the plus sign must be a U+002B PLUS SIGN, and the minus U+002D (same as the hyphen).

> The following are some examples of dates written as valid datetimes.
>
> **`"0037-12-13 00:00 Z"`**
>
> 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.
>
> **`"8592-01-01 T 02:09 +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.
>
> 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 Gregorian Calendar.
- The time and timezone components are not optional.
- Dates before the year 0 or after the year 9999 can't be represented as a datetime in this version of HTML.
- Time zones differ based on daylight savings time.

Note: Conformance checkers can use the algorithm below to determine if a datetime is a valid datetime or not.

To **parse a string as a datetime value**, a user agent must apply the following algorithm to the string. This will either return a time in UTC, with associated timezone information for round tripping or display purposes, or nothing, indicating the value is not a valid datetime. If at any point the algorithm says that it "fails", this means that it 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 exactly four characters long, then fail. Otherwise, interpret the resulting sequence as a base ten integer. Let that number be the *year*.
4. 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.
5. 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*.
6. If *month* is not a number in the range 1 ≤ *month* ≤ 12, then fail.
7. Let *maxday* be the number of days in month *month* of year *year*.
8. 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.
9. 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*.
10. If *day* is not a number in the range 1 ≤ *month* ≤ *maxday*, then fail.
11. Collect a sequence of characters that are either U+0054 LATIN CAPITAL LETTER T characters or space characters. If the collected sequence is zero characters long, or if it contains more than one U+0054 LATIN CAPITAL LETTER T character, then fail.
12. 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*.

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

14. 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.

15. 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*.

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

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

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

19. If the character at *position* is a U+003A COLON, then:

    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.

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

21. If *second* is not a number in the range 0 ≤ *hour* < 60, then fail. (The values 60 and 61 are not allowed: leap seconds cannot be represented by datetime values.)

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

23. Skip whitespace.

24. 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*.

25. 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 ≤ *timezone*$_{hours}$ ≤ 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.

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

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

28. Let *timezone* be *timezone*$_{hours}$ hours and *timezone*$_{minutes}$ minutes from UTC.

29. Return *time* and *timezone*.

3.2.3.2. Vaguer moments in time

This section defines **date or time strings**. There are two kinds, **date or time strings in content**, and **date or time strings in attributes**. The only difference is in the handling of whitespace characters.

To parse a date or time string, user agents must use the following algorithm. A date or time string is a *valid* date or time string if the following algorithm, when run on the string, doesn't say the string is invalid.

The algorithm may return nothing (in which case the string will be invalid), or it may return a date, a time, a date and a time, or a date and a time and and a timezone. Even if the algorithm returns one or more values, the string can still be invalid.

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 *results* be the collection of results that are to be returned (one or more of a date, a time, and a timezone), initially empty. If the algorithm aborts at any point, then whatever is currently in *results* must be returned as the result of the algorithm.

4. For the "in content" variant: skip Zs characters; for the "in attributes" variant: skip whitespace.

5. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is empty, then the string is invalid; abort these steps.

6. Let the sequence of characters collected in the last step be *s*.

7. If *position* is past the end of *input*, the string is invalid; abort these steps.

8. If the character at *position* is *not* a U+003A COLON character, then:

1. If the character at *position* is not a U+002D HYPHEN-MINUS ("-") character either, then the string is invalid, abort these steps.

2. If the sequence *s* is not exactly four digits long, then the string is invalid. (This does not stop the algorithm, however.)

3. Interpret the sequence of characters collected in step 5 as a base ten integer, and let that number be *year*.

4. Advance *position* past the U+002D HYPHEN-MINUS ("-") character.

5. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is empty, then the string is invalid; abort these steps.

6. If the sequence collected in the last step is not exactly two digits long, then the string is invalid.

7. Interpret the sequence of characters collected two steps ago as a base ten integer, and let that number be *month*.

8. If *month* is not a number in the range 1 ≤ *month* ≤ 12, then the string is invalid, abort these steps.

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

10. If *position* is past the end of *input*, or if the character at *position* is *not* a U+002D HYPHEN-MINUS ("-") character, then the string is invalid, abort these steps. Otherwise, advance *position* to the next character.

11. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is empty, then the string is invalid; abort these steps.

12. If the sequence collected in the last step is not exactly two digits long, then the string is invalid.

13. Interpret the sequence of characters collected two steps ago as a base ten integer, and let that number be *day*.

14. If *day* is not a number in the range 1 ≤ *day* ≤ *maxday*, then the string is invalid, abort these steps.

15. Add the date represented by *year*, *month*, and *day* to the *results*.

16. For the "in content" variant: skip Zs characters; for the "in attributes" variant: skip whitespace.

17. If the character at *position* is a U+0054 LATIN CAPITAL LETTER T, then move *position* forwards one character.

18. For the "in content" variant: skip Zs characters; for the "in attributes" variant: skip whitespace.

19. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is empty, then the string is invalid; abort these steps.

20. Let *s* be the sequence of characters collected in the last step.

9. If *s* is not exactly two digits long, then the string is invalid.

10. Interpret the sequence of characters collected two steps ago as a base ten integer, and let that number be *hour*.

11. If *hour* is not a number in the range 0 ≤ *hour* ≤ 23, then the string is invalid, abort these steps.

12. If *position* is past the end of *input*, or if the character at *position* is *not* a U+003A COLON character, then the string is invalid, abort these steps. Otherwise, advance *position* to the next character.

13. Collect a sequence of characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is empty, then the string is invalid; abort these steps.

14. If the sequence collected in the last step is not exactly two digits long, then the string is invalid.

15. Interpret the sequence of characters collected two steps ago as a base ten integer, and let that number be *minute*.

16. If *minute* is not a number in the range 0 ≤ *minute* ≤ 59, then the string is invalid, abort these steps.

17. Let *second* be 0. It may be changed to another value in the next step.

18. If *position* is not past the end of *input* and the character at *position* is a U+003A COLON character, then:

    1. Collect a sequence of characters that are either characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9) or are U+002E FULL STOP. If the collected sequence is empty, or contains more than one U+002E FULL STOP character, then the string is invalid; abort these steps.

    2. If the first character in the sequence collected in the last step is not in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), then the string is invalid.

    3. Interpret the sequence of characters collected two steps ago as a base ten number (possibly with a fractional part), and let that number be *second*.

    4. If *second* is not a number in the range 0 ≤ *minute* < 60, then the string is invalid, abort these steps.

19. Add the time represented by *hour*, *minute*, and *second* to the *results*.

20. If *results* has both a date and a time, then:

    1. For the "in content" variant: skip Zs characters; for the "in attributes" variant: skip whitespace.

    2. If *position* is past the end of *input*, then skip to the next step in the overall set of steps.

    3. Otherwise, if the character at *position* is a U+005A LATIN CAPITAL LETTER Z, then:

        1. Add the timezone corresponding to UTC (zero offset) to the *results*.

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

3. Skip to the next step in the overall set of steps.

4. 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 the string is invalid.

    4. Interpret the sequence collected in the last step as a base ten number, and let that number be $timezone_{hours}$.

    5. If $timezone_{hours}$ is not a number in the range $0 \leq timezone_{hours} \leq 23$, then the string is invalid; abort these steps.

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

    7. If *position* is beyond the end of *input* or if the character at *position* is not a U+003A COLON character, then the string is invalid; abort these steps. Otherwise, move *position* forwards one character.

    8. 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 the string is invalid.

    9. Interpret the sequence collected in the last step as a base ten number, and let that number be $timezone_{minutes}$.

    10. If $timezone_{minutes}$ is not a number in the range $0 \leq timezone_{minutes} \leq 59$, then the string is invalid; abort these steps.

    11. Add the timezone corresponding to an offset of $timezone_{hours}$ hours and $timezone_{minutes}$ minutes to the *results*.

    12. Skip to the next step in the overall set of steps.

5. Otherwise, the string is invalid; abort these steps.

21. For the "in content" variant: skip Zs characters; for the "in attributes" variant: skip whitespace.

22. If *position* is *not* past the end of *input*, then the string is invalid.

23. Abort these steps (the string is parsed).

3.2.4. Tokens

An **unordered 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 an unordered set of space-separated tokens may have leading or trailing space characters.

Note: Thus, any string is a valid representation of an unordered set of space-separated tokens.

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.

3.3. Documents and document fragments

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 only use elements, attributes, and attribute values for their appropriate semantic purposes.

> 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. 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/"><cite>Ernest</cite></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. Structure

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. Authors must only put elements inside an element if that element allows them to be there according to its content model.

> ***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 **preceeded 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 only use elements in the HTML namespace in the contexts where they are allowed, as defined for each element. For XML compound documents, these contexts could be inside elements from other namespaces, if those elements are defined as providing the relevant contexts.

> The SVG specification defines the SVG `foreignObject` element as allowing foreign namespaces to be included, thus allowing compound documents to be created by inserting

> subdocument content under that element. *This* specification defines the XHTML `html` element as being allowed where subdocument fragments are allowed in a compound document. Together, these two definitions mean that placing an XHTML `html` element as a child of an SVG `foreignObject` element is conforming.

3.3.3. Kinds of elements

Each element in HTML falls into zero or more categories that group elements with similar characteristics together. This specification uses the following categories:

- Metadata elements
- Sectioning elements
- Block-level elements
- Strictly inline-level content
- Structured inline-level elements
- Interactive elements
- Form control elements

Some elements have unique requirements and do not fit into any particular category.

In addition, some elements represent various common concepts; for example, some elements represent paragraphs.

3.3.3.1. Block-level elements

Block-level elements are used for structural grouping of page content.

There are several kinds of block-level elements:

- Some can only contain other block-level elements: `blockquote`, `section`, `article`, `header`.
- Some can only contain inline-level content: `p`, `h1`-`h6`, `address`.
- Some can contain either block-level elements or inline-level content (but not both): `nav`, `aside`, `footer`, `div`.
- Finally, some have very specific content models: `ul`, `ol`, `dl`, `table`, `script`.

There are also elements that seem to be block-level but aren't, such as `body`, `li`, `dt`, `dd`, and `td`. These elements are allowed only in specific places, not simply anywhere that block-level elements are allowed.

Some block-level elements play multiple roles. For instance, the `script` elements is allowed inside `head` elements and can also be used as inline-level content. Similarly, the `ul`, `ol`, `dl`, `table`, and `blockquote` elements play dual roles as both block-level and inline-level elements.

3.3.3.2. Inline-level content

Inline-level content consists of text and various elements to annotate the text, as well as some embedded content (such as images or sound clips).

Inline-level content comes in various types:

Strictly inline-level content

Text, embedded content, and elements that annotate the text without introducing structural grouping. For example: `a`, `meter`, `img`. Elements used in contexts allowing only strictly inline-

level content must not contain anything other than strictly inline-level content.

Structured inline-level elements

Block-level elements that can also be used as inline-level content. For example: `ol`, `blockquote`, `table`.

Some elements are defined to have as a content model **significant inline content**. This means that at least one descendant of the element must be significant text or embedded content.

Unless an element's content model explicitly states that it must contain significant inline content, simply having no text nodes and no elements satisfies an element whose content model is some kind of inline content.

Significant text, for the purposes of determining the presence of significant inline content, consists of any character other than those falling in the Unicode categories Zs, Zl, Zp, Cc, and Cf. [UNICODE]

> The following three paragraphs are non-conforming because their content model is not satisfied (they all count as empty).
>
> ```
> 
>  
> 
>  <ol>
>   <li></li>
>  </ol>
> 
> ```

Embedded content consists of elements that introduce content from other resources into the document, for example `img`. 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.

3.3.3.3. Determining if a particular element contains block-level elements or inline-level content

Some elements are defined to have content models that allow either block-level elements or inline-level content, but not both. For example, the `aside` and `li` elements.

To establish whether such an element is being used as a block-level container or as an inline-level container, for example in order to determine if a document conforms to these requirements, user agents must look at the element's child nodes. If any of the child nodes are not allowed in block-level contexts, then the element is being used for inline-level content. If all the child nodes are allowed in a block-level context, then the element is being used for block-level elements.

> For instance, in the following (non-conforming) fragment, the `li` element is being used as an inline-level element container, because the `style` element is not allowed in a block-level context. (It doesn't matter, for the purposes of determining whether it is an inline-level or block-level context, that the `style` element is not allowed in inline-level contexts either.)
>
> ```
> <ol>
>  <li>
>    Hello World 
>   <style>
>    /* This example is illegal. */
>   </style>
> ```

```
 </li>
</ol>
```

In the following fragment, the `aside` element is being used as a block-level container, because even though all the elements it contains could be considered inline-level elements, there are no nodes that can only be considered inline-level.

```
<aside>
 <ol>
  <li> ... </li>
 </ol>
 <ul>
  <li> ... </li>
 </ul>
</aside>
```

On the other hand, in the following similar fragment, the `aside` element is an inline-level container, because the text ("Foo") can only be considered inline-level.

```
<aside>
 <ol>
  <li> ... </li>
 </ol>
 Foo
</aside>
```

3.3.3.4. Interactive elements

Certain elements in HTML can be activated, for instance `a` elements, `button` elements, or `input` elements when their `type` attribute is set to `radio`. Activation of those elements can happen in various (UA-defined) ways, for instance via the mouse or keyboard.

When activation is performed via some method other than clicking the pointing device, the default action of the event that triggers the activation must, instead of being activating the element directly, be to fire a `click` event on the same element.

The default action of this `click` event, or of the real `click` event if the element was activated by clicking a pointing device, must be to dispatch yet another event, namely `DOMActivate`. It is the default action of *that* event that then performs the actual action.

For certain form controls, this process is complicated further by changes that must happen around the click event. [WF2]

Note: Most interactive elements have content models that disallowed nesting interactive elements.

Need to define how default actions actually work. For instance, if you click an event inside a link, the event is triggered on that element, but then we'd like a click is sent on the link itself. So how does that happen? Does the link have a bubbling listener that triggers that second click event? what if there are multiple nested links, which one should we send that event to?

3.3.3.5. 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 can be represented by several elements. The `address` element always represents a paragraph of contact information for its section, the `aside`, `nav`, `footer`, `li`, and `dd` elements represent paragraphs with various specific semantics when they are used as inline-level content containers, the `figure` element represents a paragraph in the form of embedded content, and the `p` element represents all the other kinds of paragraphs, for which there are no dedicated elements.

3.4. Global attributes

The following attributes are common to and may be specified on all HTML elements (even those not defined in this specification):

Global attributes:

`class`
`contenteditable`
`contextmenu`
`dir`
`draggable`
`id`
`lang`
`tabindex`
`title`

In addition, the following event handler content attributes may be specified on any HTML element:

- `onclick`
- ...more events...

3.4.1. The `id` attribute

The `id` attribute represents its element's unique identifier. The value must be unique in the subtree within which the element finds itself and must contain at least one character.

If the value is not the empty string, user agents must associate the element with the given value (exactly) for the purposes of ID matching within the subtree the element finds itself (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.

3.4.2. 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 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.

Some elements, such as `link` and `dfn`, define additional semantics for the `title` attribute beyond the semantics described above.

The **`title`** DOM attribute must reflect the `title` content attribute.

3.4.3. The `lang` (HTML only) and `xml:lang` (XML only) 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 is defined in XML. [XML]

If these attributes are omitted from an element, then it implies that the language of this element is the same as the language of the parent element. Setting the attribute to the empty string indicates that the primary language is unknown.

The `lang` attribute may only be used on elements of HTML documents. Authors must not use the `lang` attribute in XML documents.

The `xml:lang` attribute may only be used on elements of XML documents. Authors must not use the `xml:lang` attribute in HTML documents.

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 a `lang` or `xml:lang` attribute set. That 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, 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).

User agents may use the element's language to determine proper processing or rendering (e.g. in the selection of appropriate fonts or pronounciations, or for dictionary selection).

The **`lang`** DOM attribute must reflect the `lang` content attribute.

3.4.4. The `dir` attribute

redefine this in terms of a microsyntax

The `dir` attribute specifies the element's text directionality. The attribute, if specified, must have either the literal value `ltr` or the literal value `rtl`.

If the attribute has the literal value `ltr`, the element's directionality is left-to-right. If the attribute has the literal value `rtl`, the element's directionality is right-to-left. If the attribute is omitted or has another value, then the directionality is unchanged.

The processing of this attribute depends on the presentation layer. For example, CSS 2.1 defines a mapping from this attribute to the CSS 'direction' and 'unicode-bidi' properties, and defines rendering in terms of those property.

The **dir** DOM attribute must reflect the `dir` content attribute.

3.4.5. Classes

Every HTML element may have a **class** attribute specified.

The attribute, if specified, must have a value that is an unordered set of 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: Assinging 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 predefined values defined in this specification where appropriate.

Classes that are not defined in this specification can be defined as extensions, as described below.

Authors should bear in mind that using the `class` attribute does not convey any additional meaning to the element unless the classes used have been defined by this specification or registered in the Wiki. There is no semantic difference between an element *with* a class attribute and one *without*. Authors that use classes that are not so defined should make sure, therefore, that their documents make as much sense once all `class` attributes have been removed as they do with the attributes present. User agents should not derive particular meanings from `class` attribute values that are neither defined by this specification nor registered in the Wiki.

The **className** DOM attribute must reflect the `class` content attribute.

3.4.5.1. Predefined class names

The following table summarises the class names that are defined by this specification. This table is non-normative; the actual definitions for the class names are given in the next few sections.

Class name	Applicable elements	Brief description
`copyright`	`p`, `span`	Indicates that the element contains the copyright for the document.
`error`	`p`, `section`, `span`, `strong`	Indicates that the element contains an error message, e.g. in an error message to the user in an interactive application, or an error notification in the report of a conformance checker.
`example`	`aside`, `figure`, `p`, `section`, `span`	Indicates that the element contains some sample material or a worked example illustrating the neighboring content.
`issue`	`aside`, `p`, `span`	Indicates that the element contains a point in dispute or a problem with the neighboring content that needs resolving. (Thus, this is typically only found in works-in-progress or drafts, not in completed documents.)

Class name	Applicable elements	Brief description
note	aside, p, span	Indicates that the element contains an explanatory note or clarification.
search	aside, body, form, p, section, span	Indicates that the element's primary purpose is to provide the user with an interface to perform a search (e.g. of the site's pages or of a database).
warning	article, aside, body, figure, p, section, span, strong	Indicates that the element contains a warning or admonition.

3.4.5.1.1. Class name "copyright"

The copyright class name indicates that the element contains the copyright for the document.

It must only be used on the following elements: p, span

3.4.5.1.2. Class name "error"

The error class name indicates that the element contains an error message, e.g. in an error message to the user in an interactive application, or an error notification in the report of a conformance checker.

It must only be used on the following elements: p, section, span, strong

It cannot be used on the body element; if the whole document is an error message then the appropriate metadata should be included at the network layer (e.g. with HTTP using the 4xx and 5xx status codes).

3.4.5.1.3. Class name "example"

The example class name indicates that the element contains some sample material or a worked example illustrating the neighboring content.

It must only be used on the following elements: aside, figure, p, span

3.4.5.1.4. Class name "issue"

The issue class name indicates that the element contains a point in dispute or a problem with the neighboring content that needs resolving. (Thus, this is typically only found in works-in-progress or drafts, not in completed documents.)

It must only be used on the following elements: aside, p, span

3.4.5.1.5. Class name "note"

The note class name indicates that the element contains an explanatory note or clarification.

It must only be used on the following elements: aside, p, span

3.4.5.1.6. Class name "search"

The search class name indicates that the element's primary purpose is to provide the user with an

interface to perform a search (e.g. of the site's pages or of a database).

It must only be used on the following elements: `aside`, `body`, `form`, `p`, `section`, `span`

3.4.5.1.7. Class name "`warning`"

The `warning` class name indicates that the element contains a warning or admonition.

It must only be used on the following elements: `article`, `aside`, `body`, `figure`, `p`, `section`, `span`, `strong`

3.4.5.2. Other classes

Extensions to the predefined set of class keywords may be registered in the WHATWG Wiki ClassExtensions page.

Anyone is free to edit the WHATWG Wiki ClassExtensions page at any time to add a type. New classes must be specified with the following information:

Keyword

The actual class name being defined. The class name should not be confusingly similar to any other defined class name (e.g. differing only in case).

Applicable elements

A list of HTML elements to which the class applies, or one of the following values:

Metadata elements
Meaning any element described as being a metadata element.

Sectioning elements
Meaning any element described as being a sectioning element.

Block-level elements
Meaning any element described as being a block-level element, but only when that element is actually being used as a block-level element, and not, say, as a structured inline-level element.

Strictly inline-level elements
Meaning any element described as being strictly inline-level content.

Structured inline-level elements
Meaning any element described as being a structured inline-level element.

Interactive
Meaning any element described as being an interactive element.

Embedded content elements
Meaning any element described as being embedded content.

All elements
Meaning any element.

A document must not use a class defined in the Wiki on an element other than the elements that the Wiki says that class name is allowed on.

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 should 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. Someone has proposed it and is using it.

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 should use the keyword to mark up the semantic that it describes.

Unendorsed

The keyword 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 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.

Conformance checkers must use the information given on the WHATWG Wiki ClassExtensions page to establish if a value not explicitly defined in this specification is defined, and if so, whether it is being used on the right elements. When an author uses a new class name not defined by either this specification or the Wiki page, conformance checkers may 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.

3.5. Interaction

3.5.1. Activation

The **click()** method must fire a `click` event at the element.

3.5.2. Focus

When an element is *focused*, key events received by the document must be targetted at that element. There is always an element focused; in the absence of other elements being focused, the document's root element is it.

Which element within a document currently has focus is independent of whether or not the document itself has the *system focus*.

Some focusable elements might take part in *sequential focus navigation*.

3.5.2.1. Focus management

The **`focus()`** and **`blur()`** methods must focus and unfocus the element respectively, if the element is focusable.

Well that clearly needs more.

The **`activeElement`** attribute must return the element in the document that has focus. If no element specifically has focus, this must return the root element.

The **`hasFocus`** attribute must return true if the document, one of its nested browsing contexts, or any element in the document or its browsing contexts currently has the system focus.

3.5.2.2. Sequential focus navigation

The **`tabindex`** attribute specifies the relative order of elements 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.

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. If parsing the value returns an error, the attribute is ignored for the purposes of focus management (as if it wasn't specified).

A positive integer or zero specifies the index of the element in the current scope's tab order. Elements with the same index are sorted in tree order for the purposes of tabbing.

A negative integer specifies that the element should be removed from the tab order. If the element does normally take focus, it may still be focused using other means (e.g. it could be focused by a click).

If the attribute is absent (or invalid), then the user agent must treat the element as if it had the value 0 or the value -1, based on platform conventions.

> For example, a user agent might default `textarea` elements to 0, and `button` elements to -1, making text fields part of the tabbing cycle but buttons not.

When an element that does not normally take focus (i.e. whose default value would be -1) has the `tabindex` attribute specified with a positive value, then it should be added to the tab order and should be made focusable. When focused, the element matches the CSS `:focus` pseudo-class and key events are dispatched on that element in response to keyboard input.

The **`tabIndex`** DOM attribute reflects the value of the `tabIndex` content attribute. If the attribute is not present (or has an invalid value) then the DOM attribute must return the UA's default value for that element, which will be either 0 (for elements in the tab order) or -1 (for elements not in the tab order).

3.6. The root element

3.6.1. The `html` element

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.

Element-specific attributes:

None (but see prose).

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `html` element represents the root of an HTML document.

Though it has absolutely no effect and no meaning, the `html` element, in HTML documents, 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 the null 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 the null namespace specified.

3.7. Document metadata

Document metadata is represented by **metadata elements** in the document's `head` element.

3.7.1. The `head` element

Contexts in which this element may be used:

As the first element in an `html` element.

Content model:

In any order, exactly one `title` element, optionally one `base` element (HTML only), and zero or more other metadata elements (in particular, `link`, `meta`, `style`, and `script`).

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `head` element collects the document's metadata.

3.7.2. The `title` element

Metadata element.

Contexts in which this element may be used:

In a `head` element containing no other `title` elements.

Content model:

Text (for details, see prose).

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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.

> 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 `title` element must not contain any elements.

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.

3.7.3. The `base` element

Metadata element.

Contexts in which this element may be used:

In a `head` element, before any elements that use relative URIs, and only if there are no other `base` elements anywhere in the document. Only in HTML documents (never in XML documents).

Content model:

Empty.

Element-specific attributes:

`href`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLBaseElement : HTMLElement {
           attribute DOMString href;
};
```

The `base` element allows authors to specify the document's base URI for the purposes of resolving relative URIs.

The **`href`** content attribute, if specified, must contain a URI (or IRI).

User agents must use the value of the `href` attribute on the first `base` element in the document as the document entity's base URI for the purposes of section 5.1.1 of RFC 2396 ("Establishing a Base URI": "Base URI within Document Content"). [RFC2396] Note that this base URI from RFC 2396 is referred to by the algorithm given in XML Base, which is a normative part of this specification.

If the base URI given by this attribute is a relative URI, it must be resolved relative to the higher-level base URIs (i.e. the base URI from the encapsulating entity or the URI used to retrieve the entity) to obtain an absolute base URI.

The `href` content attribute must be reflected by the DOM **`href`** attribute.

Authors must not use the `base` element in XML documents. Authors should instead use the `xml:base` attribute. [XMLBASE]

3.7.4. The `link` element

Metadata element.

Contexts in which this element may be used:

In a `head` element.

Content model:

Empty.

Element-specific attributes:

`href` (required)
`rel` (required)
`media`
`hreflang`
`type`
Also, the `title` attribute has special semantics on this element.

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLLinkElement : HTMLElement {
           attribute boolean disabled;
           attribute DOMString href;
           attribute DOMString rel;
           attribute DOMString media;
           attribute DOMString hreflang;
           attribute DOMString type;
};
```

The `LinkStyle` interface defined in DOM2 Style must also be implemented by this element. [DOM2STYLE]

The `link` element allows authors to indicate explicit relationships between their document and other resources.

The destination of the link is given by the **`href`** attribute, which must be present and must contain a URI (or IRI). If the `href` attribute is absent, then the element does not define a link.

The type of link indicated (the relationship) is given by the value of the **`rel`** attribute, which must be present. The allowed values and their meanings are defined in a later section. If the `rel` attribute is absent, or if the value used is 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). User agents should process the links on a per-link basis, not a per-element basis.

The exact behaviour 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. (However, user agents may opt to only fetch such resources when they are needed, instead of pro-actively downloading all the external resources that are not applied.)

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 URI 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).

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 only apply the external resource to views while their state match the listed media.

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, user agents may use the type given in this attribute to decide whether or not to consider using the resource at all. If the UA does not support the given MIME type for the given link relationship, then the UA may opt not to download and apply the resource.

User agents must not consider the `type` attribute authoritative — upon fetching the resource, user agents must only use the Content-Type information associated with the resource to determine its type, not metadata included in the link to the resource.

If the attribute is omitted, then the UA must fetch the resource to determine its type and thus determine if it supports (and can apply) that external resource.

> If a document contains three style sheet links labelled as follows:
>
> ```
> <link rel="stylesheet" href="A" type="text/css">
> <link rel="stylesheet" href="B" type="text/plain">
> <link rel="stylesheet" href="C">
> ```
>
> ...then a compliant UA that supported only CSS style sheets would fetch the A and C files, and skip the B file (since `text/plain` is not the MIME type for CSS style sheets). For these two files, it would then check the actual types returned by the UA. For those that are sent as `text/css`, it would apply the styles, but for those labelled as `text/plain`, or any other type, it would not.

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 alternate 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.***

Some versions of HTTP defined a `Link:` header, to be processed like a series of `link` elements. When processing links, those must be taken into consideration as well. 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. Relative URIs in these headers must be resolved according to the rules given in HTTP, not relative to base URIs set by the document (e.g.

using a `base` element or `xml:base` attributes). [RFC2616] [RFC2068]

The DOM attributes **`href`**, **`rel`**, **`media`**, **`hreflang`**, and **`type`** each must reflect the respective content attributes of the same name.

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 alternate style sheets DOM. For all other `link` elements it must always return false and must do nothing on setting.

3.7.5. The `meta` element

Metadata element.

Contexts in which this element may be used:

In a `head` element.

Content model:

Empty.

Element-specific attributes:

`name`
`http-equiv` (HTML only, optional)
`content`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLMetaElement : HTMLElement {
           attribute DOMString content;
           attribute DOMString name;
};
```

The `meta` element allows authors to specify document metadata that cannot be expressed using the `title`, `base`, `link`, `style`, and `script` elements. The metadata is expressed in terms of name/value pairs: the **`name`** attribute on the `meta` element gives the name, and the **`content`** attribute on the same element gives the value.

If a `meta` element has no `name` attribute, it does not set document metadata. If a `meta` element has no `content` attribute, then the value part of the metadata name/value pair is the empty string.

The value must not be a URI. Links must be represented using the `link` element, not the `meta` element.

The DOM attributes **`name`** and **`content`** must reflect the respective content attributes of the same name.

3.7.5.1. Standard metadata names

This specification defines a few names for the `name` attribute of the `meta` element.

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.

3.7.5.2. 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.

3.7.5.3. Specifying and establishing the document's character encoding

The `meta` element may also be used, in HTML only (not in XHTML) to provide UAs with character encoding information for the file. To do this, the `meta` element must be the first element in the `head` element, it must have the **`http-equiv`** attribute set to the literal value `Content-Type`, and must have the `content` attribute set to the literal value `text/html; charset=` immediately followed by the character encoding, which must be a valid character encoding name. [IANACHARSET] When the `meta` element is used in this way, there must be no other attributes set on the element, and the `http-equiv` attribute must be listed first in the source. Other than for giving the document's character encoding in this way, the `http-equiv` attribute must not be used.

We should allow those strings to be case-insensitive, and for zero-or-more spaces where we currently require just one.

In XHTML, the XML declaration should be used for inline character encoding information, if necessary.

Authors should avoid including inline character encoding information. Character encoding information should instead be included at the transport level (e.g. using the HTTP `Content-Type` header).

3.7.6. The `style` element

Metadata element.

Contexts in which this element may be used:

In a `head` element.

Content model:

Depends on the value of the `type` attribute.

Element-specific attributes:

`media`
`type`
Also, the `title` attribute has special semantics on this element.

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLStyleElement : HTMLElement {
           attribute boolean disabled;
           attribute DOMString media;
           attribute DOMString type;
};
```

The `LinkStyle` interface defined in DOM2 Style must also be implemented by this element. [DOM2STYLE]

The `style` element allows authors to embed style information in their documents.

If the **`type`** attribute is given, it must contain a MIME type, optionally with parameters, that designates a styling language. [RFC2046] If the attribute is absent, the type defaults to `text/css`. [RFC2138]

If the UA supports the given styling language, then the UA must use the given styles as appropriate for that language.

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 only apply the styles to views while their state match the listed media.

The default, if the media attribute is omitted, is all, meaning that by default styles apply to all media.

The **title** attribute on style elements defines alternate 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 children nodes of the style element to the style system.

> ***Note: This specification does not specify a style system, but CSS is expected to be supported by most Web browsers. [CSS21]***

The DOM attributes **media** and **type** each must reflect the respective content attributes of the same name.

The DOM **disabled** attribute behaves as defined for the alternate style sheets DOM.

3.8. Sections

Sectioning elements are elements that divide the page into, for lack of a better word, sections. This section describes HTML's sectioning elements and elements that support them.

Some elements are scoped to their nearest ancestor sectioning element. For example, address elements apply just to their section. For such elements *x*, the elements that apply to a sectioning element *e* are all the *x* elements whose nearest sectioning element is *e*.

3.8.1. The body element

Sectioning element.

Contexts in which this element may be used:

As the second element in an html element.

Content model:

Zero or more block-level elements.

Element-specific attributes:

None.

Predefined classes that apply to this element:

`search`, `warning`

DOM interface:

No difference from `HTMLElement`.

The `body` element represents the main content of the document.

The `body` element potentially has a heading. See the section on headings and sections for further details.

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.

3.8.2. The `section` element

Sectioning block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more block-level elements.

Element-specific attributes:

None.

Predefined classes that apply to this element:

`error`, `example`, `search`, `warning`

DOM interface:

No difference from `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.

Each `section` element potentially has a heading. See the section on headings and sections for further details.

3.8.3. The `nav` element

Sectioning block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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.

When used as an inline-level content container, the element represents a paragraph.

Each `nav` element potentially has a heading. See the section on headings and sections for further details.

3.8.4. The `article` element

Sectioning block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more block-level elements.

Element-specific attributes:

None.

Predefined classes that apply to this element:

`warning`

DOM interface:

No difference from `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.

Each `article` element potentially has a heading. See the section on headings and sections for further details.

3.8.5. The `blockquote` element

Sectioning block-level element, and structured inline-level element.

Contexts in which this element may be used:

Where block-level elements are expected.
Where structured inline-level elements are allowed.

Content model:

Zero or more block-level elements.

Element-specific attributes:

`cite`

Predefined classes that apply to this element:

None.

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 URI, if it has one, should be cited in the **`cite`** attribute.

If the `cite` attribute is present, it must be a URI (or IRI). User agents should allow users to follow such citation links.

If a `blockquote` element is preceeded or followed by a `p` element that contains a single `cite` element and is itself not preceeded or followed by another `blockquote` element and does not itself have a `q` element descendant, then, the citation given by that `cite` element gives the source of the quotation contained in the `blockquote` element.

Each `blockquote` element potentially has a heading. See the section on headings and sections for further details.

The **`cite`** DOM attribute `reflects` the element's `cite` content attribte.

Note: The best way to represent a conversation is not with the `cite` and `blockquote` elements, but with the `dialog` element.

3.8.6. The `aside` element

Sectioning block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

None.

Predefined classes that apply to this element:

`example`, `issue`, `note`, `search`, `warning`

DOM interface:

No difference from `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.

When used as an inline-level content container, the element represents a paragraph.

Each `aside` element potentially has a heading. See the section on headings and sections for further details.

3.8.7. The `h1`, `h2`, `h3`, `h4`, `h5`, and `h6` elements

Block-level elements.

Contexts in which these elements may be used:

Where block-level elements are expected.

Content model:

Significant strictly inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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.

These elements must not be empty.

3.8.8. The `header` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected and there are no `header` ancestors.

Content model:

Zero or more block-level elements, including at least one descendant `h1`, `h2`, `h3`, `h4`, `h5`, or `h6` element, but no sectioning element descendants, no `header` element descendants, and no `footer` element descendants.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `header` element represents the header of a section. Headers may contain more than just the section's heading — for example it would be reasonable for the header to include version history information.

`header` elements must not contain any `header` elements, `footer` elements, or any sectioning elements (such as `section`) as descendants.

`header` elements must have at least one `h1`, `h2`, `h3`, `h4`, `h5`, or `h6` element as a descendant.

For the purposes of document summaries, outlines, and the like, `header` elements are equivalent to the highest ranked `h1`-`h6` element descendant (the first such element if there are multiple elements with that rank).

Other heading elements indicate subheadings or subtitles.

> 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>
>  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://w
  <dt>Previous version:</dt>
  <dd><a href="http://www.w3.org/TR/2004/WD-SVG12-20040510/">http://w
  <dt>Latest version of SVG 1.2:</dt>
  <dd><a href="http://www.w3.org/TR/SVG12/">http://www.w3.org/TR/SVG1
  <dt>Latest SVG Recommendation:</dt>
  <dd><a href="http://www.w3.org/TR/SVG/">http://www.w3.org/TR/SVG/</
  <dt>Editor:</dt>
  <dd>Dean Jackson, W3C, <a href="mailto:dean@w3.org">dean@w3.org</a>
  <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
</header>
```

The section on headings and sections defines how `header` elements are assigned to individual sections.

The rank of a `header` element is the same as for an `h1` element (the highest rank).

3.8.9. The `footer` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Either zero or more block-level elements, but with no `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `header`, or `footer` elements as descendants, and with no sectioning elements as descendants; or, inline-level content (but not both).

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `footer` element represents the 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.

`footer` elements must not contain any `footer`, `header`, `h1`, `h2`, `h3`, `h4`, `h5`, or `h6` elements, or any of the sectioning elements (such as `section`), as descendants.

When used as an inline-level content container, the element represents a paragraph.

Contact information for the section given in a `footer` should be marked up using the `address` element.

3.8.10. The `address` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `address` element represents a paragraph of contact information for the section it applies to.

> 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 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 element and its ancestor sectioning elements. The contact information is the collection of all the information given by those elements.

Note: Contact information for one sectioning element, e.g. a `aside` element, does not apply to its ancestor elements, e.g. the page's `body`.

3.8.11. Headings and sections

The `h1`-`h6` elements and the `header` element are headings.

The first heading in a sectioning element gives the header for that section. Subsequent headers of equal or higher rank start new (implied) sections, headers of lower rank start subsections that are

part of the previous one.

Sectioning elements other than `blockquote` are always considered subsections of their nearest ancestor sectioning element, regardless of what implied sections other headings may have created. However, `blockquote` elements *are* associated with implied sections. Effectively, `blockquote` elements act like sections on the inside, and act opaquely on the outside.

> 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)
>   1. Bar (heading starting implied section)
>   1. Bla (heading of explicit `blockquote` section)
>
>   Baz (paragraph)
>   2. Quux (heading starting implied section)
>   3. Thud (heading of explicit `section` section)
>
>   Grunt (paragraph)
>
> Notice how the `blockquote` nests inside an implicit section while the `section` does not (and in fact, ends the earlier implicit section so that a later paragraph 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 explictly wrap sections in sectioning elements, instead of relying on the implicit sections generated by having multiple heading in one sectioning element.

> For example, the following is correct:
>
> ```
> <body>
>  <h4>Apples</h4>
>  Apples are fruit.
>  <section>
>   <h2>Taste</h2>
>   They taste lovely.
>   <h6>Sweet</h6>
>   Red apples are sweeter than green ones.
>   <h1>Color</h1>
>   Apples come in various colors.
>  </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.

3.8.11.1. Creating an outline

Documents can be viewed as a tree of sections, which defines how each element in the tree is semantically related to the others, in terms of the overall section structure. This tree is related to the document tree, but there is not a one-to-one relationship between elements in the DOM and the document's sections.

The tree of sections should be used when generating document outlines, for example when generating tables of contents.

To derive the tree of sections from the document tree, a hypothetical tree is used, consisting of a view of the document tree containing only the `h1`-`h6` and `header` elements, and the sectioning elements other than `blockquote`. Descendants of `h1`-`h6`, `header`, and `blockquote` elements must be removed from this view.

The hypothetical tree must be rooted at the root element or at a sectioning element. In particular, while the sections inside `blockquote`s do not contribute to the document's tree of sections, `blockquote`s can have outlines of their own.

UAs must take this hypothetical tree (which will become the outline) and mutate it by walking it depth first in tree order and, for each `h1`-`h6` or `header` element that is not the first element of its parent sectioning element, inserting a new sectioning element, as follows:

↪ **If the element is a `header` element, or if it is an `h1`-`h6` node of rank equal to or higher than the first element in the parent sectioning element (assuming that is also an `h1`-`h6` node), or if the first element of the parent sectioning element is a sectioning element:**

Insert the new sectioning element as the immediately following sibling of the parent sectioning element, and move all the elements from the current heading element up to the end of the parent sectioning element into the new sectioning element.

↪ **Otherwise:**

Move the current heading element, and all subsequent siblings up to but excluding the next sectioning element, `header` element, or `h1`-`h6` of equal or higher rank, whichever comes first, into the new sectioning element, then insert the new sectioning element where the current header was.

The outline is then the resulting hypothetical tree. The ranks of the headers become irrelevant at this point: each sectioning element in the hypothetical tree contains either no or one heading element child. If there is one, then it gives the section's heading, of there isn't, the section has no heading.

Sections are nested as in the hypothetical tree. If a sectioning element is a child of another, that means it is a subsection of that other section.

When creating an interactive table of contents, entries should jump the user to the relevant section element, if it was a real element in the original document, or to the heading, if the section element was one of those created during the above process.

> 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 hypothetical tree (before mutations) could be generated by creating a `TreeWalker` with the following `NodeFilter` (described here as an anonymous ECMAScript function). [DOMTR] [ECMA262]

```
function (n) {
  // This implementation only knows about HTML elements.
  // An implementation that supports other languages might be
  // different.

  // Reject anything that isn't an element.
  if (n.nodeType != Node.ELEMENT_NODE)
    return NodeFilter.FILTER_REJECT;

  // Skip any descendants of headings.
  if (n.parentNode && n.parentNode.namespaceURI == 'http://www.w3.org
      (n.parentNode.localName == 'h1' || n.parentNode.localName == 'h
       n.parentNode.localName == 'h3' || n.parentNode.localName == 'h
       n.parentNode.localName == 'h5' || n.parentNode.localName == 'h
       n.parentNode.localName == 'header')
    return NodeFilter.FILTER_REJECT;

  // Skip any blockquotes.
  if (n.namespaceURI == 'http://www.w3.org/1999/xhtml') &&
      (n.localName == 'blockquote'))
    return NodeFilter.FILTER_REJECT;

  // Accept HTML elements in the list given in the prose above.
  if ((n.namespaceURI == 'http://www.w3.org/1999/xhtml') &&
      (n.localName == 'body' || /*n.localName == 'blockquote' ||*/
       n.localName == 'section' || n.localName == 'nav' ||
       n.localName == 'article' || n.localName == 'aside' ||
       n.localName == 'h1' || n.localName == 'h2' ||
       n.localName == 'h3' || n.localName == 'h4' ||
       n.localName == 'h5' || n.localName == 'h6' ||
       n.localName == 'header'))
```

```
      return NodeFilter.FILTER_ACCEPT;

    // Skip the rest.
    return NodeFilter.FILTER_SKIP;
  }
```

3.8.11.2. Determining which heading and section applies to a particular node

Given a particular node, user agents must use the following algorithm, *in the given order*, to determine which heading and section the node is most closely associated with. The processing of this algorithm must stop as soon as the associated section and heading are established (even if they are established to be nothing).

1. If the node has an ancestor that is a `header` element, then the associated heading is the most distant such ancestor. The associated section is that `header`'s associated section (i.e. repeat this algorithm for that `header`).

2. If the node has an ancestor that is an `h1`-`h6` element, then the associated heading is the most distant such ancestor. The associated section is that heading's section (i.e. repeat this algorithm for that heading element).

3. If the node is an `h1`-`h6` element or a `header` element, then the associated heading is the element itself. The UA must then generate the hypothetical section tree described in the previous section, rooted at the nearest section ancestor (or the root element if there is no such ancestor). If the parent of the heading in that hypothetical tree is an element in the real document tree, then that element is the associated section. Otherwise, there is no associated section element.

4. If the node is a sectioning element, then the associated section is itself. The UA must then generate the hypothetical section tree described in the previous section, rooted at the section itself. If the section element, in that hypothetical tree, has a child element that is an `h1`-`h6` element or a `header` element, then that element is the associated heading. Otherwise, there is no associated heading element.

5. If the node is a `footer` or `address` element, then the associated section is the nearest ancestor sectioning element, if there is one. The node's associated heading is the same as that sectioning element's associated heading (i.e. repeat this algorithm for that sectioning element). If there is no ancestor sectioning element, the element has no associated section nor an associated heading.

6. Otherwise, the node is just a normal node, and the document has to be examined more closely to determine its section and heading. Create a view rooted at the nearest ancestor sectioning element (or the root element if there is none) that has just `h1`-`h6` elements, `header` elements, the node itself, and sectioning elements other than `blockquote` elements. (Descendants of any of the nodes in this view can be ignored, as can any node later in the tree than the node in question, as the algorithm below merely walks backwards up this view.)

7. Let *n* be an iterator for this view, initialised at the node in question.

8. Let *c* be the current best candidate heading, initially null, and initially not used. It is used when top-level heading candidates are to be searched for (see below).

9. Repeat these steps (which effectively goes backwards through the node's previous siblings)

until an answer is found:

1. If *n* points to a node with no previous sibling, and *c* is null, then return the node's parent node as the answer. If the node has no parent node, return null as the answer.
2. Otherwise, if *n* points to a node with no previous sibling, return *c* as the answer.
3. Adjust *n* so that it points to the previous sibling of the current position.
4. If *n* is pointing at an `h1` or `header` element, then return that element as the answer.
5. If *n* is pointing at an `h2`-`h6` element, and heading candidates are not being searched for, then return that element as the answer.
6. Otherwise, if *n* is pointing at an `h2`-`h6` element, and either *c* is still null, or *c* is a heading of lower rank than this one, then set *c* to be this element, and continue going backwards through the previous siblings.
7. If *n* is pointing at a sectioning element, then from this point on top-level heading candidates are being searched for. (Specifically, we are looking for the nearest top-level header for the current section.) Continue going backwards through the previous siblings.

10. If the answer from the previous step (the loop) is null, which can only happen if the node has no preceeding headings and is not contained in a sectioning element, then there is no associated heading and no associated section.

11. Otherwise, if the answer from the earlier loop step is a sectioning element, then the associated section is that element and the associated heading is that sectioning element's associated heading (i.e. repeat this algorithm for that section).

12. Otherwise, if the answer from that same earlier step is an `h1`-`h6` element or a `header` element, then the associated heading is that element and the associated section is that heading element's associated section (i.e. repeat this algorithm for that heading).

Note: Not all nodes have an associated header or section. For example, if a section is implied, as when multiple headers are found in one sectioning element, then a node in that section has an anonymous associated section (its section is not represented by a real element), and the algorithm above does not associate that node with any particular sectioning element.

For the following fragment:

```
<body>
 <h1>X</h1>
 <h2>X</h2>
 <blockquote>
  <h3>X</h3>
 </blockquote>
 <p id="a">X</p>
 <h4>Text Node A</h4>
 <section>
  <h5>X</h5>
 </section>
 <p>Text Node B</p>
</body>
```

The associations are as follows (not all associations are shown):

Node	Associated heading	Associated section

Node	Associated heading	Associated section
<body>	<h1>	<body>
<h1>	<h1>	<body>
<h2>	<h2>	None.
<blockquote>	<h2>	None.
<h3>	<h3>	<blockquote>
<p id="a">	<h2>	None.
Text Node A	<h4>	None.
Text Node B	<h1>	<body>

3.9. Prose

3.9.1. The p element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Significant inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

copyright, error, example, issue, note, search, warning

DOM interface:

No difference from HTMLElement.

The p element represents a paragraph.

p elements can contain a mixture of strictly inline-level content, such as text, images, hyperlinks, etc, and structured inline-level elements, such as lists, tables, and block quotes. p elements must not be empty.

The following examples are conforming HTML fragments:

```
<p>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.</p>

<fieldset>
 <legend>Personal information</legend>
 <p>
   <label>Name: <input name="n"></label>
   <label><input name="anon" type="checkbox"> Hide from other users</
 </p>
 <p><label>Address: <textarea name="a"></textarea></label></p>
</fieldset>
```

```
<p>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.</p>
```

The p element should not be used when a more specific element is more appropriate.

The following example is technically correct:

```
<section>
 <!-- ... -->
 <p>Last modified: 2001-04-23</p>
 <p>Author: fred@example.com</p>
</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>
  <p>Last modified: 2001-04-23</p>
  <address>Author: fred@example.com</address>
 </footer>
</section>
```

3.9.2. The hr element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Empty.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from 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.

3.9.3. The `br` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Empty.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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 only be used 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. Sherman<br>
> 42 Wallaby Way<br>
> Sydney
> ```

`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:
>
> ```
> <a ...>34 comments.</a><br>
> <a ...>Add a comment.<a>
>
> Name: <input name="name"><br>
> Address: <input name="address">
> ```
>
> Here are alternatives to the above, which are correct:
>
> ```
> <a ...>34 comments.</a>
> <a ...>Add a comment.<a>
>
> Name: <input name="name">
> Address: <input name="address">
> ```

3.9.4. The `dialog` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more pairs of `dt` and `dd` elements.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `dialog` element represents a conversation.

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 mo
 <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 person talking were themselves quoting someone else. See the `cite`, `q`, and `blockquote` elements for other ways to cite or quote.

3.10. Preformatted text

3.10.1. The `pre` element

Block-level element, and structured inline-level element.

Contexts in which this element may be used:

Where block-level elements are expected.
Where structured inline-level elements are allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `pre` element represents a block of preformatted text, in which structure is represented by typographic conventions rather than by elements.

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.

If, ignoring text nodes consisting only of whitespace, the only child of a `pre` is a `code` element, then the `pre` element represents a block of computer code.

If, ignoring text nodes consisting only of whitespace, the only child of a `pre` is a `samp` element, then the `pre` element represents a block of computer output.

3.11. Lists

3.11.1. The `ol` element

Block-level element, and structured inline-level element.

Contexts in which this element may be used:

Where block-level elements are expected.
Where structured inline-level elements are allowed.

Content model:

Zero or more `li` elements.

Element-specific attributes:

`start`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLOListElement : HTMLElement {
           attribute long start;
};
```

The `ol` element represents an ordered list of items (which are represented by `li` elements).

The **`start`** attribute, if present, must have a value that consists of an optional U+002D HYPHEN-MINUS followed by one or more digits (U+0030 to U+0039) expressing a base ten 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.

The items of the list are the `li` element child nodes of the `ol` element, in tree order.

The first item in the list has the ordinal value given by the `ol` element's `start` attribute (unless it is further overridden by that `li` element's `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.

The **`start`** DOM attribute must reflect the value of the `start` content attribute.

3.11.2. The `ul` element

Block-level element, and structured inline-level element.

Contexts in which this element may be used:

Where block-level elements are expected.
Where structured inline-level elements are allowed.

Content model:

Zero or more `li` elements.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `ul` element represents an unordered list of items (which are represented by `li` elements).

The items of the list are the `li` element child nodes of the `ul` element.

3.11.3. The `li` element

Contexts in which this element may be used:

Inside `ol` elements.
Inside `ul` elements.
Inside `menu` elements.

Content model:

When the element is a child of an `ol` or `ul` element and the grandchild of an element that is being used as an inline-level content container, or, when the element is a child of a `menu` element: inline-level content.
Otherwise: zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

If the element is a child of an `ol` element: `value`
If the element is not the child of an `ol` element: None.

Predefined classes that apply to this element:

None.

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.

When the list item is the child of an `ol` or `ul` element, the content model of the item depends on the way that parent element was used. If it was used as structured inline content (i.e. if *that* element's parent was used as an inline-level content container), then the `li` element must only contain inline-level content. Otherwise, the element may be used either for inline content or block-level elements.

When the list item is the child of a `menu` element, the `li` element must contain only inline-level content.

When the list item is not the child of an `ol`, `ul`, or `menu` element, e.g. because it is an orphaned node not in the document, it may contain either for inline content or block-level elements.

When used as an inline-level content container, the list item represents a single paragraph.

The **`value`** attribute, if present, must have a value that consists of an optional U+002D HYPHEN-MINUS followed by one or more digits (U+0030 to U+0039) expressing a base ten integer giving the ordinal value of the first 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, it is treated as if the attribute was absent. The attribute has no default value.

The `value` attribute is processed relative to the element's parent `ol` element, 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.

3.11.4. The `dl` element

Block-level element, and structured inline-level element.

Contexts in which this element may be used:

Where block-level elements are expected.
Where structured inline-level elements are allowed.

Content model:

Zero or more groups each consisting of one or more `dt` elements followed by one or mode `dd` elements.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `dl` element introduces an unordered 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 following are all conforming HTML fragments.
>
> 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>
> ```

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, and the document is non-conforming.

If a dl element contains only dt elements, then it consists of one group with names but no values, and the document is non-conforming.

If a dl element contains only dd elements, then it consists of one group with values but no names, and the document is non-conforming.

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.

3.11.5. The dt element

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:

Strictly inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from 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.

3.11.6. The dd element

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:

When the element is a child of a dl element and the grandchild of an element that is being used as an inline-level content container: inline-level content.
Otherwise: zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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).

The content model of a `dd` element depends on the way its parent element is being used. If the parent element is a `dl` element that is being used as structured inline content (i.e. if the `dl` element's parent element is being used as an inline-level content container), then the `dd` element must only contain inline-level content.

Otherwise, the element may be used either for inline content or block-level elements.

3.12. Phrase elements

3.12.1. The `a` element

Interactive, strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed, if there are no ancestor interactive elements.

Content model:

When used in an element whose content model is only strictly inline-level content: only significant strictly inline-level content, but there must be no interactive descendants. Otherwise: any significant inline-level content, but there must be no interactive descendants.

Element-specific attributes:

`href`
`rel`
`media`
`hreflang`
`type`
`ping`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLAnchorElement : HTMLElement {
           attribute DOMString href;
           attribute DOMString rel;
           attribute DOMString media;
           attribute DOMString hreflang;
           attribute DOMString type;
           attribute DOMString ping;
};
```

The `Command` interface must also be implemented by this element.

If the a element has an href attribute, then it represents a hyperlink.

If the a element has no href attribute, then the element is a placeholder for where a link might otherwise have been placed, if it had been relevant.

> If a site uses a consistent navigation toolbar 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 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.

default action of clicking a link is to check if target was an image with ismap; if so, do server-side image map processing, otherwise, fire DOMActivate on the link.

The a element must not be empty.

The DOM attributes **href**, **rel**, **media**, **hreflang**, **type**, and **ping** each must reflect the respective content attributes of the same name.

3.12.2. The q element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

When used in an element whose content model is only strictly inline-level content: only strictly inline-level content.
Otherwise: any inline-level content.

Element-specific attributes:

cite

Predefined classes that apply to this element:

None.

DOM interface:

The q element uses the HTMLQuoteElement interface.

The q element represents a part of a paragraph quoted from another source.

Content inside a q element must be quoted from another source, whose URI, if it has one, should be cited in the **cite** attribute.

If the `cite` attribute is present, it must be a URI (or IRI). User agents should allow users to follow such citation links.

If a `q` element is contained (directly or indirectly) in a paragraph that contains a single `cite` element and has no other `q` element descendants, then, the citation given by that `cite` element gives the source of the quotation contained in the `q` element.

3.12.3. The `cite` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `cite` element represents a citation: the source, or reference, for a quote or statement made in the document.

Note: A citation is not a quote (for which the `q` element is appropriate).

This is incorrect usage:

```
<p><cite>This is wrong!</cite>, said Ian.</p>
```

This is the correct way to do it:

```
<p><q>This is correct!</q>, said <cite>Ian</cite>.</p>
```

This is also wrong, because the title and the name are not references or citations:

```
<p>My favourite book is <cite>The Reality Dysfunction</cite>
by <cite>Peter F. Hamilton</cite>.</p>
```

This is correct, because even though the source is not quoted, it is cited:

```
<p>According to <cite>the Wikipedia article on
HTML</cite>, HTML is defined in formal specifications that were
developed and published throughout the 1990s.</p>
```

Note: The `cite` element can apply to `blockquote` and `q` elements in certain cases described in the definitions of those elements.

3.12.4. The `em` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

When used in an element whose content model is only strictly inline-level content: only strictly inline-level content.
Otherwise: any inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `em` element represents stress emphasis of its contents.

The level of emphasis that a particlar 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 emphasising 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 emphasise the last word:

```
<p>Cats are cute <em>animals</em>.</p>
```

By emphasising 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 emphasising the cuteness could lead to markup such as:
>
> ```
> Cats are cute animals!
> ```

3.12.5. The `strong` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

When used in an element whose content model is only strictly inline-level content: only strictly inline-level content.
Otherwise: any inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

`error`, `warning`

DOM interface:

No difference from `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:
>
> ```
> Warning. This dungeon is dangerous.
> Avoid the ducks. Take any gold you find.
> Do not take any of the diamonds,
> they are explosive and will destroy anything within
> ten meters. You have been warned.
> ```

3.12.6. The `small` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

When used in an element whose content model is only strictly inline-level content: only strictly inline-level content.
Otherwise: any inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `small` element represents small print (part of a document often describing legal restrictions, such as copyrights or other disadvantages), or other side comments.

Note: The `small` element does not "de-emphasise" 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.

```
<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>
```

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
```

3.12.7. The `m` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

When used in an element whose content model is only strictly inline-level content: only strictly inline-level content.
Otherwise: any inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `m` element represents a run of text marked or highlighted.

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 := <m>1.1</m>;
end.</code></pre>
```

Another example of the `m` 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 <m>kitten</m>s who are visiting me
these days. They're really cute. I think they like my garden!</p>
```

3.12.8. The `dfn` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed, if there are no ancestor `dfn` elements.

Content model:

Strictly inline-level content, but there must be no descendant `dfn` elements.

Element-specific attributes:

None, but the `title` attribute has special semantics on this element.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `dfn` element represents the defining instance of a term. The paragraph, description list group, or section that contains the `dfn` element contains the definition for the term given by the contents of the `dfn` element.

`dfn` elements must not be nested.

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 only contain the term being defined.

There must only be one `dfn` element per document for each term defined (i.e. there must not be any duplicate terms).

Note: The `title` attribute of ancestor elements does not affect `dfn` elements.

The dfn element enables automatic cross-references. Specifically, any span, abbr, code, var, samp, or i element that has a non-empty title attribute whose value exactly equals the term of a dfn element in the same document, or which has no title attribute but whose textContent exactly equals the term of a dfn element in the document, and that has no interactive elements or dfn elements either as ancestors or descendants, and has no other elements as ancestors that are themselves matching these conditions, should be presented in such a way that the user can jump from the element to the first dfn element giving the defining instance of that term.

> In the following fragment, the term "GDO" is first defined in the first paragraph, then used in the second. A compliant UA could provide a link from the abbr element in the second paragraph to the dfn element in the first.
>
> ```
> The <dfn><abbr title="Garage Door Opener">GDO</abbr></dfn>
> is a device that allows off-world teams to open the iris.
> <!-- ... later in the document: -->
> Teal'c activated his <abbr title="Garage Door Opener">GDO</abbr>
> and so Hammond ordered the iris to be opened.
> ```

3.12.9. The `abbr` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None, but the title attribute has special semantics on this element.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from HTMLElement.

The abbr element represents an abbreviation or acronym. The **title** attribute should be used to provide an expansion of the abbreviation. If present, the attribute must only contain an expansion of the abbreviation.

> The paragraph below contains an abbreviation marked up with the abbr element.
>
> ```
> The <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.
> ```

The title attribute may be omitted if there is a dfn element in the document whose defining term is the abbreviation (the textContent of the abbr element).

> In the example below, the word "Zat" is used as an abbreviation in the second paragraph. The abbreviation is defined in the first, so the explanatory title attribute has been omitted. Because of the way dfn elements are defined, the second abbr element in this example would

> be connected (in some UA-specific way) to the first.
>
> ```
> <p>The <dfn><abbr>Zat</abbr></dfn>, short for Zat'ni'catel, is a weap
> <p>Jack used a <abbr>Zat</abbr> to make the boxes of evidence disappe
> ```

3.12.10. The `time` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

`datetime`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLTimeElement : HTMLElement {
           attribute DOMString datetime;
  readonly attribute DOMTimeStamp date;
  readonly attribute DOMTimeStamp time;
  readonly attribute DOMTimeStamp timezone;
};
```

The `time` element represents a date and/or a time.

The **`datetime`** attribute, if present, must contain a 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 parsing the element's `textContent` according to the rules for parsing date or time strings in content successfully extracts a date or time.

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 the following steps:

1. If the `datetime` attribute is present, then parse it according to the rules for parsing date or time strings in content, and let the result be *result*.
2. Otherwise, parse the element's `textContent` according to the rules for parsing date or time strings in content, 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.)

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:
>
> ```
> We stopped talking at <time datetime="2006-09-24 05:00 -7">5am the
> ```
>
> ...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:
>
> ```
> Many people get up at <time>08:00</time>.
> ```
>
> ...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.

These APIs may be suboptimal. Comments on making them more useful to JS authors are welcome. The primary use cases for these elements are for marking up publication dates e.g. in blog entries, and for marking event dates in hCalendar markup. Thus the DOM APIs are likely to be used as ways to generate interactive calendar widgets or some such.

3.12.11. The `meter` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

`value`
`min`
`low`

high
max
optimum

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLMeterElement : HTMLElement {
           attribute long value;
           attribute long min;
           attribute long max;
           attribute long low;
           attribute long high;
           attribute long 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.

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), or as a percentage or similar (using one of the characters such as "%"), or as a fraction.

The value, min, low, high, max, and optimum attributes are all optional. When present, they must have values that are valid floating point numbers.

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>
```

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 above results in a low boundary that is less than the minimum value, the low boundary is the minimum value.

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 above results in a high boundary that is higher than the maximum value, the high boundary is the maximum value.

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 should 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 gr
  </menu>
  <ul>
   <li>
    <p><a href="/group/comp.infosystems.www.authoring.stylesheets/view"
       <a href="/group/comp.infosystems.www.authoring.stylesheets/subsc
    <p>Group description: <strong>Layout/presentation on the WWW.</stro
    <p><meter value="0.5">Moderate activity,</meter> Usenet, 618 subscr
   </li>
   <li>
    <p><a href="/group/netscape.public.mozilla.xpinstall/view">netscape
       <a href="/group/netscape.public.mozilla.xpinstall/subscribe">joi
    <p>Group description: <strong>Mozilla XPInstall discussion.</strong
    <p><meter value="0.25">Low activity,</meter> Usenet, 22 subscribers
   </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
   </li>
  </ul>
```

Might be rendered as follows:

Suggested groups - Hide suggested groups

comp.infosystems.www.authoring.stylesheets - join
Group description: Layout/presentation on the WWW.
Usenet, 618 subscribers

netscape.public.mozilla.xpinstall - join
Group description: Mozilla XPInstall discussion.
Usenet, 22 subscribers

mozilla.dev.general - join
Usenet, 66 subscribers

The **min**, **max**, **value**, **low**, **high**, and **optimum** DOM attributes must reflect the elements' 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.

Would be cool to have the `value` DOM attribute update the `textContent` in-line...

3.12.12. The `progress` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

`value`

`max`

Predefined classes that apply to this element:

None.

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 simply 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>
>  <label>Progress: <progress>0%</progress>
>  <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.

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 elements' 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.

Would be cool to have the `value` DOM attribute update the `textContent` in-line...

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.

3.12.13. The `code` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

When used in an element whose content model is only strictly inline-level content: only strictly inline-level content.
Otherwise: any inline-level content.

Element-specific attributes:

None, but the `title` attribute has special semantics on this element when used with the `dfn` element.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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 recognise.

Note: See the `pre` element for more detais.

> The following example shows how a block of code could be marked up using the `pre` and `code` elements.
>
> ```
> <pre><code>var i: Integer;
> begin
>    i := 1;
> end.</code></pre>
> ```

3.12.14. The `var` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None, but the `title` attribute has special semantics on this element when used with the `dfn` element.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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!
> ```

3.12.15. The `samp` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

When used in an element whose content model is only strictly inline-level content: only strictly inline-level content.
Otherwise: any inline-level content.

Element-specific attributes:

None, but the `title` attribute has special semantics on this element when used with the `dfn` element.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `samp` element represents (sample) output from a program or computing system.

Note: See the `pre` and `kbd` elements for more detais.

> This example shows the `samp` element being used inline:
>
> ```
> The computer said <samp>Too much cheese in tray
> two</samp> but I didn't know what that meant.
> ```
>
> 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:~$</samp> <kbd>ssh demo.e
> Last login: Tue Apr 12 09:10:17 2005 from mowmow.example.com on pts/1
> Linux demo 2.6.10-grsec+gg3+e+fhs6b+nfs+gr0501+++p3+c4a+gr2b-reslog-v
>
> <samp class="prompt">jdoe@demo:~$</samp> <samp class="cursor">_</samp
> ```

3.12.16. The `kbd` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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:

```
<p>To make George eat an apple, press <kbd><kbd>Shift</kbd>+<kbd>F3</
```

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:

```
<p>To make George eat an apple, select
    <kbd><kbd><samp>File</samp></kbd>|<kbd><samp>Eat Apple...</samp><
</p>
```

3.12.17. The `sup` and `sub` elements

Strictly inline-level content.

Contexts in which these elements may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `sup` element represents a superscript and the `sub` element represents a subscript.

These elements must only be used to mark up typographical conventions with specific meanings, not for typographical presentation for presentation's sake. For example, it would be inappropriate for the `sup` and `sub` elements to be used in the name of the LaTeX document preparation system. In general, authors should not use these elements if the *absence* of those elements would not 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></
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.

```
<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>
```

3.12.18. The `span` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

When used in an element whose content model is only strictly inline-level content: only strictly inline-level content.
Otherwise: any inline-level content.

Element-specific attributes:

None, but the `title` attribute has special semantics on this element when used with the `dfn` element.

Predefined classes that apply to this element:

`copyright`, `error`, `example`, `issue`, `note`, `search`, `warning`

DOM interface:

No difference from `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`, or when used in conjunction with the `dfn` element.

3.12.19. The `i` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None, but the `title` attribute has special semantics on this element when used with the `dfn` element.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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>felis silvestris catus</i> is cute.</p>
<p>The <i>block-level elements</i> are 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>
```

The `i` element should be used as a last resort when no other element is more appropriate. In particular, citations should use the `cite` element, defining instances of terms should use the `dfn` element, stress emphasis should use the `em` element, importance should be denoted with the `strong` element, quotes should be marked up with the `q` element, and small print should use the `small` element.

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 italicised.

3.12.20. The `b` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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:

```
<p>The <b>frobonitor</b> and <b>barbinator</b> components are fried.<
```

The following would be *incorrect* usage:

```
<p><b>WARNING!</b> Do not frob the barbinator!</p>
```

In the previous example, the correct element to use would have been `strong`, not `b`.

In the following example, objects in a text adventure are highlighted as being special by use of the `b` element.

```
<p>You enter a small room. Your <b>sword</b> glows
brighter. A <b>rat</b> scurries past the corner wall.</p>
```

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 `m` element.

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.

3.12.21. The `bdo` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None, but the `dir` global attribute is required on this element.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `bdo` element 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]

3.13. Edits

The `ins` and `del` elements represent edits to the document.

3.13.1. The `ins` element

Block-level element, and strictly inline-level content.

Contexts in which this element may be used:

Where block-level elements is expected.
Where strictly inline-level content is allowed.

Content model:

When the element is a child of an element with only one content model (i.e. an element that only allows strictly inline-level content, or only allows inline-level content, or only allows block-level elements): same content model as the parent element.
Otherwise, when the element is a child of an element that only contains inter-element whitespace, `ins` elements, and `del` elements: same content model as the parent element, with the additional restriction that if the parent element allows a choice in content models (e.g. block or inline) then if all the children of all the sibling `ins` elements were placed directly in the parent element, the document would still be conforming.
Otherwise, when the element is a child of an element that is being used as an inline-level content container: inline-level content.
Otherwise, when the element is a child of an element that is being used as a block-level element container: block-level elements.

Otherwise: zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

`cite`
`datetime`

Predefined classes that apply to this element:

None.

DOM interface:

Uses the `HTMLModElement` interface.

The `ins` element represents an addition to the document.

The `ins` element must be used only where block-level elements or strictly inline-level content can be used.

An `ins` element must only contain content that would still be conformant if all `ins` elements were replaced by their contents.

The following would be syntactically legal:

```
<aside>
 <ins>
  <p>...</p>
 </ins>
</aside>
```

As would this:

```
<aside>
 <ins>
  <em>...</em>
 </ins>
</aside>
```

However, this last example would be illegal, as `em` and `p` cannot both be used inside an `aside` element at the same time:

```
<aside>
 <ins>
  <p>...</p>
 </ins>
 <ins>
  <em>...</em>
 </ins>
</aside>
```

3.13.2. The `del` element

Block-level element, and strictly inline-level content.

Contexts in which this element may be used:

Where block-level elements is expected.
Where strictly inline-level content is allowed.

Content model:

When the element has a parent: same content model as the parent element.
Otherwise: zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

`cite`
`datetime`

Predefined classes that apply to this element:

None.

DOM interface:

Uses the `HTMLModElement` interface.

The `del` element represents a removal from the document.

The `del` element must only contain content that would be allowed inside the parent element (regardless of what the parent element actually contains).

> The following would be syntactically legal:
>
> ```
> <aside>
>  <del>
>   ...
>  </del>
>  <ins>
>   ...
>  </ins>
> </aside>
> ```
>
> ...even though the `p` and `em` elements would never be allowed side by side in the `aside` element. This is allowed because the `del` element represents content that was removed, and it is quite possible that an edit could cause an element to go from being an inline-level container to a block-level container, or vice-versa.

3.13.3. Attributes common to `ins` and `del` elements

The **`cite`** attribute may be used to specify a URI 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 URI (or IRI) that explains the change. 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 datetime value.

User agents must parse the `datetime` attribute according to the parse a string as a datetime value 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 datetime). 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** and **datetime** DOM attributes must reflect the elements' content attributes of the same name.

3.14. Embedded content

3.14.1. The img element

Strictly inline-level embedded content.

Contexts in which this element may be used:

As the only embedded content child of a figure element.
Where strictly inline-level content is allowed.

Content model:

Empty.

Element-specific attributes:

alt (required)
src (required)
usemap
ismap (but only if one of the ancestor elements is an a element)
height
width

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLImageElement : HTMLElement {
           attribute DOMString alt;
           attribute DOMString src;
           attribute DOMString useMap;
           attribute boolean isMap;
           attribute long height;
           attribute long width;
  readonly attribute boolean complete;
};
```

Note: An instance of HTMLImageElement can be obtained using the Image constructor.

The img element represents a piece of text with an alternate graphical representation. The text is given by the **alt** attribute, which must be present, and the URI to the graphical representation of that text is given in the **src** attribute, which must also be present.

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.

When the `alt` attribute's value is the empty string, the image supplements the surrounding content. In such cases, the image could be omitted without affecting the meaning of the document.

If the `alt` attribute is omitted, user agents must treat the element as if it had an `alt` attribute set to the empty string.

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.

Guidelines on writing "alt" text here.

The `src` attribute must contain a valid URI (or IRI). If the `src` attribute is omitted, there is no alternative image representation.

When the `src` attribute is set, the user agent must immediately begin to download the specified resource, unless the user agent cannot support images, or its support for images has been disabled.

Once the download has completed, if the image is a valid image, the user agent must fire a `load` event on the `img` element. If the download fails or it completes but the image is not a valid or supported image, the user agent must fire an `error` event on the `img` element.

The remote server's response metadata (e.g. an HTTP 404 status code, or associated Content-Type headers) must be ignored when determining whether the resource obtained is a valid image or not.

Note: This allows servers to return images with error responses.

User agents must not support non-image resources with the `img` element.

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 is a server-side image map. This affects how events are handled on the corresponding `a` element.

The `ismap` attribute, when present, must have the value `ismap`. The attribute must not be specified on an element that does not have an ancestor `a` element.

The **`height`** and **`width`** attributes give the preferred rendered dimensions of the image if the image is to be shown in a visual medium.

Should we require the dimensions to be correct? Should we disallow percentages?

The values of the `height` and `width` attributes must be either valid non-negative integers or valid non-negative percentages.

To parse the attributes, user agents must use the rules for parsing dimension values. This will return either an integer length, a percentage value, or nothing. When one of these attributes has no value, it must be ignored.

The user agent requirements for processing the values obtained from parsing these attributes are described in the rendering section.

The `img` element must be empty.

The DOM attributes **`alt`**, **`src`**, **`useMap`**, and **`isMap`** each must reflect the respective content attributes of the same name.

The DOM attributes **`height`** and **`width`** must return the rendered height and width of the image, in CSS pixels, if the image is being rendered, and is being rendered to a visual medium, or 0 otherwise. [CSS21]

The DOM attribute **`complete`** must return true if the user agent has downloaded the image specified in the `src` attribute, and it is a valid image.

3.14.2. The `iframe` element

Strictly inline-level embedded content.

Contexts in which this element may be used:

As the only embedded content child of a `figure` element.
Where strictly inline-level content is allowed.

Content model:

Text (for details, see prose).

Element-specific attributes:

`src`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLIFrameElement : HTMLElement {
           attribute DOMString src;
};
```

Objects implementing the `HTMLIFrameElement` interface must also implement the `EmbeddingElement` interface defined in the Window Object specification. [WINDOW]

The `iframe` element introduces a new nested browsing context.

The **`src`** attribute, if present, must be a valid URI (or IRI) to a page that the nested browsing context is to contain. If the user navigates away from this page, the `iframe`'s corresponding `Window` object will reference new `Document` objects, but the `src` attribute will not change.

Whenever the `src` attribute is set, the nested browsing context must be navigated to the given URI.

The default value, which must be used if the `src` attribute is not set when the element is created, or if the attribute is ever removed, is `about:blank`.

When content loads in an `iframe`, 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 instead.

order of events when content is also firing its own load/error?

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.

`iframe` elements may contain any text. `iframe` elements must not contain element nodes. 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.)

restrictions for what that text must be?

Note: The HTML parser treats markup inside `iframe` elements as text.

The DOM attribute **`src`** must reflect the content attribute of the same name.

3.14.3. The `embed` element

Strictly inline-level embedded content.

Contexts in which this element may be used:

As the only embedded content child of a `figure` element.
Where strictly inline-level content is allowed.

Content model:

Empty.

Element-specific attributes:

`src` (required)
`type`
`height`
`width`
Any other attribute that has no namespace (see prose).

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLEmbedElement : HTMLElement {
           attribute DOMString src;
           attribute DOMString type;
           attribute long height;
           attribute long width;
};
```

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 must be present and contain a URI (or IRI).

If the `src` attribute is missing, then the `embed` element must be ignored.

When the `src` attribute is set, user agents are expected to find an appropriate handler for the specified resource, based on the content's type, and hand that handler the content of the resource. If the handler supports a scriptable interface, the `HTMLEmbedElement` object representing the element should expose that interfaces.

The user agent should pass the names and values of all the attributes of the `embed` element that have no namespace to the handler used. Any (namespace-less) attribute may be specified on the `embed` element.

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

The `embed` element has no fallback content. If the user agent can't display the specified resource, e.g. because the given type is not supported, then the user agent must use a default handler for the content. (This default could be as simple as saying "Unsupported Format", of course.)

The **`type`** attribute, if present, gives the MIME type of the linked resource. The value must be a valid MIME type, optionally with parameters. [RFC2046]

The **type of the content** being embedded is defined as follows:

1. If the element has a `type` attribute, then the value of the `type` attribute is the content's type.
2. Otherwise, if the specified resource has explicit type metadata, then that is the content's type.
3. Otherwise, the content has no type and there can be no appropriate handler for it.

Should we instead say that the content-sniffing that we're going to define for top-level browsing contexts should apply here?

Should we require the type attribute to match the server information?

We should say that 404s, etc, don't affect whether the resource is used or not. Not sure how to say it here though.

Browsers should take extreme care when interacting with external content intended for third-party renderers. 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.

height/width

The DOM attributes **`src`** and **`type`** each must reflect the respective content attributes of the same name.

The DOM attributes **`height`** and **`width`** must return the rendered height and width of the image, in CSS pixels, if the image is being rendered, and is being rendered to a visual medium, or 0 otherwise. [CSS21]

3.14.4. The `object` element

Strictly inline-level embedded content.

Contexts in which this element may be used:

As the only embedded content child of a `figure` element.
Where strictly inline-level content is allowed.

Content model:

When used as the child of a `figure` element, or, when used as a *`figure` fallback `object`*: Zero or more `param` elements, followed by either zero or more block-level elements or a single `object` element, which is then considered to be a *`figure` fallback `object`*.
Otherwise: Zero or more `param` elements, followed by inline-level content.

Element-specific attributes:

`data` (required if `type` is not given)
`type` (required if `data` is not given)
`usemap`
`height`
`width`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLObjectElement : HTMLElement {
           attribute DOMString data;
           attribute DOMString type;
           attribute DOMString useMap;
           attribute long height;
           attribute long width;
};
```

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.

Shouldn't allow inline-level content to be the content model when the parent's content model is strictly inline only.

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 third-party software package.

The **`data`** attribute, if present, specifies the URI of the resource. If present, the attribute must be a valid URI.

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.

Whenever the `data` attribute changes, or, if the `data` attribute is not present, whenever the `type` attribute changes, the user agent must follow the following steps to determine what the `object` element represents:

1. If the `data` attribute is present, then:

    1. Begin a load for the resource.

    2. 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 step 3 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.

    3. If the load failed (e.g. DNS error), fire an `error` event, then jump to step 3 in the overall set of steps (fallback).

    4. Determine the *resource type*, as follows:

        ↪ **If the resource has associated Content-Type metadata**

        The type is the type specified in the resource's Content-Type metadata.

        ↪ **Otherwise, if the `type` attribute is present**

        The type is the type specified in the `type` attribute.

        ↪ **Otherwise, there is no explicit type information**

        Sniffing. Default.

    5. Handle the content as given by the first of the following cases that matches:

        ↪ **If the resource requires a special handler (e.g. a plugin)**

        The user agent should find an appropriate handler for the specified resource, based on the *resource type* found in the previous step, and pass the content of the resource to that handler. If the handler supports a scriptable interface, the `HTMLObjectElement` object representing the element should expose that interface. The handler is not a nested browsing context. If no appropriate handler can be found, then jump to step 3 in the overall set of steps (fallback).

        The user agent should pass the names and values of all the parameters given by `param` elements that are children of the `object` element to the handler used.

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

        ↪ **If the type of the resource is an XML MIME type**

        ↪ **If the type of the resource is HTML**

↪ **If the type of the resource 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. (The `data` attribute of the `object` element doesn't get updated if the browsing context gets further navigated to other locations.)

↪ **If the resource is a supported image format, and support for images has not been disabled**

The `object` element represents the specified image. The image is not a nested browsing context.

↪ **Otherwise**

The `object` element represents the specified image, but the image cannot be shown. Jump to step 3 below in the overall set of steps (fallback).

6. The element's contents are not part of what the `object` element represents.

7. Once the resource is completely loaded, fire a `load` event.

2. If the `data` attribute is absent but the `type` attribute is present, and if the user agent can find a handler suitable according to the value of the `type` attribute, then that handler should be used. If the handler supports a scriptable interface, the `HTMLObjectElement` object representing the element should expose that interface. The handler is not a nested browsing context. If no suitable handler can be found, jump to the next step (fallback).

3. (Fallback.) The `object` element doesn't represent anything except what the element's contents represent, ignoring any leading `param` element children. This is the element's fallback content.

In the absence of other factors (such as style sheets), user agents must show the user what the `object` element represents. Thus, the contents of `object` elements act as fallback content, to be 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, targetting multiple user agents with different capabilities, with the user agent picking the best one it supports.

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.

height/width

The DOM attributes **`data`**, **`type`**, **`useMap`**, **`height`**, and **`width`** each must reflect the respective content attributes of the same name.

3.14.5. The `param` element

Contexts in which this element may be used:

As a child of an `object` element, before any content other than `param` elements.

Content model:

Empty.

Element-specific attributes:

`name` (required)
`value` (required)

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLParamElement : HTMLElement {
           attribute DOMString name;
           attribute DOMString value;
};
```

The `param` element defines parameters for handlers invoked by `object` elements.

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.

3.14.6. The `canvas` element

Strictly inline-level embedded content.

Contexts in which this element may be used:

As the only embedded content child of a `figure` element.
Where strictly inline-level content is allowed.

Content model:

Inline-level content.

Element-specific attributes:

`height`
`width`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLCanvasElement : HTMLElement {
         attribute long width;
         attribute long height;

  DOMString toDataURL();
  DOMString toDataURL(in DOMString type);
```

```
  DOMObject getContext(in DOMString contextID);
};
```

Shouldn't allow inline-level content to be the content model when the parent's content model is strictly inline only.

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 should 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 with scripting enabled, the canvas element is 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 must be treated as embedded content with the current image and size. Otherwise, the element's fallback content must be used instead.

In non-visual media, and in visual media with scripting disabled, the `canvas` element's fallback content must be used instead.

The `canvas` element has two attributes to control the size of the coordinate space: **`height`** and **`width`**. 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.

The canvas must initially be fully transparent black.

If the `width` and `height` attributes are dynamically modified, the bitmap and any associated contexts must be cleared back to their initial state and reinitialised with the newly specified coordinate space dimensions.

The **`width`** and **`height`** DOM attributes must reflect the content attributes of the same name.

To draw on the canvas, authors must first obtain a reference to a **context** using the **`getContext`** method of the `canvas` element.

This specification only defines one context, with the name "`2d`". If `getContext()` is called with that exact string, 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 should be literal and case-sensitive.

> ***Note: A future version of this specification will probably define a `3d` context (probably based on the OpenGL ES API).***

The **`toDataURL()`** method must, when called with no arguments, return a `data:` URI containing a representation of the image as a PNG file. [PNG].

The **`toDataURL(type)`** method (when called with one *or more* arguments) must return a `data:` URI 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.

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.***

Arguments other than the *type* must be ignored, and must not cause the user agent to raise an exception (as would normally occur if a method was called with the wrong number of arguments). A future version of this specification will probably allow extra parameters to be passed to `toDataURL()` to allow authors to more carefully control compression settings, image metadata, etc.

Security: To prevent *information leakage*, the `toDataURL()` and `getImageData()` methods should raise a security exception if the canvas ever had images painted on it that originate from a domain other than the domain of the script that painted the images onto the canvas.

3.14.6.1. 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,
  void setTransform(in float m11, in float m12, in float m21, in float m

  // compositing
           attribute float globalAlpha; // (default 1.0)
           attribute DOMString globalCompositeOperation; // (default ove

  // colors and styles
           attribute DOMObject strokeStyle; // (default black)
           attribute DOMObject fillStyle; // (default black)
  CanvasGradient createLinearGradient(in float x0, in float y0, in float
  CanvasGradient createRadialGradient(in float x0, in float y0, in float
  CanvasPattern createPattern(in HTMLImageElement image, DOMString repet
  CanvasPattern createPattern(in HTMLCanvasElement image, DOMString repe

  // line caps/joins
           attribute float lineWidth; // (default 1)
           attribute DOMString lineCap; // "butt", "round", "square" (de
           attribute DOMString lineJoin; // "round", "bevel", "miter" (d
           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 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
  void bezierCurveTo(in float cp1x, in float cp1y, in float cp2x, in flo
  void arcTo(in float x1, in float y1, in float x2, in float y2, in floa
  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,
  void fill();
  void stroke();
  void clip();
  boolean isPointInPath(in float x, in float y);

  // drawing images
  void drawImage(in HTMLImageElement image, in float dx, in float dy);
  void drawImage(in HTMLImageElement image, in float dx, in float dy, ir
  void drawImage(in HTMLImageElement image, in float sx, in float sy, ir
  void drawImage(in HTMLCanvasElement image, in float dx, in float dy);
  void drawImage(in HTMLCanvasElement image, in float dx, in float dy, i
  void drawImage(in HTMLCanvasElement image, in float sx, in float sy, i

  // pixel manipulation
  ImageData getImageData(in float sx, in float sy, in float sw, in float
  void putImageData(in ImageData image, in float dx, in float dy);

  // drawing text is not supported in this version of the API
  // (there is no way to predict what metrics the fonts will have,
  // which makes fonts very hard to use for painting)

};

interface CanvasGradient {
  // opaque object
  void addColorStop(in float offset, in DOMString color);
};

interface CanvasPattern {
  // opaque object
};

interface ImageData {
  readonly attribute long int width;
  readonly attribute long int height;
  readonly attribute int[] data;
};
```

The **canvas** attribute must return the canvas element that the context paints on.

3.14.6.1.1. The canvas state

Each context maintains a stack of drawing states. **Drawing states** consist of:

- The current transformation matrix.
- The current clip region.
- The current values of the following attributes: strokeStyle, fillStyle, globalAlpha,

`lineWidth`, `lineCap`, `lineJoin`, `miterLimit`, `shadowOffsetX`, `shadowOffsetY`, `shadowBlur`, `shadowColor`, `globalCompositeOperation`.

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.

The **`save()`** method pushes a copy of the current drawing state onto the drawing state stack.

The **`restore()`** method pops the top entry in the drawing state stack, and resets the drawing state it describes. If there is no saved state, the method does nothing.

3.14.6.1.2. Transformations

The transformation matrix is applied to all drawing operations prior to their being rendered. It is also applied when creating the clip region.

When the context is created, the transformation matrix must initially be the identity transform. It may then be adjusted using the three 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.

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:

m11	*m12*	*dx*
m21	*m22*	*dy*
0	0	1

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.

3.14.6.1.3. Compositing

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 is the shape or image being rendered, and the destination image is the current state of the bitmap.

`source-atop`

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`

Display the source image wherever both the source image and destination image are opaque. Display transparency elsewhere.

`source-out`

Display the source image wherever the source image is opaque and the destination image is transparent. Display transparency elsewhere.

`source-over` (default)

Display the source image wherever the source image is opaque. Display the destination image elsewhere.

`destination-atop`

Same as `source-atop` but using the destination image instead of the source image and vice versa.

`destination-in`

Same as `source-in` but using the destination image instead of the source image and vice versa.

`destination-out`

Same as `source-out` but using the destination image instead of the source image and vice versa.

`destination-over`

Same as `source-over` but using the destination image instead of the source image and vice versa.

`darker`

Display the sum of the source image and destination images, with color values approaching 0 as a limit.

`lighter`

Display the sum of the source image and destination image, with color values approaching 1 as a limit.

`copy`

Display the source image instead of the destination image.

`xor`

Exclusive OR of the source and destination images.

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 only recognise values that exactly match the values given above.

On setting, if the user agent does not recognise 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`.

3.14.6.1.4. Colors and styles

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 should 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: if it has alpha equal to 1.0, then the color must be returned as an uppercase 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+0041 to U+0046). If the value has alpha less than 1.0, then the value must instead be returned 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.

Otherwise, if it is not a color but a `CanvasGradient` or `CanvasPattern`, then an object supporting those interfaces must be returned. Such objects are opaque and therefore only useful for assigning to other attributes or for comparison to other gradients or patterns.

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 must be placed along it to define how the colors are distributed along the gradient. Between each such stop, the colors and the alpha component must be interpolated over the RGBA space to find the color to use at that offset. Immediately before the 0 offset and immediately after the 1 offset, transparent black stops are be assumed.

The **addColorStop(*offset*, *color*)** method on the CanvasGradient interface adds a new stop to a gradient. If the *offset* is less than 0 or greater than 1 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 be updated with the new stop information.

The **createLinearGradient(*x0*, *y0*, *x1*, *y1*)** method takes four arguments, representing the start point (*x0*, *y0*) and end point (*x1*, *y1*) of the gradient, in coordinate space units, and must return a linear CanvasGradient initialised with that line.

Linear gradients must be rendered such that at the starting point on the canvas the color at offset 0 is used, that at the ending point the color at offset 1 is used, that all points on a line perpendicular to the line between the start and end points have the color at the point where those two lines cross (interpolation happening as described above), and that any points beyond the start or end points are a transparent black.

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. The method must return a radial CanvasGradient initialised with those two circles.

Radial gradients must be rendered such that a cone is created from the two circles, so that at the circumference of the starting circle the color at offset 0 is used, that at the circumference around the ending circle the color at offset 1 is used, that the circumference of a circle drawn a certain fraction of the way along the line between the two origins with a radius the same fraction of the way between the two radii has the color at that offset (interpolation happening as described above), that the end circle appear to be above the start circle when the end circle is not completely enclosed by the start circle, and that any points not described by the gradient are a transparent black.

If a gradient has no stops defined, then the gradient must be treated as a solid transparent black. Gradients are, naturally, only painted where the stroking or filling effect requires that they be drawn.

Support for actually painting gradients is optional. Instead of painting the gradients, user agents may instead just paint the first stop's color. However, createLinearGradient() and createRadialGradient() must always return objects when passed valid arguments.

Patterns are represented by objects implementing the opaque **CanvasPattern** interface.

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 unrecognised value is given, then the user agent must raise a SYNTAX_ERR exception. User agents must recognise the four values described above exactly (e.g. they must not do case folding). The method must return a CanvasPattern object suitably initialised.

The *image* argument must be an instance of an HTMLImageElement or HTMLCanvasElement. If the *image* is of the wrong type, the implementation must raise a TYPE_MISMATCH_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 be scaled by this process; one CSS pixel of the image must be painted on one coordinate space unit. Of course,

patterns must only actually painted where the stroking or filling effect requires that they be drawn, and are affected by the current transformation matrix.

Support for patterns is optional. If the user agent doesn't support patterns, then `createPattern()` must return null.

3.14.6.1.5. Line styles

The **`lineWidth`** attribute gives the default width of lines, in coordinate space units. On setting, zero and negative 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 shall 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 is a flat edge perpendicular to the direction of the line. The `round` value means that a semi-circle with the diameter equal to the width of the line is then added on to the end of the line. The `square` value means that at the end of each line is 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. 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 that UAs will place where two lines meet. The three valid values are `round`, `bevel`, and `miter`.

On setting, any other value than the literal strings `round`, `bevel` and `miter` must be ignored, leaving the value unchanged.

When the context is created, the `lineJoin` attribute must initially have the value `miter`.

The `round` value means that a filled arc connecting the corners on the outside of the join, with the diameter equal to the line width, and the origin at the point where the inside edges of the lines touch, must be rendered at joins. The `bevel` value means that a filled triangle connecting those two corners with a straight line, the third point of the triangle being the point where the lines touch on the inside of the join, must be rendered at joins. The `miter` value means that a filled four- or five-sided polygon must be placed at the join, with two of the lines being the perpendicular edges of the joining lines, 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 limit.

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 the line width. If the miter limit would be exceeded, then a fifth line must be added to the polygon, connecting the two outside lines, such that the distance from the inside point of the join to the point in the middle of this fifth line is the maximum allowed value for the miter length.

The miter limit ratio can be explicitly set using the **`miterLimit`** attribute. On setting, zero and negative values must be ignored, leaving the value unchanged.

When the context is created, the `miterLimit` attribute must initially have the value `10.0`.

3.14.6.1.6. Shadows

All drawing operations are affected by the four global shadow attributes. Shadows form part of the source image during composition.

The **`shadowColor`** attribute sets the color of the shadow.

When the context is created, the `shadowColor` attribute initially must be fully-transparent black.

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.

When the context is created, the shadow offset attributes initially have the value `0`.

The **`shadowBlur`** attribute specifies the number of coordinate space units that the blurring is to cover. On setting, negative numbers must be ignored, leaving the attribute unmodified.

When the context is created, the `shadowBlur` attribute must initially have the value `0`.

Support for shadows is optional. When they are supported, then, when shadows are drawn, they must be rendered using the specified color, offset, and blur radius.

3.14.6.1.7. 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 and height of the rectangle, respectively.

Shapes are painted without affecting the current path, and are subject to transformations, shadow effects, global alpha, clipping paths, and global composition operators.

Negative values for width and height must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`clearRect()`** method must clear the pixels in the specified rectangle to a fully transparent black, erasing any previous image. If either height or width are zero, this method has no effect.

The **`fillRect()`** method must paint the specified rectangular area using the `fillStyle`. If either height or width are zero, this method has no effect.

The **`strokeRect()`** method must draw a rectangular outline of the specified size using the `strokeStyle`, `lineWidth`, `lineJoin`, and (if appropriate) `miterLimit` attributes. What should happen with zero heights or widths?

3.14.6.1.8. 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.

Initially, the context's path must have zero subpaths.

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*) by a quadratic curve with control point (*cpx*, *cpy*), and must then add the given point (*x*, *y*) to the subpath.

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 bezier curve with control points (*cp1x*, *cp1y*) and (*cp2x*, *cp2y*). Then, it must add the point (*x*, *y*) to the subpath.

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 behaviour depends on the arguments and the last point in the subpath.

Let the point (*x0*, *y0*) be the last point in the subpath. Let *The Arc* be the shortest arc given by circumference of the circle that has one point tangent to the line defined by the points (*x0*, *y0*) and (*x1*, *y1*), another point tangent to the line defined by the points (*x1*, *y1*) and (*x2*, *y2*), and that has radius *radius*. The points at which this circle touches these two lines are called the start and end tangent points respectively.

If the point (*x2*, *y2*) is on the line defined by the points (*x0*, *y0*) and (*x1*, *y1*) then the method must do nothing, as no arc would satisfy the above constraints.

Otherwise, the method must connect the point (*x0*, *y0*) to the start tangent point by a straight line, then connect the start tangent point to the end tangent point by *The Arc*, and finally add the start and end tangent points to the subpath.

Negative or zero values for *radius* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

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 the circle's circumference, measured in radians clockwise from the positive x-axis, are the start and end points respectively. 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.

Negative or zero 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.

Negative values for *w* and *h* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`fill()`** method must fill each subpath of the current path in turn, using `fillStyle`, and using the non-zero winding number rule. Open subpaths must be implicitly closed when being filled (without affecting the actual subpaths).

The **`stroke()`** method must stroke each subpath of the current path in turn, using the `strokeStyle`, `lineWidth`, `lineJoin`, and (if appropriate) `miterLimit` attributes.

Paths, when filled or stroked, must be painted without affecting the current path, and must be subject to transformations, shadow effects, global alpha, clipping paths, and global composition operators.

> ***Note: The transformation is applied to the path when it is drawn, not when the path is constructed. Thus, a single path can be constructed and then drawn according to different transformations without recreating the path.***

The **`clip()`** method must create a new **clipping path** by calculating the intersection of the current clipping path and the area described by the current path (after applying the current transformation), using the non-zero winding number rule. Open subpaths must be implicitly closed when computing the clipping path, without affecting the actual subpaths.

When the context is created, the initial clipping path is 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 within the area of the canvas that is inside the current path; and must return false otherwise.

3.14.6.1.9. Images

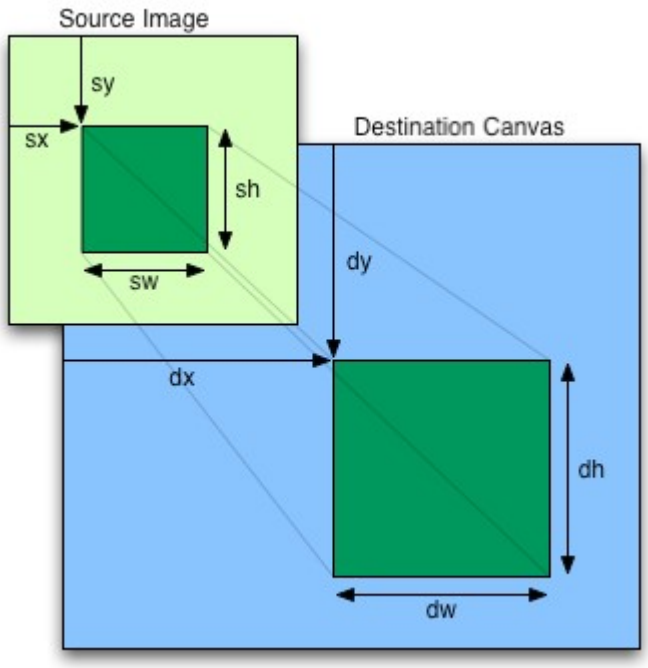
To draw images onto the canvas, the **`drawImage`** method can be used.

This method is overloaded with three variants: `drawImage(image, dx, dy)`, `drawImage(image, dx, dy, dw, dh)`, and `drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)`. (Actually it is overloaded with six; each of those three can take either an `HTMLImageElement` or an `HTMLCanvasElement` for the *image* argument.) If not specified, the *dw* and *dh* arguments 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 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` or `HTMLCanvasElement`. If

the *image* is of the wrong type, the implementation must raise a `TYPE_MISMATCH_ERR` exception. If one of the *sy*, *sw*, *sw*, and *sh* arguments is outside the size of the image, or if one of the *dw* and *dh* arguments is negative, the implementation must raise an `INDEX_SIZE_ERR` exception.

When `drawImage()` is invoked, the specified region of the image specified by the source rectangle (*sx*, *sy*, *sw*, *sh*) must be painted on the region of the canvas specified by the destination rectangle (*dx*, *dy*, *dw*, *dh*).



Images are painted without affecting the current path, and are subject to transformations, shadow effects, global alpha, clipping paths, and global composition operators.

3.14.6.1.10. Pixel manipulation

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 which has one corner at the (*sx*, *sy*) coordinate, and that has width *sw* and height *sh*. Pixels outside the canvas must be returned as transparent black.

`ImageData` objects must be initialised so that their **`height`** attribute is set to *h*, the number of rows in the image data, their **`width`** attribute is set to *w*, the number of physical device pixels per row in the image data, and the **`data`** attribute is initialised to an array of *h*×*w*×4 integers. The pixels must be represented in this array in left-to-right order, row by row, starting at the top left, with each pixel's red, green, blue, and alpha components being given in that order. 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 **`putImageData(`*`image`*`, `*`dx`*`, `*`dy`*`)`** method must take the given `ImageData` structure, and draw it at the specified location *dx*,*dy* in the canvas coordinate space, mapping each pixel represented by the `ImageData` structure into one device pixel.

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* and *y*. In other words, while user agents may round the arguments of the two methods so that they map to device pixel boundaries, any rounding performed must be performed consistently for both the `getImageData()` and `putImageData()` operations.

The current transformation matrix must not affect the `getImageData()` and `putImageData()` methods.

3.14.6.1.11. 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. The coordinates are transformed by the current transformation matrix.
2. The shape or image is rendered, creating image *A*, as described in the previous sections. For shapes, the current fill, stroke, and line styles must be honoured.
3. The shadow is rendered from image *A*, using the current shadow styles, creating image *B*.
4. Image *A* is composited over image *B* creating the source image.
5. The source image has its alpha adjusted by `globalAlpha`.
6. Within the clip region (as affected by the current transformation matrix), the source image is composited over the current canvas bitmap using the current composition operator.

3.14.7. The `map` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more block-level elements.

Element-specific attributes:

`name` (required)

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLMapElement : HTMLElement {
           attribute DOMString name;
  readonly attribute HTMLCollection areas;
};
```

The `map` element, in conjuction with any `area` element descendants, defines an image map.

The **`name`** attribute gives the name of the image map, so that it can be referenced using the `usemap` attribute of `img` and `object` elements.

The `name` attribute must be present. Its value must be a non-empty string. The value of the `name` attribute must be unique, such that no two `map` elements in a common subtree have the same value.

The **`name`** DOM attribute must reflect the content attribute of the same name.

The **`areas`** attribute must return an `HTMLCollection` rooted at the `map` element, whose filter matches only `area` elements.

3.14.8. The `area` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed, but only as a descendant of a `map` element.

Content model:

Empty.

Element-specific attributes:

`alt` (required)
`coords`
`shape`
`href`
`rel`
`media`
`hreflang`
`type`
`ping`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLAreaElement : HTMLElement {
            attribute DOMString alt;
            attribute DOMString coords;
            attribute DOMString shape;
            attribute DOMString href;
            attribute DOMString rel;
            attribute DOMString media;
            attribute DOMString hreflang;
            attribute DOMString type;
            attribute DOMString ping;
};
```

The `area` elements represents some text with corresponding to an area on an image map. The **`alt`** attribute, which must be present, specifies the text, and the `coords` and `shape` attributes specify the area.

If the `area` element has an `href` attribute, then it represents a hyperlink.

If the `area` element has no `href` attribute, then the area represented by the element cannot be

selected.

Interactive user agents should allow users to follow hyperlinks created for valid areas using the `area` element. The `href` 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 DOM attributes **`alt`**, **`coords`**, **`shape`**, **`href`**, **`rel`**, **`media`**, **`hreflang`**, **`type`**, and **`ping`** each must reflect the respective content attributes of the same name.

3.14.9. The `figure` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

In any order, exactly one `legend` element, and exactly one embedded content element.

Element-specific attributes:

None.

Predefined classes that apply to this element:

`example`, `warning`

DOM interface:

No difference from `HTMLElement`.

The `figure` element represents a paragraph consisting of embedded content and a caption.

The first embedded content element child of the `figure` element, if any, is the paragraph's content.

The first `legend` element child of the element, if any, represents the caption of the embedded content. If there is no child `legend` element, then there is no caption.

If the embedded content cannot be used, then, for the purposes of establishing what the `figure` element represents:

↪ **If the embedded content's fallback content is a single embedded content element**

The `figure` element must be treated as if that embedded content element was the `figure` element's embedded content. (If that embedded content can't be used either, then this processing must be done again, with the new embedded content's fallback content.)

↪ **If the embedded content's fallback is nothing**

The entire `figure` element (including the caption, if any) must be ignored.

↪ **If the embedded content's fallback is inline-level content**

The entire `figure` element (including the caption, if any) must be treated as being a single paragraph with that inline-level content as its content.

↪ **Otherwise**

The entire `figure` element (including the caption, if any) must be treated as being replaced by that fallback content.

3.15. Tabular data

3.15.1. The `table` element

Block-level element, and structured inline-level element.

Contexts in which this element may be used:

Where block-level elements are expected.
Where structured inline-level elements are allowed.

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).

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

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;
  readonly attribute HTMLCollection rows;
  HTMLElement insertRow(in long index);
  void deleteRow(in long index);
};
```

The `table` element represents data with more than one dimension (a table).

The children of a `table` element must be, in order:

1. Zero or one `caption` elements.
2. Zero or more `colgroup` elements.
3. Zero or one `thead` elements.
4. Zero or one `tfoot` elements, if the last element in the table is not a `tfoot` element.
5. Either:
   - Zero or more `tbody` elements, or

- One or more `tr` elements.

6. Zero or one `tfoot` element, if there are no other `tfoot` elements in the table.

The `table` element takes part in the table model.

The **`caption`** DOM attribute must return, on getting, the first `caption` element child of the `table` element. On setting, 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.

The **`createCaption()`** method must return a new `caption` element.

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. On setting, 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.

The **`createTHead()`** method must return a new `thead` element.

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. On setting, 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.

The **`createTFoot()`** method must return a new `tfoot` element.

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.

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 behaviour 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 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.

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.

3.15.2. The `caption` element

Contexts in which this element may be used:

As the first element child of a `table` element.

Content model:

Significant strictly inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `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.

3.15.3. The `colgroup` element

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.

Element-specific attributes:

`span`, but only if the element contains no `col` elements

Predefined classes that apply to this element:

None.

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. Its default value, which must be used if parsing the attribute as a non-negative integer returns either an error or zero, is 1.

The `colgroup` element and its `span` attribute take part in the table model.

The **`span`** DOM attribute must reflect the content attribute of the same name, with the exception that on setting, if the new value is 0, then an `INDEX_SIZE_ERR` exception must be raised.

3.15.4. The `col` element

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.

Element-specific attributes:

`span`

Predefined classes that apply to this element:

None.

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. Its default value, which must be used if parsing the attribute as a non-negative integer returns either an error or zero, is 1.

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, with the exception that on setting, if the new value is 0, then an `INDEX_SIZE_ERR` exception must be raised.

3.15.5. The `tbody` element

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:

One or more `tr` elements

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLTableSectionElement : HTMLElement {
  readonly attribute HTMLCollection rows;
  HTMLElement insertRow(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 `thead` element has a parent and it is a `table`.

The `tbody` element takes part in the table model.

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 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.

3.15.6. The `thead` element

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:

One or more `tr` elements

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

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.

3.15.7. The `tfoot` element

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:

One or more `tr` elements

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

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 `thead` element has a parent and it is a `table`.

The `tfoot` element takes part in the table model.

3.15.8. The `tr` element

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:

One or more `td` or `th` elements

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLTableRowElement : HTMLElement {
  readonly attribute long rowIndex;
  readonly attribute long sectionRowIndex;
  readonly attribute HTMLCollection cells;
  HTMLElement insertCell(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.

The **`rowIndex`** element 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 0.

The **`rowIndex`** element 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 0.

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 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.

3.15.9. The `td` element

Contexts in which this element may be used:

As a child of a `tr` element.

Content model:

Zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

`colspan`
`rowspan`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLTableCellElement : HTMLElement {
           attribute long colSpan;
           attribute long rowSpan;
  readonly attribute long cellIndex;
};
```

The `td` element represents a data cell in a table.

The `td` element may have a **`colspan`** content attribute specified, whose value must be a valid non-negative integer greater than zero. Its default value, which must be used if parsing the attribute as a non-negative integer returns either an error or zero, is 1.

The `td` element may also have a **`rowspan`** content attribute specified, whose value must be a valid non-negative integer. Its default value, which must be used if parsing the attribute as a non-negative integer returns an error, is also 1.

The `td` element and its `colspan` and `rowspan` attributes take part in the table model.

The **`colspan`** DOM attribute must reflect the content attribute of the same name, with the exception that on setting, if the new value is 0, then an `INDEX_SIZE_ERR` exception must be raised.

The **`rowspan`** 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.

3.15.10. The `th` element

Contexts in which this element may be used:

As a child of a `tr` element.

Content model:

Zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

`colspan`
`rowspan`
`scope`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLTableHeaderCellElement : HTMLTableCellElement {
           attribute DOMString scope;
};
```

The `th` element represents a header cell in a table.

The `th` element may have a **`colspan`** content attribute specified, whose value must be a valid non-negative integer greater than zero. Its default value, which must be used if parsing the attribute as a non-negative integer returns either an error or zero, is 1.

The `th` element may also have a **`rowspan`** content attribute specified, whose value must be a valid non-negative integer. Its default value, which must be used if parsing the attribute as a non-negative integer returns an error, is also 1.

The `th` element may have a **`scope`** content attribute specified, whose value must be one of the following literal strings: `row`, `cell`, `rowgroup`, or `cellgroup`.

The `th` element and its `colspan`, `rowspan`, and `scope` attributes take part in the table model.

The **`scope`** DOM attribute must reflect the content attribute of the same name.

The `HTMLTableHeaderCellElement` interface inherits from the `HTMLTableCellElement` interface and therefore also has the DOM attributes defined above in the `td` section.

3.15.11. 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 $1 \le x \le x_{max}$, and the y coordinates are always in the range $1 \le y \le y_{max}$. If one or both of x_{max} and y_{max} 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 $cell_y \le y < cell_y + height$. Cell can either be *data cells* or *header cells*. Data cells correspond to `td` elements, and have zero or more associated header cells. Header cells correspond to `th` elements.

A **row** is a complete set of slots from $x=1$ to $x=x_{max}$, for a particular value of y. Rows correspond to `tr` elements.

A **column** is a complete set of slots from $y=1$ to $y=y_{max}$, for a particular value of x. Columns can correspond to `col` elements, but in the absense of `col` elements are implied.

A **row group** is a set of rows anchored at a slot (1, $group_y$) with a particular *height* such that the row group covers all the slots with coordinates (x, y) where $1 \le x < x_{max}$ 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 ($group_x$, 1) 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

$1 \leq y < y_{max}$. 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 covers slots that are from two or more row groups or two or more column groups — all the slots that form part of one cell are part of either zero or one row groups and either zero or one 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.

3.15.11.1. 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_{max} and y_{max}), and to determine if there are any table model errors, user agents must use the following algorithm:

1. Let x_{max} be zero.
2. Let y_{max} be zero.
3. Let *the table* be the table represented by the `table` element. The x_{max} and y_{max} variables give *the table*'s extent. *The table* is initially empty.
4. If the `table` element has no table children, then return *the table* (which will be empty), and abort these steps.
5. 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 algorithm must be aborted at that point and the algorithm must return *the table*.
6. While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:

   - `caption`
   - `colgroup`
   - `thead`
   - `tbody`
   - `tfoot`
   - `tr`
7. If the *current element* is a `caption`, then that is the `caption` element associated with *the table*. Otherwise, it has no associated `caption` element.
8. If the *current element* is a `caption`, then 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`

(Otherwise, the *current element* will already be one of those elements.)

9. If the *current element* is a `colgroup`, follow these substeps:

    1. Let *next column* be 1.

    2. *Column groups.* Process the *current element* according to the appropriate one of the following two cases:

        ↪ If the *current element* has any `col` element children

        Follow these steps:

        1. Let x_{start} have the value x_{max}+1.

        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_{max} 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 third step of this innermost group of steps (columns).

        7. Let all the last columns in *the table* from x=x_{start} to x=x_{max} form a new column group, anchored at the slot (x_{start}, 1), with width x_{max}-x_{start}-1, 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_{max} by *span*.

        3. Let the last *span* columns in *the table* form a new column group, anchored at the slot (x_{max}-*span*+1, 1), with width *span*,

corresponding to the `colgroup` element.

3. Advance the *current element* to the next child of the `table`.
4. 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`
5. If the *current element* is a `colgroup` element, jump to step 2 in these substeps (column groups).

10. Let $y_{current}$ be zero. When the algorithm is aborted, if $y_{current}$ does not equal y_{max}, then that is a table model error.
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 (defined below), then return to the previous step (rows).
14. Otherwise, run the algorithm for ending a row group.
15. Let y_{start} have the value y_{max}+1.
16. For each `tr` element that is a child of the *current element*, in tree order, run the algorithm for processing rows (defined below).
17. If $y_{max} \geq y_{start}$, then let all the last rows in *the table* from y=y_{start} to y=y_{max} form a new row group, anchored at the slot with coordinate (1, y_{start}), with height y_{max}-y_{start}+1, corresponding to the *current element*.
18. Run the algorithm for ending a row group again.
19. Return to step 12 (rows).

The **algorithm for ending a row group**, which is invoked by the set of steps above when starting and eding a block of rows, is:

1. If $y_{current}$ is less than y_{max}, then this is a table model error.
2. While $y_{current}$ is less than y_{max}, follow these steps:
   1. Increase $y_{current}$ by 1.
   2. Run the algorithm for growing downward-growing cells.
3. 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. Increase $y_{current}$ by 1.
2. Run the algorithm for growing downward-growing cells.
3. Let $x_{current}$ be 1.
4. If the `tr` element being processed contains no `td` or `th` elements, then 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 or equal to x_{max} 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 greater than x_{max}, increase x_{max} by 1 (which will make them equal).
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 $x_{max} < x_{current}+colspan-1$, then let x_{max} be $x_{current}+colspan-1$.
12. If $y_{max} < y_{current}+rowspan-1$, then let y_{max} be $y_{current}+rowspan-1$.
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 what header cells apply to a data cell, use the algorithm for assigning header cells to data 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 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 step 5 (cells).

The **algorithm for growing downward-growing cells**, used when adding a new row, is as follows:

1. If the *list of downward-growing cells* is empty, do nothing. Abort these steps; return to the step that invoked this algorithm.
2. Otherwise, if y_{max} is less than $y_{current}$, then increase y_{max} by 1 (this will make it equal to $y_{current}$).
3. For each {*cell*, $cell_x$, *width*} tuple in the *list of downward-growing cells*, extend the cell *cell* so that it also covers the slots with coordinates (x, $y_{current}$), where $cell_x \le x < cell_x+width-1$.

If, after establishing which elements correspond to which slots, there exists a column in the table containing only slots that do not have a cell anchored to them, then this is a table model error.

3.15.11.2. Forming relationships between data cells and header cells

The **algorithm for assigning header cells to data cells** is as follows:

1. ...

3.16. Forms

This section will contain definitions of the `form` element and so forth.

This section will be a rewrite of the HTML4 Forms and Web Forms 2.0 specifications, with hopefully no normative changes.

3.16.1. The `form` element

3.16.2. The `fieldset` element

3.16.3. The `input` element

3.16.4. The `button` element

3.16.5. The `label` element

3.16.6. The `select` element

3.16.7. The `datalist` element

3.16.8. The `optgroup` element

3.16.9. The `option` element

3.16.10. The `textarea` element

3.16.11. The `output` element

3.16.12. Processing model

See WF2 for now

3.16.12.1. Form submission

See WF2 for now

3.17. Scripting

3.17.1. The `script` element

Block-level element, strictly inline-level content, and metadata element.

Contexts in which this element may be used:

In a `head` element.
Where block-level elements are expected.
Where inline-level 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 empty.

Element-specific attributes:

`src`
`defer` (if the `src` attribute is present)
`async` (if the `src` attribute is present)
`type`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLScriptElement : HTMLElement {
           attribute DOMString src;
           attribute boolean defer;
           attribute boolean async;
           attribute DOMString type;
           attribute DOMString text;
};
```

The `script` element allows authors to include dynamic script in their documents.

When the **`src`** attribute is set, the `script` element refers to an external file. The value of the attribute must be a URI.

If the `src` attribute is not set, then the script is given by the contents of the element.

The language of the script is given by the **`type`** attribute. The value must be a valid MIME type, optionally with parameters. [RFC2046]

Define defer/async attributes for authors.

Do we really want this attribute to be called async=""? Anyone have a better name?

Changing the `src`, `type`, `defer` and `async` 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 two 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.

When a script block is inserted into a document: When a `script` element whose "already executed" flag is not set is inserted into a document, the user agent must act as follows:

1. The user agent must set the element's "already executed" flag.
2. How to handle the `type` and `language` attributes should be defined here, probably with reference to the next section.
3. If the element has a `src` attribute, then a load for the specified content must be started.

   Note: Later, once the load has completed, the user agent will have to complete the steps described below.

   For performance reasons, user agents may start loading 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 load will have been effectively wasted.
4. 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**

   The element must be added to the end of the list of scripts that will execute when the document has finished parsing. The user agent must begin the next set of steps when the script is ready. 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. The user agent must jump to the next set of steps once the script is ready.

   ↪ **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 script that will execute as soon as the parser resumes. (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. The user agent must jump to the next set of steps when the script is ready.

↪ **Otherwise**

The user agent must immediately execute the script, even if other scripts are already executing.

When a script completes loading: If a script whose element was added to one of the lists mentioned above completes loading while the document is still being parsed, then the parser handles it. Otherwise, when a script completes loading, the UA must follow the following steps as soon as as any other scripts that may be executing have finished executing:

↪ **If the script's element was added to the list of scripts that will execute when the document has finished parsing:**

1. If the script's element is not the first element in the list, then do nothing yet. Stop going through these steps.
2. Otherwise, execute the script (that is, the script associated with the first element in the list).
3. Remove the script's 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's 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 (the script associated with the first element in the list).
3. Remove the script's 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's element was added to the list of scripts that will execute as soon as possible:**

1. Execute the script.
2. Remove the script's element from the list.

↪ **If the script is the script that will execute as soon as the parser resumes:**

The script will be handled when the parser resumes (amazingly enough).

need to say, in the part of the spec that fires the onload event, that the list above is triggered.

Executing a script block: If the load resulted in an error (for example a DNS error, or an HTTP 404 error), then executing the script consists of doing nothing. If the load was successful, then the behaviour depends on the scripting language.

If the script is from an external file, then that file must be used as the file to execute.

If the script is inline, then, for scripting languages that consist of pure text, user agents must use the value of the DOM `text` attribute (defined below) as the script to execute, and for XML-based scripting languages, user agents must use all the child nodes of the `script` element as the script to execute.

In any case, the user agent must execute the script according to the semantics of the relevant language specification, as determined by the `type` and `language` attributes on the `script` element when the element was inserted into the document, as described above.

Have to define that the script executes in context of global scope, scripting context, browsing context, etc.

> ***Note: The element's attributes' values might have changed between when the element was inserted into the document and when the script has finished loading, as may its other attributes; similarly, the element itself might have been taken back out of the DOM, or had other changes made. These changes do not in any way affect the above steps; only the values of the attributes at the time the `script` element is first inserted into the document matter.***

The DOM attributes **`src`**, **`type`**, **`defer`**, **`async`**, each must reflect the respective content attributes of the same name.

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.

3.17.1.1. Script languages

The following lists some MIME types and the languages to which they refer:

`text/javascript`
: 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.

3.17.2. The `noscript` element

Block-level element, and strictly inline-level 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 block-level elements are expected in HTML documents, if there are no ancestor `noscript` elements.
Where inline-level content is expected in HTML documents, if there are no ancestor `noscript` elements.

Content model:

See prose.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `noscript` element does not represent anything. 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.

The `noscript` element must not be used in XML documents.

When used in HTML documents, the allowed content model depends on whether scripting is enabled or not. If scripting is disabled, then the content model of a `noscript` element is the same as that of its parent element, with the following additional restrictions:

- Moving all the children of all the `noscript` elements in the document so that they replaced their `noscript` element parents, must not cause the document to become non-conforming.
- A `noscript` element must not have a `noscript` element as an ancestor (that is, `noscript` can't be nested).

If scripting is enabled, then the content model of a `noscript` element is 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.

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.

Note: All these contortions are required because, for historical reasons, the `noscript` element causes the HTML parser to act differently based on whether scripting is enabled or not. 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.

3.18. Interactive elements

3.18.1. The `details` element

Interactive, block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

One `legend` element followed by either one or more block-level elements or inline-level content (but not both).

Element-specific attributes:

`open`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLDetailsElement : HTMLElement {
  attribute boolean open;
};
```

The `details` element represents additional information or controls which the user can obtain on demand.

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, if present, indicates that the details should be shown to the user. The default, which applies if the attribute is absent, is that the details not be shown. If the attribute is present, its value must be the literal value `open`.

If the attribute is removed, then the details should be hidden. If the attribute is added, the details should be shown.

The user should be able to request that the details be shown or hidden.

The **open** attribute must reflect the `open` content attribute.

Rendering will be described in the Rendering section in due course. Basically CSS :open and :closed match the element, it's a block-level element by default, and when it matches :closed it renders as if it had an XBL binding attached to it whose template was just `<template>▶<content includes="legend:first-child">Details</content></template>`, and when it's :open it acts as if it had an XBL binding attached to it whose template was just `<template>▼<content includes="legend:first-child">Details</content><content/></template>` or some such.

Clicking the legend would make it open/close (and would change the content attribute). Question: Do we want the content attribute to reflect the actual state like this? I think we do, the DOM not reflecting state has been a pain in the neck before. But is it semantically ok?

3.18.2. The `datagrid` element

Interactive, block-level element.

Contexts in which this element may be used:

Where block-level elements are expected, if there are no ancestor interactive elements.

Content model:

Zero or more block-level elements.

Element-specific attributes:

`multiple`
`disabled`

Predefined classes that apply to this element:

None.

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
  void updateRowsInserted(in RowSpecification row, in unsigned long
  void updateRowsRemoved(in RowSpecification row, in unsigned long
  void updateRowChanged(in RowSpecification row);
  void updateColumnChanged(in unsigned long column);
  void updateCellChanged(in RowSpecification row, in unsigned long
};
```

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).

This element is defined as interactive, which means it can't contain other interactive elements, despite the fact that we expect it to work with other interactive elements e.g. checkboxes and input fields. It should be called something like a Leaf Interactive Element or something, which counts for ancestors looking in and not descendants looking out.

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` attribute, if present, must be either empty or have the literal value `multiple`. Similarly, the `disabled` attribute, if present, must be either empty or have the literal value `disabled`. (The actual values do not have any effect on how these attributes are processed, only the presence or absence of the attributes is important.)

The **`multiple`** and **`disabled`** DOM attributes must reflect the `multiple` and `disabled` content attributes respectively.

3.18.2.1. 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.

3.18.2.2. 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.

```
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 RowSpecifications 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.

3.18.2.3. 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).

```
// To be implemented by Web authors as a JS object
interface DataGridDataProvider {
  void initialize(in HTMLDataGridElement datagrid);
  unsigned long getRowCount(in RowSpecification row);
  unsigned long getChildAtPosition(in RowSpecification parentRow, in uns
  unsigned long getColumnCount();
  DOMString getCaptionText(in unsigned long column);
  void getCaptionClasses(in unsigned long column, in DOMTokenString clas
  DOMString getRowImage(in RowSpecification row);
  HTMLMenuElement getRowMenu(in RowSpecification row);
  void getRowClasses(in RowSpecification row, in DOMTokenString classes)
  DOMString getCellData(in RowSpecification row, in unsigned long columr
  void getCellClasses(in RowSpecification row, in unsigned long column,
  void toggleColumnSortState(in unsigned long column);
  void setCellCheckedState(in RowSpecification row, in unsigned long col
  void cycleCell(in RowSpecification row, in unsigned long column);
  void editCell(in RowSpecification row, in unsigned long column, in DOM
};
```

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 URI to 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.)

3.18.2.4. 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 behaviour 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`**

`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)`: If the row's first cell's element has an `img` element child, then the URI of the row's image is the URI of the first `img` element child of the row's first cell's element. Otherwise, the URI 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`**

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 `select` element child of the `datagrid` element, that skips all nodes other than `optgroup` and `option` elements, as well as any descendents 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 va
>        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` 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 initialised, 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 URI of the image given by 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.

> In the following example, the row with path "1,0" returns "http://example.com/a" as its image URI, 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 decsendant 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`.

3.18.2.4.1. 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` 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 descendent, 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 descendent of the `select`, then the first `option` element descendent 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 is checked, and the `unchecked` class otherwise.

The `setCellCheckedState()` method must set the `input` element's checkbox state to checked if the method's third argument is 1, and to unchecked 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.

3.18.2.5. 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 `DOMTokenString` interface, initialised to empty. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of columns. The values contained in the `DOMTokenString` 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 should be able to ask the UA to display the data 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 indentity 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. The values contained in the `DOMTokenString` object 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 is entirely up to the UA.

To obtain a URI to an image representing a row

Invoke the `getRowImage()` method with a `RowSpecification` object representing the row in question. The return value is a string representing a URI or IRI to an image. Relative URIs must be interpreted relative to the `datagrid`'s base URI. 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 `DOMTokenString` interface, initialised to empty. The second argument must be a non-negative integer less than the total number of columns. The values contained in the `DOMTokenString` object 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 how 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 appplies to the cell and the cell's checkbox is in an indeterminate state. When the `indeterminate` class appplies 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 betwen 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 honour 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 behaviour 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.

3.18.2.6. 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 indentified 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 only be called 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.

3.18.2.7. 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 should be able to toggle its 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 must be able to activate the cell to cycle its value. When the user activates this "cycling" behaviour 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 act as if the `datagrid`'s `updateCellChanged()` method had been invoked with those same arguments immediately before the provider's method was invoked.

When a cell has a checkbox, the user must be able 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 only be possible if the cell has the `indeterminate` class set). The `datagrid` must act as if the `datagrid`'s `updateCellChanged()` method had been invoked, specifying the same cell, immediately before the provider's method was invoked.

If a cell is editable, the user must be able 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 act as if the `updateCellChanged()` method had been invoked, with the same row and column specified, immediately before the provider's method was invoked.

3.18.2.8. 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;
  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. 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 must be able to select any number of rows (zero or more). If the attribute is not present, then the user must only be able to select 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.***

3.18.2.9. 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 must be able to invoke it to sort the data. 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.

3.18.3. The `command` element

Metadata element, and strictly inline-level content.

Contexts in which this element may be used:

In a `head` element.
Where strictly inline-level content is allowed.

Content model:

Empty.

Element-specific attributes:

`type`
`label`
`icon`
`hidden`
`disabled`
`checked`
`radiogroup`
`default`
Also, the `title` attribute has special semantics on this element.

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLCommandElement : HTMLElement {
           attribute DOMString type;
           attribute DOMString label;
           attribute DOMString icon;
           attribute boolean hidden;
           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's value must be either "`command`", "`checkbox`", or "`radio`", denoting each of these three types of commands respectively. The attribute may also be omitted if the element is to represent the first of these types, a simple command.

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 URI.

The **`hidden`** attribute indicates, if present, that the command is not relevant and is to be hidden. If present, the attribute must have the exact value `hidden`.

The **`disabled`** attribute indicates, if present, that the command is not available in the current state. If present, the attribute must have the exact value `disabled`.

Note: The distinction between Disabled State and Hidden State is subtle. A command should be Disabled if, in the same context, it could be enabled if only certain aspects of the situation were changed. A command should 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 indicates, if present, that the command is selected. If present, the attribute must have the exact value `checked`.

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.

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.

> 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**, **hidden**, **disabled**, **checked**, **radiogroup**, and **default** DOM attributes must reflect their respective namesake content attributes.

The **click()** method's behaviour depends on the value of the `type` attribute of the element, as follows:

↪ **If the `type` attribute has the value `checkbox`**

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 has the value `radio`**

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.

should change all the above so it actually is just trigged by a click event, then we could remove

the shadowing click() method and rely on actual events.

Need to define the command="" attribute

Note: `command` elements are not rendered unless they form part of a menu.

3.18.4. The `menu` element

Block-level element, and structured inline-level element.

Contexts in which this element may be used:

Where block-level elements are expected.
Where structured inline-level elements are allowed.

Content model:

Zero or more `li` elements, or inline-level content (but not both).

Element-specific attributes:

`type`
`label`
`autosubmit`

Predefined classes that apply to this element:

None.

DOM interface:

```
interface HTMLCommandElement : HTMLElement {
           attribute DOMString type;
           attribute DOMString label;
           attribute boolean autosubmit;
};
```

The `menu` element represents a list of commands.

The **`type`** attribute indicates the kind of menu. It must have either the value `popup` (to declare a context menu) or the value `toolbar` (to define a tool bar). The attribute may also be omitted, to indicate 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 has a `type` attribute with the value `popup`, then it 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 has a `type` attribute with the value `toolbar`, then it represents a list of active commands that the user can immediately interact with.

Otherwise, if a `menu` element has no `type` attribute, or if has a `type` attribute with a value other than `popup` or `toolbar`, then it either represents an unordered list of items (each represented by an `li` element), each of which represents a command that the user may perform or activate, or, if the element has no `li` element children, a paragraph 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 **`autosubmit`** attribute indicates whether selections made to form controls in this menu should result in the control's form being immediately submitted. If the attribute is present, its value must be `autosubmit`.

If a `change` event bubbles through a `menu` element, then, in addition to any other default action that that event might have, the UA must act as if the following was an additional default action for that event: if (when it comes time to execute the default action) the `menu` element has an `autosubmit` attribute, and the target of the event is an `input` element, and that element has a `type` attribute whose value is either `radio` or `checkbox`, and the `input` element in question has a non-null `form` DOM attribute, then the UA must invoke the `submit()` method of the `form` element indicated by that DOM attribute.

The processing model for menus is described in the next section.

3.19. Miscellaneous elements

3.19.1. The `legend` element

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:

If used as a child of a `fieldset` or `details` element: significant strictly inline-level content
If used as a child of a `figure` element: inline-level content.

Element-specific attributes:

None.

Predefined classes that apply to this element:

None.

DOM interface:

No difference from `HTMLElement`.

The `legend` element represents a title or explanatory caption for the rest of the contents of the `legend` element's parent element.

3.19.2. The `div` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more block-level elements.

Element-specific attributes:

None.

Predefined classes that apply to this element:

`error`, `example`, `issue`, `note`, `search`, `warning`

DOM interface:

No difference from `HTMLElement`.

The `div` element represents nothing at all. It can be used with the `class`, `lang`/`xml:lang`, and `title` attributes to mark up semantics common to a group of consecutive elements.

4. Processing models

4.1. Navigating across documents

This section only applies to Web browsers.

Web browsers operate on browsing contexts, which have a *session history*.

Certain actions cause the browsing context to *navigate* to a new document. A new document could be an HTML or XML `Document`, with a DOM, or it could be an image, or some other content. The new resource in question could have a URI, or it could be anonymous, for example a new document generated using the `document.write()` API.

In addition, a particular resource can have multiple entries in the session history of a browsing context.

The requirements that the session history concept puts on user agents are defined in the session history and navigation section, which is structured around the APIs provided for scripts to interact with the session history.

This section discusses the various types of content that a user agent may find itself requested to dispay in a content area, and covers how those types of content should be handled.

This section will end up defining what the UA should do when the user clicks a link, types a link in the address bar, uses a bookmark, etc. This will probably involve being honest about the fact that UAs typically content sniff for RSS/Atom feeds at this point. It should also reference the `registerProtocolHandler` and `registerContentHandler` methods and their stuff. Also involves the session history and bfcache (there's a separate section on that though).

Must define the interaction of this with document.write() pages and navigating to javascript: URIs.

Must define that the global scope is cleared/put back when navigating, so that Window objects from other windows looking at this one see the attributes change when the page navigates; this does/doesn't affect .watch(); make sure the .back()/.forward() definitions refer to this section.

4.1.1. Determining the type of a new resource in a browsing context

...

4.1.2. Page load processing model for HTML files

When an HTML page (as opposed to an XML page) is loaded in a browsing context, the user agent must begin to parse the stream of bytes as described in the parsing section.

create a `Document` node, mark it as being an HTML document, create an HTML parser, associate them together, run the parser while pushing bytes into the end of the input stream, push an explicit EOF when you have no more bytes, and require the parser to wait for an explicit EOF in this case (or some equivalent)

This will usually eventually cause a `load` event to be fired.

This needs to say that a parser is created, etc.

4.1.3. Page load processing model for XML files

Must make sure we don't step on the toes of other specifications.

4.1.4. Page load processing model for text files

Must make sure we don't step on the toes of other specifications. This section should be referred to by the document.open() stuff.

4.1.5. Page load processing model for syndication feeds

...

4.1.6. Content-Type

This is a placeholder section for some text that will define exactly how to handle Content-Type headers for top-level browsing contexts, for <img>, <embed>, <object>, etc; and will cover things like the fact that on some operating systems the extension of a URI determines the Content-Type for file:// content, or something.

4.2. Scripting

Here are some of the things I think we should list:

- onload/onunload events
- alert() blocks scripts, even those on timeouts or readystatechange events

4.2.1. Running executable code

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:` URIs (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.

User agents may provide a mechanism to enable or disable the execution of author-provided code. When the user agent is configured such that author-provided code does not execute, or if the user agent is implemented so as to never execute author-provided code, it is said that **scripting is disabled**. When author-provided code *does* execute, **scripting is enabled**. A user agent with scripting disabled is a user agent with no scripting support for the purposes of conformance.

4.2.2. Scripting contexts

Each `Document` in a browsing context has an associated **scripting context**.

4.2.3. Threads

Each scripting context is defined as having a list of zero or more **reachable scripting contexts**. These are:

- All the scripting contexts of all the `Document`s in each nested browsing context inside the scripting context's `Document`.
- The scripting context of the `Document` that contains the `Document`'s browsing context.
- All the scripting contexts of all the `Document`s associated with any top-level browsing contexts that were opened from a `Document` associated with the scripting context's `Document`'s browsing context.
- All the scripting contexts of all the `Document`s associated with the top-level browsing contexts of the browsing context of the `WindowHTML` object returned by the scripting context's `Document`'s `WindowHTML`'s `opener` attribute.

The transitive closure of all the scripting contexts that are reachable scripting contexts consists of a **unit of related scripting contexts**.

All the executable code in a unit of related scripting contexts must execute on a single conceptual thread. While a script is executing, no other scripts in that unit of related scripting contexts must be started.

Events fired by the user agent in response to user actions must be queued, as must the execution of any timers. There must be only one queue per unit of related scripting contexts. When no script in the unit of related scripting contexts is executing, the oldest entry in the queue must be processed, either by firing the appropriate event, or by executing the relevant timer callback code.

4.2.4. The security model

Web browser vendors should implement this security model, to provide Web authors with a

consistent development environment that is interoperable across different implementations. However, implementors may use any other model if desired.

The security model for Web browsers has grown organically with the development of the Web, and as such is somewhat unique.

Access to resources and APIs is granted or denied to Web content (scripts, elements, etc) based on the content's **origin**. For historical reasons, the mechanism for determining the *origin* of a particular piece of content depends on the nature of the content.

...

The **domain of a `Document` object** is the domain given by the `hostname` attribute of the `Location` object returned by the `Document` object's `location` attribute, *if* that `hostname` attribute is not the empty string. If it is, the domain of the document is UA-defined. For now. Need to be more correct about where .location is defined. It's not actually on Document.

The **domain of a script** is the domain of the `Document` object that is returned by the `document` attribute of the script's primary `Window` object (in UAs that implement ECMAScript, that is the global scope object).

The string representing the script's domain in IDNA format is obtained as follows: take the script's domain and apply the IDNA ToASCII algorithm and then the IDNA ToUnicode algorithm to each component of the domain name (with both the AllowUnassigned and UseSTD3ASCIIRules flags set both times). [RFC3490] 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 the string representing the script's domain in IDNA format cannot be obtained. (ToUnicode is defined to never fail.)

This section is *so* not complete.

Define **security exception**.

4.2.5. `javascript:` URIs

What's the scripting context for javascript: URIs in <img src>, <iframe src>, <location.href>, <a href>, @import, background-image, etc?

4.2.6. Runtime script errors

Whenever a runtime script error occurs in one of the scripts associated with the document, the value of the **`onerror`** attribute of the `window` object (defined on the `WindowHTML` interface of that object), must be processed, as follows:

↪ **If the value is a function**

The function referenced by the `onerror` attribute must be invoked with three arguments, before notifying the user of the error.

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 URI to the

resource in which the error occured, and the third must give the line number in that resource on which the error occured.

If the function returns false, then the error should not be reported to the user. Otherwise, if the function returns another value (or does not return at all), the error should be reported to the user.

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 calling the function again.

↪ **If the value is `null`**

The error should not reported to the user.

↪ **If the value is anything else**

The error should be reported to the user.

The initial value of `onerror` must be `undefined`.

4.2.7. Events

4.2.7.1. Event handler attributes

HTML elements can have event handler attributes specified. These act as bubbling event listeners for the element on which they are specified.

Each event handler attribute has two parts, an event handler content attribute and an event handler DOM attribute. Event handler attributes must initially be set to null. When their value changes (through the changing of their event handler content attribute or their event handler DOM attribute), they will either be null, or have an `EventListener` object assigned to them.

In the ECMAScript DOM binding, the ECMAScript native `Function` type must implement the `EventListener` interface such that invoking the `handleEvent()` method of that interface on the object from another language binding invokes the function itself, with the `event` argument as its only argument. In the ECMAScript binding itself, however, the `handleEvent()` method of the interface is not directly accessible on `Function` objects. Such functions, when invoked, must be called in the global scope. If the function returns false, the event's `preventDefault()` method must then invoked. Exception: for historical reasons, for the HTML `mouseover` event, the `preventDefault()` method must be called when the function returns true instead.

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, its new value must be interpreted as the body of an anonymous function with a single argument called `event`, with the new function's scope chain being linked from the activation object of the handler, to the element, to the element's `form` element if it is a form control, to the `Document` object, to the global scope. The function's `this` parameter must be the `Element` object representing the element. The resulting function must then be set as the value of the corresponding event handler attribute, and the new value must be set as the value of the content attribute. If the given function body fails to compile, then the corresponding event handler attribute must be set to null instead (the content attribute must still be updated to the new value, though).

Note: See ECMA262 Edition 3, sections 10.1.6 and 10.2.3, for more details on

activation objects. [ECMA262]

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).

The following are the event handler attributes that must be supported by all HTML elements, as both content attributes and DOM attributes:

`onclick`

Must be invoked whenever a `click` event is targetted at or bubbles through the element.

...more events...

4.2.7.2. Event firing

maybe _this_ should be moved higher up (terminology? conformance? DOM?)

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 in the `http://www.w3.org/2001/xml-events` namespace, which bubbles and is cancelable, and which uses the `MouseEvent` interface, must be dispatched at the given element. 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 `change` event means that a `change` event in the `http://www.w3.org/2001/xml-events` namespace, which bubbles but is not cancelable, and which uses the `Event` interface, must be dispatched at the given element. The event object must have its `detail` attribute set to 0.

Firing a `contextmenu` event means that a `contextmenu` event in the `http://www.w3.org/2001/xml-events` namespace, which bubbles and is cancelable, and which uses the `Event` interface, must be dispatched at the given element. The event object must have its `detail` attribute set to 0.

Firing a simple event called *e* means that an event with the name *e*, in the `http://www.w3.org/2001/xml-events` namespace, which does not bubble but is cancelable, and which uses the `Event` interface, must be dispatched at the given element. The event object must have its `detail` attribute set to 0.

Firing a `show` event means firing a simple event called `show`. **Firing a `load` event** means firing a simple event called `load`. **Firing an `error` event** means firing a simple event called `error`.

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, they should get triggered. 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.

4.2.7.3. Event handling

We need a section to define how events all work, default actions, etc. For example, how does clicking on a span in a link that is in another link actually cause a link to be followed? which one? (where should this section be?)

4.3. Links

4.3.1. 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 URI (or IRI). This URI 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.***

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. 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 recognised by the UA, 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 or has an invalid value, 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 only use 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 only use the Content-Type information associated with the resource to determine its type, not metadata included in the link to the resource.

The **`ping`** attribute, if present, gives the URIs 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 URIs. The value is used by the user agent when following hyperlinks.

4.3.2. Following hyperlinks

following hyperlinks defined here

If an `a` or `area` hyperlink element has a `ping` attribute and the user follows the hyperlink, the user agent must take the `ping` attribute's value, strip leading and trailing spaces, split the value on sequences of spaces, treat each resulting part as a URI (resolving relative URIs according to element's base URI) and then should send a request to each of the resulting URIs. 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 behaviour, for example in conjunction with a setting that disables the sending of HTTP Referrer headers. Based on the user's preferences, UAs may either ignore the `ping` attribute altogether, or selectively ignore URIs in the list (e.g. ignoring any third-party URIs).

For URIs that are HTTP URIs, the requests must be performed using the POST method (with an empty entity body in the request). User agents must ignore any entity bodies returned in the responses, but must, unless otherwise specified by the user, honour the HTTP headers — in particular, HTTP cookie headers. [RFC2965]

> ***Note: To save bandwidth, implementors might wish to consider omitting optional headers such as `Accept` from these requests.***

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 URIs.

> ***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 URI 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 optimise 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.***

4.3.3. Link types

The following table summarises 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.

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.
bookmark	*not allowed*	Hyperlink	Gives the permalink for the nearest ancestor section.
contact	Hyperlink	Hyperlink	Gives a link to contact information for the current document.
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.
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 pre-emptively 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.

Link type	Effect on...		Brief description
	`link`	**`a` and `area`**	
`sidebar`	Hyperlink	Hyperlink	Specifies that the referenced document, if retrieved, is intended to be shown in the browser's sidebar (if it has one).
`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.

4.3.3.1. 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 contains 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, 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.

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.

4.3.3.2. 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.

4.3.3.3. 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:` URI 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.

4.3.3.4. 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>
 <div id="a">
  <h2>First example</h2>
  <p><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.</p>
 </div>
 <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).
  </article>
 </article>
</body>
...
```

4.3.3.5. Link type "contact"

The `contact` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

For `a` and `area` elements, the `contact` keyword indicates that the referenced document provides further contact information for the section that the element defining the hyperlink applies to.

User agents must treat any hyperlink in an `address` element as having the `contact` link type specified.

For `link` elements, the `contact` keyword indicates that the referenced document provides further contact information for the page as a whole.

4.3.3.6. 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.

4.3.3.7. 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:
>
> ```
> You can access the planets database using Atom feeds:
> <ul>
>  <li><a href="recently-visited-planets.xml" rel="feed">Recently Visi
>  <li><a href="known-bad-planets.xml" rel="feed">Known Bad Planets</a
>  <li><a href="unexplored-planets.xml" rel="feed">Unexplored Planets<
> </ul>
> ```

4.3.3.8. 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="
> ```

For link elements, the help keyword indicates that the referenced document provides help for the page as a whole.

4.3.3.9. 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 media attribute.

4.3.3.10. 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.

4.3.3.11. 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.

4.3.3.12. 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]

4.3.3.13. 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.

4.3.3.14. 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.

4.3.3.15. Link type "`stylesheet`"

The `stylesheet` keyword may be used with `link` elements, for which it creates an external resource link.

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 alternate stylesheet.

Need more here -- defining preferred stylesheets, alternate stylesheets, persistent stylesheets, and the stuff about the alternate stylesheet API. Maybe this should all be deferred to another processing model section.

4.3.3.16. 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.

4.3.3.17. 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.

4.3.3.18. 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. A subdocument is said to be a *child* of the document it is a subdocument of. The document of which it is a subdocument is said to be its *parent*. The children of a document have a relative order; the subdocument that precedes another is its *previous sibling*, and the one that follows it is its *next sibling*. A document with no parent forms the top of the hierarchy.

4.3.3.18.1. 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 hierarchical structure, and that the link is leading to the document that is the first child of the current document's parent document.

Synonyms: For historical reasons, user agents must also treat the keywords "begin" and "start" like the first keyword.

4.3.3.18.2. 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.

Synonyms: For historical reasons, user agents must also treat the keywords "top", "contents", and "toc" like the index keyword.

4.3.3.18.3. 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 hierarchical structure, and that the link is leading to the document that is the last child of the current document's parent document.

Synonyms: For historical reasons, user agents must also treat the keyword "end" like the last keyword.

4.3.3.18.4. 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 hierarchical structure, and that the link is leading to the document that is the next sibling of the current document.

4.3.3.18.5. 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 hierarchical structure, and that the link is leading to the document that is the previous sibling of the current document.

Synonyms: For historical reasons, user agents must also treat the keyword "previous" like the prev keyword.

4.3.3.18.6. 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.

4.3.3.19. 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; it creates a hyperlink.

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.

4.3.4. Image maps

...**image map**, **valid area**, **`usemap`**, **`coords`**, **`shape`**, **server side image map**.

default action of clicking an image is to check if target was an image with usemap; if so, fire DOMActivate on the matching `area`, otherwise do nothing. (Or, do click events clicked on areas fire directly on the area and not on the img?)

4.4. 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

A graphical image that represents the action.

Hidden State

Whether the command is hidden or not (basically, whether it should be shown in menus).

Disabled State

Whether the command can be triggered or not. If the Hidden State is true (hidden) then the Disabled State will be true (disabled) regardless.

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 URI to which to navigate, or a form submission.

Triggers

The list of elements that can trigger the command. The element defining a command is always in the list of elements that can trigger the command. For anonymous commands, only the element defining the command is on the list, since other elements have no way to refer to it.

Commands are represented by elements in the DOM. Any element that can define a command also implements the `Command` interface:

```
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();
  readonly attribute HTMLCollection triggers;
  readonly attribute Command command;
};
```

The `Command` interface is implemented by any element capable of defining a command. (If an element can define a command, its definition will list this interface explicitly.) All the attributes of the `Command` interface are read-only. Elements implementing this interface may implement other interfaces that have attributes with identical names but that are mutable; in bindings that simply flatten all supported interfaces on the object, the mutable attributes must shadow the readonly attributes defined in the `Command` interface.

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 an absolute URI to 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 `command` elements.

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 **`triggers`** attribute must return a list containing the elements that can trigger the command (the command's Triggers). The list must be live. While the element does not define a command, the list must be empty.

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`.

4.4.1. 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 `a` element. If the attribute is not present, the Hint is the empty string.

The Icon of the command is the absolute URI of the first image in the element. Specifically, in a depth-first search of the children of the element, the first element that is `img` element with a `src` attribute is the one that is used as the image. The URI must be taken from the element's `src` attribute. Relative URIs must be resolved relative to the base URI of the image element. If no image

is found, then the Icon facet is left blank.

The Hidden State and Disabled State facets of the command are 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.

4.4.2. 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. Typically this is given by the element's `disabled` attribute, but certain button types become disabled at other times too (for example, the `move-up` button type is disabled when it would have no effect).

4.4.3. Using the `input` element to define a command

An `input` element whose `type` attribute is one of `submit`, `reset`, `button`, `radio`, `checkbox`, `move-up`, `move-down`, `add`, and `remove` defines a command.

The Type of the command is "radio" if the `type` attribute has the value `radio`, "checkbox" if the `type` attribute has the value `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 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 has a `label` element associated with it, the `textContent` of the first such element 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 always false. (The command is never hidden.)

The Disabled State of the command mirrors the disabled state of the control. Typically this is given by the element's `disabled` attribute, but certain input types become disabled at other times too (for example, the `move-up` input type is disabled when it would have no effect).

The Checked State of the command is true if the command is of Type "radio" or "checkbox" and the element has a `checked` attribute, and false otherwise.

The Action of the command is to fire a `click` event at the element.

4.4.4. 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 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.

There is no Icon for the command.

The Hidden State of the command is always false. (The command is never hidden.)

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's `selected` DOM attribute is true, and false otherwise.

The Action of the command depends on its Type. If the command is of Type "radio" then this must set the `selected` DOM attribute of the `option` element to true, otherwise it must toggle the state of the `selected` DOM attribute (set it to true if it is false and vice versa). Then a `change` event must be fired on the `option` element's nearest ancestor `select` element (if there is one), as if the selection had been changed directly.

4.4.5. 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 URI resulting from resolving the value of the element's `icon` attribute as a URI relative to the element's base URI. If the element has no `icon` attribute then the command has no Icon.

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 either a `disabled` attribute or a `hidden` attribute (or both), 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 behaviour described in the definition of the `click()` method of the `HTMLCommandElement` interface.

4.5. Menus

4.5.1. Introduction

This section is non-normative.

...

4.5.2. Building menus

A menu 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 element is built by iterating over its child nodes.

For each child node in tree order, the required behaviour depends on what the node is, as follows:

↪ **An element that defines a command**

Append the command to the menu. 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.

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.

4.5.3. 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.

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, as defined above.

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.

4.5.4. Toolbars

Toolbars are a kind of menu that is always visible.

When a `menu` element has a `type` attribute with the value `toolbar`, then the user agent must build the menu for that `menu` element and render it in the document in a position appropriate for that `menu` element.

The user agent must reflect changes made to the `menu`'s DOM immediately in the UI.

4.6. Repetition templates

See WF2 for now

5. The browser environment

This section describes a set of APIs that allow authors to make their documents and applications interact with the user agent, integrating with native features such as the navigation history, drag-and-drop, undo/redo, and selections.

5.1. The global scope

Many of the APIs are part of the `WindowHTML` interface. The `WindowHTML` interface must be obtainable from the `Window` object using binding-specific casting methods. [WINDOW]

```
interface WindowHTML {

  // defined in this section
  readonly attribute History history;
  readonly attribute ClientInformation navigator;
  readonly attribute UndoManager undoManager;
  Selection getSelection();
  readonly attribute Storage sessionStorage;
  readonly attribute StorageList globalStorage;

  // defined in other sections
           attribute Object onerror;

  // more...
};
```

The `WindowHTML` object must provide the following constructors:

`Audio()`

Constructs an `Audio` object.

`Image()`

`Image(in unsigned long w)`

`Image(in unsigned long w, in unsigned long h)`

Constructs an `HTMLImageElement` object (a new `img` element). If the *h* argument is present, the new object's `height` content attribute must be set to *h*. If the *w* argument is present, the new object's `width` content attribute must be set to *w*.

`Option()`

`Option(in DOMString name)`

`Option(in DOMString name, in DOMString value)`

Constructs an `HTMLOptionElement` object (a new `option` element). need to define argument processing

5.2. Session history and navigation

5.2.1. The session history of browsing contexts

`History` objects provide a representation of the pages in the session history of their `Window` object's browsing context. Each browsing context (`frame`, `iframe`, etc) has a distinct session history.

Each `DocumentUI` 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 `WindowHTML` interface must return the object implementing the `History` interface for that `WindowHTML` object's associated `DocumentUI` object.

`History` objects represent their browsing context's session history as a flat list of URIs and state objects. (This does not imply that the UI need be linear. See the notes below.)

Typically, the history list will consist of only URIs. However, a page can add **state objects** between its 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.

Entries that consist of state objects share the same `DocumentUI` as the entry for the URI itself. Contiguous entries that differ just by fragment identifier must also share the same `DocumentUI`.

> ***Note: All entries that share the same `DocumentUI` (and that are therefore merely different states of one particular document) are contiguous by definition.***

At any point, one of the entries in the session history is the **current entry**. This is the entry representing the page in this browsing context that is considered the "current" page by the UA. The current entry is usually an entry for the location of the `DocumentUI`. However, it can also be one of the entries for state objects added to the history by that document.

When the browser's navigation model differs significantly from the sequential model represented by the `History` interface, for example if separate `DocumentUI` objects in the session history are all simulatenously displayed and active, then the current entry could even be an entry unrelated to the `History` object's own `DocumentUI` object. If, when a method is invoked on a `History` object, the current entry for that browsing context's session history has a different `DocumentUI` object than the `History` object's own `DocumentUI` object, then the user agent must raise a

`NO_MODIFICATION_ALLOWED_ERR` DOM exception. (This can only happen if scripts are allowed to run in documents that are not the current document. Typically, however, user agents only allow scripts from the current entry to execute.)

User agents may **discard** the DOMs of entries other than the current entry, reloading the pages afresh when the user or script navigates back to such pages. This specification does not specify when user agents should discard pages' DOMs and when they should cache them. See the section on the `load` and `unload` events for more details.

Entries that have had their DOM 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 `DocumentUI` object with it must share the new object as well.

When a user agent discards the DOM from an entry in the session history, it must also discard all the entries from the first state object entry for that `DocumentUI` object up to and including the last entry for that `DocumentUI` object (including any non-state-object entries in that range, such as entries where the user navigated using fragment identifiers). These entries are not recreated if the user or script navigates back to the page. If there are no state object entries for that `DocumentUI` object then no entries are removed.

5.2.2. The `History` interface

```
interface History {
  readonly attribute long length;
  void go(in long delta);
  void go();
  void back();
  void forward();
  void pushState(in DOMObject data);
  void clearState();
};
```

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 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.

If the *delta* is zero, then the user agent must act as if the `location.reload()` method was called instead.

Otherwise, the user agent must cause the current browsing context to navigate to the specified entry, as described below. The specified entry is the one whose index equals the index of the current entry plus *delta*.

If there are any entries with state objects between the current entry and the specified entry (not inclusive), then the user agent must iterate through every entry between the current entry and the specified entry, starting with the entry closest to the current entry, and ending with the one closest to

the specified entry. For each entry, if the entry is a state object, the user agent must activate the state object.

If the specified entry has a different `DocumentUI` object than the current entry then the user agent must make that `DocumentUI` object the user's "current" one for that browsing context.

If the specified entry is a state object, the user agent must activate that state object. Otherwise, the user agent must update the current location object to the new location.

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.

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
`go()`	Must do the same as `go(0)`
`back()`	Must do the same as `go(-1)`
`forward()`	Must do the same as `go(1)`

The **`pushState(data)`** method adds a state object to the history.

When this method is invoked, the user agent must first remove from the session history any entries for that `DocumentUI` from the entry after the current entry up to the last entry in the session history that references the same `DocumentUI` 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 `DocumentUI` object, then no entries are removed.

Then, the user agent must add a state object entry to the session history, after the current entry, with the specified *data* as the state object.

Finally, the user agent must update the current entry to be the this newly added entry.

There has been a suggestion that pushState() should take a URI and a string; the URI to allow for the page to be bookmarked, and the string to allow the UA to give the page a meaningful title in the history state, if it shows history state.

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 `DocumentUI` 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 `DocumentUI` 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 `DocumentUI` object up to the last entry that references that same `DocumentUI` 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 DocumentUI object in the session history.

5.2.3. Activating state objects

When a state object in the session history is activated (which happens in the cases described above), the user agent must fire a **popstate** event in the http://www.w3.org/2001/xml-events namespace on the the body element using the PopStateEvent interface, with the state object in the state attribute. This event bubbles but is not cancelable and has no default action.

```
interface PopStateEvent : Event {
  readonly attribute DOMObject state;
  void initPopStateEvent(in DOMString typeArg, in boolean canBubbleArg,
  void initPopStateEventNS(in DOMString namespaceURIArg, in DOMString ty
};
```

The **initPopStateEvent()** and **initPopStateEventNS()** methods must initialise 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.

5.2.4. The Location interface

The **location** attribute of the WindowHTML interface must return an object implementing the Location interface.

For historical reasons, the **location** attribute of the DocumentUI interface must return the same object as the location attribute on its associated WindowHTML object.

Location objects provide a representation of the URI of their document, 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 hash;
  readonly attribute DOMString host;
  readonly attribute DOMString hostname;
  readonly attribute DOMString href;
  readonly attribute DOMString pathname;
  readonly attribute DOMString port;
  readonly attribute DOMString protocol;
  readonly attribute DOMString search;
  void assign(in DOMString url);
  void replace(in DOMString url);
  void reload();
};
```

In the ECMAScript DOM binding, objects implementing this interface must stringify to the same value as the href attribute.

In the ECMAScript DOM binding, the location members of the DocumentUI and WindowHTML interfaces behave as if they had a setter: user agents must treats attempts to set these location attribute as attempts at setting the href attribute of the relevant Location object instead.

The **`href`** attribute returns the address of the page represented by the associated `DocumentUI` object, as an absolute IRI reference.

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 remove all the entries after the current entry in its `DocumentUI`'s `History` object, add a new entry, with the given *url*, at the end of the list (asynchronously loading the new page if necessary), and then advance to that page as if the `history.forward()` method had been invoked.

When the **`replace(`*`url`*`)`** method is invoked, the UA must act as if the `assign()` method had been invoked, but with the additional step of removing the entry that was the current entry before the method call after the above steps (thus simply causing the current page to be replaced by the new one).

In both cases, if the location before the method call would differ from the location after the method only in terms of the fragment identifier, then the user agent must use the same `DocumentUI` object, updating only the scroll position in the document's view(s) appropriately.

Relative *url* arguments for `assign()` and `replace()` must be resolved relative to the base URI of the script that made the method call.

The component parts and .reload() are yet to be defined. If anyone can come up with a decent definition, let me know.

5.2.5. 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.

5.3. Browser state

The **navigator** attribute of the WindowHTML interface must return an instance of the ClientInformation 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 ClientInformation {
  readonly attribute boolean onLine;
  void registerProtocolHandler(in DOMString protocol, in DOMString uri,
  void registerContentHandler(in DOMString mimeType, in DOMString uri, i
};
```

5.3.1. Offline Web applications

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.

The **offline** event must be fired when the value of the navigator.onLine attribute of the WindowHTML changes from true to false.

The **online** event must be fired when the value of the navigator.onLine attribute of the WindowHTML changes from false to true.

These events are in the http://www.w3.org/2001/xml-events namespace, do bubble, are not cancelable, have no default action, and use the normal Event interface. They must be fired on the body element. (As the events bubble, they will reach the WindowHTML object.)

5.3.2. 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.

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:

***protocol* (registerProtocolHandler() only)**

A scheme, such as ftp or fax. The scheme must be treated case-insensitively by user agents for the purposes of comparing with the scheme part of URIs 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 treated case-insensitively 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.

uri

The URI of the page that will handle the requests. When the user agent uses this URI, it must replace the first occurrence of the exact literal string "`%s`" with an escaped version of the URI of the content in question (as defined below), and then fetch the resulting URI using the GET method (or equivalent for non-HTTP URIs).

To get the escaped version of the URI, first, the domain part of the URI (if any) must be converted to its punycode representation, and then, every character in the URI that is not in the ranges given in the next paragraph must be replaced by its UTF-8 byte representation, each byte being represented by a U+0025 (%) character and two digits in the range U+0030 (0) to U+0039 (9) and U+0041 (A) to U+0046 (F) giving the hexadecimal representation of the byte.

The ranges of characters that must not be escaped are: U+002D (-), U+002E (.), U+0030 (0) to U+0039 (9), U+0041 (A) to U+005A (Z), U+005F (_), U+0061 (a) to U+007A (z), and U+007E (~).

> If the user had visited a site that made the following call:
>
> ```
> navigator.registerContentHandler('application/x-soup', 'http://e
> ```
>
> ...and then clicked on a link such as:
>
> ```
> <a href="http://www.example.net/chickenkïwi.soup">Download our C
> ```
>
> ...then, assuming this `chickenkiwi.soup` file was served with the MIME type `application/x-soup`, the UA might instead navigate to the following URI:
>
> ```
> http://example.com/soup?url=http%3A%2F%2Fwww.example.net%2Fchick
> ```
>
> This site could then fetch the `chickenkiwi.soup` file and do whatever it is that it does with soup (synthesise 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 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` URIs 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 *uri* argument passed to one of these methods does not contain the exact literal string "`%s`".

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 *uri* 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.

5.3.2.1. 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 wholy 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 URIs. The mechanism described in this section can result in secret Intranet URIs 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 URI to the Intranet content.

No actual confidential file data is leaked in this manner, but the URIs themselves could contain confidential information. For example, the URI could be `https://www.corp.example.com/upcoming-aquisitions/samples.egf`, which might tell the third party that Example Corporation is intending to merge with Samples LLC. Implementors might wish to consider allowing administrators to disable this feature for certain subdomains, content types, or protocols.

Leaking secure URIs. User agents should not send HTTPS URIs 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 URIs that are escaped and included sent to the handler sites. User agents may even avoid attempting to pass to Web-based handlers the URIs 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).

5.3.2.2. 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:

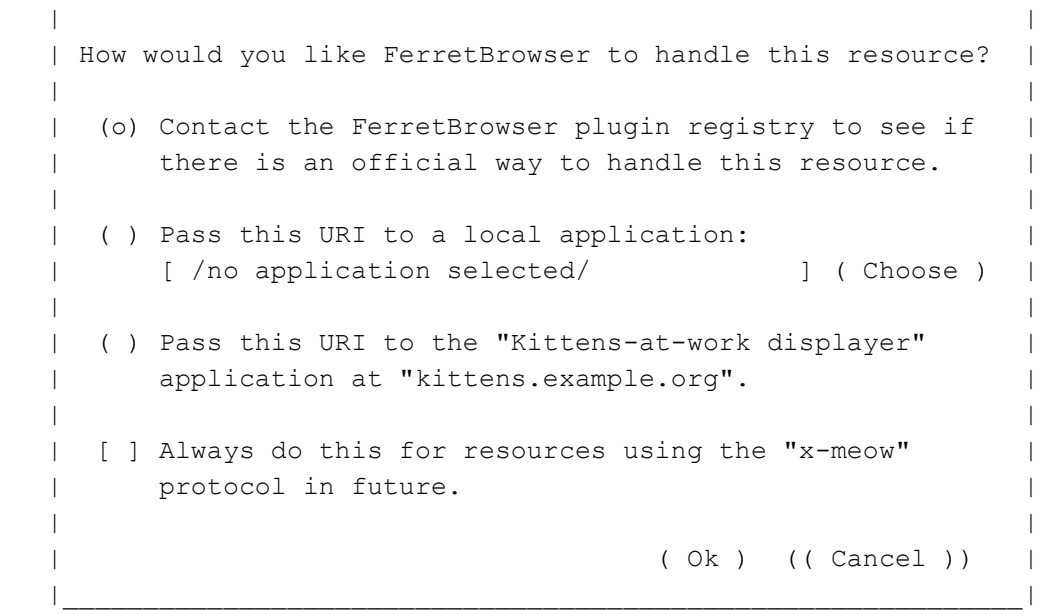
```
|||[ Protocol Handler Registration ]|||||||||||||||||||||||||||
|                                                              |
| This Web page:                                               |
|                                                              |
|    Kittens at work                                           |
|    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 URI 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 (*uri*), 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 URI that uses the "x-meow:" scheme, then it might display a dialog as follows:

```
|||[ Unknown Protocol ]|||||||||||||||||||||||||||||||||||||||||
|                                                              |
| You have attempted to access:                                |
|                                                              |
|    x-meow:S2l0dGVucyBhcmUgdGhlIGN1dGVzdCE%3D                 |
```



...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.

5.4. Client-side session and persistent storage

5.4.1. Introduction

This section is non-normative.

This specification introduces two related mechanisms, similar to HTTP session cookies [RFC2965], for storing structured data on the client side.

The first is designed for scenarios where the user is carrying out a single transaction, but could be carrying out multiple transactions in different windows at the same time.

Cookies don't really handle this case well. For example, a user could be buying plane tickets in two different windows, using the same site. If the site used cookies to keep track of which ticket the user was buying, then as the user clicked from page to page in both windows, the ticket currently being purchased would "leak" from one window to the other, potentially causing the user to buy two tickets for the same flight without really noticing.

To address this, this specification introduces the `sessionStorage` DOM attribute. Sites can add data to the session storage, and it will be accessible to any page from that domain opened in that window.

> For example, a page could have a checkbox that the user ticks to indicate that he wants insurance:
>
> ```
> <label>
>  <input type="checkbox" onchange="sessionStorage.insurance = checked"
>  I want insurance on this trip.
> ```

```
  </label>
```

A later page could then check, from script, whether the user had checked the checkbox or not:

```
if (sessionStorage.insurance) { ... }
```

If the user had multiple windows opened on the site, each one would have its own individual copy of the session storage object.

The second storage mechanism is designed for storage that spans multiple windows, and lasts beyond the current session. In particular, Web applications may wish to store megabytes of user data, such as entire user-authored documents or a user's mailbox, on the clientside for performance reasons.

Again, cookies do not handle this case well, because they are transmitted with every request.

The `globalStorage` DOM attribute is used to access the global storage areas.

The site at example.com can display a count of how many times the user has loaded its page by putting the following at the bottom of its page:

```
<p>
  You have viewed this page
  <span id="count">an untold number of</span>
  time(s).
</p>
<script>
  var storage = globalStorage['example.com'];
  if (!storage.pageLoadCount)
    storage.pageLoadCount = 0;
  storage.pageLoadCount = parseInt(storage.pageLoadCount, 10) + 1;
  document.getElementById('count').textContent = storage.pageLoadCoun
</script>
```

Each domain and each subdomain has its own separate storage area. Subdomains can access the storage areas of parent domains, and domains can access the storage areas of subdomains.

- `globalStorage['']` is accessible to all domains.
- `globalStorage['com']` is accessible to all .com domains
- `globalStorage['example.com']` is accessible to example.com and any of its subdomains
- `globalStorage['www.example.com']` is accessible to www.example.com and example.com, but not www2.example.com.

Storage areas (both session storage and global storage) store strings. To store structured data in a storage area, you must first convert it to a string.

5.4.2. The `Storage` interface

```
interface Storage {
  readonly attribute unsigned long length;
  DOMString key(in unsigned long index);
  StorageItem getItem(in DOMString key);
  void setItem(in DOMString key, in DOMString data);
  void removeItem(in DOMString key);
};
```

Each `Storage` object provides access to a list of key/value pairs, which are sometimes called items. Keys are strings, and any string (including the empty string) is a valid key. Values are strings with associated metadata, represented by `StorageItem` objects.

Each `Storage` object is associated with a list of key/value pairs when it is created, as defined in the sections on the `sessionStorage` and `globalStorage` attributes. Multiple separate objects implementing the `Storage` interface can all be associated with the same list of key/value pairs simultaneously.

Key/value pairs have associated metadata. In particular, a key/value pair can be marked as either "safe only for secure content", or as "safe for both secure and insecure content".

A key/value pair is **accessible** if either it is marked as "safe for both secure and insecure content", or it is marked as "safe only for secure content" and the script in question is running in a secure scripting context.

The **`length`** attribute must return the number of key/value pairs currently present and accessible in the list associated with the object.

The **`key(`*`n`*`)`** method must return the name of the *n*th accessible key in the list. The order of keys is user-agent defined, but must be consistent within an object between changes to the number of keys. (Thus, adding or removing a key may change the order of the keys, but merely changing the value of an existing key must not.) If *n* is less than zero or greater than or equal to the number of key/value pairs in the object, then this method must raise an `INDEX_SIZE_ERR` exception.

The **`getItem(`*`key`*`)`** method must return the `StorageItem` object representing the key/value pair with the given *key*. If the given *key* does not exist in the list associated with the object, or is not accessible, then this method must return null. Subsequent calls to this method with the same key from scripts running in the same security context must return the same instance of the `StorageItem` interface. (Such instances must not be shared across security contexts, though.)

The **`setItem(`*`key, value`*`)`** method must first check if a key/value pair with the given *key* already exists in the list associated with the object.

If it does not, then a new key/value pair must be added to the list, with the given *key* and *value*, such that any current or future `StorageItem` objects referring to this key/value pair will return the value given in the *value* argument. If the script setting the value is running in a secure scripting context, then the key/value pair must be marked as "safe only for secure content", otherwise it must be marked as "safe for both secure and insecure content".

If the given *key does* exist in the list, then, if the key/value pair with the given *key* is accessible, it must have its value updated so that any current or future `StorageItem` objects referring to this key/value pair will return the value given in the *value* argument. If it is *not* accessible, the method must raise a security exception.

When the `setItem()` method is successfully invoked (i.e. when it doesn't raise an exception), events are fired on other `HTMLDocument` objects that can access the newly stored data, as defined in the sections on the `sessionStorage` and `globalStorage` attributes.

The **`removeItem(`*`key`*`)`** method must cause the key/value pair with the given *key* to be removed from the list associated with the object, if it exists and is accessible. If no item with that key exists, the method must do nothing. If an item with that key exists but is not accessible, the method must raise a security exception.

The `setItem()` and `removeItem()` methods must be atomic with respect to failure. That is,

changes to the data storage area must either be successful, or the data storage area must not be changed at all.

In the ECMAScript DOM binding, enumerating a `Storage` object must enumerate through the currently stored and accessible keys in the list the object is associated with. (It must not enumerate the values or the actual members of the interface). In the ECMAScript DOM binding, `Storage` objects must support dereferencing such that getting a property that is not a member of the object (i.e. is neither a member of the `Storage` interface nor of `Object`) must invoke the `getItem()` method with the property's name as the argument, and setting such a property must invoke the `setItem()` method with the property's name as the first argument and the given value as the second argument.

5.4.3. The `StorageItem` interface

Items in `Storage` objects are represented by objects implementing the `StorageItem` interface.

```
interface StorageItem {
           attribute boolean secure;
           attribute DOMString value;
};
```

In the ECMAScript DOM binding, `StorageItem` objects must stringify to their `value` attribute's value.

The **`value`** attribute must return the current value of the key/value pair represented by the object. When the attribute is set, the user agent must invoke the `setItem()` method of the `Storage` object that the `StorageItem` object is associated with, with the key that the `StorageItem` object is associated with as the first argument, and the new given value of the attribute as the second argument.

`StorageItem` objects must be *live*, meaning that as the underlying `Storage` object has its key/value pairs updated, the `StorageItem` objects must always return the actual value of the key/value pair they represent.

If the key/value pair has been deleted, the `StorageItem` object must act as if its value was the empty string. On setting, the key/value pair will be recreated.

The **`secure`** attribute must raise an `INVALID_ACCESS_ERR` exception when accessed or set from a script whose script context is not considered secure. (Basically, if the page is not an SSL page.)

If the scripting context *is* secure, then the `secure` attribute must return true if the key/value pair is considered "safe only for secure content", and false if it is considered "safe for both secure and insecure content". If it is set to true, then the key/value pair must be flagged as "safe only for secure content". If it is set to false, then the key/value pair must be flagged as "safe for both secure and insecure content".

If a `StorageItem` object is obtained by a script that is not running in a secure scripting context, and the item is then marked with the "safe only for secure content" flag by a script that *is* running in a secure context, the `StorageItem` object must continue to be available to the first script, who will be able to read the value of the object. However, any attempt to *set* the value would then start raising exceptions as described in the previous section, and the key/value pair would no longer appear in the appropriate `Storage` object.

5.4.4. The `sessionStorage` attribute

The **`sessionStorage`** attribute represents the storage area specific to the current top-level browsing context.

Each top-level browsing context has a unique set of session storage areas, one for each domain.

User agents should not expire data from a browsing context's session storage areas, but may do so when the user requests that such data be deleted, or when the UA detects that it has limited storage space, or for security reasons. User agents should always avoid deleting data while a script that could access that data is running. When a top-level browsing context is destroyed (and therefore permanently inaccessible to the user) the data stored in its session storage areas can be discarded with it, as the API described in this specification provides no way for that data to ever be subsequently retrieved.

> ***Note: The lifetime of a browsing context can be unrelated to the lifetime of the actual user agent process itself, as the user agent may support resuming sessions after a restart.***

When a new `HTMLDocument` is created, the user agent must check to see if the document's top-level browsing context has allocated a session storage area for that document's domain. If it has not, a new storage area for that document's domain must be created.

The `Storage` object for the document's associated `WindowHTML` object's `sessionStorage` attribute must then be associated with the domain's session storage area.

When a new top-level browsing context is created by cloning an existing browsing context, the new browsing context must start with the same session storage areas as the original, but the two sets must from that point on be considered separate, not affecting each other in any way.

When a new top-level browsing context is created by a script in an existing browsing context, or by the user following a link in an existing browsing context, or in some other way related to a specific `HTMLDocument`, then, if the new context's first `HTMLDocument` has the same domain as the `HTMLDocument` from which the new context was created, the new browsing context must start with a single session storage area. That storage area must be a copy of that domain's session storage area in the original browsing context, which from that point on must be considered separate, with the two storage areas not affecting each other in any way.

When the `setItem()` method is called on a `Storage` object *x* that is associated with a session storage area, then, if the method does not raise a security exception, in every `HTMLDocument` object whose `WindowHTML` object's `sessionStorage` attribute's `Storage` object is associated with the same storage area, other than *x*, a `storage` event must be fired, as described below.

5.4.5. The `globalStorage` attribute

```
interface StorageList {
  Storage namedItem(in DOMString domain);
};
```

The **`globalStorage`** object provides a `Storage` object for each domain.

In the ECMAScript DOM binding, `StorageList` objects must support dereferencing such that getting a property that is not a member of the object (i.e. is neither a member of the `StorageList` interface nor of `Object`) must invoke the `namedItem()` method with the property's name as the argument.

User agents must have a set of global storage areas, one for each domain.

User agents should only expire data from the global storage areas for security reasons or when requested to do so by the user. User agents should always avoid deleting data while a script that could access that data is running. Data stored in global storage areas should be considered potentially user-critical. It is expected that Web applications will use the global storage areas for storing user-written documents.

The **`namedItem(`*`domain`*`)`** method tries to returns a `Storage` object associated with the given domain, according to the rules that follow.

The *domain* must first be split into an array of strings, by splitting the string at "." characters (U+002E FULL STOP). If the *domain* argument is the empty string, then the array is empty as well. If the *domain* argument is not empty but has no dots, then the array has one item, which is equal to the *domain* argument. If the *domain* argument contains consecutive dots, there will be empty strings in the array (e.g. the string "hello..world" becomes split into the three strings "hello", "", and "world", with the middle one being the empty string).

Each component of the array must then have the IDNA ToASCII algorithm applied to it, with both the AllowUnassigned and UseSTD3ASCIIRules flags set. [RFC3490] 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 the user agent must raise a `SYNTAX_ERR` exception. [DOM3CORE] The components after this step consist of only US-ASCII characters.

The components of the array must then be converted to lowercase. Since only US-ASCII is involved at this step, this only requires converting characters in the range A-Z to the corresponding characters in the range a-z.

The resulting array is used in a comparison with another array, as described below. In addition, its components are concatenated together, each part separated by a dot (U+002E), to form the **normalised requested domain**.

> If the original *domain* was "Åsgård.Example.Com", then the resulting array would have the three items "xn--sgrd-poac", "example", and "com", and the normalised requested domain would be "xn--sgrd-poac.example.com".

Next, the script's own domain is processed to find if it is allowed to access the requested domain.

If the script's domain name in not known, e.g. if only the server's IP address is known, and the normalised requested domain is not the empty string, then the user agent must raise a security exception.

> ***Note: If the normalised requested domain is the empty string, then the rest of this algorithm can be skipped. This is because in that situation, the comparison of the two arrays below will always find them to be the same — the first array in such a situation is also empty and so permission to access that storage area will always be given.***

If the script's domain contains no dots (U+002E) then the string "`.localdomain`" must be appended to the script's domain.

Then, the script's domain must be turned into an array, being split, converted to ASCII, and lowercased as described for the *domain* argument above.

Of the two arrays, the longest one must then be shortened to the length of the shorter one, by

dropping items from the start of the array.

> If the *domain* argument is "www.example.com" and the script's domain is "example.com" then the first array will be a three item array ("www", "example", "com"), and the second will be a two item array ("example", "com"). The first array is therefore shortened, dropping the leading parts, making both into the same array ("example", "com").

If the two arrays are not component-for-component identical in literal string comparisons, then the user agent must then raise a security exception.

Otherwise, the user agent must check to see if it has allocated global storage area for the normalised requested domain. If it has not, a new storage area for that domain must be created.

The user agent must then create a `Storage` object associated with that domain's global storage area, and return it.

When the requested *domain* is a top level domain, or the empty string, or a country-specific sub-domain like "co.uk" or "ca.us", the associated global storage area is known as **public storage area**

The `setItem()` method might be called on a `Storage` object that is associated with a global storage area for a domain *d*, created by a `StorageList` object associated with a `WindowHTML` object *x*. Whenever this occurs, if the method didn't raise an exception, a `storage` event must be fired, as described below, in every `HTMLDocument` object that matches the following conditions:

- Its `WindowHTML` object is not *x*, and
- Its `WindowHTML` object's `globalStorage` attribute's `StorageList` object's `namedItem()` method would not raise a security exception according to the rules above if it was invoked with the domain *d*.

In other words, every other document that has access to that domain's global storage area is notified of the change.

5.4.6. The `storage` event

The **`storage`** event is fired in an `HTMLDocument` when a storage area changes, as described in the previous two sections (for session storage, for global storage).

When this happens, a `storage` event in the `http://www.w3.org/2001/xml-events` namespace, which bubbles, is not cancelable, has no default action, and which uses the `StorageEvent` interface described below, must be fired on the body element.

However, it is possible (indeed, for session storage areas, likely) that the target `HTMLDocument` object is not active at that time. For example, it might not be the current entry in the session history; user agents typically stop scripts from running in pages that are in the history. In such cases, the user agent must instead delay the firing of the event until such time as the `HTMLDocument` object in question becomes active again.

When there are multiple delayed `storage` events for the same `HTMLDocument` object, user agents should coalesce events with the same `domain` value (dropping duplicates).

If the DOM of a page that has delayed `storage` events queued up is discarded, then the delayed events are dropped as well.

```
interface StorageEvent : Event {
```

```
  readonly attribute DOMString domain;
  void initStorageEvent(in DOMString typeArg, in boolean canBubbleArg, i
  void initStorageEventNS(in DOMString namespaceURIArg, in DOMString typ
};
```

The **`initStorageEvent()`** and **`initStorageEventNS()`** methods must initialise the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS]

The **`domain`** attribute of the `StorageEvent` event object must be set to the name of the domain associated with the storage area that changed if that storage area is a global storage area, or the string "`#session`" if it was a session storage area.

5.4.7. Miscellaneous implementation requirements for storage areas

5.4.7.1. Disk space

User agents should limit the total amount of space allowed for a domain based on the domain of the page setting the value.

User agents should not limit the total amount of space allowed on a per-storage-area basis, otherwise a site could just store data in any number of subdomains, e.g. storing up to the limit in a1.example.com, a2.example.com, a3.example.com, etc, circumventing per-domain limits.

User agents should consider additional quota mechanisms (for example limiting the amount of space provided to a domain's subdomains as a group) so that hostile authors can't run scripts from multiple subdomains all adding data to the global storage area in an attempted denial-of-service attack.

User agents may prompt the user when per-domain space quotas are reached, allowing the user to grant a site more space. This enables sites to store many user-created documents on the user's computer, for instance.

User agents should allow users to see how much space each domain is using.

If the storage area space limit is reached during a `setItem()` call, the user agent should raise an exception.

A mostly arbitrary limit of five megabytes per domain is recommended. Implementation feedback is welcome and will be used to update this suggestion in future.

5.4.7.2. Threads

Multiple browsing contexts must be able to access the global storage areas simultaneously in a predictable manner. Scripts must not be able to detect any concurrent script execution.

This is required to guarentee that the `length` attribute of a `Storage` object never changes while a script is executing, other than in a way that is predictable by the script itself.

There are various ways of implementing this requirement. One is that if a script running in one browsing context accesses a global storage area, the UA blocks scripts in other browsing contexts when they try to access *any* global storage area until the first script has executed to completion. (Similarly, when a script in one browsing context accesses its session storage area, any scripts that have the same top level browsing context and the same domain would block when accessing their session storage area until the first script has executed to completion.) Another (potentially more efficient but probably more complex) implementation strategy is to use optimistic transactional script execution. This specification does not require any particular implementation strategy, so long as the

requirement above is met.

5.4.8. Security and privacy

5.4.8.1. User tracking

A third-party advertiser (or any entity capable of getting content distributed to multiple sites) could use a unique identifier stored in its domain's global storage area to track a user across multiple sessions, building a profile of the user's interests to allow for highly targetted advertising. In conjunction with a site that is aware of the user's real identity (for example an e-commerce site that requires authenticated credentials), this could allow oppressive groups to target individuals with greater accuracy than in a world with purely anonymous Web usage.

The `globalStorage` object also introduces a way for sites to cooperate to track users over multiple domains, by storing identifying data in "public" top-level domain storage area, accessible by any domain.

There are a number of techniques that can be used to mitigate the risk of user tracking:

- Blocking third-party storage: user agents may restrict access to the `globalStorage` object to scripts originating at the domain of the top-level document of the browsing context.

  This blocks a third-party site from using its private storage area for tracking a user, but top-level sites could still cooperate with third parties to perferm user tracking by using the "public" storage area.

- Expiring stored data: user agents may automatically delete stored data after a period of time.

  For example, a user agent could treat the global storage area as session-only storage, deleting the data once the user had closed all the browsing contexts that could access it.

  This can restrict the ability of a site to track a user, as the site would then only be able to track the user across multiple sessions when he authenticates with the site itself (e.g. by making a purchase or logging in to a service).

- Blocking access to the top-level domain ("public") storage areas: user agents may prevent domains from storing data in and reading data from the top-level domain entries in the `globalStorage` object.

  In practice this requires a detailed list of all the "public" second-level (and third-level) domains. For example, content at the domain `www.example.com` would be allowed to access `example.com` data but not `com` data; content at the domain `example.co.uk` would be allowed access to `example.co.uk` but not `co.uk` or `uk`; and content at `example.chiyoda.tokyo.jp` would be allowed access to `example.chiyoda.tokyo.jp` but not `chiyoda.tokyo.jp`, `tokyo.jp`, or `jp`, while content at `example.metro.tokyo.jp` would be allowed access to both `example.metro.tokyo.jp` and `metro.tokyo.jp` but not `tokyo.jp` or `jp`. The problem is even more convoluted when one considers private domains with third-party subdomains such as `dyndns.org` or `uk.com`.

  Blocking access to the "public" storage areas can also prevent innocent sites from cooperating to provide services beneficial to the user.

- Treating persistent storage as cookies: user agents may present the persistent storage feature to the user in a way that does not distinguish it from HTTP session cookies.

[RFC2965]

This might encourage users to view persistent storage with healthy suspicion.

- Site-specific white-listing of access to "public" storage area: user agents may allow sites to access persistent storage for their own domain and subdomains in an unrestricted manner, but require the user to authorise access to the storage area of higher-level domains.

  For example, code at `example.com` would be always allowed to read and write data for `www.example.com` and `example.com`, but if it tried to access `com`, the user agent could display a non-modal message informing the user that the page requested access to `com` and offering to allow it.

- Origin-tracking of persistent storage data: user agents may record the domain of the script that caused data to be stored.

  If this information is then used to present the view of data currently in persistent storage, it would allow the user to make informed decisions about which parts of the persistent storage to prune. Combined with a blacklist ("delete this data and prevent this domain from ever storing data again"), the user can restrict the use of persistent storage to sites that he trusts.

- Shared blacklists: user agents may allow users to share their persistent storage domain blacklists.

  This would allow communities to act together to protect their privacy.

While these suggestions prevent trivial use of this API for user tracking, they do not block it altogether. Within a single domain, a site can continue to track the user across multiple sessions, and can then pass all this information to the third party along with any identifying information (names, credit card numbers, addresses) obtained by the site. If a third party cooperates with multiple sites to obtain such information, a profile can still be created.

However, user tracking is to some extent possible even with no cooperation from the user agent whatsoever, for instance by using session identifiers in URIs, a technique already commonly used for innocuous purposes but easily repurposed for user tracking (even retroactively). This information can then be shared with other sites, using using visitors' IP addresses and other user-specific data (e.g. user-agent headers and configuration settings) to combine separate sessions into coherent user profiles.

5.4.8.2. Cookie resurrection

If the user interface for persistent storage presents data in the persistent storage feature separately from data in HTTP session cookies, then users are likely to delete data in one and not the other. This would allow sites to use the two features as redundant backup for each other, defeating a user's attempts to protect his privacy.

5.4.8.3. Integrity of "public" storage areas

Since the "public" global storage areas are accessible by content from many different parties, it is possible for third-party sites to delete or change information stored in those areas in ways that the originating sites may not expect.

Authors must not use the "public" global storage areas for storing sensitive data. Authors must not trust information stored in "public" global storage areas.

5.4.8.4. Cross-protocol and cross-port attacks

This API makes no distinction between content served over HTTP, FTP, or other host-based protocols, and does not distinguish between content served from different ports at the same host.

Thus, for example, data stored in the global persistent storage for domain "www.example.com" by a page served from HTTP port 80 will be available to a page served in `http://example.com:18080/`, even if the latter is an experimental server under the control of a different user.

Since the data is not sent over the wire by the user agent, this is not a security risk in its own right. However, authors must take proper steps to ensure that all hosts that have fully qualified host names that are subsets of hosts dealing with sensitive information are as secure as the originating hosts themselves.

Similarly, authors must ensure that all Web servers on a host, regardless of the port, are equally trusted if any of them are to use persistent storage. For instance, if a Web server runs a production service that makes use of the persistent storage feature, then other users that have access to that machine and that can run a Web server on another port will be able to access the persistent storage added by the production service (assuming they can trick a user into visiting their page).

However, if one is able to trick users into visiting a Web server with the same host name but on a different port as a production service used by these users, then one could just as easily fake the look of the site and thus trick users into authenticating with the fake site directly, forwarding the request to the real site and stealing the credentials in the process. Thus, the persistent storage feature is considered to only minimally increase the risk involved.

5.4.8.5. DNS spoofing attacks

Because of the potential for DNS spoofing attacks, one cannot guarentee that a host claiming to be in a certain domain really is from that domain. The `secure` attribute is provided to mark certain key/value pairs as only being accessible to pages that have been authenticated using secure certificates (or similar mechanisms).

Authors must ensure that they do not mark sensitive items as "safe for both secure and insecure content". (To prevent the risk of a race condition, data stored by scripts in secure contexts default to being marked as "safe only for secure content".)

5.4.8.6. Cross-directory attacks

Different authors sharing one host name, for example users hosting content on `geocities.com`, all share one persistent storage object. There is no feature to restrict the access by pathname. Authors on shared hosts are therefore recommended to avoid using the persistent storage feature, as it would be trivial for other authors to read from and write to the same storage area.

> ***Note: Even if a path-restriction feature was made available, the usual DOM scripting security model would make it trivial to bypass this protection and access the data from any path.***

5.4.8.7. Public storage areas corresponding to hosts

If a "public" global storage area corresponds to a host, as it typically does if for private domains with third-party subdomains such as dyndns.org or uk.com, the host corresponding to the "public" domain has access to all the storage areas of its third-party subdomains. In general, authors are discouraged from using the `globalStorage` API for sensitive data unless the operators of all the domains involved are trusted.

User agents may mitigate this problem by preventing hosts corresponding to "public" global storage areas from accessing any storage areas other than their own.

5.4.8.8. Storage areas in the face of untrusted higher-level domains that do not correspond to public storage areas

Authors should not store sensitive data using the global storage APIs if there are hosts with fully-qualified domain names that are subsets of their own which they do not trust. For example, an author at `finance.members.example.net` should not store sensitive financial user data in the `finance.members.example.net` storage area if he does not trust the host that runs `example.net`.

5.4.8.9. Storage areas in the face of untrusted subdomains

If an author publishing content on one host, e.g. `example.com`, wishes to use the `globalStorage` API but does not wish any content on the host's subdomains to access the data, the author should use an otherwise non-existent subdomain name, e.g., `private.example.com`, to store the data. This will be accessible only to that host (and its parent domains), and not to any of the real subdomains (e.g. `upload.example.com`).

5.4.8.10. Implementation risks

The two primary risks when implementing this persistent storage feature are letting hostile sites read information from other domains, and letting hostile sites write information that is then read from other domains.

Letting third-party sites read data that is not supposed to be read from their domain causes *information leakage*, For example, a user's shopping wishlist on one domain could be used by another domain for targetted advertising; or a user's work-in-progress confidential documents stored by a word-processing site could be examined by the site of a competing company.

Letting third-party sites write data to the storage areas of other domains can result in *information spoofing*, which is equally dangerous. For example, a hostile site could add items to a user's wishlist; or a hostile site could set a user's session identifier to a known ID that the hostile site can then use to track the user's actions on the victim site.

A risk is also presented by servers on local domains having host names matching top-level domain names, for instance having a host called "com" or "net". Such hosts might, if implementations fail to correctly implement the `.localdomain` suffixing, have full access to all the data stored in a UA's persistent storage for that top level domain.

Thus, strictly following the model described in this specification is important for user security.

In addition, a number of optional restrictions related to the "public" global storage areas are suggested in the previous sections. The design of this API is intended to be such that not supporting these restrictions, or supporting them less than perfectly, does not result in critical security problems. However, implementations are still encouraged to create and maintain a list of "public" domains, and apply the restrictions described above.

5.5. Sound

The `Audio` interface allows scripts to play sound clips. This interface is intended for sound effects, not for streaming audio or multimedia; for the latter, the `object` element is more appropriate.

There is no markup element that corresponds to `Audio` objects, they are only accessible from script.

User agents should allow users to dynamically enable and disable sound output, but doing so must not affect how `Audio` objects act in any way other than whether sounds are physically played back or not. For instance, sound files must still be downloaded, `load` and `error` events must still fire, and if two identical clips are started with a two second interval then when the sound is reenabled they must still be two seconds out of sync.

When multiple sounds are played simultaneously, the user agent must mix the sounds together.

```
interface Audio {
           attribute EventListener onload;
           attribute EventListener onerror;
  void play();
  void loop();
  void loop(in unsigned long playCount);
  void stop();
};
```

`Audio` objects must also implement the `EventTarget` interface. [DOM3EVENTS]

In ECMAScript, an instance of `Audio` can be created using the `Audio(`*`uri`*`)` constructor:

```
var a = new Audio("test.wav");
```

The **`Audio()` constructor** takes a single argument, a URI (or IRI), which is resolved using the script context's `window.location.href` value as the base, and which returns an `Audio` object that will, at the completion of the current script, start loading that URI.

Once the URI is loaded, a `load` event must be fired on the `Audio` object.

`Audio` objects have a current position and a play count. Both are initially zero.

The `Audio` interface has the following members:

onload

An event listener that is invoked along with any other appropriate event listeners that are registered on this object when a `load` event is fired on it.

play()

Begins playing the sound at the current position, setting the play count to 1.

loop()

Begins playing the sound at the current position, setting the play count to infinity.

loop(*playCount*)

Begins playing the sound at the current position, setting the play count to *playCount*.

stop()

Stops playing the clip and resets the current position and play count to zero.

When playback of the sound reaches the end of the available data, its current position is reset to the start of the clip, and the play count is decreased by one (unless it is infinite). If the play count is greater than zero, then the sound is played again.

6. Editing

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 `contenteditable` attribute

The **`contenteditable`** attribute is a common attribute. User agents must support this attribute on all HTML elements.

redefine this in terms of a microsyntax

If an HTML element has a `contenteditable` attribute set to the empty string or the value `true` (by case-insensitive match), or if its nearest ancestor with the `contenteditable` attribute set has its attribute set to the empty string or the value `true` (by case-insensitive match), then the UA must treat the element as **editable** (as described below).

If an HTML element has a `contenteditable` attribute set but the value of the attribute is not the empty string or the literal value `true`, or if its nearest ancestor with the `contenteditable` attribute set is not *editable*, or if it has no ancestor with the `contenteditable` attribute set, then the element is not editable.

Authors must only use the exact literal values `true` and `false` with the `contenteditable` attribute.

The **`contentEditable`** DOM attribute...

If an element is editable and its parent element is not, 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.

6.2.1. 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 `mouseydown` events.

Change the selection

User agents must allow users to change the selection 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 `mouseydown` 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 inline-level content is not allowed (e.g. because the element accepts "both block-level and inline-level content but not both", and the element already contains block-level content), then the user agent must not insert the text directly at the caret position. In such cases the behaviour 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 only contain block-level content.

> 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 block 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 behaviour is UA-dependent, but user agents must not, in response to a request to break a block, 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 block, for example as in 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 inline-level content is not allowed (e.g. because the element accepts "both block-level and inline-level content but not both", and the element already contains block-level content), then the user agent must not insert the `br` element directly at the caret position. In such cases the behaviour 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, 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 behaviour 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 a way for the user to mark text as having stress emphasis and as being important, and may offer the user the ability to mark text and blocks with other semantics.

UAs should similarly offer a way for the user to insert empty semantic elements (such as, again, `em`, `strong`, and others) 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.

The exact behaviour 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 attributs 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 block at that position.

6.2.2. Making entire documents editable

This section will define document.**`designMode`**.

6.3. 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).

6.3.1. The `DragEvent` and `DataTransfer` interfaces

The drag-and-drop processing model involves several events. They all use the `DragEvent` interface.

```
interface DragEvent : Event {
  readonly attribute DataTransfer dataTransfer;
  void initDragEvent(in DOMString typeArg, in boolean canBubbleArg, in b
  void initDragEventNS(in DOMString namespaceURIArg, in DOMString typeAr
};
```

The **`initDragEvent()`** and **`initDragEventNS()`** methods must initialise the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS]

The **`dataTransfer`** attribute of the `DragEvent` interface represents the context information for the event.

When a `DragEvent` object is created, a new `DataTransfer` object must be created and assigned to the `dataTransfer` context information field of the event object.

```
interface DataTransfer {
           attribute DOMString dropEffect;
           attribute DOMString effectAllowed;
  void clearData(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 conceptually contain various kinds of data.

When a DragEvent event object is initialised, the DataTransfer object created for the event's dataTransfer member must be initialised 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 initialise 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.

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.

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, labelled 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 URI from the list must be returned. If there is no data with that format, or if there is but it has no URIs, 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.

6.3.2. 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 behaviours given in the table below, and must have the context information set up as described after the table.

Event Name	Target	Bubbles?	Cancelable?	`dataTransfer`	`effectAllowed`	`dropEffec`
`dragstart`	Source node	✓ Bubbles	✓ Cancelable	Contains source node unless a selection is being dragged, in which case it is empty	`uninitialized`	`none`
`drag`	Source node	✓ Bubbles	✓ Cancelable	Empty	Same as last event	`none`
`dragenter`	Immediate user selection or the body element	✓ Bubbles	✓ Cancelable	Empty	Same as last event	Based on `effectAllo` value
`dragleave`	Previous target element	✓ Bubbles	—	Empty	Same as last event	`none`
`dragover`	Current target element	✓ Bubbles	✓ Cancelable	Empty	Same as last event	Based on `effectAllo` value
`drop`	Current target element	✓ Bubbles	✓ Cancelable	`getData()` returns data set in `dragstart` event	Same as last event	Current drag operation
`dragend`	Source node	✓ Bubbles	—	Empty	Same as last event	Current drag operation

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`, `dragleave`, and `dragend` events (except when stated otherwise in the algorithms given in the sections below), to the value corresponding to the current drag operation for `drop` 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`

6.3.3. 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.

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 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**
`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 cancelled, then this will be the last iteration. The user agent must follow 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 with `type="text"`)**

            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 with `type="text"`), 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 with `type="text"`), 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.

6.3.3.1. 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.

6.3.3.2. 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.

6.3.4. The `draggable` attribute

All elements may have the `draggable` content attribute set. If the attribute is set, it must be set either to the empty string, to the value `true`, or to the value `false`.

redefine this in terms of a microsyntax

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 has the `draggable` content attribute set to the empty string or to the literal value `true`, the `draggable` DOM attribute must return true.

Otherwise, if an element has the `draggable` content attribute set to the literal value `false`, the `draggable` DOM attribute must return false.

Otherwise, 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`.

6.3.5. 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 only consider a drop to be successful 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.

6.4. Undo history

There has got to be a better way of doing this, surely.

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).

6.4.1. 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 {
  unsigned long add(in DOMObject data, in DOMStrong title);
  void remove(in unsigned long index);
  void clearUndo();
  void clearRedo();
  DOMObject item(in unsigned long index);
  readonly attribute unsigned long length;
  readonly attribute unsigned long position;
};
```

The **`undoManager`** attribute of the `WindowHTML` interface must return the object implementing the `UndoManager` interface for that `WindowHTML` object's associated `HTMLDocument` object.

In the ECMAScript DOM binding, objects implementing this interface must also support being dereferenced using the square bracket notation, such that dereferencing with an integer index is equivalent to invoking the `item()` method with that index (e.g. `undoManager[1]` returns the same as `undoManager.item(1)`).

`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.

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 behaviour 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.

6.4.2. 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.

6.4.3. 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.

6.4.4. The `UndoManagerEvent` interface and the `undo` and `redo` events

```
interface UndoManagerEvent : Event {
  readonly attribute DOMObject data;
  void initUndoManagerEvent(in DOMString typeArg, in boolean canBubbleAr
  void initUndoManagerEventNS(in DOMString namespaceURIArg, in DOMStrinc
};
```

The **`initUndoManagerEvent()`** and **`initUndoManagerEventNS()`** methods must initialise 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.

6.4.5. 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 offical (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.

6.5. Command APIs

The **`execCommand(`*`commandID, doShowUI, 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 *doShowUI* and *value* parameters, even if specified, are ignored unless otherwise stated.

> ***Note: In this specification, in fact, the doShowUI parameter is always ignored, regardless of its value. It is included for historical reasons only.***

When any of these methods are invoked, user agents must act as described in the list below.

For actions marked "**editing hosts only**", if the selection is not entirely within an editing host, of if there is no selection and the caret is not inside an editing host, then the user agent must do nothing.

If the *commandID* is `undo`

The user agent must move back one step in its undo transaction history, restoring the associated state. If there is no further undo information the user agent must do nothing. See the undo history.

If the *commandID* is `redo`

The user agent must move forward one step in its undo transaction history, restoring the associated state. If there is no further undo (well, "redo") information the user agent must do nothing. See the undo history.

If the *commandID* is `selectAll`

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.

If the *commandID* is `unselect`

The user agent must change the selection so that nothing is selected.

> We need some sort of way in which the user can make a selection without risk of script clobbering it.

If the *commandID* is `superscript`

Editing hosts only. 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 behaviour is UA-defined).

If the *commandID* is `subscript`

Editing hosts only. 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).

If the *commandID* is `formatBlock`

Editing hosts only. This command changes the semantics of the blocks containing the selection.

If there is no selection, then, where in the description below refers to the selection, the user agent must act as if the selection was an empty range at the caret position.

If the *value* parameter is not specified or has a value other than one of the following literal strings:

- `<address>`
- `<aside>`
- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`
- `<nav>`
- `<p>`
- `<pre>`

...then the user agent must do nothing.

Otherwise, the user agent must, for every position in the selection, take the furthest block-level element ancestor of that position that contains only inline-level content and is not being used as a structured inline-level element, and, if that element is a descendant of the editing host, rename it according to the *value*, by stripping the leading < character and the trailing > character and using the rest as the new tag name.

If the *commandID* is `delete`

Editing hosts only. The user agent must act as if the user had performed a backspace operation.

If the *commandID* is `forwardDelete`

Editing hosts only. The user agent must act as if the user had performed a forward delete operation.

If the *commandID* is `insertLineBreak`

Editing hosts only. The user agent must act as if the user had requested a line separator.

If the *commandID* is `insertParagraph`

Editing hosts only. The user agent must act as if the user had performed a break block editing action.

If the *commandID* is `insertText`

Editing hosts only. The user agent must act as if the user had inserted text corresponding to the *value* parameter.

If the *commandID* is *`vendorID-customCommandID`*

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.

If the *commandID* is something else

User agents must do nothing.

6.6. The text selection APIs

Every browsing context has **a selection**. The selection may be empty, and the selection may have more than one range (a disjointed selection). The user should be able 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**.

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

The **`getSelection()`** method on the `WindowHTML` interface must return the `Selection` object representing the selection of that `WindowHTML` object's browsing context.

For historical reasons, the **`getSelection()`** method on the `HTMLDocument` interface must return the same `Selection` object.

```
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();
  DOMString toString();
};
```

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.

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 `ownerDocument` 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 `toString()` 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).

The **`toString()`** method must return 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 language bindings where this is supported, objects implementing the `Selection` interface must stringify to the value returned by the object's `toString()` method.

> In the following document fragment, the emphasised parts indicate the selection.
>
> ```
> <p>The cute girl likes the <cite>Oxford English Dictionary</cite>.</p
> ```
>
> 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

When we define HTMLTextAreaElement and HTMLInputElement we will have to add the IDL given below to both of their IDLs.

The `input` and `textarea` elements define four 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.

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, c
> ```
>
> ...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.

When these methods and attributes are used with `input` elements that are not displaying simple text fields, they must raise an `INVALID_STATE_ERR` exception.

7. Communication

7.1. Server-sent DOM events

This section describes a mechanism for allowing servers to dispatch DOM events into documents that expect it.

7.1.1. The `event-source` element

To specify an event source in an HTML document authors use a new (empty) element `event-source`, with an attribute `src=""` that takes a URI (or IRI) to open as a stream and, if the data found at that URI is of the appropriate type, treat as an event source.

The `event-source` element may also have an `onevent=""` attribute. If present, the attribute must be treated as script representing an event handler registered as non-capture listener of events with name `event` and the namespace `uuid:755e2d2d-a836-4539-83f4-16b51156341f` or null, that are targetted at or bubble through the element.

UAs must also support all the common attributes on the `event-source` element.

7.1.2. The `RemoteEventTarget` interface

Any object that implements the `EventTarget` interface shall also implement the `RemoteEventTarget` interface.

```
interface RemoteEventTarget {
  void addEventSource(in DOMString src);
  void removeEventSource(in DOMString src);
};
```

The **`addEventSource(`*`src`*`)`** method shall register the URI (or IRI) specified in *src* as an event source on the object. The **`removeEventSource(`*`src`*`)`** method shall remove the URI (or IRI) specified in *src* from the list of event sources for that object. If a single URI is added multiple times, each instance must be handled individually. Removing a URI must only remove one instance of that URI. If the specified URI cannot be added or removed, the method must return without doing anything or raising an exception.

7.1.3. Processing model

When an `event-source` element in a document has a `src` attribute set, the UA should fetch the resource indicated by the attribute's value.

Similarly, when the `addEventSource()` method is invoked on an object, the UA should, at the completion of the script's current execution, fetch the resource identified by the method's argument (unless the `removeEventSource()` was called removing the URI from the list first).

When an `event-source` element is removed from the document, or when an event source is removed from the list of event sources for an object using the `removeEventSource()` method, the relevant connection must be closed (and not reopened unless the element is returned to the document or the `addEventSource()` method is called with the same URI again).

Since connections established to remote servers for such resources are expected to be long-lived, UAs should ensure that appropriate buffering is used. In particular, while line buffering may be safe if lines are defined to end with a single U+000A LINE FEED character, block buffering or line buffering with different expected line endings can cause delays in event dispatch.

In general, the semantics of the transport protocol specified by the "src" attribute must be followed. Clients should re-open `event-source` connections that get closed after a short interval (such as 5 seconds), unless they were closed due to problems that aren't expected to be resolved, as described in this section.

DNS errors must be considered fatal, and cause the user agent to not open any connection for the event-source.

HTTP 200 OK responses that have a Content-Type other than `application/x-dom-event-stream` must be ignored and must prevent the user agent from reopening the connection for that event-source. HTTP 200 OK responses with the right MIME type, however, should, when closed, be reopened after a small delay.

Resource with the type `application/x-dom-event-stream` must be processed line by line as described below.

HTTP 201 Created, 202 Accepted, 203 Non-Authoritative Information, and 206 Partial Content responses must be treated like HTTP 200 OK responses for the purposes of reopening event-source connections. They are, however, likely to indicate an error has occurred somewhere and may cause the user agent to emit a warning.

HTTP 204 No Content, and 205 Reset Content responses must be treated as if they were 200 OK responses with the right MIME type but no content, and should therefore cause the user agent to reopen the connection after a short delay.

HTTP 300 Multiple Choices responses should be handled automatically if possible (treating the responses as if they were 302 Moved Permanently responses pointing to the appropriate resource), and otherwise must be treated as HTTP 404 responses.

HTTP 301 Moved Permanently responses must cause the user agent to use the server specified URI instead of the one specified in the event-source's "src" attribute for future connections.

HTTP 302 Found, 303 See Other, and 307 Temporary Redirect responses must cause the user agent to use the server specified URI instead of the one specified in the event-source's "src" attribute for the next connection, but if the user agent needs to reopen the connection at a later point, it must once again start from the "src" attribute (or the last URI given by a 301 Moved Permanently response in complicated cases where such responses are chained).

HTTP 304 Not Modified responses should be handled like HTTP 200 OK responses, with the content coming from the user agent cache. A new connection attempt should then be made after a short wait.

HTTP 305 Use Proxy, HTTP 401 Unauthorized, and 407 Proxy Authentication Required should be treated transparently as for any other subresource.

HTTP 400 Bad Request, 403 Forbidden, 404 Not Found, 405 Method Not Allowed, 406 Not Acceptable, 408 Request Timeout, 409 Conflict, 410 Gone, 411 Length Required, 412 Precondition Failed, 413 Request Entity Too Large, 414 Request-URI Too Long, 415 Unsupported Media Type, 416 Requested Range Not Satisfiable, 417 Expectation Failed, 500 Internal Server Error, 501 Not Implemented, 502 Bad Gateway, 503 Service Unavailable, 504 Gateway Timeout, and 505 HTTP Version Not Supported responses, and any other HTTP response code not listed here, should cause the user agent to stop trying to process this event-source element.

For non-HTTP protocols, UAs should act in equivalent ways.

7.1.4. The event stream format

The event stream MIME type is `application/x-dom-event-stream`.

The event stream must always be encoded as UTF-8. Line must always be terminated by a single U+000A LINE FEED character.

The event stream format is (in pseudo-BNF):

```
<stream>  ::= <event>*
<event>   ::= [ <comment> | <command> | <field> ]* <newline>

<comment> ::= ';' <data> <newline>
<command> ::= ':' <data> <newline>
<field>   ::= <name> [ ':' <space>? <data> ]? <newline>

<name>    ::= one or more UNICODE characters other than ':', ';', and U+(
<data>    ::= zero or more UNICODE characters other than U+000A LINE FEEI
<space>   ::= a single U+0020 SPACE character (' ')
<newline> ::= a single U+000A LINE FEED character
```

Bytes that are not valid UTF-8 sequences must be interpreted as the U+FFFD REPLACEMENT CHARACTER.

The stream is parsed by reading everything line by line, in blocks separated by blank lines (blank lines are those consisting of just a single lone line feed character). Comment lines (those starting with the character ';') and command lines (those starting with the character ':') are ignored. Command lines are reserved for future use and should not be used.

For each non-blank, non-comment line, the field name is first taken. This is everything on the line up to but not including the first colon (':') or the line feed, whichever comes first. Then, if there was a colon, the data for that line is taken. This is everything after the colon, ignoring a single space after the colon if there is one, up to the end of the line. If there was no colon the data is the empty string.

> Examples:
>
> ```
> Field name: Field data
>
> This is a blank field
>
> 1. These two lines: have the same data
> 2. These two lines:have the same data
>
> 1. But these two lines:  do not
> 2. But these two lines: do not
> ```

If a field name occurs multiple times, the data values for those lines are concatenated with a newline between them.

> For example, the following:
>
> ```
> Test: Line 1
> Foo: Bar
> Test: Line 2
> ```
>
> ...is treated as having two fields, one called `Test` with the value `Line 1\nLine 2` (where `\n` represents a newline), and one called `Foo` with the value `Bar`.

Note: Since any random stream of characters matches the above format, there is no need to define any error handling.

7.1.5. Event stream interpretation

Once the fields have been parsed, they are interpreted as follows (these are case-sensitive exact comparisons):

- `Event` is the name of the event. For example, `load`, `DOMActivate`, `updateTicker`. If there is no field with this name, then no event will be synthesised, and the other data will be ignored.

- `Namespace` is the DOM3 namespace for the event. For normal DOM events this would be `http://www.w3.org/2001/xml-events`. If it isn't specified the event namespace is null.

- `Class` is the interface used for the event, for instance `Event`, `UIEvent`, `MutationEvent`, `KeyboardEvent`, etc. For compatibility with DOM3 Events, the values `UIEvents`, `MouseEvents`, `MutationEvents`, and `HTMLEvents` are valid values and must be treated respectively as meaning the interfaces `UIEvent`, `MouseEvent`, `MutationEvent`, and `Event`. (This value can therefore be used as the argument to `createEvent()`.) If the value is not specified it is defaulted based on the event name as follows:

    - If `Namespace` is `http://www.w3.org/2001/xml-events` or null and the `Event` field exactly matches one of the events specified by DOM3 Events in section 1.4.2 "Complete list of event types", then the Class defaults to the interface relevant for that event type. [DOM3EVENTS]

        > For example:
        >
        > ```
        > Event: click
        > ```
        >
        > ...would cause `Class` to be treated as `MouseEvent`.

    - If `Namespace` is `uuid:755e2d2d-a836-4539-83f4-16b51156341f` or null and the `Event` doesn't match any of the known events, then the `RemoteEvent` interface (described below) is used.

    - Otherwise, if the UA doesn't have special knowledge of which class to use for the given event in the given namespace, then the `Event` interface is used.

    It is quite possible to give the wrong class for an event. This is equivalent to creating an event in the DOM using the DOM Event APIs, but using the wrong interface for it.

- `Bubbles` specifies whether the event is to bubble. If it is specified and has the value `No`, the event does not bubble. If it is specified and has any other value (including `no` or `No\n`) then the event bubbles. If it is not specified it is defaulted based on the event name as follows:

    - If `Namespace` is `http://www.w3.org/2001/xml-events` or null and the `Event` field exactly matches one of the events specified by DOM3 Events in section 1.4.2 "Complete list of event types", then whether the event bubbles depends on whether the DOM3 Events spec specifies that that event should bubble or not. [DOM3EVENTS]

        > For example:
        >
        > ```
        > Event: load
        > ```
        >
        > ...would cause `Bubbles` to be treated as `No`.

    - Otherwise, if the UA doesn't have special knowledge of which class to use for the given event in the given namespace, then the event bubbles.

- `Cancelable` specifies whether the event may have its default action prevented. If it is specified and has the value `No`, the event may not have its default action prevented. If it is specified and has any other value (including `no` or `No\n`) then the event may be canceled. If it

is not specified it is defaulted based on the event name as follows:

- If `Namespace` is `http://www.w3.org/2001/xml-events` or null and the `Event` field exactly matches one of the events specified by DOM3 Events in section 1.4.2 "Complete list of event types", then whether the event is cancelable depends on whether the DOM3 Events spec specifies that that event should be cancelable or not. [DOM3EVENTS]

  > For example:
  >
  > ```
  > Event: load
  > ```
  >
  > ...would cause `Cancelable` to be treated as `No`.

- Otherwise, if the UA doesn't have special knowledge of which class to use for the given event in the given namespace, then the event may be canceled.

- `Target` is the element that the event is to be dispatched on. If its value starts with a `#` character then the remainder of the value represents an ID, and the event must be dispatched on the same node as would be obtained by the `getElementById()` method on the ownerDocument of the event-source element responsible for the event being dispatched.

  > For example,
  >
  > ```
  > Target: #test
  > ```
  >
  > ...would target the element with ID `test`.

  If the value does not start with a `#` but has the literal value `Document`, then the event is dispatched at the `ownerDocument` of the `event-source` element responsible for the event being dispatched.

  Otherwise, the event is dispatched at the `event-source` element itself.

- Other fields depend on the interface specified (or possibly implied) by the `Class` field. If the specified interface has an attribute that exactly matches the name of the field, and the value of the field can be converted (using the type conversions defined in ECMAScript) to the type of the attribute, then it must be used. Any attributes (other than the `Event` interface attributes) that do not have matching fields are initialised to zero, null, false, or the empty string.

  > For example:
  >
  > ```
  > ; ...some other fields...
  > Class: MouseEvent
  > button: 2
  > ```
  >
  > ...would result in a MouseEvent event that had `button` set to `2` but `screenX`, `screenY`, etc, set to 0, false, or null as appropriate.

  If a field does not match any of the attributes on the event, it is ignored.

  > For example:
  >
  > ```
  > Event: keypress
  > Class: MouseEvent
  > keyIdentifier: 0
  > ```

> ...would result in a `MouseEvent` event with its fields all at their default values, with the event name being `keypress`. The `ctrlKey` field would be ignored. (If the author had not included the `Class` field explicitly, it would have just worked, since the class would have defaulted as described above.)

Once a blank line is reached, an event of the appropriate type is synthesized and dispatched to the appropriate node as described by the fields above. No event is dispatched until a blank line has been received.

If the `Event` field was omitted, then no event is synthesised and the data is ignored.

> The following stream contains four blocks yet synthesises no events, since none of the blocks have a field called `Event`. (The first block has just a comment, the second block has two fields with names "load" and "Target" respectively, the third block is empty, and the fourth block has two comments.)
>
> ```
> ; test
>
> load
> Target: #image1
>
>
> ; if any real events follow this block, they will not be affected by
> ; the "Target" and "load" fields above.
> ```

7.1.6. The `RemoteEvent` interface

The `RemoteEvent` interface is defined as follows:

```
interface RemoteEvent : Event {
  readonly attribute DOMString data;
  void initRemoteEvent(in DOMString typeArg, in boolean canBubbleArg, ir
  void initRemoteEventNS(in DOMString namespaceURI, in DOMString typeArg
};
```

Events that use the `RemoteEvent` interface never have any default action associated with them.

I guess we should define those members.

7.1.7. Example

> The following event description, once followed by a blank line:
>
> ```
> Event: stock change
> data: YHOO
> data: -2
> data: 10
> ```
>
> ...would cause an event `stock change` with the interface `RemoteEvent` to be dispatched on the `event-source` element, which would then bubble up the DOM, and whose `data` attribute would contain the string `YHOO\n-2\n10` (where `\n` again represents a newline).
>
> This could be used as follows:

```
<event-source src="http://stocks.example.com/ticker.php" id="stock">
<script type="text/javascript">
document.getElementById('stock').addEventListener('stock change',
  function () {
    var data = event.data.split('\n');
    updateStocks(data[0], data[1], data[2]);
  }, false);
</script>
```

...where updateStocks is a function defined as:

```
function updateStocks(symbol, delta, value) { ... }
```

...or some such.

7.2. Network connections

To enable Web applications to communicate with each other in local area networks, and to maintain bidirectional communications with their originating server, this specification introduces the `Connection` interface.

The `WindowHTML` interface provides three constructors for creating `Connection` objects: `TCPConnection()`, for creating a direct (possibly encrypted) link to another node on the Internet using TCP/IP; `LocalBroadcastConnection()`, for creating a connection to any listening peer on a local network (which could be a local TCP/IP subnet using UDP, a Bluetooth PAN, or another kind of network infrastructure); and `PeerToPeerConnection()`, for a direct peer-to-peer connection (which could again be over TCP/IP, Bluetooth, IrDA, or some other type of network).

Note: This interface does not allow for raw access to the underlying network. For example, this interface could not be used to implement an IRC client without proxying messages through a custom server.

7.2.1. Introduction

This section is non-normative.

An introduction to the client-side and server-side of using the direct connection APIs.

An example of a party-line implementation of a broadcast service, and direct peer-to-peer chat for direct local connections.

7.2.2. The `Connection` interface

```
interface Connection {
  readonly attribute DOMString network;
  readonly attribute DOMString peer;
  readonly attribute int readyState;
           attribute EventListener onopen;
           attribute EventListener onread;
           attribute EventListener onclose;
  void send(in DOMString data);
```

```
  void disconnect();
};
```

`Connection` objects must also implement the `EventTarget` interface. [DOM3EVENTS]

When a `Connection` object is created, the UA must try to establish a connection, as described in the sections below describing each connection type.

The **`network`** attribute represents the name of the network connection (the value depends on the kind of connection being established). The **`peer`** attribute identifies the remote host for direct (non-broadcast) connections.

The `network` attribute must be set as soon as the `Connection` object is created, and keeps the same value for the lifetime of the object. The `peer` attribute must initially be set to the empty string and must be updated once, when the connection is established, after which point it must keep the same value for the lifetime of the object.

The **`readyState`** attribute represents the state of the connection. When the object is created it must be set to 0. It can have the following values:

0 Connecting

The connection has not yet been established.

1 Connected

The connection is established and communication is possible.

2 Closed

The connection has been closed.

Once a connection is established, the `readyState` attribute's value must be changed to 1, and the `open` event must be fired on the `Connection` object.

When data is received, the `read` event will be fired on the `Connection` object.

When the connection is closed, the `readyState` attribute's value must be changed to 2, and the `close` event must be fired on the `Connection` object.

The **`onopen`**, **`onread`**, and **`onclose`** attributes must, when set, register their new value as an event listener for their respective events (namely `open`, `read`, and `close`), and unregister their previous value if any.

The **`send()`** method transmits data using the connection. If the connection is not yet established, it must raise an `INVALID_STATE_ERR` exception. If the connection *is* established, then the behaviour depends on the connection type, as described below.

The **`disconnect()`** method must close the connection, if it is open. If the connection is already closed, it must do nothing. Closing the connection causes a `close` event to be fired and the `readyState` attribute's value to change, as described above.

7.2.3. Connection Events

All the events described in this section are events in the `http://www.w3.org/2001/xml-events` namespace, which do not bubble, are not cancelable, and have no default action.

The **`open`** event is fired when the connection is established. UAs must use the normal `Event`

interface when firing this event.

The **close** event is fired when the connection is closed (whether by the author, calling the `disconnect()` method, or by the server, or by a network error). UAs must use the normal `Event` interface when firing this event as well.

> ***Note: No information regarding why the connection was closed is passed to the application in this version of this specification.***

The **read** event is fired when when data is received for a connection. UAs must use the `ConnectionReadEvent` interface for this event.

```
interface ConnectionReadEvent : Event {
  readonly attribute DOMString data;
  readonly attribute DOMString source;
  void initConnectionReadEvent(in DOMString typeArg, in boolean canBubbl
  void initConnectionReadEventNS(in DOMString namespaceURI, in DOMStrinc
};
```

The **data** attribute must contain the data that was transmitted from the peer.

The **source** attribute must contain the name of the peer. This is primarily useful on broadcast connections; on direct connections it is equal to the `peer` attribute on the `Connection` object.

The **initConnectionReadEvent()** and **initConnectionReadEventNS()** methods must initialise the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS]

Events that would be fired during script execution (e.g. between the connection object being created — and thus the connection being established — and the current script completing; or, during the execution of a `read` event handler) must be buffered, and those events queued up and each one individually fired after the script has completed.

7.2.4. TCP connections

The **TCPConnection(*subdomain*, *port*, *secure*)** constructor on the `WindowHTML` interface returns a new object implementing the `Connection` interface, set up for a direct connection to a specified host on the page's domain.

When this constructor is invoked, the following steps must be followed.

First, if the script's domain is not a host name (e.g. it is an IP address) then the UA must raise a security exception.

Then, if the *subdomain* argument is null or the empty string, the target host is the script's domain. Otherwise, the *subdomain* argument is prepended to the script's domain with a dot separating the two strings, and that is the target host.

If either:

- the target host is not a valid host name, or
- the *port* argument is neither equal to 80, nor equal to 443, nor greater than or equal to 1024 and less than or equal to 65535,

...then the UA must raise a security exception.

Otherwise, the user agent must verify that the the string representing the script's domain in IDNA format can be obtained without errors. If it cannot, then the user agent must raise a security exception.

The user agent may also raise a security exception at this time if, for some reason, permission to create a direct TCP connection to the relevant host is denied. Reasons could include the UA being instructed by the user to not allow direct connections, or the UA establishing (for instance using UPnP) that the network topology will cause connections on the specified port to be directed at the wrong host.

If no exceptions are raised by the previous steps, then a new `Connection` object must be created, its `peer` attribute must be set to a string consisting of the name of the target host, a colon (U+003A COLON), and the port number as decimal digits, and its `network` attribute must be set to the same value as the `peer` attribute.

This object must then be returned.

The user agent must then begin trying to establish a connection with the target host and specified port. (This typically would begin in the backgound, while the script continues to execute.)

If the *secure* boolean argument is set to true, then the user agent must establish a secure connection with the target host and specified port using TLS or another protocol, negotiated with the server. [RFC2246] If this fails the user agent must act as if it had closed the connection.

Once a secure connection is established, or if the *secure* boolean argument is not set to true, then the user agent must continue to connect to the server using the protocol described in the section entitled clients connecting over TCP. All data on connections made using TLS must be sent as "application data".

Once the connection is established, the UA must act as described in the section entitled sending and receiving data over TCP.

User agents should allow multiple TCP connections to be established per host. In particular, user agents should not apply per-host HTTP connection limits to connections established with the `TCPConnection` constructor.

7.2.5. Broadcast connections

The **`LocalBroadcastConnection()`** constructor on the `WindowHTML` interface returns a new object implementing the `Connection` interface, set up to broadcast on the local network.

When this constructor is invoked, a new `Connection` object must be created.

The `network` attribute of the object must be set to the string representing the script's domain in IDNA format. If this string cannot be obtained, then the user agent must raise a security exception exception when the constructor is called.

The `peer` attribute must be set to the empty string.

The object must then be returned, unless, for some reason, permission to broadcast on the local network is to be denied. In the latter case, a security exception must be raised instead. User agents may deny such permission for any reason, for example a user preference.

If the object is returned (i.e. if no exception is raised), the user agent must the begin broadcasting

and listening on the local network, in the background, as described below. The user agent may define "the local network" in any way it considers appropriate and safe; for instance the user agent may ask the user which network (e.g. Bluetooth, IrDA, Ethernet, etc) the user would like to broadcast on before beginning broadcasting.

UAs may broadcast and listen on multiple networks at once. For example, the UA could broadcast on both Bluetooth and Wifi at the same time.

As soon as the object is returned, the connection has been established, which implies that the `open` event must be fired. Broadcast connections are never closed.

7.2.5.1. Broadcasting over TCP/IP

Should we drop this altogether? Letting people fill the local network with garbage seems unwise.

We need to register a UDP port for this. For now this spec refers to port 18080/udp.

> ***Note: Since this feature requires that the user agent listen to a particular port, some platforms might prevent more than one user agent per IP address from using this feature at any one time.***

On TCP/IP networks, broadcast connections transmit data using UDP over port 18080.

When the `send(`*`data`*`)` method is invoked on a `Connection` object that was created by the `LocalBroadcastConnection()` constructor, the user agent must follow these steps:

1. Create a string consisting of the value of the `network` attribute of the `Connection` object, a U+0020 SPACE character, a U+0002 START OF TEXT character, and the *data* argument.
2. Encode the string as UTF-8.
3. If the resulting byte stream is longer than 65487 bytes, raise an `INDEX_SIZE_ERR` DOM exception and stop.
4. Create a UDP packet whose data is the byte stream, with the source and destination ports being 18080, and with appropriate length and checksum fields. Transmit this packet to IPv4 address 255.255.255.255 or IPv6 address ff02::1, as appropriate. ***IPv6 applications will also have to enable reception from this address.***

When a broadcast connection is opened on a TCP/IP network, the user agent should listen for UDP packets on port 18080.

When the user agent receives a packet on port 18080, the user agent must attempt to decode that packet's data as UTF-8. If the data is not fully correct UTF-8 (i.e. if there are decoding errors) then the packet must be ignored. Otherwise, the user agent must check to see if the decoded string contains a U+0020 SPACE character. If it does not, then the packet must again be ignored (it might be a peer discovery packet from a `PeerToPeerConnection()` constructor). If it does then the user agent must split the string at the first space character. All the characters before the space are then known as *d*, and all the characters after the space are known as *s*. If *s* is not at least one character long, or if the first character of *s* is not a U+0002 START OF TEXT character, then the packet must be ignored. (This allows for future extension of this protocol.)

Otherwise, for each `Connection` object that was created by the `LocalBroadcastConnection()`

constructor and whose `network` attribute exactly matches *d*, a `read` event must be fired on the `Connection` object. The string *s*, with the first character removed, must be used as the `data`, and the source IP address of the packet as the `source`.

Making the source IP available means that if two or more machines in a private network can be made to go to a hostile page simultaneously, the hostile page can determine the IP addresses used locally (i.e. on the other side of any NAT router). Is there some way we can keep link-local IP addresses secret while still allowing for applications to distinguish between multiple participants?

7.2.5.2. Broadcasting over Bluetooth

Does anyone know enough about Bluetooth to write this section?

7.2.5.3. Broadcasting over IrDA

Does anyone know enough about IrDA to write this section?

7.2.6. Peer-to-peer connections

The **`PeerToPeerConnection()`** constructor on the `WindowHTML` interface returns a new object implementing the `Connection` interface, set up for a direct connection to a user-specified host.

When this constructor is invoked, a new `Connection` object must be created.

The `network` attribute of the object must be set to the string representing the script's domain in IDNA format. If this string cannot be obtained, then the user agent must raise a security exception exception when the constructor is called.

The `peer` attribute must be set to the empty string.

The object must then be returned, unless, for some reason, permission to establish peer-to-peer connections is generally disallowed, for example due to administrator settings. In the latter case, a security exception must be raised instead.

The user agent must then, typically while the script resumes execution, find a remote host to establish a connection to. To do this it must start broadcasting and listening for peer discovery messages and listening for incoming connection requests on all the supported networks. How this is performed depends on the type of network and is described below.

The UA should inform the user of the clients that are detected, and allow the user to select one to connect to. UAs may also allow users to explicit specify hosts that were not detected, e.g. by having the user enter an IP address.

If an incoming connection is detected before the user specifies a target host, the user agent should ask the user to confirm that this is the host they wish to connect to. If it is, the connection should be accepted and the UA will act as the *server* in this connection. (Which UA acts as the server and which acts as the client is not discernible at the DOM API level.)

If no incoming connection is detected and if the user specifies a particular target host, a connection should be established to that host, with the UA acting as the *client* in the connection.

No more than one connection must be established per `Connection` object, so once a connection

has been established, the user agent must stop listening for further connections (unless, or until such time as, another `Connection` object is being created).

If at any point the user cancels the connection process or the remote host refuses the connection, then the user agent must act as if it had closed the connection, and stop trying to connect.

7.2.6.1. Peer-to-peer connections over TCP/IP

Should we replace this section with something that uses Rendez-vous/zeroconf or equivalent?

We need to register ports for this. For now this spec refers to port 18080/udp and 18080/tcp.

Note: Since this feature requires that the user agent listen to a particular port, some platforms might prevent more than one user agent per IP address from using this feature at any one time.

When using TCP/IP, broadcasting peer discovery messages must be done by creating UDP packets every few seconds containing as their data the value of the connection's `network` attribute, encoded as UTF-8, with the source and destination ports being set to 18080 and appropriate length and checksum fields, and sending these packets to address (in IPv4) 255.255.255.255 or (in IPv6) ff02::1, as appropriate.

Listening for peer discovery messages must be done by examining incoming UDP packets on port 18080. ***IPv6 applications will also have to enable reception from the ff02::1 address.*** If their payload is exactly byte-for-byte equal to a UTF-8 encoded version of the value of the connection's `network` attribute, then the source address of that packet represents the address of a host that is ready to accept a peer-to-peer connection, and it should therefore be offered to the user.

Incoming connection requests must be listened for on TCP port 18080. If an incoming connection is received, the UA must act as a *server*, as described in the section entitled servers accepting connections over TCP.

If no incoming connection requests are accepted and the user instead specifies a target host to connect to, the UA acts as a *client*: the user agent must attempt to connect to the user-specified host on port 18080, as described in the section entitled clients connecting over TCP.

Once the connection is established, the UA must act as described in the section entitled sending and receiving data over TCP.

Note: This specification does not include a way to establish secure (encrypted) peer-to-peer connections at this time. ***If you can see a good way to do this, let me know.***

7.2.6.2. Peer-to-peer connections over Bluetooth

Does anyone know enough about Bluetooth to write this section?

7.2.6.3. Peer-to-peer connections over IrDA

Does anyone know enough about IrDA to write this section?

7.2.7. The common protocol for TCP-based connections

The same protocol is used for `TCPConnection` and `PeerToPeerConnection` connection types. This section describes how such connections are established from the client and server sides, and then describes how data is sent and received over such connections (which is the same for both clients and servers).

7.2.7.1. Clients connecting over TCP

This section defines the client-side requirements of the protocol used by the `TCPConnection` and `PeerToPeerConnection` connection types.

If a TCP connection to the specified target host and port cannot be established, for example because the target host is a domain name that cannot be resolved to an IP address, or because packets cannot be routed to the host, the user agent should retry creating the connection. If the user agent gives up trying to connect, the user agent must act as if it had closed the connection.

> ***Note: No information regarding the state of the connection is passed to the application while the connection is being established in this version of this specification.***

Once a TCP/IP connection to the remote host is established, the user agent must transmit the following sequence of bytes, represented here in hexadecimal form:

```
0x48 0x65 0x6C 0x6C 0x6F 0x0A
```

> ***Note: This represents the string "Hello" followed by a newline, encoded in UTF-8.***

The user agent must then read all the bytes sent from the remote host, up to the first 0x0A byte (inclusive). That string of bytes is then compared byte-for-byte to the following string of bytes:

```
0x57 0x65 0x6C 0x63 0x6F 0x6E 0x65 0x0A
```

> ***Note: This says "Welcome".***

If the server sent back a string in any way different to this, then the user agent must close the connection and give up trying to connect.

Otherwise, the user agent must then take the string representing the script's domain in IDNA format, encode it as UTF-8, and send that to the remote host, followed by a 0x0A byte (a U+000A LINE FEED in UTF-8).

The user agent must then read all the bytes sent from the remote host, up to the first 0x0A byte (inclusive). That string of bytes must then be compared byte-for-byte to the string that was just sent to the server (the one with the IDNA domain name and ending with a newline character). If the server sent back a string in any way different to this, then the user agent must close the connection and give up trying to connect.

Otherwise, the connection has been established (and events and so forth get fired, as described above).

If at any point during this process the connection is closed prematurely, then the user agent must close the connection and give up trying to connect.

7.2.7.2. Servers accepting connections over TCP

This section defines the server side of the protocol described in the previous section. For authors, it should be used as a guide for how to implement servers that can communicate with Web pages over TCP. For UAs these are the requirements for the server part of `PeerToPeerConnection`s.

Once a TCP/IP connection from a remote host is established, the user agent must transmit the following sequence of bytes, represented here in hexadecimal form:

```
0x57 0x65 0x6C 0x63 0x6F 0x6E 0x65 0x0A
```

Note: This says "Welcome" and a newline in UTF-8.

The user agent must then read all the bytes sent from the remote host, up to the first 0x0A byte (inclusive). That string of bytes is then compared byte-for-byte to the following string of bytes:

```
0x48 0x65 0x6C 0x6C 0x6F 0x0A
```

Note: "Hello" and a newline.

If the remote host sent back a string in any way different to this, then the user agent must close the connection and give up trying to connect.

Otherwise, the user agent must then take the string representing the script's domain in IDNA format, encode it as UTF-8, and send that to the remote host, followed by a 0x0A byte (a U+000A LINE FEED in UTF-8).

The user agent must then read all the bytes sent from the remote host, up to the first 0x0A byte (inclusive). That string of bytes must then be compared byte-for-byte to the string that was just sent to that host (the one with the IDNA domain name and ending with a newline character). If the remote host sent back a string in any way different to this, then the user agent must close the connection and give up trying to connect.

Otherwise, the connection has been established (and events and so forth get fired, as described above).

Note: For author-written servers (as opposed to the server side of a peer-to-peer connection), the script's domain would be replaced by the hostname of the server. Alternatively, such servers might instead wait for the client to send its domain string, and then simply echo it back. This would allow connections from pages on any domain, instead of just pages originating from the same host. The client compares the two strings to ensure they are the same before allowing the connection to be used by author script.

If at any point during this process the connection is closed prematurely, then the user agent must close the connection and give up trying to connect.

7.2.7.3. Sending and receiving data over TCP

When the `send(`*`data`*`)` method is invoked on the connection's corresponding `Connection` object, the user agent must take the *data* argument, replace any U+0000 NULL and U+0017 END OF

TRANSMISSION BLOCK characters in it with U+FFFD REPLACEMENT CHARACTER characters, then transmit a U+0002 START OF TEXT character, this new *data* string and a single U+0017 END OF TRANSMISSION BLOCK character (in that order) to the remote host, all encoded as UTF-8.

When the user agent receives bytes on the connection, the user agent must buffer received bytes until it receives a 0x17 byte (a U+0017 END OF TRANSMISSION BLOCK character). If the first buffered byte is not a 0x02 byte (a U+0002 START OF TEXT character encoded as UTF-8) then all the data up to the 0x17 byte, inclusive, must be dropped. (This allows for future extension of this protocol.) Otherwise, all the data from (but not including) the 0x02 byte and up to (but not including) the 0x17 byte must be taken, interpreted as a UTF-8 string, and a `read` event must be fired on the `Connection` object with that string as the `data`. If that string cannot be decoded as UTF-8 without errors, the packet should be ignored.

> ***Note: This protocol does not yet allow binary data (e.g. an image or video data) to be efficiently transmitted. A future version of this protocol might allow this by using the prefix character U+001F INFORMATION SEPARATOR ONE, followed by binary data which uses a particular byte (e.g. 0xFF) to encode byte 0x17 somehow (since otherwise 0x17 would be treated as transmission end by down-level UAs).***

7.2.8. Security

Need to write this section.

If you have an unencrypted page that is (through a man-in-the-middle attack) changed, it can access a secure service that is using IP authentication and then send that data back to the attacker. Ergo we should probably stop unencrypted pages from accessing encrypted services, on the principle that the actual level of security is zero. Then again, if we do that, we prevent insecure sites from using SSL as a tunneling mechanism.

Should consider dropping the subdomain-only restriction. It doesn't seem to add anything, and prevents cross-domain chatter.

7.2.9. Relationship to other standards

Should have a section talking about the fact that we blithely ignoring IANA's port assignments here.

Should explain why we are not reusing HTTP for this. (HTTP is too heavy-weight for such a simple need; requiring authors to implement an HTTP server just to have a party line is too much of a barrier to entry; cannot rely on prebuilt components; having a simple protocol makes it much easier to do RAD; HTTP doesn't fit the needs and doesn't have the security model needed; etc)

7.3. 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.

7.3.1. Processing model

When a script invokes the **`postMessage(`*`message`*`)`** method on a `Document` object, the user agent must create an event that uses the `CrossDocumentMessageEvent` interface, with the event name **`message`**, which bubbles, is cancelable, and has no default action. The **`data`** attribute must be set to the value passed as the *message* argument to the `postMessage()` method, the **`domain`** attribute must be set to the domain of the document that the script that invoked the methods is associated with, the **`uri`** attribute must be set to the URI of that document, and the **`source`** attribute must be set to the `Document` object representing that document.

⚠Warning! Authors should check the `domain` 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.

> For example, if document A contains an `object` element that contains document B, and script in document A calls `postMessage()` on document B, then a message event will be fired on that element, marked as originating from document A. The script in document A might look like:
>
> ```
> var o = document.getElementsByTagName('object')[0];
> o.contentDocument.postMessage('Hello world');
> ```
>
> 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:
>
> ```
> document.addEventListener('message', receiver, false);
> function receiver(e) {
>   if (e.domain == 'example.com') {
>     if (e.data == 'Hello world') {
>       e.source.postMessage('Hello');
>     } 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.

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.

7.3.2. Event definitions

The postMessage() method causes an event to be dispatched (as defined above). This event uses the following interface:

```
interface CrossDocumentMessageEvent : Event {
  readonly attribute DOMString data;
  readonly attribute DOMString domain;
  readonly attribute DOMString uri;
  readonly attribute Document source;
  void initCrossDocumentMessageEvent(in DOMString typeArg, in boolean ca
  void initCrossDocumentMessageEventNS(in DOMString namespaceURI, in DOM
};
```

The **data** attribute represents the message being sent.

The **domain** attribute represents the domain of the document from which the message came.

The **uri** attribute represents the address of the document from which the message came.

The **source** attribute represents the `Document` from which the message came.

The **initCrossDocumentMessageEvent()** and **initCrossDocumentMessageEventNS()** methods must initialise the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS]

8. The HTML 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. Any number of comments and space characters.
2. A DOCTYPE.
3. Any number of comments and space characters.
4. The root element, in the form of an `html` element.
5. Any number of comments and space characters.

The various types of content mentioned above are described in the next few sections.

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 U+0044 LATIN CAPITAL LETTER D or U+0064 LATIN SMALL LETTER D character.
4. A U+004F LATIN CAPITAL LETTER O or U+006F LATIN SMALL LETTER O character.
5. A U+0043 LATIN CAPITAL LETTER C or U+0063 LATIN SMALL LETTER C character.
6. A U+0054 LATIN CAPITAL LETTER T or U+0074 LATIN SMALL LETTER T character.
7. A U+0059 LATIN CAPITAL LETTER Y or U+0079 LATIN SMALL LETTER Y character.
8. A U+0050 LATIN CAPITAL LETTER P or U+0070 LATIN SMALL LETTER P character.
9. A U+0045 LATIN CAPITAL LETTER E or U+0065 LATIN SMALL LETTER E character.
10. One or more space characters.
11. A U+0048 LATIN CAPITAL LETTER H or U+0068 LATIN SMALL LETTER H character.
12. A U+0054 LATIN CAPITAL LETTER T or U+0074 LATIN SMALL LETTER T character.
13. A U+004D LATIN CAPITAL LETTER M or U+006D LATIN SMALL LETTER M character.
14. A U+004C LATIN CAPITAL LETTER L or U+006C LATIN SMALL LETTER L character.
15. Zero or more space characters.
16. A U+003E GREATER-THAN SIGN (>) character.

Note: In other words, `<!DOCTYPE HTML>`, case-insensitively.

8.1.2. Elements

There are four different kinds of **elements**: void elements, CDATA elements, RCDATA elements, and normal elements.

Void elements

`base`, `link`, `meta`, `hr`, `br`, `img`, `embed`, `param`, `area`, `col`, `input`

CDATA elements

`style`, `script`

RCDATA elements

`title`, `textarea`

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.

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 four 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, but the text must not contain the two character sequence "`</`" (U+003C LESS-THAN SIGN, U+002F SOLIDUS).

RCDATA elements can have text and character entity references, but the text must not contain the character U+003C LESS-THAN SIGN (<) or the character U+0026 AMPERSAND (`&`).

Normal elements can have text, character entity references, other elements, and comments, but the text must not contain the character U+003C LESS-THAN SIGN (<) or the character U+0026 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+0061 LATIN SMALL LETTER A .. U+007A LATIN SMALL LETTER Z, or, in uppercase, 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, then there may be a single U+002F SOLIDUS (/) character. This character has no effect except to appease the markup gods. As this character is therefore just a symbol of faith, atheists should omit it.
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** use characters in the range U+0061 LATIN SMALL LETTER A .. U+007A LATIN SMALL LETTER Z, or, in uppercase, U+0041 LATIN CAPITAL LETTER A .. U+005A LATIN CAPITAL LETTER Z, and U+002D HYPHEN-MINUS (-). In the HTML

syntax, attribute names may be written with any mix of lower- and uppercase letters that, when converted to all-lowercase, matches the attribute's name; attribute names are case-insensitive.

Attribute values are a mixture of text and character entity references, except with the additional restriction that the text cannot contain a U+0026 AMPERSAND (`&`) character.

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, U+003E GREATER-THAN SIGN (`>`) characters, or U+003C LESS-THAN SIGN (`<`) characters, and must not, furthermore, start with either a literal U+0022 QUOTATION MARK (`"`) character or a literal U+0027 APOSTROPHE (`'`) character.

> 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'>
> ```

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">
> ```

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 space character or a comment.

An `html` element's end tag may be omitted if the `html` element is not immediately followed by a space character or 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.

A `body` element's end tag may be omitted if the `body` element is not immediately followed by a space character or 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`, `blockquote`, `dl`, `fieldset`, `form`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `hr`, `menu`, `ol`, `p`, `pre`, `table`, or `ul` 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 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 preceeded 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 preceeded 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 `p` element must not contain `blockquote`, `dl`, `menu`, `ol`, `pre`, `table`, or `ul` elements, even though these elements are technically allowed inside `p` elements according to the content models described in this specification. (In fact, if one of those elements is put inside a `p` element in the markup, it will instead imply a `p` element end tag before it.)

An `optgroup` element must not contain `optgroup` elements, even though these elements are technically allowed to be nested according to the content models described in this specification. (If an `optgroup` element is put inside another in the markup, it will in fact imply an `optgroup` end tag before it.)

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.)

8.1.3. Text

Text is allowed inside elements, attributes, and comments. Text must consist of valid Unicode characters other than U+0000. Text should 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 entity references

In certain cases described in other sections, text may be mixed with **character entity references**. These can be used to escape characters that couldn't otherwise legally be included in text.

Character entity references must start with a U+0026 AMPERSAND (`&`). Following this, there are three possible kinds of character entity references:

Named entities

The ampersand must be followed by one of the names given in the entities section, using the same case. Finally, after the name, the entity must be terminated by a U+003B SEMICOLON character (`;`).

Decimal numeric entities

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 valid Unicode code point that isn't U+0000. The digits must then be followed by a U+003B SEMICOLON character (`;`).

Hexadecimal numeric entities

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 valid Unicode code point that isn't U+0000. The digits must then be followed by a U+003B SEMICOLON character (`;`).

8.1.5. 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 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 XML and Namespaces in XML specifications, and are out of scope of this specification.

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 resulted in this version of HTML returning to a non-SGML basis.***
>
> ***Authors interested in using SGML tools in their authoring pipeline are encouraged to use the XML serialisation of HTML5 instead of the HTML serialisation.***

This specification defines the parsing rules for HTML documents, whether they are syntactically valid 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.

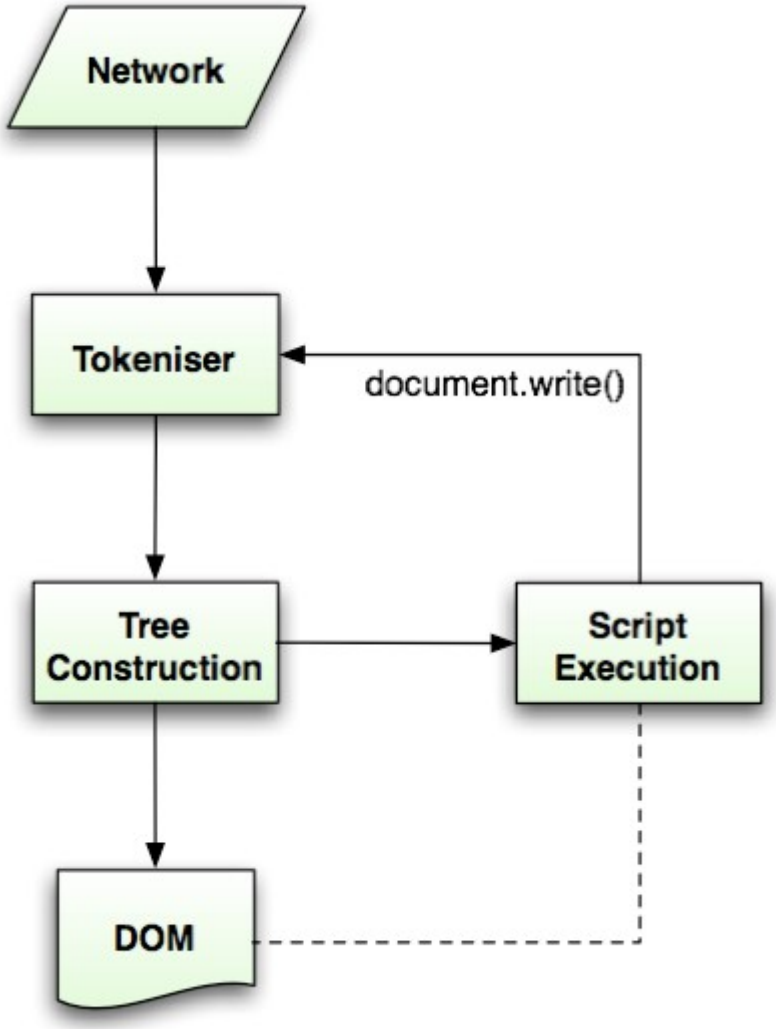
> ***Note: Parse errors are only errors with the syntax of HTML. In addition to checking for parse errors, conformance checkers will also verify that the document obeys all the other conformance requirements described in this specification.***

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 tokenisation 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 tokenisation 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>
> ...
> ```

8.2.2. The input stream

The stream of Unicode characters that consists the input to the tokenisation 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.

For HTML, user agents must use the following algorithm in determining the character encoding of a

document:

1. If the transport layer specifies an encoding, use that.

2. Otherwise, if the user agent can find a `meta` element that specifies character encoding information, then use that. (The exact parsing rules for finding and using this information are not yet described in this specification.) This needs to be fleshed out a whole heck of a lot more.

3. Otherwise, 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 use that.

4. Otherwise, use an implementation-defined or user-specified default character encoding (`ISO-8859-1`, `windows-1252`, and `UTF-8` are recommended as defaults, and can in many cases be identified by inspection as they have different ranges of valid bytes).

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]

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.

A leading U+FEFF BYTE ORDER MARK (BOM) must be dropped if present.

All U+0000 NULL characters in the input must be replaced by U+FFFD REPLACEMENT 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 tokenisation 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 **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 uninitialised.

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" charecter is not a real character in the stream, but rather the lack of any further characters.

8.2.3. Tokenisation

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 behaviour and can consume several characters before switching to another state.

The exact behaviour 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 is in the PCDATA state.

The output of the tokenisation 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 names and can be either correct or in error. Start and end tag tokens have a tagname and a list of attributes, each of which has a name and a value. 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.)

Before each step of the tokeniser, the user agent must check to see if any scripts are ready. ...

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: switch to the entity data state.
Otherwise: treat it as per the "anything else" entry below.

↪ **U+003C LESS-THAN SIGN (<)**
When the content model flag is set to a state other than the PLAINTEXT state: switch to the tag open state.
Otherwise: treat it as per the "anything else" entry below.

↪ **EOF**
Emit an end-of-file token.

↪ **Anything else**
Emit the input character as a character token. Stay in the data state.

Entity data state

(This cannot happen if the content model flag is set to the CDATA state.)

Attempt to consume an entity.

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.

Tag open state

The behaviour of this state depends on the content model flag.

If the content model flag is set to the RCDATA or CDATA states
If the next input character is a U+002F SOLIDUS (/) character, consume it and switch to the close tag open state. If the next input character is not a U+002F SOLIDUS (/) character, 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.

Close tag open state

If the content model flag is set to the RCDATA or CDATA states then examine the next few characters. If they do not match the tag name of the last start tag token emitted (case insensitively), 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+000B LINE TABULATION
- U+000C FORM FEED (FF)
- U+0020 SPACE
- U+003E GREATER-THAN SIGN (>)
- U+002F SOLIDUS (/)
- U+003C LESS-THAN SIGN (<)
- EOF

...then there is a parse error. Emit a U+003C LESS-THAN SIGN character token, a U+002F SOLIDUS character token, and reconsume the current input character in the data state.

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.

Tag name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000B LINE TABULATION**
↪ **U+000C FORM FEED (FF)**
↪ **U+0020 SPACE**
Switch to the before attribute name 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.

↪ **U+003C LESS-THAN SIGN (<)**
↪ **EOF**
Parse error. Emit the current tag token. Reconsume the character in the data state.

↪ **U+002F SOLIDUS (/)**
Parse error unless this is a permitted slash. Switch to the before attribute name state.

↪ **Anything else**
Append the current input character to the current tag token's tag name. Stay in the tag name state.

Before attribute name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000B LINE TABULATION**
↪ **U+000C FORM FEED (FF)**
↪ **U+0020 SPACE**
Stay in the before attribute name 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+002F SOLIDUS (/)**
Parse error unless this is a permitted slash. Stay in the before attribute name state.

↪ **U+003C LESS-THAN SIGN (<)**

↪ **EOF**
Parse error. Emit the current tag token. Reconsume the 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.

Attribute name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**
Switch to the after attribute name 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+002F SOLIDUS (/)**
Parse error unless this is a permitted slash. Switch to the before attribute name state.

↪ **U+003C LESS-THAN SIGN (<)**

↪ **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 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).

After attribute name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

- **U+000A LINE FEED (LF)**
- **U+000B LINE TABULATION**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Stay in the after attribute name 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+002F SOLIDUS (/)**

  Parse error unless this is a permitted slash. Switch to the before attribute name state.
- **U+003C LESS-THAN SIGN (<)**
- **EOF**

  Parse error. Emit the current tag token. Reconsume the 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.

Before attribute value state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000B LINE TABULATION**
- **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 (>)**

  Emit the current tag token. Switch to the data state.
- **U+003C LESS-THAN SIGN (<)**
- **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.

Attribute value (double-quoted) state

Consume the next input character:

- **U+0022 QUOTATION MARK (")**
  Switch to the before attribute name state.
- **U+0026 AMPERSAND (&)**
  Switch to the entity in attribute value state.
- **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.

Attribute value (single-quoted) state

Consume the next input character:

- **U+0027 APOSTROPHE (')**
  Switch to the before attribute name state.
- **U+0026 AMPERSAND (&)**
  Switch to the entity in attribute value state.
- **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.

Attribute value (unquoted) state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000B LINE TABULATION**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
  Switch to the before attribute name state.
- **U+0026 AMPERSAND (&)**
  Switch to the entity in attribute value state.
- **U+003E GREATER-THAN SIGN (>)**
  Emit the current tag token. Switch to the data state.
- **U+003C LESS-THAN SIGN (<)**
- **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.

Entity in attribute value state

Attempt to consume an entity.

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.

Bogus comment state

(This can only happen if the content model flag is set to the PCDATA state.)

Consume every character up to 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 last consumed character before the U+003E character, if any, or up to the end of the file otherwise. (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.

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 state.

Otherwise if the next seven chacacters are a case-insensitive match for the word "DOCTYPE", then consume those characters and switch to the DOCTYPE state.

Otherwise, is 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.

Comment state

Consume the next input character:

↪ **U+002D HYPHEN-MINUS (-)**
Switch to the comment 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.

Comment 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.

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.

DOCTYPE state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **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.

Before DOCTYPE name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the before DOCTYPE name state.

↪ **U+0061 LATIN SMALL LETTER A through to U+007A LATIN SMALL LETTER Z**

Create a new DOCTYPE token. Set the token's name name to the uppercase version of the current input character (subtract 0x0020 from the character's code point), and mark it as being in error. Switch to the DOCTYPE name state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Emit a DOCTYPE token whose name is the empty string and that is marked as being in error. Switch to the data state.

↪ **EOF**

Parse error. Emit a DOCTYPE token whose name is the empty string and that is marked as being in error. Reconsume the EOF character in the data state.

↪ **Anything else**
Create a new DOCTYPE token. Set the token's name name to the current input character, and mark it as being in error. Switch to the DOCTYPE name state.

DOCTYPE name state

First, consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000B LINE TABULATION**
↪ **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+0061 LATIN SMALL LETTER A through to U+007A LATIN SMALL LETTER Z**
Append the uppercase version of the current input character (subtract 0x0020 from the character's code point) to the current DOCTYPE token's name. Stay in the DOCTYPE name state.
↪ **EOF**
Parse error. Emit the current 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.

Then, if the name of the DOCTYPE token is exactly the four letters "HTML", then mark the token as being correct. Otherwise, mark it as being in error.

Note: Because lowercase letters in the name are uppercased by the algorithm above, the "HTML" letters are actually case-insensitive relative to the markup.

After DOCTYPE name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000B LINE TABULATION**
↪ **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. Emit the current DOCTYPE token. Reconsume the EOF character in the data state.
↪ **Anything else**
Parse error. Mark the DOCTYPE token as being in error, if it is not already. Switch to the bogus DOCTYPE state.

Bogus DOCTYPE state

Consume the next input character:

↪ **U+003E GREATER-THAN SIGN (>)**
Emit the current DOCTYPE token. Switch to the data state.

↪ **EOF**
Parse error. Emit the current DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**
Stay in the bogus DOCTYPE state.

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.

A **permitted slash** is a U+002F SOLIDUS character that is immediately followed by a U+003E GREATER-THAN SIGN, if, and only if, the current token being processed is a start tag token whose tag name is one of the following: `base`, `link`, `meta`, `hr`, `br`, `img`, `embed`, `param`, `area`, `col`, `input`

8.2.3.1. Tokenising entities

This section defines how to **consume an entity**. This definition is used when parsing entities in text and in attributes.

The behaviour depends on the identity of the next character (the one immediately after the U+0026 AMPERSAND character):

↪ **U+0023 NUMBER SIGN (#)**
Consume the U+0023 NUMBER SIGN.

The behaviour 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+0078 LATIN SMALL LETTER F, and U+0041 LATIN CAPITAL LETTER A, through to U+0058 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), and return a character token for the Unicode character whose code point is that number. If the number is not a valid Unicode character (e.g. if the number is higher than 1114111), or if the number is zero, then return a character token for the U+FFFD REPLACEMENT CHARACTER character instead.

↪ **Anything else**

Consume the maximum number of characters possible, with the consumed characters case-sensitively matching one of the identifiers in the first column of the entities table.

If no match can be made, then this is a parse error. No characters are consumed, and 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.

Return a character token for the character corresponding to the entity name (as given by the second column of the entities table).

> If the markup contains `I'm &notit without you`, the entity is parsed as "not", as in, `I'm ¬it without you`. But if the markup was `I'm &notin without you`, the entity would be parsed as "notin", resulting in `I'm ∉ without you`.

This isn't quite right. For some entities, UAs require a semicolon, for others they don't. We probably need to do the same for backwards compatibility. If we do that we might be able to add more entities, e.g. for mathematics. Probably the way to mark whether or not an entity requires a semicolon is with an additional column in the entity table lower down.

8.2.4. Tree construction

The input to the tree construction stage is a sequence of tokens from the tokenisation 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.

Tree construction passes through several phases. Initially, UAs must act according to the steps described as being those of the initial phase.

This specification does not define when an interactive user agent has to render the `Document` available to the user, or when it has to begin accepting user input.

When the steps below require the UA to **append a character** to a node, the UA must collect it and all subsequent consecutive characters that would be appended to that node, and insert one `Text` node whose data is the concatenation of all those characters.

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 implementators are encouraged to avoid arbitrary limits, it is recognised that practical concerns will likely force user agents to impose nesting depths.

8.2.4.1. The initial phase

Initially, the tree construction stage must handle each token emitted from the tokenisation stage as follows:

- **A DOCTYPE token that is marked as being in error**
- **A comment token**
- **A start tag token**
- **An end tag token**
- **A character token that is not one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**
- **An end-of-file token**

  This specification does not define how to handle this case. In particular, user agents may ignore the entirety of this specification altogether for such documents, and instead invoke special parse modes with a greater emphasis on backwards compatibility.

  Browsers in particular have generally used DOCTYPE-based sniffing to invoke an "alternative conformance mode" known as quirks mode on certain documents. In this mode, emphasis is put on legacy compatibility rather than on standards compliance. This specification takes no position on this behaviour; documents without DOCTYPEs or with DOCTYPEs that do not conform to the syntax allowed by this specification are considered to be out of scope of this specification.

As far as parsing goes, the quirks I know of are:

- Comment parsing is different.
- The following is considered one script block (!):

```
<script><!-- document.write('</script>'); --></script>
```

- `</br>` and `</p>` do magical things.
- `p` can contain `table`
- Safari and IE have special parsing rules for <% ... %> (even in standards mode, though clearly this should be quirks-only).

Maybe we should just adopt all those and be done with it. One parsing mode to rule them all. Or legitimise/codify the quirks mode parsing in some way.

Would be interesting to do a search to see how many pages hit each of the above.

↪ **A DOCTYPE token marked as being correct**

Append a `DocumentType` node to the `Document` node, with the `name` attribute set to the name given in the DOCTYPE token (which will be "HTML"), and the other attributes specific to `DocumentType` objects set to null, empty lists, or the empty string as appropriate.

Then, switch to the root element phase of the tree construction stage.

↪ **A character token that *is* one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append that character to the `Document` node.

8.2.4.2. The root element phase

After the initial phase, as each token is emitted from the tokenisation stage, it must be processed as described in this section.

↪ **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 one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append that character to the `Document` node.

↪ **A character token that is *not* one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

↪ **A start tag token**

↪ **An end tag token**

↪ **An end-of-file token**

Create an `HTMLElement` node with the tag name `html`, in the HTML namespace. Append it to the `Document` object. Switch to the main phase and reprocess the current token.

Should probably make end tags be ignored, so that "</head><!-- --><html>" puts the comment befor 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.

8.2.4.3. The main phase

After the root element phase, each token emitted from the tokenisation stage must be processed as described in *this* section. This is by far the most involved part of parsing an HTML document.

The tree construction stage in this phase has several pieces of state: a stack of open elements, a list of active formatting elements, a `head` element pointer, a `form` element pointer, and an insertion mode.

We could just fold insertion modes and phases into one concept (and duplicate the two rules common to all insertion modes into all of them).

8.2.4.3.1. THE STACK OF OPEN ELEMENTS

Initially the **stack of open elements** contains just the `html` root element node created in the last phase before switching to *this* phase (or, in the `innerHTML` case, the `html` element created to represent the element whose `innerHTML` attribute is being set). That's the topmost node of the stack. It never gets popped off the stack. (This stack grows downwards.)

The **current node** is the bottommost node in this stack.

Elements in the stack fall into the following categories:

Special

The following HTML elements have varying levels of special parsing rules: `address`, `area`, `base`, `basefont`, `bgsound`, `blockquote`, `body`, `br`, `center`, `col`, `colgroup`, `dd`, `dir`, `div`, `dl`, `dt`, `embed`, `fieldset`, `form`, `frame`, `frameset`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `head`, `hr`, `iframe`, `image`, `img`, `input`, `isindex`, `li`, `link`, `listing`, `menu`, `meta`, `noembed`, `noframes`, `noscript`, `ol`, `optgroup`, `option`, `p`, `param`, `plaintext`, `pre`, `script`, `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: `button`, `caption`, `html`, `marquee`, `object`, `table`, `td` and `th`.

Formatting

The following HTML elements are those that end up in the list of active formatting elements: `a`, `b`, `big`, `em`, `font`, `i`, `nobr`, `s`, `small`, `strike`, `strong`, `tt`, and `u`.

Phrasing

All other elements found while parsing an HTML document.

Still need to add these new elements to the lists: `event-source`, `section`, `nav`, `article`, `aside`, `header`, `footer`, `datagrid`, `command`

The stack of open elements is said to **have an element in scope** or **have an element in *table* scope** when the following algorithm terminates in a match state:

1. Initialise *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 a `table` element, terminate in a failure state.
4. Otherwise, if the algorithm is the "has an element in scope" variant (rather than the "has an element in table scope" variant), and *node* is one of the following, terminate in a failure state:
    - `caption`
    - `td`
    - `th`
    - `button`
    - `marquee`
    - `object`
5. Otherwise, if *node* is an `html` element, terminate in a failure state. (This can only happen if the *node* is the topmost node of the stack of open elements, and prevents the next step from being invoked if there are no more elements in the stack.)
6. 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 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.

8.2.4.3.2. 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 buttons, `object` elements, marquees, table cells, and table captions, and are used to prevent formatting from "leaking" into tables, buttons, `object` elements, and marquees.

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.

8.2.4.3.3. Creating and inserting HTML elements

When the steps below require the UA to **create an element for a token**, the UA must create a node implementing the interface appropriate for the element type corresponding to the tag name of the token (as given in the section of this specification that defines that element, e.g. for an `a` element it would be the `HTMLAnchorElement` interface), with the tag name being the name of that element, with the node being in the HTML namespace, and with the attributes on the node being those given in the given token.

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, 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 create an element for the token and then insert or append the new node in the location specified. (This happens in particular during the parsing of tables with invalid content.)

The interface appropriate for an element that is not defined in this specification is `HTMLElement`.

8.2.4.3.4. Closing elements that have implied end tags

When the steps below require the UA to **generate implied end tags**, then, if the current node is a `dd` element, a `dt` element, an `li` element, a `p` element, a `td` element, a `th` element, or a `tr` element, the UA must act as if an end tag with the respective tag name had been seen and then generate implied end tags again.

The step that 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.

8.2.4.3.5. 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.

8.2.4.3.6. The insertion mode

Initially the **insertion mode** is "before head". It can change to "in head", "after head", "in body", "in table", "in caption", "in column group", "in table body", "in row", "in cell", "in select", "after body", "in frameset", and "after frameset" during the course of the parsing, as described below. It affects how certain tokens are processed.

If the tree construction stage is switched from the main phase to the trailing end phase and back again, the various pieces of state are not reset; the UA must act as if the state was maintained.

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. If the element whose `innerHTML` attribute is being set is neither a `td` element nor a `th` element, then set *node* to the element whose `innerHTML` attribute is being set. (`innerHTML` case)
4. If *node* is a `select` element, then switch the insertion mode to "in select" and abort these steps. (`innerHTML` case)
5. If *node* is a `td` or `th` element, 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. (`innerHTML` case)

10. If *node* is a `table` element, then switch the insertion mode to "in table" and abort these steps.

11. If *node* is a `head` element, then switch the insertion mode to "in body" ("in body"! *not "in head"*!) and abort these steps. (`innerHTML` case)

12. If *node* is a `body` element, then switch the insertion mode to "in body" and abort these steps.

13. If *node* is a `frameset` element, then switch the insertion mode to "in frameset" and abort these steps. (`innerHTML` case)

14. 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. (`innerHTML` case)

15. If *last* is true, then set the insertion mode to "in body" and abort these steps. (`innerHTML` case)

16. Let *node* now be the node before *node* in the stack of open elements.

17. Return to step 3.

8.2.4.3.7. How to handle tokens in the main phase

Tokens in the main phase must be handled as follows:

↪ A DOCTYPE token

Parse error. Ignore the token.

↪ A start tag token with the tag name "html"

If this start tag token was not the first start tag token, then it is a 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.

↪ An end-of-file token

Generate implied end tags.

If there are more than two nodes on the stack of open elements, or if there are two nodes but the second node is not a `body` node, this is a parse error.

Otherwise, if the parser was originally created in order to handle the setting of an element's `innerHTML` attribute, and there's more than one element in the stack of open elements, and the second node on the stack of open elements is not a `body` node, then this is a parse error. (`innerHTML` case)

Stop parsing.

This fails because it doesn't imply HEAD and BODY tags. We should probably expand out the insertion modes and merge them with phases and then put the three things here into each insertion mode instead of trying to factor them out so carefully.

↪ **Anything else**

Depends on the insertion mode:

↪ **If the insertion mode is "before head"**

Handle the token as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character to 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 start tag token with the tag name "head"**

Create an element for the token.

Set the `head` element pointer to this new element node.

Append the new element to the current node and push it onto the stack of open elements.

Change the insertion mode to "in head".

↪ **A start tag token whose tag name is one of: "base", "link", "meta", "script", "style", "title"**

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 a `head` element being generated, and with the current token being reprocessed in the "in head" insertion mode.

↪ **An end tag with the tag name "html"**

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.

↪ **A character token that is *not* one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

↪ **Any other start tag token**

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.

↪ **If the insertion mode is "in head"**

Handle the token as follows.

Note: The rules for handling "title", "style", and "script" start tags are similar, but not identical.

Note: It is possible for the tree construction stage's main phase to be in the "in head" insertion mode without the current node being a `head` element, e.g. if a `head` end tag is immediately followed by a `meta` start tag.

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character to 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 start tag with the tag name "title"**

Create an element for the token.

Append the new element to the node pointed to by the `head` element pointer, or, if that is null (`innerHTML` case), to the current node.

Switch the tokeniser's content model flag to the RCDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token.

If this process resulted in a collection of character tokens, append a single `Text` node to the `title` element node whose contents is the concatenation of all those tokens' characters.

The tokeniser's content model flag will have switched back to the PCDATA state.

If the next token is an end tag token with the tag name "title", ignore it. Otherwise, this is a parse error.

↪ **A start tag with the tag name "style"**

Create an element for the token.

Append the new element to the node pointed to by the `head` element pointer, or, if that is null (`innerHTML` case), to the current node.

Switch the tokeniser's content model flag to the CDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token, or until it stops tokenising.

If this process resulted in a collection of character tokens, append a single `Text` node to the `style` element node whose contents is the concatenation of all those tokens' characters.

The tokeniser's content model flag will have switched back to the PCDATA state.

If the next token is an end tag token with the tag name "style", ignore it. Otherwise, this is a parse error.

↪ **A start tag with the tag name "script"**

Create an element for the token.

Mark the element as being "parser-inserted". 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.

Switch the tokeniser's content model flag to the CDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token, or until it stops tokenising.

If this process resulted in a collection of character tokens, append a single `Text` node to the `script` element node whose contents is the concatenation of all those tokens' characters.

The tokeniser's content model flag will have switched back to the PCDATA state.

If the next token is not an end tag token with the tag name "script", then this is a parse error; mark the `script` element as "already executed". Otherwise, the token is the `script` element's end tag, so ignore it.

If the parser was originally created in order to handle the setting of a node's `innerHTML` attribute, then mark the `script` element as "already executed", and skip the rest of the processing described for this token (including the part below where "scripts that will execute as soon as the parser resumes" are executed). (`innerHTML` case)

> ***Note: Marking the `script` element as "already executed" prevents it from executing when it is inserted into the document a few paragraphs below. Scripts missing their end tags and scripts that were inserted using `innerHTML` aren't executed.***

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.

Append the new element to the current node, unless the insertion mode is "in head" and the `head` element pointer is not null, in which case append it to the node pointed to by the `head` element pointer. Special processing occurs when a `script` element is inserted into a document that might cause some script to execute, which might cause new characters to be inserted into the tokeniser.

Let the insertion point have the value of the *old insertion point*. (In other words, restore the insertion point to the value it had before the previous paragraph. This value might be the "undefined" value.)

At this stage, if there is a script that will execute as soon as the parser resumes, then:

↪ **If the tree construction stage is being called reentrantly, say from a call to `document.write()`:**

Abort the processing of any nested invokations of the tokeniser, yielding control back to the caller. (Tokenisation will resume when the caller returns to the "outer" tree construction stage.)

↪ **Otherwise:**

Follow these steps:

1. Let *the script* be the script that will execute as soon as the parser resumes. There is no longer a script that will execute as soon as the parser resumes.
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 script that will execute as soon as the parser resumes, then repeat these steps from step 1.

↪ **A start tag with the tag name "base", "link", or "meta"**

Create an element for the token.

Append the new element to the node pointed to by the `head` element pointer, or, if that is null (`innerHTML` case), to the current node.

Need to cope with second and subsequent `base` elements affecting subsequent elements magically.

↪ **An end tag with the tag name "head"**

If the current node is a `head` element, pop the current node off the stack of open elements. Otherwise, this is a parse error.

Change the insertion mode to "after head".

↪ **An end tag with the tag name "html"**

Act as described in the "anything else" entry below.

↪ **A start tag with the tag name "head"**

↪ **Any other end tag**

Parse error. Ignore the token.

↪ **Anything else**

If the current node is a `head` element, act as if an end tag token with the tag name "head" had been seen.

Otherwise, change the insertion mode to "after head".

Then, 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?

↪ **If the insertion mode is "after head"**

Handle the token as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character to 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 start tag token with the tag name "body"**

Insert a `body` element for the token.

Change the insertion mode to "in body".

↪ **A start tag token with the tag name "frameset"**

Insert a `frameset` element for the token.

Change the insertion mode to "in frameset".

↪ **A start tag token whose tag name is one of: "base", "link", "meta", "script", "style", "title"**

Parse error. Switch the insertion mode back to "in head" and reprocess 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.

↪ **If the insertion mode is "in body"**

Handle the token as follows:

↪ **A character token**

Reconstruct the active formatting elements, if any.

Append the token's character to 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 start tag token whose tag name is "script"**

Process the token as if the insertion mode had been "in head".

↪ **A start tag token whose tag name is one of: "base", "link", "meta", "style", "title"**

Parse error. Process the token as if the insertion mode had been "in head".

↪ **A start tag token with the tag name "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. (`innerHTML` 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.

↪ **An end tag with the tag name "body"**

If the second element in the stack of open elements is not a `body` element, this is a parse error. Ignore the token. (`innerHTML` case)

Otherwise:

If the current node is not the `body` element, then this is a parse error.

Change the insertion mode to "after body".

↪ **An end tag with the tag name "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 `innerHTML` case.

↪ **A start tag whose tag name is one of: "address", "blockquote", "center", "dir", "div", "dl", "fieldset", "listing", "menu", "ol", "p", "pre", "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 "form"**

If the `form` element pointer is not null, ignore the token with a parse error.

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"**

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.

Run the following algorithm:

1. Initialise *node* to be the current node (the bottommost node of the stack).
2. If *node* is an `li` element, then pop all the nodes from the current node up to *node*, including *node*, then stop this algorithm.
3. If *node* is not in the formatting category, and is not in the phrasing category, and is not an `address` or `div` element, then stop this algorithm.
4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2.

Finally, insert an `li` element.

↪ **A start tag whose tag name is "dd" or "dt"**

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.

Run the following algorithm:

1. Initialise *node* to be the current node (the bottommost node of the stack).
2. If *node* is a `dd` or `dt` element, then pop all the nodes from the current node up to *node*, including *node*, then stop this algorithm.
3. If *node* is not in the formatting category, and is not in the phrasing category, and is not an `address` or `div` element, then stop this algorithm.
4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2.

Finally, insert an HTML element with the same tag name as the

token's.

↪ **A start tag token 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", "blockquote", "center", "dir", "div", "dl", "fieldset", "listing", "menu", "ol", "pre", "ul"**

If the stack of open elements has an element in scope with the same tag name as that of the token, then generate implied end tags.

Now, if the current node is not an element with the same tag name as that of the token, then this is a parse error.

If the stack of open elements has an element in scope with the same tag name as that of the token, then pop elements from this stack until an element with that tag name has been popped from the stack.

↪ **An end tag whose tag name is "form"**

If the stack of open elements has an element in scope with the same tag name as that of the token, then generate implied end tags.

Now, if the current node is not an element with the same tag name as that of the token, then this is a parse error.

If the stack of open elements has an element in scope with the same tag name as that of the token, then pop elements from this stack until an element with that tag name has been popped from the stack.

In any case, set the form element pointer to null.

↪ **An end tag whose tag name is "p"**

If the stack of open elements has a p element in scope, then generate implied end tags, except for p elements.

If the current node is not a p element, then this is a parse error.

If the stack of open elements has a p element in scope, then pop elements from this stack until the stack no longer has a p element in scope.

↪ **An end tag whose tag name is "dd", "dt", or "li"**

If the stack of open elements has an element in scope whose tag name matches the tag name of the token, then generate implied end tags, except for elements with the same tag name as the token.

If the current node is not an element with the same tag name as the token, then this is a parse error.

If the stack of open elements has an element in scope whose tag name matches the tag name of the token, then pop elements from this stack until an element with that tag name has been popped from the stack.

↪ **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 stack of open elements has in scope an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then this is a parse error; pop elements from the stack until an element with one of those tag names has been popped from the stack.

Insert an HTML element for the token.

↪ **An end tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

If the stack of open elements has in scope an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then generate implied end tags.

Now, if the current node is not an element with the same tag name as that of the token, then this is a parse error.

If the stack of open elements has in scope an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then pop elements from the stack until an element with one of those tag names has been popped from the stack.

↪ **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", "em", "font", "i", "nobr", "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.

↪ **An end tag whose tag name is one of: "a", "b", "big", "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. Abort these steps. The token is ignored.

   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 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. If the *furthest block* has a parent node, then remove the *furthest block* from its parent node.

6. 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.

7. Let *node* and *last node* be the *furthest block*. Follow these steps:

    1. Let *node* be the element immediately prior to *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. If *node* has any children, 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.

8. Insert whatever *last node* ended up being in the previous step into the *common ancestor* node, first removing it from its previous parent node if any.

9. Perform a shallow clone of the *formatting element*.

10. Take all of the child nodes of the *furthest block* and append them to the clone created in the last step.

11. Append that clone to the *furthest block*.

12. 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.

13. Remove the *formatting element* from the stack of open elements, and insert the clone into the stack of open

elements immediately after (i.e. in a more deeply nested position than) the position of the *furthest block* in that stack.

14. 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 token 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.

↪ **A start tag token whose tag name is one of: "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.

↪ **An end tag token whose tag name is one of: "button", "marquee", "object"**

If the stack of open elements has in scope an element whose tag name is the same as the tag name of the token, then 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.

Now, if the stack of open elements has an element in scope whose tag name matches the tag name of the token, then pop elements from the stack until that element has been popped from the stack, and clear the list of active formatting elements up to the last marker.

↪ **A start tag token whose tag name is "xmp"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Switch the content model flag to the CDATA state.

↪ **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.

Change the insertion mode to "in table".

↪ **A start tag whose tag name is one of: "area", "basefont", "bgsound", "br", "embed", "img", "param", "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.

↪ **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.

↪ **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 "input"**

Reconstruct the active formatting elements, if any.

Insert an `input` element for the token.

If the `form` element pointer is not null, then associate the `input` element with the `form` element pointed to by the `form` element pointer.

Pop that `input` element off the stack of open elements.

↪ **A start tag whose tag name is "isindex"**

Parse error.

If the `form` element pointer is not null, then ignore the token.

Otherwise:

Act as if a start tag token with the tag name "form" had been seen.

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 with the "name" attribute set to the value "isindex" (ignoring any explicit "name" attribute).

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.

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.

Then need to specify that if the form submission causes just a single form control, whose name is "isindex", to be submitted, then we submit just the value part, not the "isindex=" part.

↪ **A start tag whose tag name is "textarea"**

Create an element for the token.

If the `form` element pointer is not null, then associate the `textarea` element with the `form` element pointed to by the `form` element pointer.

Append the new element to the current node.

Switch the tokeniser's content model flag to the RCDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token, or until it stops tokenising.

If this process resulted in a collection of character tokens, append a single `Text` node, whose contents is the concatenation of all those tokens' characters, to the new element node.

The tokeniser's content model flag will have switched back to the PCDATA state.

If the next token is an end tag token with the tag name "textarea", ignore it.

↪ **A start tag whose tag name is one of: "iframe", "noembed", "noframes"**

↪ **A start tag whose tag name is "noscript", if scripting is enabled:**

Create an element for the token.

For "iframe" tags, the node must be an `HTMLIFrameElement` object, for the other tags it must be an `HTMLElement` object.

Append the new element to the current node.

Switch the tokeniser's content model flag to the CDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token, or until it stops tokenising.

If this process resulted in a collection of character tokens, append a single `Text` node, whose contents is the concatenation of all those tokens' characters, to the new element node.

The tokeniser's content model flag will have switched back to the PCDATA state.

If the next token is an end tag token with the same tag name as the start tag token, ignore it.

↪ **A start tag whose tag name is "select"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Change the insertion mode to "in select".

↪ **A start or end tag whose tag name is one of: "caption", "col", "colgroup", "frame", "frameset", "head", "option", "optgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

↪ **An end tag whose tag name is one of: "area", "basefont", "bgsound", "br", "embed", "hr", "iframe", "image", "img", "input", "isindex", "noembed", "noframes", "param", "select", "spacer", "table", "textarea", "wbr"**

↪ **An end tag whose tag name is "noscript", if scripting is enabled:**

Parse error. Ignore the token.

↪ **A start or end tag whose tag name is one of: "event-source", "section", "nav", "article", "aside", "header", "footer", "datagrid", "command"**

Work in progress!

↪ **A start tag token not covered by the previous entries**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Note: This element will be a phrasing element.

↪ **An end tag token not covered by the previous entries**

Run the following algorithm:

1. Initialise *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 this algorithm.

3. Otherwise, if *node* is in neither the formatting category nor the phrasing category, then this is a parse error. Stop this algorithm. The end tag token is ignored.

4. Set *node* to the previous entry in the stack of open elements.

5. Return to step 2.

↪ **If the insertion mode is "in table"**

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character to 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 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 `innerHTML` 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. (`innerHTML` case)

Otherwise:

Generate implied end tags.

Now, if the current node is not a `table` element, then this is a parse error.

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.

↪ **Anything else**

Parse error. Process the token as if the insertion mode was "in body", with the following exception:

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 inserted into the *foster parent element*.

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 (`innerHTML` 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 the new node must be inserted immediately *before* the last `table` element in the stack of open elements in the foster parent element; otherwise, the new

node must be *appended* to the foster parent element.

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. If this causes any elements to be popped from the stack, then this is a parse error.

Note: The current node being an `html` element after this process is an `innerHTML` case.

↪ **If the insertion mode is "in caption"**

↪ **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. (`innerHTML` 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 `innerHTML` 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 as if the insertion mode was "in body".

↪ **If the insertion mode is "in column group"**

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character to 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 start tag whose tag name is "col"**

Insert a `col` element for the token. Immediately pop the current node off the stack of open elements.

↪ **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. (`innerHTML` 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.

↪ **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 `innerHTML` case.

↪ **If the insertion mode is "in table body"**

↪ **A start tag whose tag name is "tr"**

Clear the stack back to a table body context. (See below.)

Insert a `tr` 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. (`innerHTML` 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 as if the insertion mode was "in table".

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. If this causes any elements to be popped from the stack, then this is a parse error.

Note: The current node being an `html` element after this process is an `innerHTML` case.

↪ **If the insertion mode is "in row"**

↪ **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. (`innerHTML` 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 `innerHTML` 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 as if the insertion mode was "in table".

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. If this causes any elements to be popped from the stack, then this is a parse error.

Note: The current node being an `html` element after this process is an `innerHTML` case.

↪ **If the insertion mode is "in cell"**

↪ **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, except for elements with the same tag name as the token.

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. (`innerHTML` 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 `innerHTML` 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 as if the insertion mode was "in body".

Where the steps above say to **close the cell**, they mean to follow 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".

↪ **If the insertion mode is "in select"**

Handle the token as follows:

↪ **A character token**

Append the token's character to 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 start tag token 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 token 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 token 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 token 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. (`innerHTML` 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.

↪ **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**

Parse error. Ignore the token.

↪ **If the insertion mode is "after body"**

Handle the token as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Process the token as it would be processed if the insertion mode was "in body".

↪ **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.

- **An end tag with the tag name "html"**

  If the parser was originally created in order to handle the setting of *an element*'s `innerHTML` attribute, this is a parse error; ignore the token. (The element will be an `html` element in this case.) (`innerHTML` case)

  Otherwise, switch to the trailing end phase.

- **Anything else**

  Parse error. Set the insertion mode to "in body" and reprocess the token.

- **If the insertion mode is "in frameset"**

  Handle the token as follows:

  - **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

    Append the character to 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 start tag with the tag name "frameset"**

    Insert a `frameset` element for the token.

  - **An end tag with the tag name "frameset"**

    If the current node is the root `html` element, then this is a parse error; ignore the token. (`innerHTML` case)

    Otherwise, pop the current node from the stack of open elements.

    If the parser was *not* originally created in order to handle the setting of an element's `innerHTML` attribute (`innerHTML` case), and the current node is no longer a `frameset` element, then change the insertion mode to "after frameset".

  - **A start tag with the tag name "frame"**

    Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

  - **A start tag with the tag name "noframes"**

    Process the token as if the insertion mode had been "in body".

  - **Anything else**

    Parse error. Ignore the token.

- **If the insertion mode is "after frameset"**

  Handle the token as follows:

  - **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

    Append the character to 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.

↪ **An end tag with the tag name "html"**

Switch to the trailing end phase.

↪ **A start tag with the tag name "noframes"**

Process the token as if the insertion mode had been "in body".

↪ **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.

8.2.4.4. The trailing end phase

After the main phase, as each token is emitted from the tokenisation stage, it must be processed as described in this section.

↪ **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 one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Process the token as it would be processed in the main phase.

↪ **A character token that is *not* one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

↪ **A start tag token**

↪ **An end tag token**

Parse error. Switch back to the main phase and reprocess the token.

↪ **An end-of-file token**

Stop parsing.

Need to define the term **stop parsing**. have to run all pending scripts (so document.write()s work) and say something about the load event. Also need to execute any pending loads of scripts in the "list of scripts that will execute when the document has finished parsing".

8.3. Namespaces

The **HTML namespace** is: `http://www.w3.org/1999/xhtml`

8.4. Entities

This table lists the entity names that are supported by HTML, and the code points to which they refer. It is referenced by the previous sections.

Entity Name	Character
AElig	U+00C6
Aacute	U+00C1
Acirc	U+00C2
Agrave	U+00C0
Alpha	U+0391
Aring	U+00C5
Atilde	U+00C3
Auml	U+00C4
Beta	U+0392
Ccedil	U+00C7
Chi	U+03A7
Dagger	U+2021
Delta	U+0394
ETH	U+00D0
Eacute	U+00C9
Ecirc	U+00CA
Egrave	U+00C8
Epsilon	U+0395
Eta	U+0397
Euml	U+00CB
Gamma	U+0393
Iacute	U+00CD
Icirc	U+00CE
Igrave	U+00CC
Iota	U+0399
Iuml	U+00CF
Kappa	U+039A
Lambda	U+039B
Mu	U+039C
Ntilde	U+00D1
Nu	U+039D
OElig	U+0152
Oacute	U+00D3
Ocirc	U+00D4
Ograve	U+00D2

Entity Name	Character
Omega	U+03A9
Omicron	U+039F
Oslash	U+00D8
Otilde	U+00D5
Ouml	U+00D6
Phi	U+03A6
Pi	U+03A0
Prime	U+2033
Psi	U+03A8
Rho	U+03A1
Scaron	U+0160
Sigma	U+03A3
THORN	U+00DE
Tau	U+03A4
Theta	U+0398
Uacute	U+00DA
Ucirc	U+00DB
Ugrave	U+00D9
Upsilon	U+03A5
Uuml	U+00DC
Xi	U+039E
Yacute	U+00DD
Yuml	U+0178
Zeta	U+0396
aacute	U+00E1
acirc	U+00E2
acute	U+00B4
aelig	U+00E6
agrave	U+00E0
alefsym	U+2135
alpha	U+03B1
amp	U+0026
AMP	U+0026
and	U+2227
ang	U+2220
apos	U+0027
aring	U+00E5

Entity Name	Character
asymp	U+2248
atilde	U+00E3
auml	U+00E4
bdquo	U+201E
beta	U+03B2
brvbar	U+00A6
bull	U+2022
cap	U+2229
ccedil	U+00E7
cedil	U+00B8
cent	U+00A2
chi	U+03C7
circ	U+02C6
clubs	U+2663
cong	U+2245
copy	U+00A9
COPY	U+00A9
crarr	U+21B5
cup	U+222A
curren	U+00A4
dArr	U+21D3
dagger	U+2020
darr	U+2193
deg	U+00B0
delta	U+03B4
diams	U+2666
divide	U+00F7
eacute	U+00E9
ecirc	U+00EA
egrave	U+00E8
empty	U+2205
emsp	U+2003
ensp	U+2002
epsilon	U+03B5
equiv	U+2261
eta	U+03B7
eth	U+00F0

Entity Name	Character
euml	U+00EB
euro	U+20AC
exist	U+2203
fnof	U+0192
forall	U+2200
frac12	U+00BD
frac14	U+00BC
frac34	U+00BE
frasl	U+2044
gamma	U+03B3
ge	U+2265
gt	U+003E
GT	U+003E
hArr	U+21D4
harr	U+2194
hearts	U+2665
hellip	U+2026
iacute	U+00ED
icirc	U+00EE
iexcl	U+00A1
igrave	U+00EC
image	U+2111
infin	U+221E
int	U+222B
iota	U+03B9
iquest	U+00BF
isin	U+2208
iuml	U+00EF
kappa	U+03BA
lArr	U+21D0
lambda	U+03BB
lang	U+2329
laquo	U+00AB
larr	U+2190
lceil	U+2308
ldquo	U+201C
le	U+2264

Entity Name	Character
lfloor	U+230A
lowast	U+2217
loz	U+25CA
lrm	U+200E
lsaquo	U+2039
lsquo	U+2018
lt	U+003C
LT	U+003C
macr	U+00AF
mdash	U+2014
micro	U+00B5
middot	U+00B7
minus	U+2212
mu	U+03BC
nabla	U+2207
nbsp	U+00A0
ndash	U+2013
ne	U+2260
ni	U+220B
not	U+00AC
notin	U+2209
nsub	U+2284
ntilde	U+00F1
nu	U+03BD
oacute	U+00F3
ocirc	U+00F4
oelig	U+0153
ograve	U+00F2
oline	U+203E
omega	U+03C9
omicron	U+03BF
oplus	U+2295
or	U+2228
ordf	U+00AA
ordm	U+00BA
oslash	U+00F8
otilde	U+00F5

Entity Name	Character
otimes	U+2297
ouml	U+00F6
para	U+00B6
part	U+2202
permil	U+2030
perp	U+22A5
phi	U+03C6
pi	U+03C0
piv	U+03D6
plusmn	U+00B1
pound	U+00A3
prime	U+2032
prod	U+220F
prop	U+221D
psi	U+03C8
quot	U+0022
QUOT	U+0022
rArr	U+21D2
radic	U+221A
rang	U+232A
raquo	U+00BB
rarr	U+2192
rceil	U+2309
rdquo	U+201D
real	U+211C
reg	U+00AE
REG	U+00AE
rfloor	U+230B
rho	U+03C1
rlm	U+200F
rsaquo	U+203A
rsquo	U+2019
sbquo	U+201A
scaron	U+0161
sdot	U+22C5
sect	U+00A7
shy	U+00AD

Entity Name	Character
sigma	U+03C3
sigmaf	U+03C2
sim	U+223C
spades	U+2660
sub	U+2282
sube	U+2286
sum	U+2211
sup	U+2283
sup1	U+00B9
sup2	U+00B2
sup3	U+00B3
supe	U+2287
szlig	U+00DF
tau	U+03C4
there4	U+2234
theta	U+03B8
thetasym	U+03D1
thinsp	U+2009
thorn	U+00FE
tilde	U+02DC
times	U+00D7
trade	U+2122
uArr	U+21D1
uacute	U+00FA
uarr	U+2191
ucirc	U+00FB
ugrave	U+00F9
uml	U+00A8
upsih	U+03D2
upsilon	U+03C5
uuml	U+00FC
weierp	U+2118
xi	U+03BE
yacute	U+00FD
yen	U+00A5
yuml	U+00FF
zeta	U+03B6

Entity Name	Character
zwj	U+200D
zwnj	U+200C

9. Rendering

This section will probably include details on how to render DATAGRID, drag-and-drop, etc, in a visual medium, in concert with CSS. Terms that need to be defined include: **sizing of embedded content**

CSS UAs in visual media must, when scrolling a page to a fragment identifier, align the top of the viewport with the target element's top border edge.

9.1. Rendering and the DOM

This section is wrong. mediaMode will end up on Window, I think. All views implement Window.

Any object implement the `AbstractView` interface must also implement the `MediaModeAbstractView` interface.

```
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` inteface, during event propagation. There is no way currently to enumerate all the views.

10. 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.

10.1. Localisation

If you wish to create localised versions of an HTML application, the best solution is to preprocess the files on the server, and then use HTTP content negotation to serve the appropriate language.

10.2. Declarative 2D vector graphics and animation

Embedding vector graphics into XHTML documents is the domain of SVG.

10.3. Declarative 3D scenes

Embedding 3D imagery into XHTML documents is the domain of X3D, or technologies based on X3D that are namespace-aware.

10.4. Alternate style sheets: the `DocumentStyle` interface

This section describes an extension to the `DocumentStyle` interface introduced in DOM2 Style. [DOM2STYLE]

It is expected that this section will be moved to a W3C CSS working group or WebAPI working group specification in the next few months.

```
// Introduced in DOM Level 2: [DOM2STYLE]
interface DocumentStyle {
  readonly attribute StyleSheetList styleSheets;

  // New in this specification:
           attribute DOMString selectedStyleSheetSet;
  readonly attribute DOMString lastStyleSheetSet;
  readonly attribute DOMString preferredStyleSheetSet;
  readonly attribute DOMStringList styleSheetSets;
  void enableStyleSheetsForSet(in DOMString name);
};
```

Any object implementing the `HTMLDocument` interface must also implement the `DocumentStyle` interface.

For this interface, the `DOMString` values "null" and "the empty string" are distinct, and must not be considered equivalent.

A style sheet is said to **have a title** if the `title` attribute or pseudo-attribute of the DOM node that introduced the style sheet is present and has a non-empty value (i.e. if the `title` attribute of the `StyleSheet` object returned by the `sheet` attribute of the `LinkStyle` interface of that DOM node is neither null nor the empty string).

The new members are defined as follows:

`selectedStyleSheetSet` of type `DOMString`

> This attribute indicates which style sheet set ([HTML4]) is in use. This attribute is live; changing the disabled attribute on style sheets directly will change the value of this attribute.
>
> If all the sheets that are enabled and have a title have the *same* title (by case-sensitive comparisons) then the value of this attribute must be exactly equal to the title of the first

enabled style sheet with a title in the `styleSheets` list. Otherwise, if style sheets from different sets are enabled, then the return value must be null (there is no way to determine what the currently selected style sheet set is in those conditions). Otherwise, either all style sheets that have a title are disabled, or there are no alternate style sheets, and `selectedStyleSheetSet` must return the empty string.

Setting this attribute to the null value must have no effect.

Setting this attribute to a non-null value must call `enableStyleSheetsForSet()` with that value as the function's argument, and set `lastStyleSheetSet` to that value.

From the DOM's perspective, all views have the same `selectedStyleSheetSet`. If a UA supports multiple views with different selected alternate style sheets, then this attribute (and the `StyleSheet` interface's `disabled` attribute) must return and set the value for the default view.

`lastStyleSheetSet` of type `DOMString`, readonly

This property must initially have the value null. Its value changes when the `selectedStyleSheetSet` attribute is set.

`preferredStyleSheetSet` of type `DOMString`, readonly

This attribute must return the preferred style sheet set as set by the author. It is determined from the order of style sheet declarations and the `Default-Style` HTTP headers, as eventually defined elsewhere in this specification. If there is no preferred style sheet set, this attribute must return the empty string. The case of this attribute must exactly match the case given by the author where the preferred style sheet is specified or implied. This attribute must never return null.

`styleSheetSets` of type `DOMStringList`, readonly

This must return the live list of the currently available style sheet sets. This list is constructed by enumerating all the style sheets for this document available to the implementation, in the order they are listed in the `styleSheets` attribute, adding the title of each style sheet with a title to the list, avoiding duplicates by dropping titles that match (case-sensitively) titles that have already been added to the list.

`enableStyleSheetsForSet(`*`name`*`)`, method

Calling this method must change the `disabled` attribute on each `StyleSheet` object with a title attribute with a length greater than 0 in the `styleSheets` attribute, so that all those whose title matches the *name* argument are enabled, and all others are disabled. Title matches must be case-sensitive. Calling this method with the empty string disables all alternate and preferred style sheets (but does not change the state of persistent style sheets, that is those with no title attribute).

Calling this method with a null value must have no effect.

Style sheets that do not have a title are never affected by this method. This method does not change the values of the `lastStyleSheetSet` or `preferredStyleSheetSet` attributes.

10.4.1. Dynamically adding new style sheets

If new style sheets with titles are added to the document, the UA must decide whether or not the style sheets should be initially enabled or not. How this happens depends on the exact state of the document at the time the style sheet is added, as follows.

10.4.1.1. Adding style sheets

First, if the style sheet is a preferred style sheet (it has a title, but is not marked as alternate), and there is no current preferred style sheet (the `preferredStyleSheetSet` attribute is equal to the empty string) then the `preferredStyleSheetSet` attribute is set to the exact value of this style sheet's title. (This changes the preferred style sheet set, which causes further changes — see below.)

Then, for all sheets, if any of the following is true, then the style sheet must be enabled:

- The style sheet has an empty title.
- The `lastStyleSheetSet` is null, and the style sheet's title matches (by case-sensitive match) the value of the `preferredStyleSheetSet` attribute.
- The style sheet's title matches (by case-sensitive match) the value of the `lastStyleSheetSet` attribute.

Otherwise, the style sheet must be disabled.

10.4.1.2. Changing the preferred style sheet set

The first time the preferred style sheet set is set, which is either before any alternate style sheets are seen (e.g. using a "Default-Style" HTTP header), or is the first time a titled, non-alternate style sheet is seen (in the absence of information to the contrary, the first titled non-alternate sheet sets the name of the preferred set), the `preferredStyleSheetSet` attribute's value must be set to the name of that preferred style sheet set. This does not change the `lastStyleSheetSet` attribute.

If the UA has the preferred style sheet set changed, for example if it receives a "Default-Style:" HTTP header after it receives HTTP "Link:" headers implying another preferred style sheet, then the `preferredStyleSheetSet` attribute's value must be changed appropriately, and, if the `lastStyleSheetSet` is null, the `enableStyleSheetsForSet()` method must be called with the new `preferredStyleSheetSet` value. (The `lastStyleSheetSet` attribute is, again, not changed.)

10.4.1.3. Examples

Thus, in the following HTML snippet:

```
<link rel="alternate stylesheet" title="foo" href="a">
<link rel="alternate stylesheet" title="bar" href="b">
<script>
  document.selectedStyleSheetSet = 'foo';
  document.styleSheets[1].disabled = false;
</script>
<link rel="alternate stylesheet" title="foo" href="c">
<link rel="alternate stylesheet" title="bar" href="d">
```

...the style sheets that end up enabled are style sheets "a", "b", and "c", the `selectedStyleSheetSet` attribute would return null, `lastStyleSheetSet` would return "foo", and `preferredStyleSheetSet` would return "".

Similarly, in the following HTML snippet:

```
<link rel="alternate stylesheet" title="foo" href="a">
<link rel="alternate stylesheet" title="bar" href="b">
```

```
<script>
  var before = document.preferredStyleSheetSet;
  document.styleSheets[1].disabled = false;
</script>
<link rel="stylesheet" title="foo" href="c">
<link rel="alternate stylesheet" title="bar" href="d">
<script>
  var after = document.preferredStyleSheetSet;
</script>
```

...the "before" variable will be equal to the empty string, the "after" variable will be equal to "foo", and style sheets "a" and "c" will be enabled. This is the case even though the first script block sets style sheet "b" to be enabled, because upon parsing the following `<link>` element, the `preferredStyleSheetSet` is set and the `enableStyleSheetsForSet()` method is called (since `selectedStyleSheetSet` was never set explicitly, leaving `lastStyleSheetSet` at null throughout), which changes which style sheets are enabled and which are not.

10.4.2. Interaction with the User Interface

The user interface of Web browsers that support style sheets should list the style sheet titles given in the `styleSheetSets` list, showing the `selectedStyleSheetSet` as the selected style sheet set, leaving none selected if it is null or the empty string, and selecting an extra option "Basic Page Style" (or similar) if it is the empty string and the `preferredStyleSheetSet` is the empty string as well.

Selecting a style sheet from this list should set the `selectedStyleSheetSet` attribute. This (by definition) affects the `lastStyleSheetSet` attribute.

10.4.2.1. Persisting the selected style sheet set

If UAs persist the selected style sheet set, they should use the value of the `selectedStyleSheetSet` attribute, or if that is null, the `lastStyleSheetSet` attribute, when leaving the page (or at some other time) to determine the set name to store. If that is null then the style sheet set should not be persisted.

When re-setting the style sheet set to the persisted value (which can happen at any time, typically at the first time the style sheets are needed for styling the document, after the `<head>` of the document has been parsed, after any scripts that are not dependent on computed style have executed), the style sheet set should be set by setting the `selectedStyleSheetSet` attribute as if the user had selected the set manually.

Note: This specification does not give any suggestions on how UAs should decide to persist the style sheet set or whether or how to persist the selected set across pages.

10.4.3. Future compatibility

Future versions of CSS may introduce ways of having alternate style sheets declared at levels lower than the top level, i.e. embedded within other style sheets. Implementations of this specification that also support this proposed declaration of alternate style sheets are expected to perform depth-first traversals of the `styleSheets` list, not simply enumerations of the `styleSheets` list that only contains the top level.

10.5. Timers

This section is expected to be moved to the Window Object specification in due course.

```
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 langu
  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 lang
  void clearInterval(in long handle);
};

interface TimeoutHandler {
  void handleEvent(arguments...);
};
```

The `WindowTimers` interface must be obtainable from any `Window` object using binding-specific casting methods.

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 invoke `handleEvent()` on the *handler* object. If any *arguments...* were provided, they must be passed to the *handler* as arguments to the `handleEvent()` function.

In the ECMAScript DOM binding, the ECMAScript native `Function` type must implement the `TimeoutHandler` interface such that invoking the `handleEvent()` method of that interface on the object from another language binding invokes the function itself, with the arguments passed to `handleEvent()` as the arguments passed to the function. In the ECMAScript DOM binding itself, however, the `handleEvent()` method of the interface is not directly accessible on `Function` objects. Such functions must be called in the global scope.

Alternatively, **`setTimeout(`*`code`*`, `*`timeout`*`[, `*`language`*`])`** may be used. This variant takes a string instead of a `TimeoutHandler` object. That string must be parsed using the specified *language* (defaulting to ECMAScript if the third argument is omitted) and executed in the global scope.

Need to define *language* values.

The **`setInterval(...)`** variants must work in the same way as the `setTimeout` variants except that the *handler* or `code` must be invoked 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.

Timeouts must never fire while another script is executing. (Thus the HTML scripting model is strictly single-threaded and not reentrant.)

References

This section will be written in a future draft.

Acknowledgements

Thanks to Aankhen, Aaron Leventhal, Alexey Feldgendler, Anne van Kesteren, Anthony Hickson, Asbjørn Ulsberg, Ben Godfrey, Ben Meadowcroft, Benjamin Hawkes-Lewis, Bjoern Hoehrmann, Boris Zbarsky, Brad Fults, Brad Neuberg, Brendan Eich, Brett Wilson, Channy Yun, Charl van Niekerk, Christian Biesinger, Christian Johansen, Chriswa, Daniel Peng, Darin Fisher, Dave Townsend, David Baron, David Flanagan, David Hyatt, Derek Featherstone, Dimitri Glazkov, dolphinling, Doron Rosenberg, Eira Monstad, Elliotte Harold, Erik Arvidsson, fantasai, Franck 'Shift' Quélain, Håkon Wium Lie, Henri Sivonen, Henrik Lied, Ignacio Javier, J. King, James Graham, James M Snell, James Perrett, Jan-Klaas Kollhof, Jasper Bryant-Greene, Jens Bannmann, Joel Spolsky, Johnny Stenback, Jon Perlow, Jonathan Worent, Jukka K. Korpela, Kai Hendry, Kornel Lesinski, Lachlan Hunt, Larry Page, Laurens Holst, Lenny Domnitser, Léonard Bouchet, Leons Petrazickis, Logan, Maciej Stachowiak, Malcolm Rowe, Mark Nottingham, Mark Schenk, Martijn Wargers, Martin Honnen, Matthew Mastracci, Matthew Raymond, Matthew Thomas, Mattias Waldau, Max Romantschuk, Michael 'Ratt' Iannarelli, Michael A. Nachbaur, Michael Gratton, Michel Fortin, Mihai Şucan, Mike Schinkel, Mike Shaver, Mikko Rantalainen, Neil Deakin, Olav Junker Kjær, Rimantas Liubertas, Robert O'Callahan, Robert Sayre, Roman Ivanov, S. Mike Dierken, Sam Ruby, Shadow2531, Shaun Inman, Simon Pieters, Stephen Ma, Steve Runyon, Steven Garrity, Stewart Brodie, Stuart Parmenter, Tantek Çelik, Thomas Broyer, Thomas O'Connor, Tim Altman, Vladimir Vukićević, William Swanson, and everyone on the WHATWG mailing list for their useful and substantial comments.

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 also the Microsoft blogging community for some ideas, to the attendees of the W3C Workshop on Web Applications and Compound Documents for inspiration, and to the #mrt crew, the #mrt.no crew, and the cabal for their ideas and support.