key information in here too (shift/ctrl, etc), like with mouse events and like with the context menu event.

should we be consistent with?

ignored (maybe? or at least should say they have no label so that they are dropped below), and `select` elements inside `label` elements may need special processing.

HTML 5

Draft Recommendation — 24 July 2008

You can take part in this work. Join the working group's discussion list.
Web designers! We have a FAQ, a forum, and a help mailing list for you!

Multiple-page version:

http://whatwg.org/html5

One-page version:

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

PDF print versions:

A4: http://www.whatwg.org/specs/web-apps/current-work/html5-a4.pdf
Letter: http://www.whatwg.org/specs/web-apps/current-work/html5-letter.pdf

Version history:

Twitter messages (non-editorial changes only): http://twitter.com/WHATWG
Commit-Watchers mailing list: http://lists.whatwg.org/listinfo.cgi/commit-watchers-whatwg.org
Interactive Web interface: http://html5.org/tools/web-apps-tracker
Subversion interface: http://svn.whatwg.org/
HTML diff with the last version in Subversion: http://whatwg.org/specs/web-apps/current-work/index-diff

Issues:

To send feedback: whatwg@whatwg.org
To view and vote on feedback: http://www.whatwg.org/issues/

Editor:

Ian Hickson, Google, ian@hixie.ch

Abstract

This specification evolves HTML and its related APIs to ease the authoring of Web-based applications. Additions include context menus, a direct-mode graphics canvas, a full duplex client-server communication channel, more semantics, audio and video, various features for offline Web applications, sandboxed iframes, and scoped styling. Heavy emphasis is placed on keeping the language backwards compatible with existing legacy user agents and on keeping user agents backwards compatible with existing legacy documents.

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.

The current focus is in responding to the outstanding feedback. (There is a chart showing current progress.)

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 specification is also being produced by the W3C HTML WG. The two specifications are identical from the table of contents onwards.

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.

Some of the more major known issues are marked like this. There are many other issues that have been raised as well; the issues given in this document are not the only known issues! There are also some spec-wide issues that have not yet been addressed: case-sensitivity is a very poorly handled topic right now, and the firing of events needs to be unified (right now some bubble, some don't, they all use different text to fire events, etc). It would also be nice to unify the rules on downloading content when attributes change (e.g. `src` attributes) - should they initiate downloads when the element immediately, is inserted in the document, when active scripts end, etc. This matters e.g. if an attribute is set twice in a row (does it hit the network twice).

Table of contents

1. Introduction

1.1 Background

This section is non-normative.

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

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

1.2 Scope

This section is non-normative.

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

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

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

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

For sophisticated cross-platform applications, there already exist several proprietary solutions (such as Mozilla's XUL, Adobe's Flash, or Microsoft's Silverlight). 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.

1.3 Relationships to other specifications

1.3.1 Relationship to HTML 4.01 and DOM2 HTML

This section is non-normative.

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

1.3.2 Relationship to XHTML 1.x

This section is non-normative.

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

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

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

1.3.3 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.

1.3.4 Relationship to Web Forms 2.0 and XForms

This section is non-normative.

This specification will eventually supplant Web Forms 2.0. The current Web Forms 2.0 draft can be considered part of this specification for the time being; its features will eventually be merged into this specification. [WF2]

As it stands today, this specification is unrelated and orthognoal to XForms. When the forms features defined

in HTML4 and Web Forms 2.0 are merged into this specification, then the relationship to XForms described in the Web Forms 2.0 draft will apply to this specification. [XForms]

1.3.5 Relationship to XUL, Flash, Silverlight, and other proprietary UI languages

This section is non-normative.

This specification is independent of the various proprietary UI languages that various vendors provide. As an open, vendor-neutral language, HTML provides for a solution to the same problems without the risk of vendor lock-in.

1.4 HTML vs XHTML

This section is non-normative.

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

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

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

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

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

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

1.5 Structure of this specification

This section is non-normative.

This specification is divided into the following major sections:

Common Infrastructure

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

The DOM

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

Elements

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

Web Browsers

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

User Interaction

HTML documents can provide a number of mechanisms for users to interact with and modify content, which are described in this section.

The Communication APIs

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

Repetition Templates

A mechanism to support repeating sections in forms.

The Language Syntax

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

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

1.5.1 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.

1.5.2 Typographic conventions

This is a definition, requirement, or explanation.

Note: This is a note.

> This is an example.

This is an open issue.

⚠Warning! ***This is a warning.***

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

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

Other code fragments are marked up `like this`.

Variables are marked up like *this*.

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

2. Common infrastructure

2.1 Terminology

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

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

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

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

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

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, albeit without letting the user interact with Web pages where that would involve invoking any script.

2.1.1 XML

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

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

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

`xml`

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

`html`

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

svg

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

Attribute names are said to be **XML-compatible** if they match the `Name` production defined in XML, they contain no U+003A COLON (:) characters, and they do not start with three characters "`xml`". [XML]

2.1.2 DOM trees

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

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.

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

2.1.3 Scripting

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

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

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

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

2.1.4 Plugins

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

> One example of a plugin would be a PDF viewer that is instantiated in a browsing context when the user navigates to a PDF file. This would count as a plugin regardless of whether the party that implemented

> the PDF viewer component was the same as that which implemented the user agent itself. However, a PDF viewer application that launches separate from the user agent (as opposed to using the same interface) is not a plugin by this definition.

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

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

2.2 Conformance requirements

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

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

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

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

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

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

Web browsers and other interactive user agents

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

> A conforming XHTML processor would, upon finding an XHTML `script` element in an XML document, execute the script contained in that element. However, if the element is found within 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 labeled 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 entirely) are exempt from supporting the events and DOM interfaces mentioned in this specification. For the parts of this specification that are defined in terms of an events model or in terms of the DOM, such user agents must still act as if events and the DOM were supported.

Note: Scripting can form an integral part of an application. Web browsers that do not support scripting, or that have scripting disabled, might be unable to fully convey the author's intent.

Conformance checkers

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

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

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

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

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

1. ***Criteria that can be expressed in a DTD.***

2. ***Criteria that cannot be expressed by a DTD, but can still be checked by a machine.***

3. ***Criteria that can only be checked by a human.***

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

Data mining tools

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

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

Authoring tools and markup generators

Authoring tools and markup generators must generate conforming documents. Conformance criteria that apply to authors also apply to authoring tools, where appropriate.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

> ***Note: According to the XML specification, XML processors are not guaranteed to process the external DTD subset referenced in the DOCTYPE. This means, for example, that using entity references for characters in XHTML documents is unsafe (except for `<`, `>`, `&`, `&quot;` and `&apos;`).***

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

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

2.2.1 Dependencies

This specification relies on several other underlying specifications.

XML

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

DOM

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

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

ECMAScript

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

Media Queries

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

This specification does not require support of any particular network transport protocols, style sheet language, scripting language, or any of the DOM and WebAPI specifications beyond those described above. However, the language described by this specification is biased towards CSS as the styling language, ECMAScript as the scripting language, and HTTP as the network protocol, and several features assume that those languages and protocols are in use.

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

2.2.2 Features defined in other specifications

this section will be removed at some point

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

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 interface **`DOMTimeStamp`** is defined in DOM3 Core. [DOM3CORE]

The term **activation behavior** is used as defined in the DOM3 Events specification. [DOM3EVENTS] At the time of writing, DOM3 Events hadn't yet been updated to define that phrase.

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

See http://dev.w3.org/cvsweb/~checkout~/csswg/cssom/Overview.html?content-type=text/html;%20charset=utf-8

2.2.3 Common conformance requirements for APIs exposed to JavaScript

This section will eventually be removed in favour of WebIDL.

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

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

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

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

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

2.3 URLs

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

2.3.1 Terminology

A **URL** is a string used to identify a resource. A URL is always associated with a `Document`, either explicitly when the URL is created or defined; or through a DOM node, in which case the associated `Document` is the node's `Document`; or through a script, in which case the associated `Document` is the script's script document context.

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

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

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

2.3.2 Parsing URLs

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

1. Strip leading and trailing space characters from *url*.
2. Parse *url* in the manner defined by RFC 3986, with the following exceptions:
    - Add all characters with codepoints less than or equal to U+0020 or greater than or equal to U+007F to the <unreserved> production.
    - Add the characters U+0022, U+003C, U+003E, U+005B .. U+005E, U+0060, and U+007B .. U+007D to the <unreserved> production.
    - Add a single U+0025 PERCENT SIGN character as a second alternative way of matching the <pct-encoded> production, except when the <pct-encoded> is used in the <reg-name> production.
    - Add the U+0023 NUMBER SIGN character to the characters allowed in the <fragment> production.
3. If *url* doesn't match the <URI-reference> production, even after the above changes are made to the ABNF definitions, then parsing the URL fails with an error. [RFC3986]

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

   <scheme>

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

   <host>

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

   <port>

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

   <hostport>

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

   <path>

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

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

   <query>

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

<fragment>

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

2.3.3 Resolving URLs

Relative URLs are resolved relative to a base URL. The **base URL** of a URL is the absolute URL obtained as follows:

↪ **If the URL to be resolved was passed to an API**

The base URL is the document base URL of the script's script document context.

↪ **If the URL to be resolved is from the value of a content attribute**

The base URL is the *base URI of the element* that the attribute is on, as defined by the XML Base specification, with *the base URI of the document entity* being defined as the document base URL of the `Document` that owns the element.

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

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

↪ **If the URL to be resolved was found in an offline application cache manifest**

The base URL is the URL of the application cache manifest.

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

1. If there is no `base` element that is both a child of the `head` element and has an `href` attribute, then the document base URL is the document's address.
2. Otherwise, let *url* be the value of the `href` attribute of the first such element.
3. Resolve the *url* URL, using the document's address as the base URL (thus, the `base` `href` attribute isn't affect by `xml:base` attributes).
4. The document base URL is the result of the previous step if it was successful; otherwise it is the document's address.

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

1. Let *url* be the URL being resolved.
2. Let *document* be the `Document` associated with *url*.
3. Let *encoding* be the character encoding of *document*.
4. If *encoding* is UTF-16, then change it to UTF-8.

5. Let *base* be the base URL for *url*. (This is an absolute URL.)

6. Parse *url* into its component parts.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14. Return *result*.

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

2.3.4 Dynamic changes to base URLs

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

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

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

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

↪ **If the element is a hyperlink element**

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

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

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

↪ **If the element is a `blockquote`, `q`, `ins`, or `del` element with a `cite` attribute**

If the absolute URL identified by the `cite` attribute is being shown to the user, or if any data derived from that URL is affecting the display, then the it should be reresolved and the UI updated appropriately.

↪ **Otherwise**

The element is not directly affected.

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

2.3.5 Interfaces for URL manipulation

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

```
           attribute DOMString protocol;
           attribute DOMString host;
           attribute DOMString hostname;
```

```
           attribute DOMString port;
           attribute DOMString pathname;
           attribute DOMString search;
           attribute DOMString hash;
```

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

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

The seven URL decomposition attributes have similar requirements.

On getting, if the input fulfills the condition given in the "getter condition" column corresponding to the attribute in the table below, the user agent must return the part of the input URL given in the "component" column, with any prefixes specified in the "prefix" column appropriately added to the start of the string and any suffixes specified in the "suffix" column appropriately added to the end of the string. Otherwise, the attribute must return the empty string.

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

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

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

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

Attribute	Component	Getter Condition	Prefix	Suffix	Setter Preprocessor	Setter Condition
protocol	<scheme>	—	—	U+003A COLON (":")	Remove all trailing U+003A COLON (":") characters	The new value is not the empty string
host	<hostport>	input is hierarchical and uses a server-based naming authority	—	—	—	—
hostname	<host>	input is hierarchical and uses a server-based naming authority	—	—	Remove all leading U+002F SOLIDUS ("/") characters	—
port	<port>	input is hierarchical, uses a server-based naming authority, and contained a <port> component (possibly an empty one)	—	—	Remove any characters in the new value that are not in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE. If the resulting string is empty, set it to a single U+0030 DIGIT ZERO character ('0').	—

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

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

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

2.4 Common microsyntaxes

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

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

2.4.1 Common parser idioms

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

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]

2.4.2 Boolean attributes

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

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

2.4.3 Numbers

2.4.3.1. 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*.

2.4.3.2. Signed integers

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

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

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

2.4.3.3. Real numbers

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

2.4.3.4. Ratios

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

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

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

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

1. If the string is empty, then return nothing and abort these steps.

2. Find a number in the string according to the algorithm below, starting at the start of the string.

3. If the sub-algorithm in step 2 returned nothing or returned an error condition, return nothing and abort these steps.

4. Set *number1* to the number returned by the sub-algorithm in step 2.

5. Starting with the character immediately after the last one examined by the sub-algorithm in step 2, skip 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 guaranteed to be a valid floating point number).
6. Return *number*.

2.4.3.5. Percentages and dimensions

valid positive non-zero integers rules for parsing dimension values (only used by height/width on img, embed, object — lengths in css pixels or percentages)

2.4.3.6. Lists of integers

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

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

1. Let *input* be the string being parsed.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Let *numbers* be an initially empty list of integers. This list will be the result of this algorithm.

4. If there is a character in the string *input* at position *position*, and it is either a U+0020 SPACE, U+002C COMMA, or U+003B SEMICOLON character, then advance *position* to the next character in *input*, or to beyond the end of the string if there are no more characters.
5. If *position* points to beyond the end of *input*, return *numbers* and abort.
6. If the character in the string *input* at position *position* is a U+0020 SPACE, U+002C COMMA, or U+003B SEMICOLON character, then return to step 4.
7. Let *negated* be false.
8. Let *value* be 0.
9. Let *started* be false. This variable is set to true when the parser sees a number or a "−" character.
10. Let *got number* be false. This variable is set to true when the parser sees a number.
11. Let *finished* be false. This variable is set to true to switch parser into a mode where it ignores characters until the next separator.
12. Let *bogus* be false.
13. *Parser:* If the character in the string *input* at position *position* is:

    ↪ **A U+002D HYPHEN-MINUS character**

    Follow these substeps:

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

    ↪ **A character in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE**

    Follow these substeps:

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

    ↪ **A U+0020 SPACE character**

    ↪ **A U+002C COMMA character**

    ↪ **A U+003B SEMICOLON character**

    Follow these substeps:

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

↪ **A U+002E FULL STOP character**

Follow these substeps:

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

↪ **Any other character**

Follow these substeps:

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

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

2.4.4 Dates and times

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

2.4.4.1. 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), optionally the

seconds (which, if included, must consist of another colon, two digits (the integer part of the seconds), and 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.

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

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*, subtracting $timezone_{hours}$ 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*.

2.4.4.2. 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 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 might 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 ≤ $timezone_{hours}$ ≤ 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).

2.4.5 Time offsets

valid time offset, **rules for parsing time offsets**, **time offset serialization rules**; in the format "5d4h3m2s1ms" or "3m 9.2s" or "00:00:00.00" or similar.

2.4.6 Tokens

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

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

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

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

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

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

1. Let *input* be the string being parsed.
2. Let *position* be a pointer into *input*, initially pointing at the start of the string.
3. Let *tokens* be a list of tokens, initially empty.
4. Skip whitespace
5. While *position* is not past the end of *input*:
    1. Collect a sequence of characters that are not space characters.
    2. Add the string collected in the previous step to *tokens*.
    3. Skip whitespace
6. Return *tokens*.

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

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

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

2.4.7 Keywords and enumerated attributes

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

If an enumerated attribute is specified, the attribute's value must be one of the given keywords that are not said to be non-conforming, with no leading or trailing whitespace. The keyword may use any mix of uppercase and lowercase letters.

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

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

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

2.4.8 References

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

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

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

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

3. Return the first element of type *type* that has an `id` or `name` attribute whose value case-insensitively

matches *s*.

2.5 Common DOM interfaces

2.5.1 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 whose content attribute is defined to contain a URL, then on getting, the DOM attribute must resolve the value of the content attribute and return the resulting absolute URL if that was successful, or the empty string otherwise; and on setting, must set the content attribute to the specified literal value. If the content attribute is absent, the DOM attribute must return the default value, if the content attribute has one, or else the empty string.

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

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

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

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

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

If a reflecting DOM attribute is an *unsigned* integer type (`unsigned long`) then, on getting, the content attribute must be parsed according to the rules for parsing unsigned integers, and if that is successful, the resulting value must be returned. If, on the other hand, it 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 the shortest possible string representing the number as a valid non-negative integer in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is an unsigned integer type (`unsigned long`) that is **limited to only positive non-zero numbers**, then the behavior is similar to the previous case, but zero is not allowed. On getting, the content attribute must first be parsed according to the rules for parsing unsigned integers, and if that is successful, the resulting value must be returned. If, on the other hand, it fails, or if the attribute is absent, the default value must be returned instead, or 1 if there is no default value. On setting, if the value is zero, the user agent must fire an `INDEX_SIZE_ERR` exception. Otherwise, the given value must be converted to the shortest possible string representing the number as a valid non-negative integer in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is a floating point number type (`float`) and the content attribute is defined to contain a time offset, then, on getting, the content attribute must be parsed according to the rules for parsing time offsets, and if that is successful, the resulting value, in seconds, must be returned. If that fails, or if the attribute is absent, the default value must be returned, or the not-a-number value (NaN) if there is no default value. On setting, the given value, interpreted as a time offset in seconds, must be converted to a string using the time offset serialization rules, and that string must be used as the new content attribute value.

If a reflecting DOM attribute is a floating point number type (`float`) and it doesn't fall into one of the earlier categories, then, on getting, the content attribute must be parsed according to the rules for parsing floating point number values, and if that is successful, the resulting value must be returned. If, on the other hand, it fails, or if the attribute is absent, the default value must be returned instead, or 0.0 if there is no default value. On setting, the given value must be converted to the shortest possible string representing the number as a valid floating point number 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 `DOMTokenList`, then on getting it must return a `DOMTokenList` object whose underlying string is the element's corresponding content attribute. When the `DOMTokenList` object mutates its underlying string, the content attribute must itself be immediately mutated. When the attribute is absent, then the string represented by the `DOMTokenList` object is the empty string; when the object mutates this empty string, the user agent must first add the corresponding content attribute, and then mutate that attribute instead. `DOMTokenList` attributes are always read-only. The same `DOMTokenList` object must be returned every time for each attribute.

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

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

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

2.5.2 Collections

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

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

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

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

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

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

2.5.2.1. HTMLCollection

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

```
interface HTMLCollection {
  readonly attribute unsigned long length;
  [IndexGetter] Element item(in unsigned long index);
  [NameGetter] 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.

2.5.2.2. HTMLFormControlsCollection

The `HTMLFormControlsCollection` interface represents a collection of form controls.

```
interface HTMLFormControlsCollection {
  readonly attribute unsigned long length;
```

```
  [IndexGetter] HTMLElement item(in unsigned long index);
  [NameGetter] 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.

2.5.2.3. HTMLOptionsCollection

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

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

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

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

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

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

The **`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.

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.

2.5.3 DOMTokenList

The `DOMTokenList` interface represents an interface to an underlying string that consists of an unordered set of unique space-separated tokens.

Which string underlies a particular `DOMTokenList` object is defined when the object is created. It might be a content attribute (e.g. the string that underlies the `classList` object is the `class` attribute), or it might be an anonymous string (e.g. when a `DOMTokenList` object is passed to an author-implemented callback in the `datagrid` APIs).

```
[Stringifies] interface DOMTokenList {
  readonly attribute unsigned long length;
  [IndexGetter] DOMString item(in unsigned long index);
  boolean has(in DOMString token);
  void add(in DOMString token);
  void remove(in DOMString token);
  boolean toggle(in DOMString token);
};
```

The **`length`** attribute must return the number of *unique* tokens that result from splitting the underlying string on spaces.

The **`item(index)`** method must split the underlying string on spaces, sort the resulting list of tokens by Unicode codepoint, remove exact duplicates, and then return the *index*th item in this list. If *index* is equal to or greater than the number of tokens, then the method must return null.

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

1. If the *token* argument contains any space characters, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.

2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.
3. If the token indicated by *token* is one of the tokens in the object's underlying string then return true and stop this algorithm.
4. Otherwise, return false.

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

1. If the *token* argument contains any space characters, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.
2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.
3. If the given *token* is already one of the tokens in the `DOMTokenList` object's underlying string then stop the algorithm.
4. Otherwise, if the `DOMTokenList` object's underlying string is not the empty string and the last character of that string is not a space character, then append a U+0020 SPACE character to the end of that string.
5. Append the value of *token* to the end of the `DOMTokenList` object's underlying string.

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

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

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

1. If the *token* argument contains any space characters, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.
2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.
3. If the given *token* is already one of the tokens in the `DOMTokenList` object's underlying string then remove the given *token* from the underlying string, and stop the algorithm, returning false.
4. Otherwise, if the `DOMTokenList` object's underlying string is not the empty string and the last character of that string is not a space character, then append a U+0020 SPACE character to the end of that string.
5. Append the value of *token* to the end of the `DOMTokenList` object's underlying string.
6. Return true.

Objects implementing the `DOMTokenList` interface must **stringify** to the object's underlying string representation.

2.5.4 DOMStringMap

The `DOMStringMap` interface represents a set of name-value pairs. When a `DOMStringMap` object is instanced, it is associated with three algorithms, one for getting values from names, one for setting names to certain values, and one for deleting names.

The names of the methods on this interface are temporary and will be fixed when the Web IDL / "Language Bindings for DOM Specifications" spec is ready to handle this case.

```
interface DOMStringMap {
  [NameGetter] DOMString XXX1(in DOMString name);
  [NameSetter] void XXX2(in DOMString name, in DOMString value);
  [XXX] boolean XXX3(in DOMString name);
};
```

The **`XXX1(name)`** method must call the algorithm for getting values from names, passing *name* as the name, and must return the corresponding value, or null if *name* has no corresponding value.

The **`XXX2(name, value)`** method must call the algorithm for setting names to certain values, passing *name* as the name and *value* as the value.

The **`XXX3(name)`** method must call the algorithm for deleting names, passing *name* as the name, and must return true.

2.5.5 DOM feature strings

DOM3 Core defines mechanisms for checking for interface support, and for obtaining implementations of interfaces, using feature strings. [DOM3CORE]

A DOM application can use the **`hasFeature(feature, version)`** method of the `DOMImplementation` interface with parameter values "`HTML`" and "`5.0`" (respectively) to determine whether or not this module is supported by the implementation. In addition to the feature string "`HTML`", the feature string "`XHTML`" (with version string "`5.0`") can be used to check if the implementation supports XHTML. User agents should respond with a true value when the `hasFeature` method is queried with these values. Authors are cautioned, however, that UAs returning true might not be perfectly compliant, and that UAs returning false might well have support for features in this specification; in general, therefore, use of this method is discouraged.

The values "`HTML`" and "`XHTML`" (both with version "`5.0`") should also be supported in the context of the `getFeature()` and `isSupported()` methods, as defined by DOM3 Core.

Note: The interfaces defined in this specification are not always supersets of the interfaces defined in DOM2 HTML; some features that were formerly deprecated, poorly supported, rarely used or considered unnecessary have been removed. Therefore it is not guaranteed that an implementation that supports "`HTML`" "`5.0`" also supports "`HTML`" "`2.0`".

2.6 Fetching resources

replace all instances of the word 'fetch' or 'download' with a reference to this section, and put something

here that talks about caching, that redirects to the offline storage stuff when appropriate, that defines that before fetching a URL you have to resolve the URL, so that every case of fetching doesn't have to independently say to resolve the URL, etc; "once fetched, a resource might have to have its type determined", pointing to the next section but also explicitly saying that it's up to the part of the spec doing the fetching to determine how the type is established

2.7 Determining the type of a resource

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

2.7.1 Content-Type metadata

What explicit **Content-Type metadata** is associated with the resource (the resource's type information) depends on the protocol that was used to fetch the resource.

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

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

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

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

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

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

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

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

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

5. Process the next character as follows:

↪ **If it is a U+0022 QUOTATION MARK ("") and there is a later U+0022 QUOTATION MARK ("") in *s***

↪ **If it is a U+0027 APOSTROPHE ('') and there is a later U+0027 APOSTROPHE ('') in *s***

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

↪ **If it is an unmatched U+0022 QUOTATION MARK ("")**

↪ **If it is an unmatched U+0027 APOSTROPHE ('')**

↪ **If there is no next character**

Return nothing.

↪ **Otherwise**

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

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

2.7.2 Content-Type sniffing: Web pages

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

1. Let *official type* be the type given by the Content-Type metadata for the resource (in lowercase, ignoring any parameters). If there is no such type, jump to the *unknown type* step below.

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

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

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

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

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

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

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

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

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

2.7.3 Content-Type sniffing: text or binary

1. The user agent may wait for 512 or more bytes of the resource to be available.
2. Let *n* be the smaller of either 512 or the number of bytes already available.
3. If *n* is 4 or more, and the first bytes of the resource match one of the following byte sets:

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

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

4. If none of the first *n* bytes of the resource are binary data bytes then the sniffed type of the resource is "text/plain". Abort these steps.
5. If the first bytes of the resource match one of the byte sequences in the "pattern" column of the table in the *unknown type* section below, ignoring any rows whose cell in the "security" column says "scriptable" (or "n/a"), then the sniffed type of the resource is the type given in the corresponding cell in the "sniffed type" column on that row; abort these steps.

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

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

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

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

2.7.4 Content-Type sniffing: unknown type

1. The user agent may wait for 512 or more bytes of the resource to be available.
2. Let *stream length* be the smaller of either 512 or the number of bytes already available.
3. For each row in the table below:

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

   1. Let *pattern length* be the length of the pattern (number of bytes described by the cell in the second column of the row).
   2. If *stream length* is smaller than *pattern length* then skip this row.
   3. Apply the "and" operator to the first *pattern length* bytes of the resource and the given mask (the bytes in the cell of first column of that row), and let the result be the *data*.
   4. If the bytes of the *data* matches the given pattern bytes exactly, then the sniffed type

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

↪ **If the row has a "*WS*" byte:**

1. Let $index_{pattern}$ be an index into the mask and pattern byte strings of the row.
2. Let $index_{stream}$ be an index into the byte stream being examined.
3. *Loop*: If $index_{stream}$ points beyond the end of the byte stream, then this row doesn't match, skip this row.
4. Examine the $index_{stream}$th byte of the byte stream as follows:

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

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

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

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

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

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

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

5. If $index_{pattern}$ does not point beyond the end of the mask and pattern byte strings, then jump back to the *loop* step in this algorithm.
6. Otherwise, the sniffed type of the resource is the type given in the cell of the third column in that row; abort these steps.

4. If none of the first *n* bytes of the resource are binary data bytes then the sniffed type of the resource is "text/plain". Abort these steps.
5. Otherwise, the sniffed type of the resource is "application/octet-stream".

The table used by the above algorithm is:

Bytes in Hexadecimal		Sniffed type	Security	Comment
Mask	**Pattern**			
FF FF DF DF DF DF DF DF DF FF DF DF DF DF	3C 21 44 4F 43 54 59 50 45 20 48 54 4D 4C	text/html	Scriptable	The string "`<!DOCTYPE HTML`" in US-ASCII or compatible encodings, case-insensitively.
FF FF DF DF DF DF	*WS* 3C 48 54 4D 4C	text/html	Scriptable	The string "`<HTML`" in US-ASCII or compatible encodings, case-insensitively, possibly with leading spaces.

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

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

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

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

2.7.5 Content-Type sniffing: image

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

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

User agents must ignore any rows for image types that they do not support.

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

2.7.6 Content-Type sniffing: feed or HTML

1. The user agent may wait for 512 or more bytes of the resource to be available.
2. Let *s* be the stream of bytes, and let *s*[*i*] represent the byte in *s* with position *i*, treating *s* as zero-indexed (so the first byte is at *i*=0).
3. If at any point this algorithm requires the user agent to determine the value of a byte in *s* which is not yet available, or which is past the first 512 bytes of the resource, or which is beyond the end of the resource, the user agent must stop this algorithm, and assume that the sniffed type of the resource is "text/html".

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

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

   ↪ **If it is 0x09 (ASCII tab), 0x20 (ASCII space), 0x0A (ASCII LF), or 0x0D (ASCII CR)**

   Increase *pos* by 1 and repeat this step.

   ↪ **If it is 0x3C (ASCII "<")**

   Increase *pos* by 1 and go to the next step.

   ↪ **If it is anything else**

   The sniffed type of the resource is "text/html". Abort these steps.

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

   1. Increase *pos* by 3.
   2. If the bytes with positions *pos* to *pos*+2 in *s* are exactly equal to 0x2D, 0x2D, 0x3E respectively (ASCII for "-->"), then increase *pos* by 3 and jump back to the previous step (the step labeled *loop start*) in the overall algorithm in this section.
   3. Otherwise, increase *pos* by 1.
   4. Return to step 2 in these substeps.

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

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

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

1. Increase *pos* by 1.

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

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

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

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

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

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

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

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

3. Semantics and structure of HTML documents

3.1 Introduction

This section is non-normative.

An introduction to marking up a document.

3.2 Documents

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

3.2.1 Documents in the DOM

`Document` objects are assumed to be **XML documents** unless they are flagged as being **HTML documents** when they are created. Whether a document is an HTML document or an XML document affects the behavior of certain APIs, as well as a few CSS rendering rules. [CSS21]

> ***Note: A `Document` object created by the `createDocument()` API on the `DOMImplementation` object is initially an XML document, but can be made into an HTML document by calling `document.open()` on it.***

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
  [PutForwards=href] readonly attribute Location location;
  readonly attribute DOMString URL;
           attribute DOMString domain;
  readonly attribute DOMString referrer;
           attribute DOMString cookie;
  readonly attribute DOMString lastModified;
  readonly attribute DOMString compatMode;
           attribute DOMString charset;
  readonly attribute DOMString characterSet;
  readonly attribute DOMString defaultCharset;
```

```
  readonly attribute DOMString readyState;

  // DOM tree accessors
           attribute DOMString title;
           attribute DOMString dir;
           attribute HTMLElement body;
  readonly attribute HTMLCollection images;
  readonly attribute HTMLCollection embeds;
  readonly attribute HTMLCollection plugins;
  readonly attribute HTMLCollection links;
  readonly attribute HTMLCollection forms;
  readonly attribute HTMLCollection anchors;
  readonly attribute HTMLCollection scripts;
  NodeList getElementsByName(in DOMString elementName);
  NodeList getElementsByClassName(in DOMString classNames);

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

  // user interaction
  Selection getSelection();
  readonly attribute Element activeElement;
  boolean hasFocus();
           attribute boolean designMode;
  boolean execCommand(in DOMString commandId);
  boolean execCommand(in DOMString commandId, in boolean showUI);
  boolean execCommand(in DOMString commandId, in boolean showUI, in
DOMString value);
  boolean queryCommandEnabled(in DOMString commandId);
  boolean queryCommandIndeterm(in DOMString commandId);
  boolean queryCommandState(in DOMString commandId);
  boolean queryCommandSupported(in DOMString commandId);
  DOMString queryCommandValue(in DOMString commandId);
  readonly attribute HTMLCollection commands;
};
```

Since the `HTMLDocument` interface holds methods and attributes related to a number of disparate features, the members of this interface are described in various different sections.

3.2.2 Security

User agents must raise a security exception whenever any of the members of an `HTMLDocument` object are accessed by scripts whose effective script origin is not the same as the `Document`'s effective script origin.

3.2.3 Resource metadata management

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

The **`referrer`** attribute must return either the address of the active document of the source browsing context at the time the navigation was started (that is, the page which navigated the browsing context to the current document), or the empty string if there is no such originating page, or if the UA has been configured not to report referrers in this case, or if the navigation was initiated for a hyperlink with a `noreferrer` keyword.

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

> ***Note: Typically user agents are configured to not report referrers in the case where the referrer uses an encrypted protocol and the current page does not (e.g. when navigating from an `https:` page to an `http:` page).***

The **`cookie`** attribute represents the cookies of the resource.

On getting, if the sandboxed origin browsing context flag is set on the browsing context of the document, the user agent must raise a security exception. Otherwise, it must return the same string as the value of the `Cookie` HTTP header it would include if fetching the resource indicated by the document's address over HTTP, as per RFC 2109 section 4.3.4 or later specifications. [RFC2109] [RFC2965]

On setting, if the sandboxed origin browsing context flag is set on the browsing context of the document, the user agent must raise a security exception. Otherwise, the user agent must act as it would when processing cookies if it had just attempted to fetch the document's address over HTTP, and had received a response with a `Set-Cookie` header whose value was the specified value, as per RFC 2109 sections 4.3.1, 4.3.2, and 4.3.3 or later specifications. [RFC2109] [RFC2965]

> ***Note: Since the `cookie` attribute is accessible across frames, the path restrictions on cookies are only a tool to help manage which cookies are sent to which parts of the site, and are not in any way a security feature.***

The **`lastModified`** attribute, on getting, must return the date and time of the `Document`'s source file's last modification, in the user's local timezone, in the following format:

1. The month component of the date.
2. A U+002F SOLIDUS character ('/').
3. The day component of the date.
4. A U+002F SOLIDUS character ('/').
5. The year component of the date.

6. A U+0020 SPACE character.
7. The hours component of the time.
8. A U+003A COLON character (':').
9. The minutes component of the time.
10. A U+003A COLON character (':').
11. The seconds component of the time.

All the numeric components above, other than the year, must be given as two digits in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE representing the number in base ten, zero-padded if necessary.

The `Document`'s source file's last modification date and time must be derived from relevant features of the networking protocols used, e.g. from the value of the HTTP `Last-Modified` header of the document, or from metadata in the file system for local files. If the last modification date and time are not known, the attribute must return the string `01/01/1970 00:00:00`.

A `Document` is always set to one of three modes: **no quirks mode**, the default; **quirks mode**, used typically for legacy documents; and **limited quirks mode**, also known as "almost standards" mode. The mode is only ever changed from the default by the HTML parser, based on the presence, absence, or value of the DOCTYPE string.

The **`compatMode`** DOM attribute must return the literal string "`CSS1Compat`" unless the document has been set to quirks mode by the HTML parser, in which case it must instead return the literal string "`BackCompat`".

As far as parsing goes, the quirks I know of are:

- Comment parsing is different.
- `p` can contain `table`
- Safari and IE have special parsing rules for <% ... %> (even in standards mode, though clearly this should be quirks-only).

Documents have an associated **character encoding**. When a `Document` object is created, the document's character encoding must be initialized to UTF-16. Various algorithms during page loading affect this value, as does the `charset` setter. [IANACHARSET]

The **`charset`** DOM attribute must, on getting, return the preferred MIME name of the document's character encoding. On setting, if the new value is an IANA-registered alias for a character encoding, the document's character encoding must be set to that character encoding. (Otherwise, nothing happens.)

The **`characterSet`** DOM attribute must, on getting, return the preferred MIME name of the document's character encoding.

The **`defaultCharset`** DOM attribute must, on getting, return the preferred MIME name of a character encoding, possibly the user's default encoding, or an encoding associated with the user's current geographical location, or any arbitrary encoding name.

Each document has a **current document readiness**. When a `Document` object is created, it must have its current document readiness set to the string "loading". Various algorithms during page loading affect this value. When the value is set, the user agent must fire a simple event called `readystatechanged` at the `Document` object.

The **`readyState`** DOM attribute must, on getting, return the current document readiness.

3.2.4 DOM tree accessors

The `html` element of a document is the document's root element, if there is one and it's an `html` element, or null otherwise.

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

The `title` element of a document is the first `title` element in the document (in tree order), if there is one, or null otherwise.

The **`title`** attribute must, on getting, run the following algorithm:

1. If the root element is an `svg` element in the "`http://www.w3.org/2000/svg`" namespace, and the user agent supports SVG, then the getter must return the value that would have been returned by the DOM attribute of the same name on the `SVGDocument` interface.
2. Otherwise, it must return a concatenation of the data of all the child text nodes of the `title` element, in tree order, or the empty string if the `title` element is null.

On setting, the following algorithm must be run:

1. If the root element is an `svg` element in the "`http://www.w3.org/2000/svg`" namespace, and the user agent supports SVG, then the setter must defer to the setter for the DOM attribute of the same name on the `SVGDocument` interface. Stop the algorithm here.
2. If the `title` element is null and the `head` element is null, then the attribute must do nothing. Stop the algorithm here.
3. If the `title` element is null, then a new `title` element must be created and appended to the `head` element.
4. The children of the `title` element (if any) must all be removed.
5. A single `Text` node whose data is the new value being assigned must be appended to the `title` element.

The `title` attribute on the `HTMLDocument` interface should shadow the attribute of the same name on the `SVGDocument` interface when the user agent supports both HTML and SVG.

The body element of a document is the first child of the `html` element that is either a `body` element or a `frameset` element. If there is no such element, it is null. If the body element is null, then when the specification requires that events be fired at "the body element", they must instead be fired at the `Document` object.

The **`body`** attribute, on getting, must return the body element of the document (either a `body` element, a `frameset` element, or null). On setting, the following algorithm must be run:

1. If the new value is not a `body` or `frameset` element, then raise a `HIERARCHY_REQUEST_ERR` exception and abort these steps.
2. Otherwise, if the new value is the same as the body element, do nothing. Abort these steps.
3. Otherwise, if the body element is not null, then replace that element with the new value in the DOM, as if the root element's `replaceChild()` method had been called with the new value and the incumbent body element as its two arguments respectively, then abort these steps.
4. Otherwise, the the body element is null. Append the new value to the root element.

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

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

The **`plugins`** attribute must return the same object as that returned by the `embeds` attribute.

The **`links`** attribute must return an `HTMLCollection` rooted at the `Document` node, whose filter matches only `a` elements with `href` attributes and `area` elements with `href` attributes.

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

The **`anchors`** attribute must return an `HTMLCollection` rooted at the `Document` node, whose filter matches only `a` elements with `name` attributes.

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

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 a string that contains an unordered set of unique space-separated tokens representing classes. When called, the method must return a live `NodeList` object containing all the elements in the document that have all the classes specified in that argument, having obtained the classes by splitting a string on spaces. If there are no tokens specified in the argument, then the method must return an empty `NodeList`.

The **`getElementsByClassName()`** method on the `HTMLElement` interface must return a live `NodeList` with the nodes that the `HTMLDocument` `getElementsByClassName()` method would return when passed the same argument(s), excluding any elements that are not descendants of the `HTMLElement` object on which the method was invoked.

HTML, SVG, and MathML elements define which classes they are in by having an attribute in the per-element partition with the name `class` containing a space-separated list of classes to which the element belongs. Other specifications may also allow elements in their namespaces to be labeled as being in specific classes.

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('bbb  ccc ')` would return the same thing.
>
> A call to `getElementsByClassName('aaa,bbb')` would return no nodes; none of the elements above are in the "aaa,bbb" class.

Note: The `dir` attribute on the `HTMLDocument` interface is defined along with the `dir` content attribute.

3.3 Elements

3.3.1 Semantics

Elements, attributes, and attribute values in HTML are defined (by this specification) to have certain meanings (semantics). For example, the `ol` element represents an ordered list, and the `lang` attribute represents the language of the content.

Authors must not use elements, attributes, and attribute values for purposes other than their appropriate intended semantic purpose.

> For example, the following document is non-conforming, despite being syntactically correct:
>
> ```
> <!DOCTYPE html>
> <html lang="en-GB">
>  <head> <title> Demonstration </title> </head>
>  <body>
>   <table>
>    <tr> <td> My favourite animal is the cat. </td> </tr>
>    <tr>
>     <td>
>      —<a href="http://example.org/~ernest/"><cite>Ernest</cite></a>,
>      in an essay from 1992
>     </td>
>    </tr>
> ```

```
  </table>
 </body>
</html>
```

...because the data placed in the cells is clearly not tabular data (and the `cite` element mis-used). A corrected version of this document might be:

```
<!DOCTYPE html>
<html lang="en-GB">
 <head> <title> Demonstration </title> </head>
 <body>
  <blockquote>
   <p> My favourite animal is the cat. </p>
  </blockquote>
  <p>
   —<a href="http://example.org/~ernest/">Ernest</a>,
   in an essay from 1992
  </p>
 </body>
</html>
```

This next document fragment, intended to represent the heading of a corporate site, is similarly non-conforming because the second line is not intended to be a heading of a subsection, but merely a subheading or subtitle (a subordinate heading for the same section).

```
<body>
 <h1>ABC Company</h1>
 <h2>Leading the way in widget design since 1432</h2>
 ...
```

The `header` element should be used in these kinds of situations:

```
<body>
 <header>
  <h1>ABC Company</h1>
  <h2>Leading the way in widget design since 1432</h2>
 </header>
 ...
```

Through scripting and using other mechanisms, the values of attributes, text, and indeed the entire structure of the document may change dynamically while a user agent is processing it. The semantics of a document at an instant in time are those represented by the state of the document at that instant in time, and the semantics of a document can therefore change over time. User agents must update their presentation of the document as this occurs.

> HTML has a `progress` element that describes a progress bar. If its "value" attribute is dynamically updated by a script, the UA would update the rendering to show the progress changing.

3.3.2 Elements in the DOM

The nodes representing HTML elements in the DOM must implement, and expose to scripts, the interfaces

listed for them in the relevant sections of this specification. This includes HTML elements in XML documents, even when those documents are in another context (e.g. inside an XSLT transform).

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

> For example, an `ol` element represents an ordered list.

The basic interface, from which all the HTML elements' interfaces inherit, and which must be used by elements that have no additional requirements, is the `HTMLElement` interface.

```
interface HTMLElement : Element {
  // DOM tree accessors
  NodeList getElementsByClassName(in DOMString classNames);

  // dynamic markup insertion
           attribute DOMString innerHTML;

  // metadata attributes
           attribute DOMString id;
           attribute DOMString title;
           attribute DOMString lang;
           attribute DOMString dir;
           attribute DOMString className;
  readonly attribute DOMTokenList classList;
  readonly attribute DOMStringMap dataset;

  // user interaction
           attribute boolean irrelevant;
  void click();
  void scrollIntoView();
  void scrollIntoView(in boolean top);
           attribute long tabIndex;
  void focus();
  void blur();
           attribute boolean draggable;
           attribute DOMString contentEditable;
  readonly attribute DOMString isContentEditable;
           attribute HTMLMenuElement contextMenu;

  // styling
  readonly attribute CSSStyleDeclaration style;

  // data templates
           attribute DOMString template;
  readonly attribute HTMLDataTemplateElement templateElement;
           attribute DOMString ref;
  readonly attribute Node refNode;
           attribute DOMString registrationMark;
  readonly attribute DocumentFragment originalContent;
```

```
  // event handler DOM attributes
           attribute EventListener onabort;
           attribute EventListener onbeforeunload;
           attribute EventListener onblur;
           attribute EventListener onchange;
           attribute EventListener onclick;
           attribute EventListener oncontextmenu;
           attribute EventListener ondblclick;
           attribute EventListener ondrag;
           attribute EventListener ondragend;
           attribute EventListener ondragenter;
           attribute EventListener ondragleave;
           attribute EventListener ondragover;
           attribute EventListener ondragstart;
           attribute EventListener ondrop;
           attribute EventListener onerror;
           attribute EventListener onfocus;
           attribute EventListener onkeydown;
           attribute EventListener onkeypress;
           attribute EventListener onkeyup;
           attribute EventListener onload;
           attribute EventListener onmessage;
           attribute EventListener onmousedown;
           attribute EventListener onmousemove;
           attribute EventListener onmouseout;
           attribute EventListener onmouseover;
           attribute EventListener onmouseup;
           attribute EventListener onmousewheel;
           attribute EventListener onresize;
           attribute EventListener onscroll;
           attribute EventListener onselect;
           attribute EventListener onstorage;
           attribute EventListener onsubmit;
           attribute EventListener onunload;

};
```

The `HTMLElement` interface holds methods and attributes related to a number of disparate features, and the members of this interface are therefore described in various different sections of this specification.

3.3.3 Global attributes

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

Global attributes:

`class`
`contenteditable`
`contextmenu`

dir
draggable
id
irrelevant
lang
ref
registrationmark
style
tabindex
template
title

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

Event handler content attributes:

onabort
onbeforeunload
onblur
onchange
onclick
oncontextmenu
ondblclick
ondrag
ondragend
ondragenter
ondragleave
ondragover
ondragstart
ondrop
onerror
onfocus
onkeydown
onkeypress
onkeyup
onload
onmessage
onmousedown
onmousemove
onmouseout
onmouseover
onmouseup
onmousewheel
onresize
onscroll
onselect
onstorage
onsubmit

```
onunload
```

Also, custom data attributes (e.g. `data-foldername` or `data-msgid`) can be specified on any HTML element, to store custom data specific to the page.

In HTML documents, elements in the HTML namespace may have an `xmlns` attribute specified, if, and only if, it has the exact value "`http://www.w3.org/1999/xhtml`". This does not apply to XML documents.

> ***Note: In HTML, the `xmlns` attribute has absolutely no effect. It is basically a talisman. It is allowed merely to make migration to and from XHTML mildly easier. When parsed by an HTML parser, the attribute ends up in no namespace, not the "`http://www.w3.org/2000/xmlns/`" namespace like namespace declaration attributes in XML do.***

> ***Note: In XML, an `xmlns` attribute is part of the namespace declaration mechanism, and an element cannot actually have an `xmlns` attribute in no namespace specified.***

3.3.3.1. The `id` attribute

The `id` attribute represents its element's unique identifier. The value must be unique in the subtree within which the element finds itself and must contain at least one character. The value must not contain any space characters.

If the value is not the empty string, user agents must associate the element with the given value (exactly, including any space characters) for the purposes of ID matching within the 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.

3.3.3.2. The `title` attribute

The `title` attribute represents advisory information for the element, such as would be appropriate for a tooltip. On a link, this could be the title or a description of the target resource; on an image, it could be the image credit or a description of the image; on a paragraph, it could be a footnote or commentary on the text; on a citation, it could be further information about the source; and so forth. The value is text.

If this attribute is omitted from an element, then it implies that the `title` attribute of the nearest ancestor HTML element with a `title` attribute set is also relevant to this element. Setting the attribute overrides this, explicitly stating that the advisory information of any ancestors is not relevant to this element. Setting the attribute to the empty string indicates that the element has no advisory information.

If the `title` attribute's value contains U+000A LINE FEED (LF) characters, the content is split into multiple lines. Each U+000A LINE FEED (LF) character represents a line break.

Some elements, such as `link` and `abbr`, define additional semantics for the `title` attribute beyond the

semantics described above.

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

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

The `xml:lang` attribute may 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 an `xml:lang` attribute set or is an HTML element and has a `lang` attribute set. That attribute specifies the language of the node.

If both the `xml:lang` attribute and the `lang` attribute are set on an element, user agents must use the `xml:lang` attribute, and the `lang` attribute must be ignored for the purposes of determining the element's language.

If no explicit language is given for the root element, 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 pronunciations, or for dictionary selection).

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

3.3.3.4. The `xml:base` attribute (XML only)

The `xml:base` attribute is defined in XML Base. [XMLBASE]

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

3.3.3.5. The `dir` attribute

The `dir` attribute specifies the element's text directionality. The attribute is an enumerated attribute with the keyword `ltr` mapping to the state *ltr*, and the keyword `rtl` mapping to the state *rtl*. The attribute has no defaults.

If the attribute has the state *ltr*, the element's directionality is left-to-right. If the attribute has the state *rtl*, the element's directionality is right-to-left. Otherwise, the element's directionality is the same as its parent element, or *ltr* if there is no parent element.

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 properties.

The **`dir`** DOM attribute on an element must reflect the `dir` content attribute of that element, limited to only known values.

The **`dir`** DOM attribute on `HTMLDocument` objects must reflect the `dir` content attribute of the `html` element, if any, limited to only known values. If there is no such element, then the attribute must return the empty string and do nothing on setting.

3.3.3.6. The `class` attribute

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

The attribute, if specified, must have a value that is an unordered set of unique space-separated tokens representing the various classes that the element belongs to.

The classes that an HTML element has assigned to it consists of all the classes returned when the value of the `class` attribute is split on spaces.

> ***Note: Assigning classes to an element affects class matching in selectors in CSS, the `getElementsByClassName()` method in the DOM, and other such features.***

Authors may use any value in the `class` attribute, but are encouraged to use the values that describe the nature of the content, rather than values that describe the desired presentation of the content.

The **`className`** and **`classList`** DOM attributes must both reflect the `class` content attribute.

3.3.3.7. The `style` attribute

All elements may have the `style` content attribute set. If specified, the attribute must contain only a list of zero or more semicolon-separated (;) CSS declarations. [CSS21]

The attribute, if specified, must be parsed and treated as the body (the part inside the curly brackets) of a declaration block in a rule whose selector matches just the element on which the attribute is set. For the purposes of the CSS cascade, the attribute must be considered to be a 'style' attribute at the author level.

Documents that use `style` attributes on any of their elements must still be comprehensible and usable if those attributes were removed.

> ***Note: In particular, using the `style` attribute to hide and show content, or to convey meaning that is otherwise not included in the document, is non-conforming.***

The **`style`** DOM attribute must return a `CSSStyleDeclaration` whose value represents the declarations specified in the attribute, if present. Mutating the `CSSStyleDeclaration` object must create a `style`

attribute on the element (if there isn't one already) and then change its value to be a value representing the serialized form of the `CSSStyleDeclaration` object. [CSSOM]

> In the following example, the words that refer to colors are marked up using the `span` element and the `style` attribute to make those words show up in the relevant colors in visual media.
>
> ```
> My sweat suit is <span style="color: green; background:
> transparent">green and my eyes are <span style="color: blue;
> background: transparent">blue.
> ```

3.3.3.8. Embedding custom non-visible data

A **custom data attribute** is an attribute whose name starts with the string "**`data-`**", has at least one character after the hyphen, is XML-compatible, and has no namespace.

Custom data attributes are intended to store custom data private to the page or application, for which there are no more appropriate attributes or elements.

Every HTML element may have any number of custom data attributes specified, with any value.

The **`dataset`** DOM attribute provides convenient accessors for all the `data-*` attributes on an element. On getting, the `dataset` DOM attribute must return a `DOMStringMap` object, associated with the following three algorithms, which expose these attributes on their element:

The algorithm for getting values from names

1. Let *name* be the concatenation of the string `data-` and the name passed to the algorithm.
2. If the element does not have an attribute with the name *name*, then the name has no corresponding value, abort.
3. Otherwise, return the value of the attribute with the name *name*.

The algorithm for setting names to certain values

1. Let *name* be the concatenation of the string `data-` and the name passed to the algorithm.
2. Let *value* be the value passed to the algorithm.
3. Set the value of the attribute with the name *name*, to the value *value*, replacing any previous value if the attribute already existed. If `setAttribute()` would have raised an exception when setting an attribute with the name *name*, then this must raise the same exception.

The algorithm for deleting names

1. Let *name* be the concatenation of the string `data-` and the name passed to the algorithm.
2. Remove the attribute with the name *name*, if such an attribute exists. Do nothing otherwise.

> If a Web page wanted an element to represent a space ship, e.g. as part of a game, it would have to use the `class` attribute along with `data-*` attributes:
>
> ```
> <div class="spaceship" data-id="92432"
> ```

```
        data-weapons="laser 2" data-shields="50%"
        data-x="30" data-y="10" data-z="90">
    <button class="fire"
            onclick="spaceships[this.parentNode.dataset.id].fire()">
     Fire
    </button>
   </div>
```

Authors should carefully design such extensions so that when the attributes are ignored and any associated CSS dropped, the page is still usable.

User agents must not derive any implementation behavior from these attributes or values. Specifications intended for user agents must not define these attributes to have any meaningful values.

3.4 Content models

All the elements in this specification have a defined content model, which describes what nodes are allowed inside the elements, and thus what the structure of an HTML document or fragment must look like.

> ***Note: As noted in the conformance and terminology sections, for the purposes of determining if an element matches its content model or not, `CDATASection` nodes in the DOM are treated as equivalent to `Text` nodes, and entity reference nodes are treated as if they were expanded in place.***

The space characters are always allowed between elements. User agents represent these characters between elements in the source markup as text nodes in the DOM. Empty text nodes and text nodes consisting of just sequences of those characters are considered **inter-element whitespace**.

Inter-element whitespace, comment nodes, and processing instruction nodes must be ignored when establishing whether an element matches its content model or not, and must be ignored when following algorithms that define document and element semantics.

An element *A* is said to be **preceded or followed** by a second element *B* if *A* and *B* have the same parent node and there are no other element nodes or text nodes (other than inter-element whitespace) between them.

Authors must not use elements in the HTML namespace anywhere except where they are explicitly allowed, as defined for each element, or as explicitly required by other specifications. For XML compound documents, these contexts could be inside elements from other namespaces, if those elements are defined as providing the relevant contexts.

> The 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. [SVG]

> The Atom specification defines the Atom `content` element, when its `type` attribute has the value `xhtml`, as requiring that it contains a single HTML `div` element. Thus, a `div` element is allowed in that context, even though this is not explicitly normatively stated by this specification. [ATOM]

In addition, elements in the HTML namespace may be orphan nodes (i.e. without a parent node).

> For example, creating a `td` element and storing it in a global variable in a script is conforming, even though `td` elements are otherwise only supposed to be used inside `tr` elements.
>
> ```
> var data = {
>   name: "Banana",
>   cell: document.createElement('td'),
> };
> ```

3.4.1 Kinds of content

Each element in HTML falls into zero or more categories that group elements with similar characteristics together. The following categories are used in this specification:

- Metadata content
- Flow content
- Sectioning content
- Heading content
- Phrasing content
- Embedded content
- Form control content
- Interactive content

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

3.4.1.1. Metadata content

Metadata content is content that sets up the presentation or behavior of the rest of the content, or that sets up the relationship of the document with other documents, or that conveys other "out of band" information.

Elements from other namespaces whose semantics are primarily metadata-related (e.g. RDF) are also metadata content.

3.4.1.2. Flow content

Most elements that are used in the body of documents and applications are categorized as **flow content**.

As a general rule, elements whose content model allows any flow content should have either at least one descendant text node that is not inter-element whitespace, or at least one descendant element node that is embedded content. For the purposes of this requirement, `del` elements and their descendants must not be counted as contributing to the ancestors of the `del` element.

This requirement is not a hard requirement, however, as there are many cases where an element can be empty legitimately, for example when it is used as a placeholder which will later be filled in by a script, or when the element is part of a template and would on most pages be filled in but on some pages is not relevant.

3.4.1.3. Sectioning content

Sectioning content is content that defines the scope of headers, footers, and contact information.

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

3.4.1.4. Heading content

Heading content defines the header of a section (whether explicitly marked up using sectioning content elements, or implied by the heading content itself).

3.4.1.5. Phrasing content

Phrasing content is the text of the document, as well as elements that mark up that text at the intra-paragraph level. Runs of phrasing content form paragraphs.

All phrasing content is also flow content. Any content model that expects flow content also expects phrasing content.

As a general rule, elements whose content model allows any phrasing content should have either at least one descendant text node that is not inter-element whitespace, or at least one descendant element node that is embedded content. For the purposes of this requirement, nodes that are descendants of `del` elements must not be counted as contributing to the ancestors of the `del` element.

> ***Note: Most elements that are categorized as phrasing content can only contain elements that are themselves categorized as phrasing content, not any flow content.***

Text nodes that are not inter-element whitespace are phrasing content.

3.4.1.6. Embedded content

Embedded content is content that imports another resource into the document, or content from another vocabulary that is inserted into the document.

All embedded content is also phrasing content (and flow content). Any content model that expects phrasing content (or flow content) also expects embedded content.

Elements that are from namespaces other than the HTML namespace and that convey content but not metadata, are embedded content for the purposes of the content models defined in this specification. (For example, MathML, or SVG.)

Some embedded content elements can have **fallback content**: content that is to be used when the external resource cannot be used (e.g. because it is of an unsupported format). The element definitions state what the fallback is, if any.

3.4.1.7. Interactive content

Parts of this section should eventually be moved to DOM3 Events.

Interactive content is content that is specifically intended for user interaction.

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 fire a further `DOMActivate` event at the same element, whose own default action is to go through all the elements the `DOMActivate` event bubbled through (starting at the target node and going towards the `Document` node), looking for an element with an activation behavior; the first element, in reverse tree order, to have one, must have its activation behavior executed.

> ***Note: The above doesn't happen for arbitrary synthetic events dispatched by author script. However, the `click()` method can be used to make it happen programmatically.***

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 disallow nesting interactive elements.***

3.4.2 Transparent content models

Some elements are described as **transparent**; they have "transparent" as their content model. Some elements are described as **semi-transparent**; this means that part of their content model is "transparent" but that is not the only part of the content model that must be satisfied.

When a content model includes a part that is "transparent", those parts must not contain content that would not be conformant if all transparent and semi-transparent elements in the tree were replaced, in their parent element, by the children in the "transparent" part of their content model, retaining order.

When a transparent or semi-transparent element has no parent, then the part of its content model that is "transparent" must instead be treated as accepting any flow content.

3.5 Paragraphs

A **paragraph** is typically a block of text with one or more sentences that discuss a particular topic, as in typography, but can also be used for more general thematic grouping. For instance, an address is also a paragraph, as is a part of a form, a byline, or a stanza in a poem.

Paragraphs in flow content are defined relative to what the document looks like without the `ins` and `del` elements complicating matters. Let *view* be a view of the DOM that replaces all `ins` and `del` elements in the document with their contents. Then, in *view*, for each run of phrasing content uninterrupted by other types of content, in an element that accepts content other than phrasing content, let *first* be the first node of the run, and let *last* be the last node of the run. For each run, a paragraph exists in the original DOM from immediately before *first* to immediately after *last*. (Paragraphs can thus span across `ins` and `del` elements.)

A paragraph is also formed by `p` elements.

Note: The `p` element can be used to wrap individual paragraphs when there would otherwise not be any content other than phrasing content to separate the paragraphs from each other.

> In the following example, there are two paragraphs in a section. There is also a header, which contains phrasing content that is not a paragraph. Note how the comments and intra-element whitespace do not form paragraphs.
>
> ```
> <section>
>   <h1>Example of paragraphs</h1>
>   This is the first paragraph in this example.
>   This is the second.
>   <!-- This is not a paragraph. -->
> </section>
> ```
>
> The following example takes that markup and puts `ins` and `del` elements around some of the markup to show that the text was changed (though in this case, the changes don't really make much sense, admittedly). Notice how this example has exactly the same paragraphs as the previous one, despite the `ins` and `del` elements.
>
> ```
> <section>
>   <ins><h1>Example of paragraphs</h1>
>   This is the first paragraph in</ins> this example<del>.
>   This is the second. </del>
>   <!-- This is not a paragraph. -->
> </section>
> ```

3.6 APIs in HTML documents

For HTML documents, and for HTML elements in HTML documents, certain APIs defined in DOM3 Core become case-insensitive or case-changing, as sometimes defined in DOM3 Core, and as summarized or required below. [DOM3CORE].

This does not apply to XML documents or to elements that are not in the HTML namespace despite being in HTML documents.

`Element.tagName`, `Node.nodeName`, and `Node.localName`

These attributes return tag names in all uppercase and attribute names in all lowercase, regardless of the case with which they were created.

`Document.createElement()`

The canonical form of HTML markup is all-lowercase; thus, this method will lowercase the argument before creating the requisite element. Also, the element created must be in the HTML namespace.

Note: This doesn't apply to `Document.createElementNS()`. Thus, it is possible, by passing this last method a tag name in the wrong case, to create an element that claims to have the tag name of an element defined in this specification, but doesn't support its interfaces, because it really has another tag name not accessible from the DOM APIs.

`Element.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.

3.7 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 behavior 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(...)`** 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 line feed character (U+000A).

3.7.1 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 and return the `Document` object on which the method was invoked.

   Note: This basically causes `document.open()` to be ignored when it's called in an inline script found during the parsing of data sent over the network, while still letting it have an effect when called asynchronously or on a document that is itself being spoon-fed using these APIs.

4. onbeforeunload, onunload, reset timers, empty event queue, kill any pending transactions, XMLHttpRequests, etc
5. If the document has an active parser, then stop that parser, and throw away any pending content in the input stream. what about if it doesn't, because it's either like a text/plain, or Atom, or PDF, or XHTML, or image document, or something?
6. Remove all child nodes of the document.
7. Change the document's character encoding to UTF-16.
8. 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).
9. Mark the document as being an HTML document (it might already be so-marked).
10. If *type* does not have the value "`text/html`", then act as if the tokeniser had emitted a start tag token with the tag name "pre", then set the HTML parser's tokenization stage's content model flag to *PLAINTEXT*.
11. If *replace* is false, then:
    1. Remove all the entries in the browsing context's session history after the current entry in its `Document`'s `History` object
    2. Remove any earlier entries that share the same `Document`
    3. Add a new entry just before the last entry that is associated with the text that was parsed by the previous parser associated with the `Document` object, as well as the state of the document at the start of these steps. (This allows the user to step backwards in the session history to see the page before it was blown away by the `document.open()` call.)
12. Finally, set the insertion point to point at just before the end of the input stream (which at this point will be empty).
13. Return the `Document` on which the method was invoked.

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 `Window` 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, and return whatever that method returned. 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.

3.7.2 Dynamic markup insertion in HTML

In HTML, the **`document.write(...)`** 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 consisting of the concatenation of all the arguments to the method must be inserted into the input stream just before the insertion point.
3. If there is a pending external script, 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 serialization 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 HTML fragment serialization algorithm on the node.

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

1. 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. Remove the children nodes of the `Document` whose `innerHTML` attribute is being set.
3. Create a new HTML parser, in its initial state, and associate it with the `Document` node.
4. Place into the input stream for the HTML parser just created the string being assigned into the `innerHTML` attribute.
5. 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. Invoke the HTML fragment parsing algorithm, with the element whose `innerHTML` attribute is being set as the *context* element, and the string being assigned into the `innerHTML` attribute as the *input*. Let *new children* be the result of this algorithm.
2. Remove the children of the element whose `innerHTML` attribute is being set.
3. Let *target document* be the `ownerDocument` of the `Element` node whose `innerHTML` attribute is being set.
4. Set the `ownerDocument` of all the nodes in *new children* to the *target document*.
5. Append all the *new children* nodes to the node whose `innerHTML` attribute is being set, preserving their order.

Note: `script` elements inserted using `innerHTML` do not execute when they are inserted.

3.7.3 Dynamic markup insertion in XML

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

On the other hand, however, the **`innerHTML`** attribute is indeed usable in an XML context.

In an XML context, the `innerHTML` DOM attribute on `HTMLElement`s must return a string in the form of an internal general parsed entity, and on `HTMLDocument`s must return a string in the form of a document entity. The string returned must be XML namespace-well-formed and must be an isomorphic serialization of all of that node's child nodes, in document order. User agents may adjust prefixes and namespace declarations in the serialization (and indeed might be forced to do so in some cases to obtain namespace-well-formed XML). For the `innerHTML` attribute on `HTMLElement` objects, if any of the elements in the serialization are in no namespace, the default namespace in scope for those elements must be explicitly declared as the empty string. (This doesn't apply to the `innerHTML` attribute on `HTMLDocument` objects.) [XML] [XMLNS]

If any of the following cases are found in the DOM being serialized, the user agent must raise an `INVALID_STATE_ERR` exception:

- A `Document` node with no child element nodes.
- A `DocumentType` node that has an external subset public identifier or an external subset system identifier that contains both a U+0022 QUOTATION MARK ("") and a U+0027 APOSTROPHE ("").
- A node with a prefix or local name containing a U+003A COLON (":").
- An `Attr` node, `Text` node, `CDATASection` node, `Comment` node, or `ProcessingInstruction` node whose data contains characters that are not matched by the XML `Char` production. [XML]
- A `CDATASection` node whose data contains the string "`]]>`".
- A `Comment` node whose data contains two adjacent U+002D HYPHEN-MINUS (-) characters or ends with such a character.
- A `ProcessingInstruction` node whose target name is the string "`xml`" (case insensitively).
- A `ProcessingInstruction` node whose target name contains a U+003A COLON (":").
- A `ProcessingInstruction` node whose data contains the string "`?>`".

Note: These are the only ways to make a DOM unserializable. The DOM enforces all the other XML constraints; for example, trying to set an attribute with a name that contains an equals sign (=) will raised an `INVALID_CHARACTER_ERR` exception.

On setting, in an XML context, the `innerHTML` DOM attribute on `HTMLElement`s and `HTMLDocument`s must run the following algorithm:

1. The user agent must create a new XML parser.
2. 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.
3. The user agent must feed the parser just created the string being assigned into the `innerHTML` attribute.
4. 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.
5. If the parser found a well-formedness error, the attribute's setter must raise a `SYNTAX_ERR` exception and abort these steps.
6. The user agent must remove the children nodes of the node whose `innerHTML` attribute is being set.
7. If the attribute is being set on a `Document` node, let *new children* be the children of the document, preserving their order. Otherwise, the attribute is being set on an `Element` node; let *new children* be the children of the document's root element, preserving their order.

8. If the attribute is being set on a `Document` node, let *target document* be that `Document` node. Otherwise, the attribute is being set on an `Element` node; let *target document* be the `ownerDocument` of that `Element`.

9. Set the `ownerDocument` of all the nodes in *new children* to the *target document*.

10. Append all the *new children* nodes to the node whose `innerHTML` attribute is being set, preserving their order.

Note: `script` elements inserted using `innerHTML` do not execute when they are inserted.

4. The elements of HTML

4.1 The root element

4.1.1 The `html` element

Categories

None.

Contexts in which this element may be used:

As the root element of a document.
Wherever a subdocument fragment is allowed in a compound document.

Content model:

A `head` element followed by a `body` element.

Element-specific attributes:

`manifest`

DOM interface:

Uses `HTMLElement`.

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

The **`manifest`** attribute gives the address of the document's application cache manifest, if there is one. If the attribute is present, the attribute's value must be a valid URL.

The `manifest` attribute only has an effect during the early stages of document load. Changing the attribute dynamically thus has no effect (and thus, no DOM API is provided for this attribute).

> ***Note: Later `base` elements don't affect the resolving of relative URLs in `manifest` attributes, as the attributes are processed before those elements are seen.***

4.2 Document metadata

4.2.1 The `head` element

Categories

None.

Contexts in which this element may be used:

As the first element in an `html` element.

Content model:

One or more elements of metadata content, of which exactly one is a `title` element.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

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

4.2.2 The `title` element

Categories

Metadata content.

Contexts in which this element may be used:

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

Content model:

Text.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `title` element represents the document's title or name. Authors should use titles that identify their documents even when they are used out of context, for example in a user's history or bookmarks, or in search results. The document's title is often different from its first header, since the first header does not have to stand alone when taken out of context.

There must be no more than one `title` element per document.

The `title` element must not contain any elements.

> Here are some examples of appropriate titles, contrasted with the top-level headers that might be used on those same pages.
>
> ```
> <title>Introduction to The Mating Rituals of Bees</title>
>   ...
> <h1>Introduction</h1>
> <p>This companion guide to the highly successful
> <cite>Introduction to Medieval Bee-Keeping</cite> book is...
> ```
>
> The next page might be a part of the same site. Note how the title describes the subject matter unambiguously, while the first header assumes the reader knowns what the context is and therefore won't wonder if the dances are Salsa or Waltz:
>
> ```
> <title>Dances used during bee mating rituals</title>
>   ...
> <h1>The Dances</h1>
> ```

The string to use as the document's title is given by the `document.title` DOM attribute. User agents should use the document's title when referring to the document in their user interface.

4.2.3 The `base` element

Categories

Metadata content.

Contexts in which this element may be used:

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

Content model:

Empty.

Element-specific attributes:

`href`
`target`

DOM interface:

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

The `base` element allows authors to specify the document base URL for the purposes of resolving relative URLs, and the name of the default browsing context for the purposes of following hyperlinks.

There must be no more than one `base` element per document.

A `base` element must have either an `href` attribute, a `target` attribute, or both.

The **`href`** content attribute, if specified, must contain a valid URL.

A `base` element, if it has an `href` attribute, must come before any other elements in the tree that have attributes defined as taking URLs, except the `html` element (its `manifest` attribute isn't affected by `base` elements).

Note: If there are multiple `base` elements with `href` attributes, all but the first are ignored.

The **`target`** attribute, if specified, must contain a valid browsing context name or keyword. User agents use this name when following hyperlinks.

A `base` element, if it has a `target` attribute, must come before any elements in the tree that represent hyperlinks.

Note: If there are multiple `base` elements with `target` attributes, all but the first are ignored.

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

4.2.4 The `link` element

Categories

Metadata content.

Contexts in which this element may be used:

Where metadata content is expected.
In a `noscript` element that is a child of a `head` element.

Content model:

Empty.

Element-specific attributes:

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

DOM interface:

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

The `LinkStyle` interface must also be implemented by this element, the styling processing model defines how. [CSSOM]

The `link` element allows authors to link their document to other resources.

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

The type of link indicated (the relationship) is given by the value of the **`rel`** attribute, which must be present, and must have a value that is a set of space-separated tokens. The allowed values and their meanings are defined in a later section. If the `rel` attribute is absent, or if the 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); exactly which and how many links are created depends on the keywords given in the `rel` attribute. User agents must process the links on a per-link basis, not a per-element basis.

The exact behavior for links to external resources depends on the exact relationship, as defined for the relevant link type. Some of the attributes control whether or not the external resource is to be applied (as defined below). For external resources that are represented in the DOM (for example, style sheets), the DOM representation must be made available even if the resource is not applied. (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.)

HTTP semantics must be followed when fetching external resources. (For example, redirects must be followed and 404 responses must cause the external resource to not be applied.)

Interactive user agents should provide users with a means to follow the hyperlinks created using the `link` element, somewhere within their user interface. The exact interface is not defined by this specification, but it should include the following information (obtained from the element's attributes, again as defined below), in some form or another (possibly simplified), for each hyperlink created with each `link` element in the document:

- The relationship between this document and the resource (given by the `rel` attribute)
- The title of the resource (given by the `title` attribute).
- The address of the resource (given by the `href` attribute).
- The language of the resource (given by the `hreflang` attribute).
- The optimum media for the resource (given by the `media` attribute).

User agents may also include other information, such as the type of the resource (as given by the `type` attribute).

> ***Note: Hyperlinks created with the `link` element and its `rel` attribute apply to the whole page. This contrasts with the `rel` attribute of `a` and `area` elements, which indicates the type of a link whose context is given by the link's location within the document.***

The **`media`** attribute says which media the resource applies to. The value must be a valid media query. [MQ]

If the link is a hyperlink then the `media` attribute is purely advisory, and describes for which media the document in question was designed.

However, if the link is an external resource link, then the `media` attribute is prescriptive. The user agent must apply the external resource to views while their state match the listed media and the other relevant conditions apply, and must not apply them otherwise.

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

The **`hreflang`** attribute on the `link` element has the same semantics as the `hreflang` attribute on hyperlink elements.

The **`type`** attribute gives the MIME type of the linked resource. It is purely advisory. The value must be a valid

MIME type, optionally with parameters. [RFC2046]

For external resource links, the `type` attribute is used as a hint to user agents so that they can avoid downloading resources they do not support. If the attribute is present, then the user agent must assume that the resource is of the given type. If the attribute is omitted, but the external resource link type has a default type defined, then the user agent must assume that the resource is of that type. If the UA does not support the given MIME type for the given link relationship, then the UA should not download the resource; if the UA does support the given MIME type for the given link relationship, then the UA should download the resource. If the attribute is omitted, and the external resource link type does not have a default type defined, but the user agent would fetch the resource if the type was known and supported, then the user agent should fetch the resource under the assumption that it will be supported.

User agents must not consider the `type` attribute authoritative — upon fetching the resource, user agents must not use the `type` attribute to determine its actual type. Only the actual type (as defined in the next paragraph) is used to determine whether to *apply* the resource, not the aforementioned assumed type.

If the resource is expected to be an image, user agents may apply the image sniffing rules, with the *official type* being the type determined from the resource's Content-Type metadata, and use the resulting sniffed type of the resource as if it was the actual type. Otherwise, if the resource is not expected to be an image, or if the user agent opts not to apply those rules, then the user agent must use the resource's Content-Type metadata to determine the type of the resource. If there is no type metadata, but the external resource link type has a default type defined, then the user agent must assume that the resource is of that type.

Once the user agent has established the type of the resource, the user agent must apply the resource if it is of a supported type and the other relevant conditions apply, and must ignore the resource otherwise.

> If a document contains style sheet links labeled as follows:
>
> ```
> <link rel="stylesheet" href="A" type="text/plain">
> <link rel="stylesheet" href="B" type="text/css">
> <link rel="stylesheet" href="C">
> ```
>
> ...then a compliant UA that supported only CSS style sheets would fetch the B and C files, and skip the A file (since `text/plain` is not the MIME type for CSS style sheets).
>
> For files B and C, it would then check the actual types returned by the server. For those that are sent as `text/css`, it would apply the styles, but for those labeled as `text/plain`, or any other type, it would not.
>
> If one the two files was returned without a Content-Type metadata, or with a syntactically incorrect type like `Content-Type: "null"`, then the default type for `stylesheet` links would kick in. Since that default type is `text/css`, the style sheet *would* nonetheless be applied.

The **`title`** attribute gives the title of the link. With one exception, it is purely advisory. The value is text. The exception is for style sheet links, where the `title` attribute defines alternative style sheet sets.

> ***Note: The `title` attribute on `link` elements differs from the global `title` attribute of most other elements in that a link without a title does not inherit the title of the parent element: it merely has no title.***

The `sizes` attribute is used with the `icon` link type. The attribute must not be specified on `link` elements

that do not have a `rel` attribute that specifies the `icon` keyword.

Some versions of HTTP defined a `Link:` header, to be processed like a series of `link` elements. If supported, for the purposes of ordering links defined by HTTP headers must be assumed to come before any links in the document, in the order that they were given in the HTTP entity header. (URIs in these headers are to be processed and resolved according to the rules given in HTTP; the rules of *this* specification don't apply.) [RFC2616] [RFC2068]

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

The DOM attribute **`relList`** must reflect the `rel` content attribute.

The DOM attribute **`disabled`** only applies to style sheet links. When the `link` element defines a style sheet link, then the `disabled` attribute behaves as defined for the alternative style sheets DOM. For all other `link` elements it always return false and does nothing on setting.

4.2.5 The `meta` element

Categories

Metadata content.

Contexts in which this element may be used:

If the `charset` attribute is present, or if the element is in the Encoding declaration state: as the first element in a `head` element.
If the `http-equiv` attribute is present, and the element is not in the Encoding declaration state: in a `head` element.
If the `http-equiv` attribute is present, and the element is not in the Encoding declaration state: in a `noscript` element that is a child of a `head` element.
If the `name` attribute is present: where metadata content is expected.

Content model:

Empty.

Element-specific attributes:

`name`
`http-equiv`
`content`
`charset` (HTML only)

DOM interface:

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

The `meta` element represents various kinds of metadata that cannot be expressed using the `title`, `base`,

`link`, `style`, and `script` elements.

The `meta` element can represent document-level metadata with the `name` attribute, pragma directives with the `http-equiv` attribute, and the file's character encoding declaration when an HTML document is serialized to string form (e.g. for transmission over the network or for disk storage) with the `charset` attribute.

Exactly one of the `name`, `http-equiv`, and `charset` attributes must be specified.

If either `name` or `http-equiv` is specified, then the `content` attribute must also be specified. Otherwise, it must be omitted.

The **`charset`** attribute specifies the character encoding used by the document. This is called a character encoding declaration.

The `charset` attribute may be specified in HTML documents only, it must not be used in XML documents. If the `charset` attribute is specified, the element must be the first element in the `head` element of the file.

The **`content`** attribute gives the value of the document metadata or pragma directive when the element is used for those purposes. The allowed values depend on the exact context, as described in subsequent sections of this specification.

If a `meta` element has a **`name`** attribute, it sets document metadata. Document metadata is expressed in terms of name/value pairs, the `name` attribute on the `meta` element giving the name, and the `content` attribute on the same element giving the value. The name specifies what aspect of metadata is being set; valid names and the meaning of their values are described in the following sections. If a `meta` element has no `content` attribute, then the value part of the metadata name/value pair is the empty string.

If a `meta` element has the `http-equiv` attribute specified, it must be either in a `head` element or in a `noscript` element that itself is in a `head` element. If a `meta` element does not have the `http-equiv` attribute specified, it must be in a `head` element.

The DOM attributes **`name`** and **`content`** must reflect the respective content attributes of the same name. The DOM attribute **`httpEquiv`** must reflect the content attribute `http-equiv`.

4.2.5.1. Standard metadata names

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

application-name

> The value must be a short free-form string that giving the name of the Web application that the page represents. If the page is not a Web application, the `application-name` metadata name must not be used. User agents may use the application name in UI in preference to the page's `title`, since the title might include status messages and the like relevant to the status of the page at a particular moment in time instead of just being the name of the application.

description

> The value must be a free-form string that describes the page. The value must be appropriate for use in a directory of pages, e.g. in a search engine.

generator

> The value must be a free-form string that identifies the software used to generate the document. This

value must not be used on hand-authored pages.

4.2.5.2. Other metadata names

Extensions to the predefined set of metadata names may be registered in the WHATWG Wiki MetaExtensions page.

Anyone is free to edit the WHATWG Wiki MetaExtensions page at any time to add a type. These new names must be specified with the following information:

Keyword

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

Brief description

A short description of what the metadata name's meaning is, including the format the value is required to be in.

Link to more details

A link to a more detailed description of the metadata name's semantics and requirements. It could be another page on the Wiki, or a link to an external page.

Synonyms

A list of other names that have exactly the same processing requirements. Authors should not use the names defined to be synonyms, they are only intended to allow user agents to support legacy content.

Status

One of the following:

Proposal

The name has not received wide peer review and approval. Someone has proposed it and is using it.

Accepted

The name has received wide peer review and approval. It has a specification that unambiguously defines how to handle pages that use the name, including when they use it in incorrect ways.

Unendorsed

The metadata name has received wide peer review and it has been found wanting. Existing pages are using this keyword, but new pages should avoid it. The "brief description" and "link to more details" entries will give details of what authors should use instead, if anything.

If a metadata name is added with the "proposal" status and found to be redundant with existing values, it should be removed and listed as a synonym for the existing value.

Conformance checkers must use the information given on the WHATWG Wiki MetaExtensions page to establish if a value not explicitly defined in this specification is allowed or not. When an author uses a new type not defined by either this specification or the Wiki page, conformance checkers should offer to add the value to the Wiki, with the details described above, with the "proposal" status.

This specification does not define how new values will get approved. It is expected that the Wiki will have a community that addresses this.

Metadata names whose values are to be URLs must not be proposed or accepted. Links must be represented using the `link` element, not the `meta` element.

4.2.5.3. Pragma directives

When the **`http-equiv`** attribute is specified on a `meta` element, the element is a pragma directive.

The `http-equiv` attribute is an enumerated attribute. The following table lists the keywords defined for this attribute. The states given in the first cell of the rows with keywords give the states to which those keywords map.

State	Keywords
Encoding declaration	`Content-Type`
Default style	`default-style`
Refresh	`refresh`

When a `meta` element is inserted into the document, if its `http-equiv` attribute is present and represents one of the above states, then the user agent must run the algorithm appropriate for that state, as described in the following list:

Encoding declaration state

The Encoding declaration state's user agent requirements are all handled by the parsing section of the specification. The state is just an alternative form of setting the `charset` attribute: it is a character encoding declaration.

For `meta` elements in the Encoding declaration state, the `content` attribute must have a value that is a case-insensitive match of a string that consists of the literal string "`text/html;`", optionally followed by any number of space characters, followed by the literal string "`charset=`", followed by the character encoding name of the character encoding declaration.

If the document contains a `meta` element in the Encoding declaration state then that element must be the first element in the document's `head` element, and the document must not contain a `meta` element with the `charset` attribute present.

The Encoding declaration state may be used in HTML documents only, elements in that state must not be used in XML documents.

Default style state

1. ...

Refresh state

1. If another `meta` element in the Refresh state has already been successfully processed (i.e. when it was inserted the user agent processed it and reached the last step of this list of steps), then abort these steps.
2. If the `meta` element has no `content` attribute, or if that attribute's value is the empty string, then abort these steps.
3. Let *input* be the value of the element's `content` attribute.

4. Let *position* point at the first character of *input*.
5. Skip whitespace.
6. Collect a sequence of characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE, and parse the resulting string using the rules for parsing non-negative integers. If the sequence of characters collected is the empty string, then no number will have been parsed; abort these steps. Otherwise, let *time* be the parsed number.
7. Collect a sequence of characters in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE and U+002E FULL STOP ("`.`"). Ignore any collected characters.
8. Skip whitespace.
9. Let *url* be the address of the current page.
10. If the character in *input* pointed to by *position* is a U+003B SEMICOLON ("`;`"), then advance *position* to the next character. Otherwise, jump to the last step.
11. Skip whitespace.
12. If the character in *input* pointed to by *position* is one of U+0055 LATIN CAPITAL LETTER U or U+0075 LATIN SMALL LETTER U, then advance *position* to the next character. Otherwise, jump to the last step.
13. If the character in *input* pointed to by *position* is one of U+0052 LATIN CAPITAL LETTER R or U+0072 LATIN SMALL LETTER R, then advance *position* to the next character. Otherwise, jump to the last step.
14. If the character in *input* pointed to by *position* is one of U+004C LATIN CAPITAL LETTER L or U+006C LATIN SMALL LETTER L, then advance *position* to the next character. Otherwise, jump to the last step.
15. Skip whitespace.
16. If the character in *input* pointed to by *position* is a U+003D EQUALS SIGN ("`=`"), then advance *position* to the next character. Otherwise, jump to the last step.
17. Skip whitespace.
18. Let *url* be equal to the substring of *input* from the character at *position* to the end of the string.
19. Strip any trailing space characters from the end of *url*.
20. Strip any U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), and U+000D CARRIAGE RETURN (CR) characters from *url*.
21. Resolve the *url* value to an absolute URL. (For the purposes of determining the base URL, the *url* value comes from the value of a content attribute of the `meta` element.) If this fails, abort these steps.
22. Perform one or more of the following steps:
    - Set a timer so that in *time* seconds, adjusted to take into account user or user agent

preferences, if the user has not canceled the redirect, the user agent navigates the document's browsing context to *url*, with replacement enabled, and with the document's browsing context as the source browsing context.

- Provide the user with an interface that, when selected, navigates a browsing context to *url*, with the document's browsing context as the source browsing context.
- Do nothing.

In addition, the user agent may, as with anything, inform the user of any and all aspects of its operation, including the state of any timers, the destinations of any timed redirects, and so forth.

For `meta` elements in the Refresh state, the `content` attribute must have a value consisting either of:

- just a valid non-negative integer, or
- a valid non-negative integer, followed by a U+003B SEMICOLON (`;`), followed by one or more space characters, followed by either a U+0055 LATIN CAPITAL LETTER U or a U+0075 LATIN SMALL LETTER U, a U+0052 LATIN CAPITAL LETTER R or a U+0072 LATIN SMALL LETTER R, a U+004C LATIN CAPITAL LETTER L or a U+006C LATIN SMALL LETTER L, a U+003D EQUALS SIGN (`=`), and then a valid URL.

In the former case, the integer represents a number of seconds before the page is to be reloaded; in the latter case the integer represents a number of seconds before the page is to be replaced by the page at the given URL.

There must not be more than one `meta` element with any particular state in the document at a time.

4.2.5.4. Specifying the document's character encoding

A **character encoding declaration** is a mechanism by which the character encoding used to store or transmit a document is specified.

The following restrictions apply to character encoding declarations:

- The character encoding name given must be the name of the character encoding used to serialize the file.
- The value must be a valid character encoding name, and must be the preferred name for that encoding. [IANACHARSET]
- The character encoding declaration must be serialized without the use of character references or character escapes of any kind.

If the document does not start with a BOM, and if its encoding is not explicitly given by Content-Type metadata, then the character encoding used must be an ASCII-compatible character encoding, and, in addition, if that encoding isn't US-ASCII itself, then the encoding must be specified using a `meta` element with a `charset` attribute or a `meta` element in the Encoding declaration state.

If the document contains a `meta` element with a `charset` attribute or a `meta` element in the Encoding declaration state, then the character encoding used must be an ASCII-compatible character encoding.

An **ASCII-compatible character encoding** is one that is a superset of US-ASCII (specifically,

ANSI_X3.4-1968) for bytes in the set 0x09, 0x0A, 0x0C, 0x0D, 0x20 - 0x22, 0x26, 0x27, 0x2C - 0x3F, 0x41 - 0x5A, and 0x61 - 0x7A.

Authors should not use JIS_X0212-1990, x-JIS0208, and encodings based on EBCDIC. Authors should not use UTF-32. Authors must not use the CESU-8, UTF-7, BOCU-1 and SCSU encodings. [CESU8] [UTF7] [BOCU1] [SCSU]

Authors are encouraged to use UTF-8. Conformance checkers may advise against authors using legacy encodings.

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

4.2.6 The `style` element

Categories

Metadata content.
If the `scoped` attribute is present: flow content.

Contexts in which this element may be used:

If the `scoped` attribute is absent: where metadata content is expected.
If the `scoped` attribute is absent: in a `noscript` element that is a child of a `head` element.
If the `scoped` attribute is present: where flow content is expected, but before any sibling elements other than `style` elements and before any text nodes other than inter-element whitespace.

Content model:

Depends on the value of the `type` attribute.

Element-specific attributes:

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

DOM interface:

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

The `LinkStyle` interface must also be implemented by this element, the styling processing model defines how. [CSSOM]

The `style` element allows authors to embed style information in their documents. The `style` element is one of several inputs to the styling processing model.

If the **`type`** attribute is given, it must contain a valid MIME type, optionally with parameters, that designates a

styling language. [RFC2046] If the attribute is absent, the type defaults to `text/css`. [RFC2138]

When examining types to determine if they support the language, user agents must not ignore unknown MIME parameters — types with unknown parameters must be assumed to be unsupported.

The **`media`** attribute says which media the styles apply to. The value must be a valid media query. [MQ] User agents must apply the styles to views while their state match the listed media, and must not apply them otherwise. [DOM3VIEWS]

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

The **`scoped`** attribute is a boolean attribute. If the attribute is present, then the user agent must apply the specified style information only to the `style` element's parent element (if any), and that element's child nodes. Otherwise, the specified styles must, if applied, be applied to the entire document.

The **`title`** attribute on `style` elements defines alternative style sheet sets. If the `style` element has no `title` attribute, then it has no title; the `title` attribute of ancestors does not apply to the `style` element.

> ***Note: The `title` attribute on `style` elements, like the `title` attribute on `link` elements, differs from the global `title` attribute in that a `style` block without a title does not inherit the title of the parent element: it merely has no title.***

All descendant elements must be processed, according to their semantics, before the `style` element itself is evaluated. For styling languages that consist of pure text, user agents must evaluate `style` elements by passing the concatenation of the contents of all the text nodes that are direct children of the `style` element (not any other nodes such as comments or elements), in tree order, to the style system. For XML-based styling languages, user agents must pass all the 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 **`media`**, **`type`** and **`scoped`** DOM attributes must reflect the respective content attributes of the same name.

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

4.2.7 Styling

The `link` and `style` elements can provide styling information for the user agent to use when rendering the document. The DOM Styling specification specifies what styling information is to be used by the user agent and how it is to be used. [CSSOM]

The `style` and `link` elements implement the `LinkStyle` interface. [CSSOM]

For `style` elements, if the user agent does not support the specified styling language, then the `sheet` attribute of the element's `LinkStyle` interface must return null. Similarly, `link` elements that do not represent external resource links that contribute to the styling processing model (i.e. that do not have a `stylesheet` keyword in their `rel` attribute), and `link` elements whose specified resource has not yet been downloaded, or is not in a supported styling language, must have their `LinkStyle` interface's `sheet` attribute

return null.

Otherwise, the `LinkStyle` interface's `sheet` attribute must return a `StyleSheet` object with the attributes implemented as follows: [CSSOM]

The content type (`type` DOM attribute)

The content type must be the same as the style's specified type. For `style` elements, this is the same as the `type` content attribute's value, or `text/css` if that is omitted. For `link` elements, this is the Content-Type metadata of the specified resource.

The location (`href` DOM attribute)

For `link` elements, the location must be the result of resolving the URL given by the element's `href` content attribute, or the empty string if that fails. For `style` elements, there is no location.

The intended destination media for style information (`media` DOM attribute)

The media must be the same as the value of the element's `media` content attribute.

The style sheet title (`title` DOM attribute)

The title must be the same as the value of the element's `title` content attribute. If the attribute is absent, then the style sheet does not have a title. The title is used for defining **alternative style sheet sets**.

The **`disabled`** DOM attribute on `link` and `style` elements must return false and do nothing on setting, if the `sheet` attribute of their `LinkStyle` interface is null. Otherwise, it must return the value of the `StyleSheet` interface's `disabled` attribute on getting, and forward the new value to that same attribute on setting.

4.3 Sections

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

4.3.1 The `body` element

Categories

Sectioning content.

Contexts in which this element may be used:

As the second element in an `html` element.

Content model:

Flow content.

Element-specific attributes:

None.

DOM interface:

```
interface HTMLBodyElement : HTMLElement {};
```

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

In conforming documents, there is only one `body` element. The `document.body` DOM attribute provides scripts with easy access to a document's `body` element.

Note: Some DOM operations (for example, parts of the drag and drop model) are defined in terms of "the body element". This refers to a particular element in the DOM, as per the definition of the term, and not any arbitrary `body` element.

4.3.2 The `section` element

Categories

Flow content.
Sectioning content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `section` element represents a generic document or application section. A section, in this context, is a thematic grouping of content, typically with a header, possibly with a footer.

> Examples of sections would be chapters, the various tabbed pages in a tabbed dialog box, or the numbered sections of a thesis. A Web site's home page could be split into sections for an introduction, news items, contact information.

4.3.3 The `nav` element

Categories

Flow content.
Sectioning content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `nav` element represents a section of a page that links to other pages or to parts within the page: a section with navigation links. Not all groups of links on a page need to be in a `nav` element — only sections that consist of primary navigation blocks are appropriate for the `nav` element. In particular, it is common for footers to have a list of links to various key parts of a site, but the `footer` element is more appropriate in such cases.

In the following example, the page has several places where links are present, but only one of those places is considered a navigation section.

```
<body>
 <header>
  <h1>Wake up sheeple!</h1>
  <p><a href="news.html">News</a> -
     <a href="blog.html">Blog</a> -
     <a href="forums.html">Forums</a></p>
 </header>
 <nav>
  <h1>Navigation</h1>
  <ul>
   <li><a href="articles.html">Index of all articles</a><li>
   <li><a href="today.html">Things sheeple need to wake up for
today</a><li>
   <li><a href="successes.html">Sheeple we have managed to wake</a><li>
  </ul>
 </nav>
 <article>
  <p>...page content would be here...</p>
 </article>
 <footer>
  <p>Copyright © 2006 The Example Company</p>
  <p><a href="about.html">About</a> -
     <a href="policy.html">Privacy Policy</a> -
     <a href="contact.html">Contact Us</a></p>
 </footer>
</body>
```

4.3.4 The `article` element

Categories

Flow content.
Sectioning content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `article` element represents a section of a page that consists of a composition that forms an independent part of a document, page, or site. This could be a forum post, a magazine or newspaper article, a Web log entry, a user-submitted comment, or any other independent item of content.

Note: An `article` element is "independent" in that its contents could stand alone, for example in syndication. However, the element is still associated with its ancestors; for instance, contact information that applies to a parent `body` element still covers the `article` as well.

When `article` elements are nested, the inner `article` elements represent articles that are in principle related to the contents of the outer article. For instance, a Web log entry on a site that accepts user-submitted comments could represent the comments as `article` elements nested within the `article` element for the Web log entry.

Author information associated with an `article` element (q.v. the `address` element) does not apply to nested `article` elements.

4.3.5 The `aside` element

Categories

Flow content.
Sectioning content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `aside` element represents a section of a page that consists of content that is tangentially related to the content around the `aside` element, and which could be considered separate from that content. Such sections are often represented as sidebars in printed typography.

> The following example shows how an aside is used to mark up background material on Switzerland in a much longer news story on Europe.

```
<aside>
 <h1>Switzerland</h1>
 <p>Switzerland, a land-locked country in the middle of geographic
 Europe, has not joined the geopolitical European Union, though it is
 a signatory to a number of European treaties.</p>
</aside>
```

The following example shows how an aside is used to mark up a pull quote in a longer article.

```
...

<p>He later joined a large company, continuing on the same work.
<q>I love my job. People ask me what I do for fun when I'm not at
work. But I'm paid to do my hobby, so I never know what to
answer. Some people wonder what they would do if they didn't have to
work... but I know what I would do, because I was unemployed for a
year, and I filled that time doing exactly what I do
now.</q></p>

<aside>
 <q> People ask me what I do for fun when I'm not at work. But I'm
 paid to do my hobby, so I never know what to answer. </q>
</aside>

<p>Of course his work — or should that be hobby? —
isn't his only passion. He also enjoys other pleasures.</p>

...
```

4.3.6 The `h1`, `h2`, `h3`, `h4`, `h5`, and `h6` elements

Categories

Flow content.
Heading content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

These elements define headers for their sections.

The semantics and meaning of these elements are defined in the section on headings and sections.

These elements have a **rank** given by the number in their name. The `h1` element is said to have the highest rank, the `h6` element has the lowest rank, and two elements with the same name have equal rank.

4.3.7 The `header` element

Categories

Flow content.
Heading content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content, including at least one descendant that is heading content, but no sectioning content descendants, no `header` element descendants, and no `footer` element descendants.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `header` element represents the header of a section. The element is typically used to group a set of `h1`–`h6` elements to mark up a page's title with its subtitle or tagline. However, `header` elements may contain more than just the section's headings and subheadings — for example it would be reasonable for the header to include version history information.

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

Other heading elements in the `header` element indicate subheadings or subtitles.

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

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

> Here are some examples of valid headers. In each case, the emphasised text represents the text that would be used as the header in an application extracting header data and ignoring subheadings.
>
> ```
> <header>
>  <h1>The reality dysfunction</h1>
>  <h2>Space is not the only void</h2>
> </header>
>
> <header>
>  <h1>Dr. Strangelove</h1>
>  <h2>Or: How I Learned to Stop Worrying and Love the Bomb</h2>
> </header>
>
> <header>
> ```

```
 <p>Welcome to...</p>
 <h1>Voidwars!</h1>
</header>

<header>
 <h1>Scalable Vector Graphics (SVG) 1.2</h1>
 <h2>W3C Working Draft 27 October 2004</h2>
 <dl>
  <dt>This version:</dt>
  <dd><a href="http://www.w3.org/TR/2004/WD-SVG12-20041027/">http:
//www.w3.org/TR/2004/WD-SVG12-20041027/</a></dd>
  <dt>Previous version:</dt>
  <dd><a href="http://www.w3.org/TR/2004/WD-SVG12-20040510/">http:
//www.w3.org/TR/2004/WD-SVG12-20040510/</a></dd>
  <dt>Latest version of SVG 1.2:</dt>
  <dd><a href="http://www.w3.org/TR/SVG12/">http://www.w3.org/TR/SVG12
/</a></dd>
  <dt>Latest SVG Recommendation:</dt>
  <dd><a href="http://www.w3.org/TR/SVG/">http://www.w3.org/TR/SVG
/</a></dd>
  <dt>Editor:</dt>
  <dd>Dean Jackson, W3C, <a href="mailto:dean@w3.org">dean@w3.org</a></dd>
  <dt>Authors:</dt>
  <dd>See <a href="#authors">Author List</a></dd>
 </dl>
 <p class="copyright"><a href="http://www.w3.org/Consortium/Legal/ipr-
notic ...
</header>
```

4.3.8 The `footer` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content, but with no heading content descendants, no sectioning content descendants, and no `footer` element descendants.

Element-specific attributes:

None.

DOM interface:

Uses `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.

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

Footers don't necessarily have to appear at the end of a section, though they usually do.

> Here is a page with two footers, one at the top and one at the bottom, with the same content:
>
> ```
> <body>
>  <footer><a href="../">Back to index...</a></footer>
>  <h1>Lorem ipsum</h1>
>  A dolor sit amet, consectetur adipisicing elit, sed do eiusmod
>  tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
>  veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
>  ea commodo consequat. Duis aute irure dolor in reprehenderit in
>  voluptate velit esse cillum dolore eu fugiat nulla
>  pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
>  culpa qui officia deserunt mollit anim id est laborum.
>  <footer><a href="../">Back to index...</a></footer>
> </body>
> ```

4.3.9 The `address` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content, but with no heading content descendants, no sectioning content descendants, no `footer` element descendants, and no `address` element descendants.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `address` element represents the contact information for the section it applies to. If it applies to the `body` element, then it instead applies to the document as a whole.

> For example, a page at the W3C Web site related to HTML might include the following contact information:
>
> ```
> <ADDRESS>
>  <A href="../People/Raggett/">Dave Raggett</A>,
>  <A href="../People/Arnaud/">Arnaud Le Hors</A>,
>  contact persons for the <A href="Activity">W3C HTML Activity</A>
> </ADDRESS>
> ```

The `address` element must not be used to represent arbitrary addresses (e.g. postal addresses), unless those addresses are contact information for the section. (The `p` element is the appropriate element for marking

up such addresses.)

The `address` element must not contain information other than contact information.

> For example, the following is non-conforming use of the `address` element:
>
> ```
> <ADDRESS>Last Modified: 1999/12/24 23:37:50</ADDRESS>
> ```

Typically, the `address` element would be included with other information in a `footer` element.

To determine the contact information for a sectioning content element (such as a document's `body` element, which would give the contact information for the page), UAs must collect all the `address` elements that apply to that sectioning content element and its ancestor sectioning content elements. The contact information is the collection of all the information given by those elements.

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

4.3.10 Headings and sections

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

The first element of heading content in an element of sectioning content 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 content elements are always considered subsections of their nearest ancestor element of sectioning content, regardless of what implied sections other headings may have created.

Certain elements are said to be **sectioning roots**, including `blockquote` and `td` elements. These elements can have their own outlines, but the sections and headers inside these elements do not contribute to the outlines of their ancestors.

> For the following fragment:
>
> ```
> <body>
>  <h1>Foo</h1>
>  <h2>Bar</h2>
>  <blockquote>
>   <h3>Bla</h3>
>  </blockquote>
>  Baz
>  <h2>Quux</h2>
>  <section>
>   <h3>Thud</h3>
>  </section>
>  Grunt
> </body>
> ```
>
> ...the structure would be:

1. Foo (heading of explicit `body` section, containing the "Grunt" paragraph)
    1. Bar (heading starting implied section, containing a block quote and the "Baz" paragraph)
    2. Quux (heading starting implied section)
    3. Thud (heading of explicit `section` section)

Notice how the `section` ends the earlier implicit section so that a later paragraph ("Grunt") is back at the top level.

Sections may contain headers of any rank, but authors are strongly encouraged to either use only `h1` elements, or to use elements of the appropriate rank for the section's nesting level.

Authors are also encouraged to explicitly wrap sections in elements of sectioning content, instead of relying on the implicit sections generated by having multiple heading in one element of sectioning content.

For example, the following is correct:

```
<body>
 <h4>Apples</h4>
 <p>Apples are fruit.</p>
 <section>
  <h2>Taste</h2>
  <p>They taste lovely.</p>
  <h6>Sweet</h6>
  <p>Red apples are sweeter than green ones.</p>
  <h1>Color</h1>
  <p>Apples come in various colors.</p>
 </section>
</body>
```

However, the same document would be more clearly expressed as:

```
<body>
 <h1>Apples</h1>
 <p>Apples are fruit.</p>
 <section>
  <h2>Taste</h2>
  <p>They taste lovely.</p>
  <section>
   <h3>Sweet</h3>
   <p>Red apples are sweeter than green ones.</p>
  </section>
 </section>
 <section>
  <h2>Color</h2>
  <p>Apples come in various colors.</p>
 </section>
</body>
```

Both of the documents above are semantically identical and would produce the same outline in compliant user agents.

4.3.10.1. Creating an outline

This section defines an algorithm for creating an outline for a sectioning content element or a sectioning root element. It is defined in terms of a walk over the nodes of a DOM tree, in tree order, with each node being visited when it is *entered* and when it is *exited* during the walk.

The **outline** for a sectioning content element or a sectioning root element consists of a list of one or more potentially nested sections. A **section** is a container that corresponds to some nodes in the original DOM tree. Each section can have one heading associated with it, and can contain any number of further nested sections. The algorithm for the outline also associates each node in the DOM tree with a particular section and potentially a heading. (The sections in the outline aren't `section` elements, though some may correspond to such elements — they are merely conceptual sections.)

> The following markup fragment:
>
> ```
> <body>
>  <h1>A</h1>
>  B
>  <h2>C</h2>
>  D
>  <h2>E</h2>
>  F
> </body>
> ```
>
> ...results in the following outline being created for the `body` node (and thus the entire document):
>
> 1. Section created for `body` node.
>   Associated with heading "A".
>   Also associated with paragraph "B".
>   Nested sections:
>
>   1. Section implied for first `h2` element.
>   Associated with heading "C".
>   Also associated with paragraph "D".
>   No nested sections.
>   2. Section implied for second `h2` element.
>   Associated with heading "E".
>   Also associated with paragraph "F".
>   No nested sections.

The algorithm that must be followed during a walk of a DOM subtree rooted at a sectioning content element or a sectioning root element to determine that element's outline is as follows:

1. Let *current outlinee* be null. (It holds the element whose outline is being created.)

2. Let *current section* be null. (It holds a pointer to a section, so that elements in the DOM can all be associated with a section.)

3. Create a stack to hold elements, which is used to handle nesting. Initialize this stack to empty.

4. As you walk over the DOM in tree order, trigger the first relevant step below for each element as you enter and exit it.

   ↪ If the top of the stack is an element, and you are exiting that element

Note: The element being exited is a heading content element.

Pop that element from the stack.

↪ **If the top of the stack is a heading content element**

Do nothing.

↪ **When entering a sectioning content element or a sectioning root element**

If *current outlinee* is not null, push *current outlinee* onto the stack.

Let *current outlinee* be the element that is being entered.

Let *current section* be a newly created section for the *current outlinee* element.

Let there be a new outline for the new *current outlinee*, initialized with just the new *current section* as the only section in the outline.

↪ **When exiting a sectioning content element, if the stack is not empty**

Pop the top element from the stack, and let the *current outlinee* be that element.

Let *current section* be the last section in the outline of the *current outlinee* element.

Append the outline of the sectioning content element being exited to the *current section*. (This does not change which section is the last section in the outline.)

↪ **When exiting a sectioning root element, if the stack is not empty**

Run these steps:

1. Pop the top element from the stack, and let the *current outlinee* be that element.
2. Let *current section* be the last section in the outline of the *current outlinee* element.
3. *Finding the deepest child:* If *current section* has no child sections, stop these steps.
4. Let *current section* be the last child section of the current *current section*.
5. Go back to the substep labeled *finding the deepest child*.

↪ **When exiting a sectioning content element or a sectioning root element**

Note: The current outlinee is the element being exited.

Let *current section* be the first section in the outline of the *current outlinee* element.

Skip to the next step in the overall set of steps. (The walk is over.)

↪ **If the *current outlinee* is null.**

Do nothing.

↪ **When entering a heading content element**

If the *current section* has no heading, let the element being entered be the heading for the *current section*.

Otherwise, if the element being entered has a rank equal to or greater than the heading of the last section of the outline of the *current outlinee*, then create a new section and append it to the outline of the *current outlinee* element, so that this new section is the new last section of that outline. Let *current section* be that new section. Let the element being entered be the new heading for the *current section*.

Otherwise, run these substeps:

1. Let *candidate section* be *current section*.
2. If the element being entered has a rank lower than the rank of the heading of the *candidate section*, then create a new section, and append it to *candidate section*. (This does not change which section is the last section in the outline.) Let *current section* be this new section. Let the element being entered be the new heading for the *current section*. Abort these substeps.
3. Let *new candidate section* be the section that contains *candidate section* in the outline of *current outlinee*.
4. Let *candidate section* be *new candidate section*.
5. Return to step 2.

Push the element being entered onto the stack. (This causes the algorithm to skip any descendants of the element.)

Note: Recall that `h1` has the highest rank, and `h6` has the lowest rank.

↪ **Otherwise**

Do nothing.

In addition, whenever you exit a node, after doing the steps above, if *current section* is not null, associate the node with the section *current section*.

5. If the *current outlinee* is null, then there was no sectioning content element or sectioning root element in the DOM. There is no outline. Abort these steps.
6. Associate any nodes that were not associated a section in the steps above with *current outlinee* as their section.
7. Associate all nodes with the heading of the section with which they are associated, if any.
8. If *current outlinee* is the `body` element, then the outline created for that element is the outline of the entire document.

The tree of sections created by the algorithm above, or a proper subset thereof, must be used when generating document outlines, for example when generating tables of contents.

When creating an interactive table of contents, entries should jump the user to the relevant sectioning content element, if the section was created for a real element in the original document, or to the relevant heading content element, if the section in the tree was generated for a heading in the above process.

Note: Selecting the first section of the document therefore always takes the user to the top of

the document, regardless of where the first header in the `body` is to be found.

The following JavaScript function shows how the tree walk could be implemented. The root argument is the root of the tree to walk, and the enter and exit arguments are callbacks that are called with the nodes as they are entered and exited. [ECMA262]

```
function (root, enter, exit) {
  var node = root;
  start: do while (node) {
    enter(node);
    if (node.firstChild) {
      node = node.firstChild;
      continue start;
    }
    while (node) {
      exit(node);
      if (node.nextSibling) {
        node = node.nextSibling;
        continue start;
      }
      if (node == root)
        node = null;
      else
        node = node.parentNode;
    }
  }
}
```

4.3.10.2. Distinguishing site-wide headings from page headings

Given the outline of a document, but ignoring any sections created for `nav` and `aside` elements, and any of their descendants, if the only root of the tree is the `body` element's section, and it has only a single subsection which is created by an `article` element, then the heading of the `body` element should be assumed to be a site-wide heading, and the heading of the `article` element should be assumed to be the page's heading.

If a page starts with a heading that is common to the whole site, the document must be authored such that, in the document's outline, ignoring any sections created for `nav` and `aside` elements and any of their descendants, the tree has only one root section, the `body` element's section, its heading is the site-wide heading, the `body` element has just one subsection, that subsection is created by an `article` element, and that `article`'s heading is the page heading.

If a page does not contain a site-wide heading, then the page must be authored such that, in the document's outline, ignoring any sections created for `nav` and `aside` elements and any of their descendants, either the `body` element has no subsections, or it has more than one subsection, or it has a single subsection but that subsection is not created by an `article` element, or there is more than one section at the root of the outline.

Note: Conceptually, a site is thus a document with many articles — when those articles are

split into many pages, the heading of the original single page becomes the heading of the site, repeated on every page.

4.4 Grouping content

4.4.1 The `p` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `p` element represents a paragraph.

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</label>
 </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>
```

4.4.2 The `hr` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Empty.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `hr` element represents a paragraph-level thematic break, e.g. a scene change in a story, or a transition to another topic within a section of a reference book.

4.4.3 The `br` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Empty.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `br` element represents a line break.

`br` elements must be empty. Any content inside `br` elements must not be considered part of the surrounding text.

`br` elements must be used only for line breaks that are actually part of the content, as in poems or addresses.

> The following example is correct usage of the `br` element:
>
> ```
> P. 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">
> ```

If a paragraph consists of nothing but a single `br` element, it represents a placeholder blank line (e.g. as in a template). Such blank lines must not be used for presentation purposes.

4.4.4 The `pre` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

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

Note: In the HTML serialization, a leading newline character immediately following the `pre` element start tag is stripped.

Some examples of cases where the `pre` element could be used:

- Including an e-mail, with paragraphs indicated by blank lines, lists indicated by lines prefixed with a bullet, and so on.
- Including fragments of computer code, with structure indicated according to the conventions of that language.
- Displaying ASCII art.

To represent a block of computer code, the `pre` element can be used with a `code` element; to represent a block of computer output the `pre` element can be used with a `samp` element. Similarly, the `kbd` element can be used within a `pre` element to indicate text that the user is to enter.

In the following snippet, a sample of computer code is presented.

```
<p>This is the <code>Panel</code> constructor:</p>
<pre><code>function Panel(element, canClose, closeHandler) {
  this.element = element;
  this.canClose = canClose;
  this.closeHandler = function () { if (closeHandler) closeHandler() };
}</code></pre>
```

In the following snippet, `samp` and `kbd` elements are mixed in the contents of a `pre` element to show a session of Zork I.

```
<pre><samp>You are in an open field west of a big white house with a
boarded
front door.
There is a small mailbox here.

></samp> <kbd>open mailbox</kbd>

<samp>Opening the mailbox reveals:
A leaflet.

></samp></pre>
```

The following shows a contemporary poem that uses the `pre` element to preserve its unusual formatting, which forms an intrinsic part of the poem itself.

```
<pre>                maxling

it is with a          heart
               heavy

that i admit loss of a feline
        so           loved

a friend lost to the
        unknown
                                  (night)

~cdr 11dec07</pre>
```

4.4.5 The `dialog` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Zero or more pairs of one `dt` element followed by one `dd` element.

Element-specific attributes:

None.

DOM interface:

Uses `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 money?
 <dt> Abbott
 <dd> Every dollar of it.
</dialog>
```

Note: Text in a `dt` element in a `dialog` element is implicitly the source of the text given in the following `dd` element, and the contents of the `dd` element are implicitly a quote from that speaker. There is thus no need to include `cite`, `q`, or `blockquote` elements in this markup. Indeed, a `q` element inside a `dd` element in a conversation would actually imply the people talking were themselves quoting another work. See the `cite`, `q`, and `blockquote` elements for other ways to cite or quote.

4.4.6 The `blockquote` element

Categories

Flow content.
Sectioning root.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Element-specific attributes:

`cite`

DOM interface:

```
interface HTMLQuoteElement : HTMLElement {
           attribute DOMString cite;
};
```

Note: The `HTMLQuoteElement` interface is also used by the `q` element.

The `blockquote` element represents a section that is quoted from another source.

Content inside a `blockquote` must be quoted from another source, whose address, if it has one, should be cited in the **`cite`** attribute.

If the `cite` attribute is present, it must be a valid URL. User agents should allow users to follow such citation links.

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

The **cite** DOM attribute must reflect the element's cite content attribute.

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

4.4.7 The ol element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Zero or more li elements.

Element-specific attributes:

reversed
start

DOM interface:

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

The ol element represents a list of items, where the items have been intentionally ordered, such that changing the order would change the meaning of the document.

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

The **reversed** attribute is a boolean attribute. If present, it indicates that the list is a descending list (..., 3, 2, 1). If the attribute is omitted, the list is an ascending list (1, 2, 3, ...).

The **start** attribute, if present, must be a valid integer giving the ordinal value of the first list item.

If the start attribute is present, user agents must parse it as an integer, in order to determine the attribute's value. The default value, used if the attribute is missing or if the value cannot be converted to a number according to the referenced algorithm, is 1 if the element has no reversed attribute, and is the number of child li elements otherwise.

The first item in the list has the ordinal value given by the ol element's start attribute, unless that li element has a value attribute with a value that can be successfully parsed, in which case it has the ordinal value given by that value attribute.

Each subsequent item in the list has the ordinal value given by its value attribute, if it has one, or, if it doesn't, the ordinal value of the previous item, plus one if the reversed is absent, or minus one if it is present.

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

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

> The following markup shows a list where the order matters, and where the `ol` element is therefore appropriate. Compare this list to the equivalent list in the `ul` section to see an example of the same items using the `ul` element.
>
> ```
> I have lived in the following countries (given in the order of when
> I first lived there):
> <ol>
>  <li>Switzerland
>  <li>United Kingdom
>  <li>United States
>  <li>Norway
> </ol>
> ```
>
> Note how changing the order of the list changes the meaning of the document. In the following example, changing the relative order of the first two items has changed the birthplace of the author:
>
> ```
> I have lived in the following countries (given in the order of when
> I first lived there):
> <ol>
>  <li>United Kingdom
>  <li>Switzerland
>  <li>United States
>  <li>Norway
> </ol>
> ```

4.4.8 The `ul` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Zero or more `li` elements.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `ul` element represents a list of items, where the order of the items is not important — that is, where changing the order would not materially change the meaning of the document.

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

The following markup shows a list where the order does not matter, and where the `ul` element is therefore appropriate. Compare this list to the equivalent list in the `ol` section to see an example of the same items using the `ol` element.

```
<p>I have lived in the following countries:</p>
<ul>
 <li>Norway
 <li>Switzerland
 <li>United Kingdom
 <li>United States
</ul>
```

Note that changing the order of the list does not change the meaning of the document. The items in the snippet above are given in alphabetical order, but in the snippet below they are given in order of the size of their current account balance in 2007, without changing the meaning of the document whatsoever:

```
<p>I have lived in the following countries:</p>
<ul>
 <li>Switzerland
 <li>Norway
 <li>United Kingdom
 <li>United States
</ul>
```

4.4.9 The `li` element

Categories

None.

Contexts in which this element may be used:

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

Content model:

When the element is a child of a `menu` element: phrasing content.
Otherwise: flow content.

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.

DOM interface:

```
interface HTMLLIElement : HTMLElement {
           attribute long value;
};
```

The `li` element represents a list item. If its parent element is an `ol`, `ul`, or `menu` element, then the element is

an item of the parent element's list, as defined for those elements. Otherwise, the list item has no defined list-related relationship to any other `li` element.

The **`value`** attribute, if present, must be a valid integer giving the ordinal value of the list item.

If the `value` attribute is present, user agents must parse it as an integer, in order to determine the attribute's value. If the attribute's value cannot be converted to a number, the attribute must be treated as if it was absent. The attribute has no default value.

The `value` attribute is processed relative to the element's parent `ol` element (q.v.), if there is one. If there is not, the attribute has no effect.

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

> The following example, the top ten movies are listed (in reverse order). Note the way the list is given a title by using a `figure` element and its `legend`.
>
> ```
> <figure>
>  <legend>The top 10 movies of all time</legend>
>  <ol>
>   <li value="10"><cite>Josie and the Pussycats</cite>, 2001</li>
>   <li value="9"><cite lang="sh">Црна мачка, бели мачор</cite>, 1998</li>
>   <li value="8"><cite>A Bug's Life</cite>, 1998</li>
>   <li value="7"><cite>Toy Story</cite>, 1995</li>
>   <li value="6"><cite>Monsters, Inc</cite>, 2001</li>
>   <li value="5"><cite>Cars</cite>, 2006</li>
>   <li value="4"><cite>Toy Story 2</cite>, 1999</li>
>   <li value="3"><cite>Finding Nemo</cite>, 2003</li>
>   <li value="2"><cite>The Incredibles</cite>, 2004</li>
>   <li value="1"><cite>Ratatouille</cite>, 2007</li>
>  </ol>
> </figure>
> ```
>
> The markup could also be written as follows, using the `reversed` attribute on the `ol` element:
>
> ```
> <figure>
>  <legend>The top 10 movies of all time</legend>
>  <ol reversed>
>   <li><cite>Josie and the Pussycats</cite>, 2001</li>
>   <li><cite lang="sh">Црна мачка, бели мачор</cite>, 1998</li>
>   <li><cite>A Bug's Life</cite>, 1998</li>
>   <li><cite>Toy Story</cite>, 1995</li>
>   <li><cite>Monsters, Inc</cite>, 2001</li>
>   <li><cite>Cars</cite>, 2006</li>
>   <li><cite>Toy Story 2</cite>, 1999</li>
>   <li><cite>Finding Nemo</cite>, 2003</li>
>   <li><cite>The Incredibles</cite>, 2004</li>
>   <li><cite>Ratatouille</cite>, 2007</li>
>  </ol>
> </figure>
> ```

If the `li` element is the child of a `menu` element and itself has a child that defines a command, then the `li`

element must match the `:enabled` and `:disabled` pseudo-classes in the same way as the first such child element does.

4.4.10 The `dl` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

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

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `dl` element introduces an association list consisting of zero or more name-value groups (a description list). Each group must consist of one or more names (`dt` elements) followed by one or more values (`dd` elements).

Name-value groups may be terms and definitions, metadata topics and values, or any other groups of name-value data.

The values within a group are alternatives; multiple paragraphs forming part of the same value must all be given within the same `dd` element.

The order of the list of groups, and of the names and values within each group, may be significant.

If a `dl` element is empty, it contains no groups.

If a `dl` element contains non-whitespace text nodes, or elements other than `dt` and `dd`, then those elements or text nodes do not form part of any groups in that `dl`.

If a `dl` element contains only `dt` elements, then it consists of one group with names but no values.

If a `dl` element contains only `dd` elements, then it consists of one group with values but no names.

If a `dl` element starts with one or more `dd` elements, then the first group has no associated name.

If a `dl` element ends with one or more `dt` elements, then the last group has no associated value.

Note: When a `dl` element doesn't match its content model, it is often due to accidentally using `dd` elements in the place of `dt` elements and vice versa. Conformance checkers can spot such mistakes and might be able to advise authors how to correctly use the markup.

In the following example, one entry ("Authors") is linked to two values ("John" and "Luke").

```
<dl>
 <dt> Authors
 <dd> John
 <dd> Luke
 <dt> Editor
 <dd> Frank
</dl>
```

In the following example, one definition is linked to two terms.

```
<dl>
 <dt lang="en-US"> <dfn>color</dfn> </dt>
 <dt lang="en-GB"> <dfn>colour</dfn> </dt>
 <dd> A sensation which (in humans) derives from the ability of
 the fine structure of the eye to distinguish three differently
 filtered analyses of a view. </dd>
</dl>
```

The following example illustrates the use of the `dl` element to mark up metadata of sorts. At the end of the example, one group has two metadata labels ("Authors" and "Editors") and two values ("Robert Rothman" and "Daniel Jackson").

```
<dl>
 <dt> Last modified time </dt>
 <dd> 2004-12-23T23:33Z </dd>
 <dt> Recommended update interval </dt>
 <dd> 60s </dd>
 <dt> Authors </dt>
 <dt> Editors </dt>
 <dd> Robert Rothman </dd>
 <dd> Daniel Jackson </dd>
</dl>
```

The following example shows the `dl` element used to give a set of instructions. The order of the instructions here is important (in the other examples, the order of the blocks was not important).

```
<p>Determine the victory points as follows (use the
first matching case):</p>
<dl>
 <dt> If you have exactly five gold coins </dt>
 <dd> You get five victory points </dd>
 <dt> If you have one or more gold coins, and you have one or more silver
coins </dt>
 <dd> You get two victory points </dd>
 <dt> If you have one or more silver coins </dt>
 <dd> You get one victory point </dd>
 <dt> Otherwise </dt>
 <dd> You get no victory points </dd>
</dl>
```

The following snippet shows a `dl` element being used as a glossary. Note the use of `dfn` to indicate the

word being defined.

```
<dl>
 <dt><dfn>Apartment</dfn>, n.</dt>
 <dd>An execution context grouping one or more threads with one or
 more COM objects.</dd>
 <dt><dfn>Flat</dfn>, n.</dt>
 <dd>A deflated tire.</dd>
 <dt><dfn>Home</dfn>, n.</dt>
 <dd>The user's login directory.</dd>
</dl>
```

Note: The `dl` element is inappropriate for marking up dialogue, since dialogue is ordered (each speaker/line pair comes after the next). For an example of how to mark up dialogue, see the `dialog` element.

4.4.11 The `dt` element

Categories

None.

Contexts in which this element may be used:

Before `dd` or `dt` elements inside `dl` elements.
Before a `dd` element inside a `dialog` element.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `dt` element represents the term, or name, part of a term-description group in a description list (`dl` element), and the talker, or speaker, part of a talker-discourse pair in a conversation (`dialog` element).

Note: The `dt` element itself, when used in a `dl` element, does not indicate that its contents are a term being defined, but this can be indicated using the `dfn` element.

If the `dt` element is the child of a `dialog` element, and it further contains a `time` element, then that `time` element represents a timestamp for when the associated discourse (`dd` element) was said, and is not part of the name of the talker.

The following extract shows how an IM conversation log could be marked up.

```
<dialog>
 <dt> <time>14:22</time> egof
 <dd> I'm not that nerdy, I've only seen 30% of the star trek episodes
```

```
 <dt> <time>14:23</time> kaj
 <dd> if you know what percentage of the star trek episodes you have seen,
you are inarguably nerdy
 <dt> <time>14:23</time> egof
 <dd> it's unarguably
 <dt> <time>14:24</time> kaj
 <dd> you are not helping your case
</dialog>
```

4.4.12 The `dd` element

Categories

None.

Contexts in which this element may be used:

After `dt` or `dd` elements inside `dl` elements.
After a `dt` element inside a `dialog` element.

Content model:

Flow content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `dd` element represents the description, definition, or value, part of a term-description group in a description list (`dl` element), and the discourse, or quote, part in a conversation (`dialog` element).

4.5 Text-level semantics

4.5.1 The `a` element

Categories

Phrasing content.
Interactive content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content, but there must be no interactive content descendant.

Element-specific attributes:

`href`
`target`
`ping`
`rel`

media
hreflang
type

DOM interface:

```
[Stringifies=href] interface HTMLAnchorElement : HTMLElement {
           attribute DOMString href;
           attribute DOMString target;
           attribute DOMString ping;
           attribute DOMString rel;
  readonly attribute DOMTokenList relList;
           attribute DOMString media;
           attribute DOMString hreflang;
           attribute DOMString type;
};
```

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

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

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.

The `target`, `ping`, `rel`, `media`, `hreflang`, and `type` attributes must be omitted if the `href` attribute is not present.

If a site uses a consistent navigation 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`, `target` and `ping` attributes decide how the link is followed. The `rel`, `media`, `hreflang`, and `type` attributes may be used to indicate to the user the likely nature of the target resource before the user follows the link.

The activation behavior of `a` elements that represent hyperlinks is to run the following steps:

1. If the `DOMActivate` event in question is not trusted (i.e. a `click()` method call was the reason for the event being dispatched), and the `a` element's `target` attribute is ... then raise an `INVALID_ACCESS_ERR` exception and abort these steps.

2. If the target of the `DOMActivate` event is an `img` element with an `ismap` attribute specified, then server-side image map processing must be performed, as follows:

    1. If the `DOMActivate` event was dispatched as the result of a real pointing-device-triggered `click` event on the `img` element, then let *x* be the distance in CSS pixels from the left edge of the image to the location of the click, and let *y* be the distance in CSS pixels from the top edge of the image to the location of the click. Otherwise, let *x* and *y* be zero.

    2. Let the ***hyperlink suffix*** be a U+003F QUESTION MARK character, the value of *x* expressed as a base-ten integer using ASCII digits (U+0030 DIGIT ZERO to U+0039 DIGIT NINE), a U+002C COMMA character, and the value of *y* expressed as a base-ten integer using ASCII digits.

3. Finally, the user agent must follow the hyperlink defined by the `a` element. If the steps above defined a *hyperlink suffix*, then take that into account when following the hyperlink.

Note: One way that a user agent can enable users to follow hyperlinks is by allowing `a` elements to be clicked, or focussed and activated by the keyboard. This will cause the aforementioned activation behavior to be invoked.

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

The DOM attribute **`relList`** must reflect the `rel` content attribute.

4.5.2 The `q` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

`cite`

DOM interface:

The `q` element uses the `HTMLQuoteElement` interface.

The `q` element represents some phrasing content quoted from another source.

Quotation punctuation (such as quotation marks), if any, must be placed inside the `q` element.

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

If the `cite` attribute is present, it must be a valid URL. 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 title of the work given by that `cite` element gives the source of the quotation contained in the `q` element.

> Here is a simple example of the use of the `q` element:
>
> ```
> The man said <q>"Things that are impossible just take
> longer"</q>. I disagreed with him.
> ```

> Here is an example with both an explicit citation link in the `q` element, and an explicit citation outside:
>
> ```
> The W3C page <cite>About W3C</cite> says the W3C's
> mission is <q cite="http://www.w3.org/Consortium/">"To lead the
> World Wide Web to its full potential by developing protocols and
> guidelines that ensure long-term growth for the Web"</q>. I
> disagree with this mission.
> ```

> In the following example, the quotation itself contains a quotation:
>
> ```
> In <cite>Example One</cite>, he writes <q>"The man
> said <q>'Things that are impossible just take longer'</q>. I
> disagreed with him"</q>. Well, I disagree even more!
> ```

> In the following example, there are no quotation marks:
>
> ```
> His best argument: <q>I disagree!</q>
> ```

4.5.3 The `cite` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `cite` element represents the title of a work (e.g. a book, a paper, an essay, a poem, a score, a song, a script, a film, a TV show, a game, a sculpture, a painting, a theatre production, a play, an opera, a musical, an exhibition, etc). This can be a work that is being quoted or referenced in detail (i.e. a citation), or it can just be a work that is mentioned in passing.

A person's name is not the title of a work — even if people call that person a piece of work — and the element must therefore not be used to mark up people's names. (In some cases, the `b` element might be appropriate for names; e.g. in a gossip article where the names of famous people are keywords rendered with a different

style to draw attention to them. In other cases, if an element is *really* needed, the `span` element can be used.)

A ship is similarly not a work, and the element must not be used to mark up ship names (the `i` element can be used for that purpose).

> This next example shows a typical use of the `cite` element:
>
> ```
> My favourite book is <cite>The Reality Dysfunction</cite> by
> Peter F. Hamilton. My favourite comic is <cite>Pearls Before
> Swine</cite> by Stephan Pastis. My favourite track is <cite>Jive
> Samba</cite> by the Cannonball Adderley Sextet.
> ```

> This is correct usage:
>
> ```
> According to the Wikipedia article <cite>HTML</cite>, as it
> stood in mid-February 2008, leaving attribute values unquoted is
> unsafe. This is obviously an over-simplification.
> ```
>
> The following, however, is incorrect usage, as the `cite` element here is containing far more than the title of the work:
>
> ```
> <!-- do not copy this example, it is an example of bad usage! -->
> According to <cite>the Wikipedia article on HTML</cite>, as it
> stood in mid-February 2008, leaving attribute values unquoted is
> unsafe. This is obviously an over-simplification.
> ```

> The `cite` element is obviously a key part of any citation in a bibliography, but it is only used to mark the title:
>
> ```
> <cite>Universal Declaration of Human Rights</cite>, United Nations,
> December 1948.  Adopted by General Assembly resolution 217 A (III).
> ```

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

> This is incorrect usage, because `cite` is not for quotes:
>
> ```
> <cite>This is wrong!</cite>, said Ian.
> ```
>
> This is also incorrect usage, because a person is not a work:
>
> ```
> <q>This is still wrong!</q>, said <cite>Ian</cite>.
> ```
>
> The correct usage does not use a `cite` element:
>
> ```
> <q>This is correct</q>, said Ian.
> ```
>
> As mentioned above, the `b` element might be relevant for marking names as being keywords in certain kinds of documents:
>
> ```
> And then Ian said <q>this might be right, in a
> gossip column, maybe!</q>.
> ```

Note: The `cite` element can apply to `blockquote` and `q` elements in certain cases described

in the definitions of those elements.

This next example shows the use of `cite` alongside `blockquote`:

```
<p>His next piece was the aptly named <cite>Sonnet 130</cite>:</p>
<blockquote>
  <p>My mistress' eyes are nothing like the sun,<br>
  Coral is far more red, than her lips red,
  ...
```

4.5.4 The `em` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

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

The level of emphasis that a particular piece of content has is given by its number of ancestor `em` elements.

The placement of emphasis changes the meaning of the sentence. The element thus forms an integral part of the content. The precise way in which emphasis is used in this way depends on the language.

These examples show how changing the emphasis changes the meaning. First, a general statement of fact, with no emphasis:

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

By emphasizing the first word, the statement implies that the kind of animal under discussion is in question (maybe someone is asserting that dogs are cute):

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

Moving the emphasis to the verb, one highlights that the truth of the entire sentence is in question (maybe someone is saying cats are not cute):

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

By moving it to the adjective, the exact nature of the cats is reasserted (maybe someone suggested cats were *mean* animals):

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

Similarly, if someone asserted that cats were vegetables, someone correcting this might emphasize the last word:

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

By emphasizing the entire sentence, it becomes clear that the speaker is fighting hard to get the point across. This kind of emphasis also typically affects the punctuation, hence the exclamation mark here.

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

Anger mixed with emphasizing the cuteness could lead to markup such as:

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

4.5.5 The `strong` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `strong` element represents strong importance for its contents.

The relative level of importance of a piece of content is given by its number of ancestor `strong` elements; each `strong` element increases the importance of its contents.

Changing the importance of a piece of text with the `strong` element does not change the meaning of the sentence.

Here is an example of a warning notice in a game, with the various parts marked up according to how important they are:

```
<p><strong>Warning.</strong> This dungeon is dangerous.
<strong>Avoid the ducks.</strong> Take any gold you find.
<strong><strong>Do not take any of the diamonds</strong>,
they are explosive and <strong>will destroy anything within
ten meters.</strong></strong> You have been warned.</p>
```

4.5.6 The `small` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `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-emphasize" or lower the importance of text emphasised by the `em` element or marked as important with the `strong` element.

In this example the footer contains contact information and a copyright.

```
<footer>
 <address>
  For more details, contact
  <a href="mailto:js@example.com">John Smith</a>.
 </address>
 <p><small>© copyright 2038 Example Corp.</small></p>
</footer>
```

In this second example, the `small` element is used for a side comment.

```
<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.</small></strong></p>
```

4.5.7 The `mark` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `mark` element represents a run of text in one document marked or highlighted for reference purposes, due to its relevance in another context. When used in a quotation or other block of text referred to from the prose, it indicates a highlight that was not originally present but which has been added to bring the reader's attention to a part of the text that might not have been considered important by the original author when the block was originally written, but which is now under previously unexpected scrutiny. When used in the main prose of a document, it indicates a part of the document that has been highlighted due to its likely relevance to the user's current activity.

The rendering section will eventually suggest that user agents provide a way to let users jump between `mark` elements. Suggested rendering is a neon yellow background highlight, though UAs maybe should allow this to be toggled.

This example shows how the `mark` example can be used to bring attention to a particular part of a quotation:

```
<p lang="en-US">Consider the following quote:</p>
<blockquote lang="en-GB">
 <p>Look around and you will find, no-one's really
 <mark>colour</mark> blind.</p>
</blockquote>
<p lang="en-US">As we can tell from the <em>spelling</em> of the word,
the person writing this quote is clearly not American.</p>
```

Another example of the `mark` element is highlighting parts of a document that are matching some search string. If someone looked at a document, and the server knew that the user was searching for the word "kitten", then the server might return the document with one paragraph modified as follows:

```
<p>I also have some <mark>kitten</mark>s who are visiting me
these days. They're really cute. I think they like my garden! Maybe I
should adopt a <mark>kitten</mark>.</p>
```

In the following snippet, a paragraph of text refers to a specific part of a code fragment.

```
<p>The highlighted part below is where the error lies:</p>
<pre><code>var i: Integer;
begin
   i := <mark>1.1</mark>;
end.</code></pre>
```

This is another example showing the use of `mark` to highlight a part of quoted text that was originally not emphasised. In this example, common typographic conventions have led the author to explicitly style `mark` elements in quotes to render in italics.

```
<article>
 <style>
  blockquote mark, q mark {
    font: inherit; font-style: italic;
    text-decoration: none;
    background: transparent; color: inherit;
  }
  .bubble em {
    font: inherit; font-size: larger;
    text-decoration: underline;
  }
 </style>
 <h1>She knew</h1>
 <p>Did you notice the subtle joke in the joke on panel 4?</p>
 <blockquote>
  <p class="bubble">I didn't <em>want</em> to believe. <mark>Of course
  on some level I realized it was a known-plaintext attack.</mark> But I
  couldn't admit it until I saw for myself.</p>
 </blockquote>
 <p>(Emphasis mine.) I thought that was great. It's so pedantic, yet it
 explains everything neatly.</p>
</article>
```

Note, incidentally, the distinction between the `em` element in this example, which is part of the original text being quoted, and the `mark` element, which is highlighting a part for comment.

The following example shows the difference between denoting the *importance* of a span of text (`strong`) as opposed to denoting the *relevance* of a span of text (`mark`). It is an extract from a textbook, where the extract has had the parts relevant to the exam highlighted. The safety warnings, important though they may be, are apparently not relevant to the exam.

```
<h3>Wormhole Physics Introduction</h3>

<p><mark>A wormhole in normal conditions can be held open for a
maximum of just under 39 minutes.</mark> Conditions that can increase
the time include a powerful energy source coupled to one or both of
the gates connecting the wormhole, and a large gravity well (such as a
black hole).</p>

<p><mark>Momentum is preserved across the wormhole. Electromagnetic
radiation can travel in both directions through a wormhole,
but matter cannot.</mark></p>

<p>When a wormhole is created, a vortex normally forms.
<strong>Warning: The vortex caused by the wormhole opening will
annihilate anything in its path.</strong> Vortexes can be avoided when
using sufficiently advanced dialing technology.</p>

<p><mark>An obstruction in a gate will prevent it from accepting a
wormhole connection.</mark></p>
```

4.5.8 The `dfn` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content, but there must be no descendant `dfn` elements.

Element-specific attributes:

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

DOM interface:

Uses `HTMLElement`.

The `dfn` element represents the defining instance of a term. The paragraph, description list group, or section that is the nearest ancestor of the `dfn` element must also contain the definition(s) for the term given by the `dfn` element.

Defining term: If the `dfn` element has a **`title`** attribute, then the exact value of that attribute is the term being defined. Otherwise, if it contains exactly one element child node and no child text nodes, and that child element is an `abbr` element with a `title` attribute, then the exact value of *that* attribute is the term being defined. Otherwise, it is the exact `textContent` of the `dfn` element that gives the term being defined.

If the `title` attribute of the `dfn` element is present, then it must contain only the term being defined.

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

An `a` element that links to a `dfn` element represents an instance of the term defined by the `dfn` element.

> In the following fragment, the term "GDO" is first defined in the first paragraph, then used in the second.
>
> ```
> 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.
> ```
>
> With the addition of an `a` element, the reference can be made explicit:
>
> ```
> The <dfn id=gdo><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 <a href=#gdo><abbr title="Garage Door
> Opener">GDO</abbr></a>
> and so Hammond ordered the iris to be opened.
> ```

4.5.9 The `abbr` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

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

DOM interface:

Uses `HTMLElement`.

The `abbr` element represents an abbreviation or acronym, optionally with its expansion. The **`title`** attribute may be used to provide an expansion of the abbreviation. The attribute, if specified, must contain an expansion of the abbreviation, and nothing else.

> The paragraph below contains an abbreviation marked up with the `abbr` element. This paragraph defines the term "Web Hypertext Application Technology Working Group".
>
> ```
> The <dfn id=whatwg><abbr title="Web Hypertext Application
> Technology Working Group">WHATWG</abbr></dfn> is a loose
> unofficial collaboration of Web browser manufacturers and interested
> parties who wish to develop new technologies designed to allow authors
> to write and deploy Applications over the World Wide Web.
> ```

> This paragraph has two abbreviations. Notice how only one is defined; the other, with no expansion associated with it, does not use the `abbr` element.
>
> ```
> The <abbr title="Web Hypertext Application Technology Working
> Group">WHATWG</abbr> started working on HTML5 in 2004.
> ```

> This paragraph links an abbreviation to its definition.
>
> ```
> The <a href="#whatwg"><abbr title="Web Hypertext Application
> Technology Working Group">WHATWG</abbr></a> community does not
> have much representation from Asia.
> ```

> This paragraph marks up an abbreviation without giving an expansion, possibly as a hook to apply styles for abbreviations (e.g. smallcaps).
>
> ```
> Philip` and Dashiva both denied that they were going to
> get the issue counts from past revisions of the specification to
> backfill the <abbr>WHATWG</abbr> issue graph.
> ```

If an abbreviation is pluralized, the expansion's grammatical number (plural vs singular) must match the grammatical number of the contents of the element.

> Here the plural is outside the element, so the expansion is in the singular:

```
<p>Two <abbr title="Working Group">WG</abbr>s worked on
this specification: the <abbr>WHATWG</abbr> and the
<abbr>HTMLWG</abbr>.</p>
```

Here the plural is inside the element, so the expansion is in the plural:

```
<p>Two <abbr title="Working Groups">WGs</abbr> worked on
this specification: the <abbr>WHATWG</abbr> and the
<abbr>HTMLWG</abbr>.</p>
```

4.5.10 The `time` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

`datetime`

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 these steps:

1. If the `datetime` attribute is present, then parse it according to the rules for parsing date or time strings in attributes, 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 next
> morning</time>.
> ```
>
> ...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.

4.5.11 The `progress` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

`value`
`max`

DOM interface:

```
interface HTMLProgressElement : HTMLElement {
           attribute float value;
           attribute float max;
  readonly attribute float position;
};
```

The `progress` element represents the completion progress of a task. The progress is either indeterminate, indicating that progress is being made but that it is not clear how much more work remains to be done before the task is complete (e.g. because the task is waiting for a remote host to respond), or the progress is a number in the range zero to a maximum, giving the fraction of work that has so far been completed.

There are two attributes that determine the current task completion represented by the element.

The **`value`** attribute specifies how much of the task has been completed, and the **`max`** attribute specifies how much work the task requires in total. The units are arbitrary and not specified.

Instead of using the attributes, authors are recommended to include the current value and the maximum value inline as text inside the element.

Here is a snippet of a Web application that shows the progress of some automated task:

```
<section>
 <h2>Task Progress</h2>
 <p>Progress: <progress><span id="p">0</span>%</progress></p>
 <script>
  var progressBar = document.getElementById('p');
  function updateProgress(newValue) {
    progressBar.textContent = newValue;
  }
 </script>
</section>
```

(The `updateProgress()` method in this example would be called by some other code on the page to update the actual progress bar as the task progressed.)

Author requirements: The `max` and `value` attributes, when present, must have values that are valid floating point numbers. The `max` attribute, if present, must have a value greater than zero. The `value` attribute, if

present, must have a value equal to or greater than zero, and less than or equal to the value of the `max` attribute, if present.

> ***Note: The `progress` element is the wrong element to use for something that is just a gauge, as opposed to task progress. For instance, indicating disk space usage using `progress` would be inappropriate. Instead, the `meter` element is available for such use cases.***

User agent requirements: User agents must parse the `max` and `value` attributes' values according to the rules for parsing floating point number values.

If the `value` attribute is omitted, then user agents must also parse the `textContent` of the `progress` element in question using the steps for finding one or two numbers of a ratio in a string. These steps will return nothing, one number, one number with a denominator punctuation character, or two numbers.

Using the results of this processing, user agents must determine whether the progress bar is an indeterminate progress bar, or whether it is a determinate progress bar, and in the latter case, what its current and maximum values are, all as follows:

1. If the `max` attribute is omitted, and the `value` is omitted, and the results of parsing the `textContent` was nothing, then the progress bar is an indeterminate progress bar. Abort these steps.
2. Otherwise, it is a determinate progress bar.
3. If the `max` attribute is included, then, if a value could be parsed out of it, then the maximum value is that value.
4. Otherwise, if the `max` attribute is absent but the `value` attribute is present, or, if the `max` attribute is present but no value could be parsed from it, then the maximum is 1.
5. Otherwise, if neither attribute is included, then, if the `textContent` contained one number with an associated denominator punctuation character, then the maximum value is the value associated with that denominator punctuation character; otherwise, if the `textContent` contained two numbers, the maximum value is the higher of the two values; otherwise, the maximum value is 1.
6. If the `value` attribute is present on the element and a value could be parsed out of it, that value is the current value of the progress bar. Otherwise, if the attribute is present but no value could be parsed from it, the current value is zero.
7. Otherwise if the `value` attribute is absent and the `max` attribute is present, then, if the `textContent` was parsed and found to contain just one number, with no associated denominator punctuation character, then the current value is that number. Otherwise, if the `value` attribute is absent and the `max` attribute is present then the current value is zero.
8. Otherwise, if neither attribute is present, then the current value is the lower of the one or two numbers that were found in the `textContent` of the element.
9. If the maximum value is less than or equal to zero, then it is reset to 1.
10. If the current value is less than zero, then it is reset to zero.
11. Finally, if the current value is greater than the maximum value, then the current value is reset to the maximum value.

UA requirements for showing the progress bar: When representing a `progress` element to the user, the UA should indicate whether it is a determinate or indeterminate progress bar, and in the former case, should indicate the relative position of the current value relative to the maximum value.

The **`max`** and **`value`** DOM attributes must reflect the 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.

4.5.12 The `meter` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

`value`
`min`
`low`
`high`
`max`
`optimum`

DOM interface:

```
interface HTMLMeterElement : HTMLElement {
           attribute float value;
           attribute float min;
           attribute float max;
           attribute float low;
           attribute float high;
           attribute float optimum;
};
```

The `meter` element represents a scalar measurement within a known range, or a fractional value; for example disk usage, the relevance of a query result, or the fraction of a voting population to have selected a particular candidate.

This is also known as a gauge.

Note: The `meter` element should not be used to indicate progress (as in a progress bar). For that role, HTML provides a separate `progress` element.

Note: The `meter` element also does not represent a scalar value of arbitrary range — for example, it would be wrong to use this to report a weight, or height, unless there is a known maximum value.

There are six attributes that determine the semantics of the gauge represented by the element.

The **`min`** attribute specifies the lower bound of the range, and the **`max`** attribute specifies the upper bound. The **`value`** attribute specifies the value to have the gauge indicate as the "measured" value.

The other three attributes can be used to segment the gauge's range into "low", "medium", and "high" parts, and to indicate which part of the gauge is the "optimum" part. The **`low`** attribute specifies the range that is considered to be the "low" part, and the **`high`** attribute specifies the range that is considered to be the "high" part. The **`optimum`** attribute gives the position that is "optimum"; if that is higher than the "high" value then this indicates that the higher the value, the better; if it's lower than the "low" mark then it indicates that lower values are better, and naturally if it is in between then it indicates that neither high nor low values are good.

Authoring requirements: The recommended way of giving the value is to include it as contents of the element, either as two numbers (the higher number represents the maximum, the other number the current value, and the minimum is assumed to be zero), or as a percentage or similar (using one of the characters such as "%"), or as a fraction.

The `value`, `min`, `low`, `high`, `max`, and `optimum` attributes are all optional. When present, they must have values that are valid floating point numbers, and their values must satisfy the following inequalities:

- `min` ≤ `value` ≤ `max`
- `min` ≤ `low` ≤ `high` ≤ `max`
- `min` ≤ `optimum` ≤ `max`

> The following examples all represent a measurement of three quarters (of the maximum of whatever is being measured):
>
> ```
> <meter>75%</meter>
> <meter>750%</meter>
> <meter>3/4</meter>
> <meter>6 blocks used (out of 8 total)</meter>
> <meter>max: 100; current: 75</meter>
> <meter><object data="graph75.png">0.75</object></meter>
> <meter min="0" max="100" value="75"></meter>
> ```
>
> The following example is incorrect use of the element, because it doesn't give a range (and since the default maximum is 1, both of the gauges would end up looking maxed out):
>
> ```
> The grapefruit pie had a radius of <meter>12cm</meter>
> and a height of <meter>2cm</meter>. <!-- BAD! -->
> ```
>
> Instead, one would either not include the meter element, or use the meter element with a defined range to give the dimensions in context compared to other pies:
>
> ```
> The grapefruit pie had a radius of 12cm and a height of
> ```

```
2cm.</p>
<dl>
 <dt>Radius: <dd> <meter min=0 max=20 value=12>12cm</meter>
 <dt>Height: <dd> <meter min=0 max=10 value=2>2cm</meter>
</dl>
```

There is no explicit way to specify units in the `meter` element, but the units may be specified in the `title` attribute in free-form text.

The example above could be extended to mention the units:

```
<dl>
 <dt>Radius: <dd> <meter min=0 max=20 value=12
title="centimeters">12cm</meter>
 <dt>Height: <dd> <meter min=0 max=10 value=2
title="centimeters">2cm</meter>
</dl>
```

User agent requirements: User agents must parse the `min`, `max`, `value`, `low`, `high`, and `optimum` attributes using the rules for parsing floating point number values.

If the `value` attribute has been omitted, the user agent must also process the `textContent` of the element according to the steps for finding one or two numbers of a ratio in a string. These steps will return nothing, one number, one number with a denominator punctuation character, or two numbers.

User agents must then use all these numbers to obtain values for six points on the gauge, as follows. (The order in which these are evaluated is important, as some of the values refer to earlier ones.)

The minimum value

If the `min` attribute is specified and a value could be parsed out of it, then the minimum value is that value. Otherwise, the minimum value is zero.

The maximum value

If the `max` attribute is specified and a value could be parsed out of it, the maximum value is that value.

Otherwise, if the `max` attribute is specified but no value could be parsed out of it, or if it was not specified, but either or both of the `min` or `value` attributes *were* specified, then the maximum value is 1.

Otherwise, none of the `max`, `min`, and `value` attributes were specified. If the result of processing the `textContent` of the element was either nothing or just one number with no denominator punctuation character, then the maximum value is 1; if the result was one number but it had an associated denominator punctuation character, then the maximum value is the value associated with that denominator punctuation character; and finally, if there were two numbers parsed out of the `textContent`, then the maximum is the higher of those two numbers.

If the above machinations result in a maximum value less than the minimum value, then the maximum value is actually the same as the minimum value.

The actual value

If the `value` attribute is specified and a value could be parsed out of it, then that value is the actual value.

If the `value` attribute is not specified but the `max` attribute *is* specified and the result of processing the `textContent` of the element was one number with no associated denominator punctuation character, then that number is the actual value.

If neither of the `value` and `max` attributes are specified, then, if the result of processing the `textContent` of the element was one number (with or without an associated denominator punctuation character), then that is the actual value, and if the result of processing the `textContent` of the element was two numbers, then the actual value is the lower of the two numbers found.

Otherwise, if none of the above apply, the actual value is zero.

If the above procedure results in an actual value less than the minimum value, then the actual value is actually the same as the minimum value.

If, on the other hand, the result is an actual value greater than the maximum value, then the actual value is the maximum value.

The low boundary

If the `low` attribute is specified and a value could be parsed out of it, then the low boundary is that value. Otherwise, the low boundary is the same as the minimum value.

If the 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:

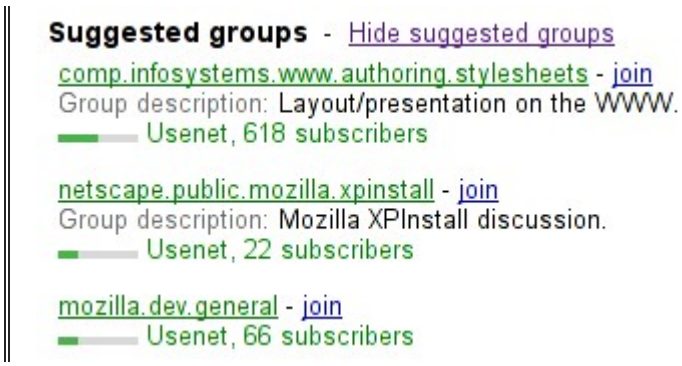
- minimum value ≤ actual value ≤ maximum value
- minimum value ≤ low boundary ≤ high boundary ≤ maximum value
- minimum value ≤ optimum point ≤ maximum value

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
> groups</a>
> </menu>
> <ul>
>  <li>
>   <a href="/group/comp.infosystems.www.authoring.stylesheets
> /view">comp.infosystems.www.authoring.stylesheets</a> -
>      <a href="/group/comp.infosystems.www.authoring.stylesheets
> /subscribe">join</a>
>   Group description: Layout/presentation on the
> WWW.
>   <meter value="0.5">Moderate activity,</meter> Usenet, 618
> subscribers
>  </li>
>  <li>
>   <a href="/group/netscape.public.mozilla.xpinstall
> /view">netscape.public.mozilla.xpinstall</a> -
>      <a href="/group/netscape.public.mozilla.xpinstall
> /subscribe">join</a>
>   Group description: Mozilla XPInstall discussion.
>   <meter value="0.25">Low activity,</meter> Usenet, 22 subscribers
>  </li>
>  <li>
>   <a href="/group/mozilla.dev.general/view">mozilla.dev.general</a> -
>      <a href="/group/mozilla.dev.general/subscribe">join</a>
>   <meter value="0.25">Low activity,</meter> Usenet, 66 subscribers
>  </li>
> </ul>
> ```
>
> Might be rendered as follows:



User agents may combine the value of the `title` attribute and the other attributes to provide context-sensitive help or inline text detailing the actual values.

> For example, the following snippet:
>
> ```
> <meter min=0 max=60 value=23.2 title=seconds></meter>
> ```
>
> ...might cause the user agent to display a gauge with a tooltip saying "Value: 23.2 out of 60." on one line and "seconds" on a second line.

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

4.5.13 The `code` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `code` element represents a fragment of computer code. This could be an XML element name, a filename, a computer program, or any other string that a computer would recognize.

Although there is no formal way to indicate the language of computer code being marked up, authors who wish to mark `code` elements with the language used, e.g. so that syntax highlighting scripts can use the right rules, may do so by adding a class prefixed with "`language-`" to the element.

The following example shows how the element can be used in a paragraph to mark up element names and computer code, including punctuation.

```
<p>The <code>code</code> element represents a fragment of computer
code.</p>

<p>When you call the <code>activate()</code> method on the
<code>robotSnowman</code> object, the eyes glow.</p>

<p>The example below uses the <code>begin</code> keyword to indicate
the start of a statement block. It is paired with an <code>end</code>
keyword, which is followed by the <code>.</code> punctuation character
(full stop) to indicate the end of the program.</p>
```

The following example shows how a block of code could be marked up using the `pre` and `code` elements.

```
<pre><code class="language-pascal">var i: Integer;
begin
   i := 1;
end.</code></pre>
```

A class is used in that example to indicate the language used.

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

4.5.14 The `var` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `var` element represents a variable. This could be an actual variable in a mathematical expression or programming context, or it could just be a term used as a placeholder in prose.

In the paragraph below, the letter "n" is being used as a variable in prose:

```
<p>If there are <var>n</var> pipes leading to the ice
cream factory then I expect at <em>least</em> <var>n</var>
flavours of ice cream to be available for purchase!</p>
```

4.5.15 The `samp` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

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

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

This example shows the `samp` element being used inline:

```
<p>The computer said <samp>Too much cheese in tray
two</samp> but I didn't know what that meant.</p>
```

This second example shows a block of sample output. Nested `samp` and `kbd` elements allow for the styling of specific elements of the sample output using a style sheet.

```
<pre><samp><samp class="prompt">jdoe@mowmow:~$</samp> <kbd>ssh
demo.example.com</kbd>
Last login: Tue Apr 12 09:10:17 2005 from mowmow.example.com on pts/1
Linux demo 2.6.10-grsec+gg3+e+fhs6b+nfs+gr0501+++p3+c4a+gr2b-reslog-v6.189
#1 SMP Tue Feb 1 11:22:36 PST 2005 i686 unknown

<samp class="prompt">jdoe@demo:~$</samp> <samp class="cursor">_</samp>
</samp></pre>
```

4.5.16 The `kbd` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `kbd` element represents user input (typically keyboard input, although it may also be used to represent other input, such as voice commands).

When the `kbd` element is nested inside a `samp` element, it represents the input as it was echoed by the system.

When the `kbd` element *contains* a `samp` element, it represents input based on system output, for example invoking a menu item.

When the `kbd` element is nested inside another `kbd` element, it represents an actual key or other single unit of input as appropriate for the input mechanism.

> Here the `kbd` element is used to indicate keys to press:
>
> ```
> To make George eat an apple, press <kbd><kbd>Shift</kbd>+<kbd>F3</kbd>
> </kbd>
> ```
>
> In this second example, the user is told to pick a particular menu item. The outer `kbd` element marks up a block of input, with the inner `kbd` elements representing each individual step of the input, and the `samp` elements inside them indicating that the steps are input based on something being displayed by the system, in this case menu labels:
>
> ```
> To make George eat an apple, select
>     <kbd><kbd><samp>File</samp></kbd>|<kbd><samp>Eat Apple...</samp></kbd>
> </kbd>
> 
> ```

4.5.17 The `sub` and `sup` elements

Categories

Phrasing content.

Contexts in which these elements may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

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

These elements must be used only to mark up typographical conventions with specific meanings, not for typographical presentation for presentation's sake. For example, it would be inappropriate for the `sub` and `sup`

elements to be used in the name of the LaTeX document preparation system. In general, authors should use these elements only if the *absence* of those elements would change the meaning of the content.

When the `sub` element is used inside a `var` element, it represents the subscript that identifies the variable in a family of variables.

```
<p>The coordinate of the <var>i</var>th point is
(<var>x<sub><var>i</var></sub></var>, <var>y<sub><var>i</var></sub>
</var>).
For example, the 10th point has coordinate
(<var>x<sub>10</sub></var>, <var>y<sub>10</sub></var>).</p>
```

In certain languages, superscripts are part of the typographical conventions for some abbreviations.

```
<p>The most beautiful women are
<span lang="fr"><abbr>M<sup>lle</sup></abbr> Gwendoline</span> and
<span lang="fr"><abbr>M<sup>me</sup></abbr> Denise</span>.</p>
```

Mathematical expressions often use subscripts and superscripts. Authors are encouraged to use MathML for marking up mathematics, but authors may opt to use `sub` and `sup` if detailed mathematical markup is not desired. [MathML]

```
<var>E</var>=<var>m</var><var>c</var><sup>2</sup>

f(<var>x</var>, <var>n</var>) = log<sub>4</sub><var>x</var>
<sup><var>n</var></sup>
```

4.5.18 The `span` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `span` element doesn't mean anything on its own, but can be useful when used together with other attributes, e.g. `class`, `lang`, or `dir`.

4.5.19 The `i` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `i` element represents a span of text in an alternate voice or mood, or otherwise offset from the normal prose, such as a taxonomic designation, a technical term, an idiomatic phrase from another language, a thought, a ship name, or some other prose whose typical typographic presentation is italicized.

Terms in languages different from the main text should be annotated with `lang` attributes (`xml:lang` in XML).

The examples below show uses of the `i` element:

```
<p>The <i class="taxonomy">Felis silvestris catus</i> is cute.</p>
<p>The term <i>prose content</i> is defined above.</p>
<p>There is a certain <i lang="fr">je ne sais quoi</i> in the air.</p>
```

In the following example, a dream sequence is marked up using `i` elements.

```
<p>Raymond tried to sleep.</p>
<p><i>The ship sailed away on Thursday</i>, he
dreamt. <i>The ship had many people aboard, including a beautiful
princess called Carey. He watched her, day-in, day-out, hoping she
would notice him, but she never did.</i></p>
<p><i>Finally one night he picked up the courage to speak with
her—</i></p>
<p>Raymond woke with a start as the fire alarm rang out.</p>
```

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.

Authors are encouraged to use the `class` attribute on the `i` element to identify why the element is being used, so that if the style of a particular use (e.g. dream sequences as opposed to taxonomic terms) is to be changed at a later date, the author doesn't have to go through the entire document (or series of related documents) annotating each use.

Note: Style sheets can be used to format `i` elements, just like any other element can be restyled. Thus, it is not the case that content in `i` elements will necessarily be italicized.

4.5.20 The `b` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `b` element represents a span of text to be stylistically offset from the normal prose without conveying any extra importance, such as key words in a document abstract, product names in a review, or other spans of text whose typical typographic presentation is boldened.

> The following example shows a use of the `b` element to highlight key words without marking them up as important:
>
> ```
> The frobonitor and barbinator components are fried.
> ```

> In the following example, objects in a text adventure are highlighted as being special by use of the `b` element.
>
> ```
> You enter a small room. Your sword glows
> brighter. A rat scurries past the corner wall.
> ```

> Another case where the `b` element is appropriate is in marking up the lede (or lead) sentence or paragraph. The following example shows how a BBC article about kittens adopting a rabbit as their own could be marked up using HTML5 elements:
>
> ```
> <article>
>  <h2>Kittens 'adopted' by pet rabbit</h2>
>  Six abandoned kittens have found an unexpected new
>  mother figure — a pet rabbit.
>  Veterinary nurse Melanie Humble took the three-week-old
>  kittens to her Aberdeen home.
> [...]
> ```

The `b` element should be used as a last resort when no other element is more appropriate. In particular, headers should use the `h1` to `h6` elements, stress emphasis should use the `em` element, importance should be denoted with the `strong` element, and text marked or highlighted should use the `mark` element.

> The following would be *incorrect* usage:
>
> ```
> WARNING! Do not frob the barbinator!
> ```
>
> In the previous example, the correct element to use would have been `strong`, not `b`.

Note: Style sheets can be used to format `b` elements, just like any other element can be restyled. Thus, it is not the case that content in `b` elements will necessarily be boldened.

4.5.21 The `bdo` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None, but the `dir` global attribute has special requirements on this element.

DOM interface:

Uses `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]

4.5.22 The `ruby` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected.

Content model:

One or more groups of: phrasing content followed either by a single `rt` element, or an `rp` element, an `rt` element, and another `rp` element.

Element-specific attributes:

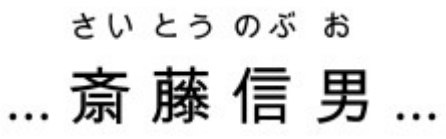
None.

DOM interface:

Uses `HTMLElement`.

The `ruby` element allows one or more spans of phrasing content to be marked with ruby annotations.

A `ruby` element represents the spans of phrasing content it contains, ignoring all the child `rt` and `rp` elements and their descendants. Those spans of phrasing content have associated annotations created using the `rt` element.

> In this example, each ideograph in the text 斎藤信男 is annotated with its reading.
>
> ```
> ... <ruby>
>  斎 <rt> さい </rt>
>  藤 <rt> とう </rt>
>  信 <rt> のぶ </rt>
>  男 <rt> お       </rt>
> </ruby> ...
> ```
>
> This might be rendered as:
>
> 
>

4.5.23 The `rt` element

Categories

None.

Contexts in which this element may be used:

As a child of a `ruby` element.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `rt` element marks the ruby text component of a ruby annotation.

An `rt` element that is a child of a `ruby` element represents an annotation (given by its children) for the zero or more nodes of phrasing content that immediately precedes it in the `ruby` element, ignoring `rp` elements.

An `rt` element that is not a child of a `ruby` element represents the same thing as its children.

4.5.24 The `rp` element

Categories

None.

Contexts in which this element may be used:

As a child of a `ruby` element, either immediately before or immediately after an `rt` element.

Content model:

If the `rp` element is immediately after an `rt` element that is immediately preceded by another `rp` element: a single character from Unicode character class Pe.
Otherwise: a single character from Unicode character class Ps.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `rp` element can be used to provide parentheses around a ruby text component of a ruby annotation, to be shown by user agents that don't support ruby annotations.

An `rp` element that is a child of a `ruby` element represents nothing and it and its contents must be ignored.
An `rp` element whose parent element is not a `ruby` element represents the same thing as its children.

> The example above, in which each ideograph in the text 斎藤信男 is annotated with its reading, could be expanded to use `rp` so that in legacy user agentthe readings are in parentheses:
>
> ```
> ... <ruby>
>  斎 <rp>(</rp><rt>さい</rt><rp>)</rp>
>  藤 <rp>(</rp><rt>とう</rt><rp>)</rp>
>  信 <rp>(</rp><rt>のぶ</rt><rp>)</rp>
>  男 <rp>(</rp><rt>お</rt><rp>)</rp>
> </ruby> ...
> ```
>
> In conforming user agents the rendering would be as above, but in user agents that do not support ruby, the rendering would be:
>
> ... 斎（さい）藤（とう）信（のぶ）男（お）...

4.5.25 Usage summary

We need to summarize the various elements, in particular to distinguish b/i/em/strong/var/q/mark/cite.

4.5.26 Footnotes

HTML does not have a dedicated mechanism for marking up footnotes. Here are the recommended alternatives.

For short inline annotations, the `title` attribute should be used.

> In this example, two parts of a dialog are annotated.
>
> ```
> <dialog>
>  <dt>Customer
>  <dd>Hello! I wish to register a complaint. Hello. Miss?
>  <dt>Shopkeeper
>  <dd><span title="Colloquial pronunciation of 'What do you'"
>  >Watcha mean, miss?
>  <dt>Customer
>  <dd>Uh, I'm sorry, I have a cold. I wish to make a complaint.
>  <dt>Shopkeeper
>  <dd>Sorry, we're
>  closing for lunch.
> </dialog>
> ```

For longer annotations, the `a` element should be used, pointing to an element later in the document. The convention is that the contents of the link be a number in square brackets.

> In this example, a footnote in the dialog links to a paragraph below the dialog. The paragraph then reciprocally links back to the dialog, allowing the user to return to the location of the footnote.
>
> ```
> <dialog>
>  <dt>Announcer
>  <dd>Number 16: The hand.
>  <dt>Interviewer
>  <dd>Good evening. I have with me in the studio tonight Mr
>  Norman St John Polevaulter, who for the past few years has
>  been contradicting people. Mr Polevaulter, why do
>  you contradict people?
>  <dt>Norman
>  <dd>I don't. <a href="#fn1" id="r1">[1]</a>
>  <dt>Interviewer
>  <dd>You told me you did!
> </dialog>
> <section>
>  <a href="#r1">[1]</a> This is, naturally, a lie,
>  but paradoxically if it were true he could not say so without
>  contradicting the interviewer and thus making it false.
> </section>
> ```

For side notes, longer annotations that apply to entire sections of the text rather than just specific words or sentences, the `aside` element should be used.

> In this example, a sidebar is given after a dialog, giving some context to the dialog.
>
> ```
> <dialog>
>  <dt>Customer
>  <dd>I will not buy this record, it is scratched.
>  <dt>Shopkeeper
>  <dd>I'm sorry?
> ```

```
 <dt>Customer
 <dd>I will not buy this record, it is scratched.
 <dt>Shopkeeper
 <dd>No no no, this's'a tobacconist's.
</dialog>
<aside>
 <p>In 1970, the British Empire lay in ruins, and foreign
 nationalists frequented the streets — many of them Hungarians
 (not the streets — the foreign nationals). Sadly, Alexander
 Yalt has been publishing incompetently-written phrase books.
</aside>
```

4.6 Edits

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

4.6.1 The `ins` element

Categories

When the element only contains phrasing content: phrasing content.
Otherwise: flow content.

Contexts in which this element may be used:

When the element only contains phrasing content: where phrasing content is expected.
Otherwise: where flow content is expected.

Content model:

Transparent.

Element-specific attributes:

`cite`
`datetime`

DOM interface:

Uses the `HTMLModElement` interface.

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

The following represents the addition of a single paragraph:

```
<aside>
 <ins>
  <p> I like fruit. </p>
 </ins>
</aside>
```

As does this, because everything in the `aside` element here counts as phrasing content and therefore there is just one paragraph:

```
<aside>
 <ins>
  Apples are <em>tasty</em>.
 </ins>
 <ins>
  So are pears.
 </ins>
</aside>
```

`ins` elements should not cross implied paragraph boundaries.

The following example represents the addition of two paragraphs, the second of which was inserted in two parts. The first `ins` element in this example thus crosses a paragraph boundary, which is considered poor form.

```
<aside>
 <ins datetime="2005-03-16T00:00Z">
  <p> I like fruit. </p>
  Apples are <em>tasty</em>.
 </ins>
 <ins datetime="2007-12-19T00:00Z">
  So are pears.
 </ins>
</aside>
```

Here is a better way of marking this up. It uses more elements, but none of the elements cross implied paragraph boundaries.

```
<aside>
 <ins datetime="2005-03-16T00:00Z">
  <p> I like fruit. </p>
 </ins>
 <ins datetime="2005-03-16T00:00Z">
  Apples are <em>tasty</em>.
 </ins>
 <ins datetime="2007-12-19T00:00Z">
  So are pears.
 </ins>
</aside>
```

4.6.2 The `del` element

Categories

When the element only contains phrasing content: phrasing content.
Otherwise: flow content.

Contexts in which this element may be used:

When the element only contains phrasing content: where phrasing content is expected.
Otherwise: where flow content is expected.

Content model:

Transparent.

Element-specific attributes:

`cite`
`datetime`

DOM interface:

Uses the `HTMLModElement` interface.

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

`del` elements should not cross implied paragraph boundaries.

4.6.3 Attributes common to `ins` and `del` elements

The **`cite`** attribute may be used to specify the address of a document that explains the change. When that document is long, for instance the minutes of a meeting, authors are encouraged to include a fragment identifier pointing to the specific part of that document that discusses the change.

If the `cite` attribute is present, it must be a valid URL that explains the change. 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`** DOM attribute must reflect the element's `cite` content attribute. The **`dateTime`** DOM attribute must reflect the element's `datetime` content attribute.

4.6.4 Edits and paragraphs

Since the `ins` and `del` elements do not affect paragraphing, it is possible, in some cases where paragraphs are implied (without explicit `p` elements), for an `ins` or `del` element to span both an entire paragraph or other non-phrasing content elements and part of another paragraph.

For example:

```
<section>
 <ins>
  <p>
   This is a paragraph that was inserted.
  </p>
  This is another paragraph whose first sentence was inserted
  at the same time as the paragraph above.
 </ins>
 This is a second sentence, which was there all along.
</section>
```

By only wrapping some paragraphs in `p` elements, one can even get the end of one paragraph, a whole second paragraph, and the start of a third paragraph to be covered by the same `ins` or `del` element (though this is very confusing, and not considered good practice):

```
<section>
 This is the first paragraph. <ins>This sentence was
 inserted.
 <p>This second paragraph was inserted.</p>
 This sentence was inserted too.</ins> This is the
 third paragraph in this example.
</section>
```

However, due to the way implied paragraphs are defined, it is not possible to mark up the end of one paragraph and the start of the very next one using the same `ins` or `del` element. You instead have to use one (or two) `p` element(s) and two `ins` or `del` elements:

For example:

```
<section>
 <p>This is the first paragraph. <del>This sentence was
 deleted.</del></p>
 <p><del>This sentence was deleted too.</del> That
 sentence needed a separate &lt;del&gt; element.</p>
</section>
```

Partly because of the confusion described above, authors are strongly recommended to always mark up all paragraphs with the `p` element, and to not have any `ins` or `del` elements that cross across any implied paragraphs.

4.6.5 Edits and lists

The content models of the `ol` and `ul` elements do not allow `ins` and `del` elements as children. Lists always represent all their items, including items that would otherwise have been marked as deleted.

To indicate that an item is inserted or deleted, an `ins` or `del` element can be wrapped around the contents of the `li` element. To indicate that an item has been replaced by another, a single `li` element can have one or more `del` elements followed by one or more `ins` elements.

> In the following example, a list that started empty had items added and removed from it over time. The bits in the example that have been emphasised show the parts that are the "current" state of the list. The

list item numbers don't take into account the edits, though.

```
<h1>Stop-ship bugs</h1>
<ol>
 <li><ins datetime="2008-02-12 15:20 Z">Bug 225: Rain detector
 doesn't work in snow</ins></li>
 <li><del datetime="2008-03-01 20:22 Z"><ins datetime="2008-02-14
 12:02 Z">Bug 228: Water buffer overflows in April</ins></del></li>
 <li><ins datetime="2008-02-16 13:50 Z">Bug 230: Water heater
 doesn't use renewable fuels</ins></li>
 <li><del datetime="2008-02-20 21:15 Z"><ins datetime="2008-02-16
 14:25 Z">Bug 232: Carbon dioxide emissions detected after
 startup</ins></del></li>
</ol>
```

In the following example, a list that started with just fruit was replaced by a list with just colors.

```
<h1>List of <del>fruits</del><ins>colors</ins></h1>
<ul>
 <li><del>Lime</del><ins>Green</ins></li>
 <li><del>Apple</del></li>
 <li>Orange</li>
 <li><del>Pear</del></li>
 <li><ins>Teal</ins></li>
 <li><del>Lemon</del><ins>Yellow</ins></li>
 <li>Olive</li>
 <li><ins>Purple</ins>
</ul>
```

4.7 Embedded content

4.7.1 The `figure` element

Categories

Flow content.
Sectioning root.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Either: one `legend` element followed by flow content.
Or: Flow content followed by one `legend` element.
Or: Flow content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `figure` element represents some flow content, optionally with a caption, which can be moved away from the main flow of the document without affecting the document's meaning.

The element can thus be used to annotate illustrations, diagrams, photos, code listings, etc, that are referred to from the main content of the document, but that could, without affecting the flow of the document, be moved away from that primary content, e.g. to the side of the page, to dedicated pages, or to an appendix.

The first `legend` element child of the element, if any, represents the caption of the `figure` element's contents. If there is no child `legend` element, then there is no caption.

The remainder of the element's contents, if any, represents the content.

> This example shows the `figure` element to mark up a code listing.
>
> ```
> In <a href="#l4">listing 4</a> we see the primary core interface
> API declaration.
> <figure id="l4">
>  <legend>Listing 4. The primary core interface API declaration.</legend>
>  <pre><code>interface PrimaryCore {
>   boolean verifyDataLine();
>   void sendData(in sequence<byte> data);
>   void initSelfDestruct();
> }</code></pre>
> </figure>
> The API is designed to use UTF-8.
> ```

> Here we see a `figure` element to mark up a photo.
>
> ```
> <figure>
>  <img src="bubbles-work.jpeg"
>       alt="Bubbles, sitting in his office chair, works on his
>            latest project intently.">
>  <legend>Bubbles at work</legend>
> </figure>
> ```

> In this example, we see an image that is *not* a figure, as well as an image and a video that are.
>
> ```
> <h2>Malinko's comics</h2>
>
> This case centered on some sort of "intellectual property"
> infringement related to a comic (see Exhibit A). The suit started
> after a trailer ending with these words:
>
> <img src="promblem-packed-action.png" alt="ROUGH COPY! Promblem-Packed
> Action!">
>
> ...was aired. A lawyer, armed with a Bigger Notebook, launched a
> preemptive strike using snowballs. A complete copy of the trailer is
> included with Exhibit B.
>
> <figure>
>  <img src="ex-a.png" alt="Two squiggles on a dirty piece of paper.">
> ```

```
 <legend>Exhibit A. The alleged <cite>rough copy</cite> comic.</legend>
</figure>

<figure>
 <video src="ex-b.mov"></video>
 <legend>Exhibit A. The alleged <cite>rough copy</cite> comic.</legend>
</figure>

<p>The case was resolved out of court.</p>
```

Here, a part of a poem is marked up using `figure`.

```
<figure>
 <p>'Twas brillig, and the slithy toves<br>
Did gyre and gimble in the wabe;<br>
All mimsy were the borogoves,<br>
And the mome raths outgrabe.</p>
 <legend><cite>Jabberwocky</cite> (first verse). Lewis Carroll,
1832-98</legend>
</figure>
```

In this example, which could be part of a much larger work discussing a castle, the figure has three images in it.

```
<figure>
 <img src="castle1423.jpeg" title="Etching. Anonymous, ca. 1423."
      alt="The castle has one tower, and a tall wall around it.">
 <img src="castle1858.jpeg" title="Oil-based paint on canvas. Maria Towle,
1858."
      alt="The castle now has two towers and two walls.">
 <img src="castle1999.jpeg" title="Film photograph. Peter Jankle, 1999."
      alt="The castle lies in ruins, the original tower all that remains
in one piece.">
 <legend>The castle through the ages: 1423, 1858, and 1999 respectively.
</legend>
</figure>
```

4.7.2 The `img` element

Categories

Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Empty.

Element-specific attributes:

`alt`

`src`
`usemap`
`ismap`
`width`
`height`

DOM interface:

```
[NamedConstructor=Image(),
 NamedConstructor=Image(in unsigned long width),
 NamedConstructor=Image(in unsigned long width, in unsigned long
height)]
interface HTMLImageElement : HTMLElement {
           attribute DOMString alt;
           attribute DOMString src;
           attribute DOMString useMap;
           attribute boolean isMap;
           attribute long width;
           attribute long height;
  readonly attribute boolean complete;
};
```

An `img` element represents an image.

The image given by the **`src`** attribute is the embedded content, and the value of the **`alt`** attribute is the `img` element's fallback content.

Authoring requirements: The `src` attribute must be present, and must contain a valid URL.

Should we restrict the URL to pointing to an image? What's an image? Is PDF an image? (Safari supports PDFs in <img> elements.) How about SVG? (Opera supports those). WMFs? XPMs? HTML?

The requirements for the `alt` attribute depend on what the image is intended to represent:

A phrase or paragraph with an alternative graphical representation

Sometimes something can be more clearly stated in graphical form, for example as a flowchart, a diagram, a graph, or a simple map showing directions. In such cases, an image can be given using the `img` element, but the lesser textual version must still be given, so that users who are unable to view the image (e.g. because they have a very slow connection, or because they are using a text-only browser, or because they are listening to the page being read out by a hands-free automobile voice Web browser, or simply because they are blind) are still able to understand the message being conveyed.

The text must be given in the `alt` attribute, and must convey the same message as the image specified in the `src` attribute.

> In the following example we have a flowchart in image form, with text in the `alt` attribute rephrasing the flowchart in prose form:
>
> ```
> <p>In the common case, the data handled by the tokenization stage
> ```

```
comes from the network, but it can also come from script.</p>
<p><img src="images/parsing-model-overview.png" alt="The network
passes data to the Tokeniser stage, which passes data to the Tree
Construction stage. From there, data goes to both the DOM and to
Script Execution. Script Execution is linked to the DOM, and, using
document.write(), passes data to the Tokeniser."></p>
```

Here's another example, showing a good solution and a bad solution to the problem of including an image in a description.

First, here's the good solution. This sample shows how the alternative text should just be what you would have put in the prose if the image had never existed.

```
<!-- This is the correct way to do things. -->
<p>
 You are standing in an open field west of a house.
 <img src="house.jpeg" alt="The house is white, with a boarded front
door.">
 There is a small mailbox here.
</p>
```

Second, here's the bad solution. In this incorrect way of doing things, the alternative text is simply a description of the image, instead of a textual replacement for the image. It's bad because when the image isn't shown, the text doesn't flow as well as in the first example.

```
<!-- This is the wrong way to do things. -->
<p>
 You are standing in an open field west of a house.
 <img src="house.jpeg" alt="A white house, with a boarded front
door.">
 There is a small mailbox here.
</p>
```

It is important to realize that the alternative text is a *replacement* for the image, not a description of the image.

Icons: a short phrase or label with an alternative graphical representation

A document can contain information in iconic form. The icon is intended to help users of visual browsers to recognize features at a glance.

In some cases, the icon is supplemental to a text label conveying the same meaning. In those cases, the `alt` attribute must be present but must be empty.

Here the icons are next to text that conveys the same meaning, so they have an empty `alt` attribute:

```
<nav>
 <p><a href="/help/"><img src="/icons/help.png" alt=""> Help</a></p>
 <p><a href="/configure/"><img src="/icons/configuration.png" alt="">
 Configuration Tools</a></p>
</nav>
```

In other cases, the icon has no text next to it describing what it means; the icon is supposed to be self-explanatory. In those cases, an equivalent textual label must be given in the `alt` attribute.

> Here, posts on a news site are labeled with an icon indicating their topic.
>
> ```
> <body>
>  <article>
>   <header>
>    <h1>Ratatouille wins Best Movie of the Year award</h1>
>    <img src="movies.png" alt="Movies">
>   </header>
>   Pixar has won yet another Best Movie of the Year award,
>   making this its 8th win in the last 12 years.
>  </article>
>  <article>
>   <header>
>    <h1>Latest TWiT episode is online</h1>
>    <img src="podcasts.png" alt="Podcasts">
>   </header>
>   The latest TWiT episode has been posted, in which we hear
>   several tech news stories as well as learning much more about the
>   iPhone. This week, the panelists compare how reflective their
>   iPhones' Apple logos are.
>  </article>
> </body>
> ```

Many pages include logos, insignia, flags, or emblems, which stand for a particular entity such as a company, organization, project, band, software package, country, or some such.

If the logo is being used to represent the entity, the `alt` attribute must contain the name of the entity being represented by the logo. The `alt` attribute must *not* contain text like the word "logo", as it is not the fact that it is a logo that is being conveyed, it's the entity itself.

If the logo is being used next to the name of the entity that it represents, then the logo is supplemental, and its `alt` attribute must instead be empty.

If the logo is merely used as decorative material (as branding, or, for example, as a side image in an article that mentions the entity to which the logo belongs), then the entry below on purely decorative images applies. If the logo is actually being discussed, then it is being used as a phrase or paragraph (the description of the logo) with an alternative graphical representation (the logo itself), and the first entry above applies.

> In the following snippets, all four of the above cases are present. First, we see a logo used to represent a company:
>
> ```
> <h1><img src="XYZ.gif" alt="The XYZ company"></h1>
> ```
>
> Next, we see a paragraph which uses a logo right next to the company name, and so doesn't have any alternative text:
>
> ```
> <article>
>  <h2>News</h2>
> ```

```
<p>We have recently been looking at buying the <img src="alpha.gif"
alt=""> ΑΒΓ company, a small Greek company
specializing in our type of product.</p>
```

In this third snippet, we have a logo being used in an aside, as part of the larger article discussing the acquisition:

```
<aside><p><img src="alpha-large.gif" alt=""></p></aside>
<p>The ΑΒΓ company has had a good quarter, and our
pie chart studies of their accounts suggest a much bigger blue slice
than its green and orange slices, which is always a good sign.</p>
</article>
```

Finally, we have an opinion piece talking about a logo, and the logo is therefore described in detail in the alternative text.

```
<p>Consider for a moment their logo:</p>

<p><img src="/images/logo" alt="It consists of a green circle with a
green question mark centered inside it."></p>

<p>How unoriginal can you get? I mean, oooooh, a question mark, how
<em>revolutionary</em>, how utterly <em>ground-breaking</em>, I'm
sure everyone will rush to adopt those specifications now! They could
at least have tried for some sort of, I don't know, sequence of
rounded squares with varying shades of green and bold white outlines,
at least that would look good on the cover of a blue book.</p>
```

This example shows how the alternative text should be written such that if the image isn't available, and the text is used instead, the text flows seamlessly into the surrounding text, as if the image had never been there in the first place.

A graphical representation of some of the surrounding text

In many cases, the image is actually just supplementary, and its presence merely reinforces the surrounding text. In these cases, the `alt` attribute must be present but its value must be the empty string.

A flowchart that repeats the previous paragraph in graphical form:

```
<p>The network passes data to the Tokeniser stage, which
passes data to the Tree Construction stage. From there, data goes
to both the DOM and to Script Execution. Script Execution is
linked to the DOM, and, using document.write(), passes data to
the Tokeniser.</p>
<p><img src="images/parsing-model-overview.png" alt=""></p>
```

A graph that repeats the previous paragraph in graphical form:

```
<p>According to a study covering several billion pages,
about 62% of documents on the Web in 2007 triggered the Quirks
rendering mode of Web browsers, about 30% triggered the Almost
Standards mode, and about 9% triggered the Standards mode.</p>
```

```
<p><img src="rendering-mode-pie-chart.png" alt=""></p>
```

In general, an image falls into this category if removing the image doesn't make the page any less useful, but including the image makes it a lot easier for users of visual browsers to understand the concept.

A purely decorative image that doesn't add any information but is still specific to the surrounding content

In some cases, the image isn't discussed by the surrounding text, but it has some relevance. Such images are decorative, but still form part of the content. In these cases, the `alt` attribute must be present but its value must be the empty string.

> Examples where the image is purely decorative despite being relevant would include things like a photo of the Black Rock City landscape in a blog post about an event at Burning Man, or an image of a painting inspired by a poem, on a page reciting that poem. The following snippet shows an example of the latter case (only the first verse is included in this snippet):
>
> ```
> <h1>The Lady of Shalott</h1>
> <img src="shalott.jpeg" alt="">
> On either side the river lie<br>
> Long fields of barley and of rye,<br>
> That clothe the wold and meet the sky;<br>
> And through the field the road run by<br>
> To many-tower'd Camelot;<br>
> And up and down the people go,<br>
> Gazing where the lilies blow<br>
> Round an island there below,<br>
> The island of Shalott.
> ```

In general, if an image is decorative but isn't especially page-specific, for example an image that forms part of a site-wide design scheme, the image should be specified in the site's CSS, not in the markup of the document.

A key part of the content

In some cases, the image is a critical part of the content. This could be the case, for instance, on a page that is part of a photo gallery. The image is the whole *point* of the page containing it.

When it is possible for alternative text to be provided, for example if the image is part of a series of screenshots in a magazine review, or part of a comic strip, or is a photograph in a blog entry about that photograph, text that conveys can serve as a substitute for the image must be given as the contents of the `alt` attribute.

In a rare subset of these cases, there might be no alternative text available. This could be the case, for instance, on a photo upload site, if the site has received 8000 photos from a user without the user annotating any of them. In such cases, the `alt` attribute may be omitted, but the `alt` attribute should be included, with a useful value, if at all possible.

In any case, if an image is a key part of the content, the `alt` attribute must not be specified with an empty value.

> A screenshot in a gallery of screenshots for a new OS, with some alternative text:

```
<figure>
 <img src="KDE%20Light%20desktop.png"
      alt="The desktop is blue, with icons along the left hand side
in
           two columns, reading System, Home, K-Mail, etc. A window
is
           open showing that menus wrap to a second line if they
           cannot fit in the window. The window has a list of icons
           along the top, with an address bar below it, a list of
           icons for tabs along the left edge, a status bar on the
           bottom, and two panes in the middle. The desktop has a bar
           at the bottom of the screen with a few buttons, a pager, a
           list of open applications, and a clock.">
 <legend>Screenshot of a KDE desktop.</legend>
</figure>
```

A photo on a photo-sharing site, if the site received the image with no metadata other than the caption:

```
<figure>
 <img src="1100670787_6a7c664aef.jpg">
 <legend>Bubbles traveled everywhere with us.</legend>
</figure>
```

In this case, though, it would be better if a detailed description of the important parts of the image obtained from the user and included on the page.

Sometimes there simply is no text that can do justice to an image. For example, there is little that can be said to usefully describe a Rorschach inkblot test.

```
<figure>
 <img src="/commons/a/a7/Rorschach1.jpg">
 <legend>A black outline of the first of the ten cards
 in the Rorschach inkblot test.</legend>
</figure>
```

Note that the following would be a very bad use of alternative text:

```
<!-- This example is wrong. Do not copy it. -->
<figure>
 <img src="/commons/a/a7/Rorschach1.jpg" alt="A black outline
 of the first of the ten cards in the Rorschach inkblot test.">
 <legend>A black outline of the first of the ten cards
 in the Rorschach inkblot test.</legend>
</figure>
```

Including the caption in the alternative text like this isn't useful because it effectively duplicates the caption for users who don't have images, taunting them twice yet not helping them any more than if they had only read or heard the caption once.

Note: Since some users cannot use images at all (e.g. because they have a very slow

connection, or because they are using a text-only browser, or because they are listening to the page being read out by a hands-free automobile voice Web browser, or simply because they are blind), the `alt` attribute is only allowed to be omitted when no alternative text is available and none can be made available, e.g. on automated image gallery sites.

An image in an e-mail or document intended for a specific person who is known to be able to view images

When an image is included in a communication (such as an HTML e-mail) aimed at someone who is known to be able to view images, the `alt` attribute may be omitted. However, even in such cases it is strongly recommended that alternative text be included (as appropriate according to the kind of image involved, as described in the above entries), so that the e-mail is still usable should the user use a mail client that does not support images, or should the e-mail be forwarded on to other users whose abilities might not include easily seeing images.

The `img` must not be used as a layout tool. In particular, `img` elements should not be used to display fully transparent images, as they rarely convey meaning and rarely add anything useful to the document.

There has been some suggestion that the `longdesc` attribute from HTML4, or some other mechanism that is more powerful than `alt=""`, should be included. This has not yet been considered.

User agent requirements: When the `alt` attribute is present and its value is the empty string, the image supplements the surrounding content. In such cases, the image may be omitted without affecting the meaning of the document.

When the `alt` attribute is present and its value is not the empty string, the image is a graphical equivalent of the string given in the `alt` attribute. In such cases, the image may be replaced in the rendering by the string given in the attribute without significantly affecting the meaning of the document.

When the `alt` attribute is missing, the image represents a key part of the content. Non-visual user agents should apply image analysis heuristics to help the user make sense of the image.

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.

If the `src` attribute is omitted, the image represents whatever string is given by the element's `alt` attribute, if any, or nothing, if that attribute is empty or absent.

When an `img` is created with a `src` attribute, and whenever the `src` attribute is set subsequently, the user agent must fetch the resource specifed by the `src` attribute's value, unless the user agent cannot support images, or its support for images has been disabled.

Fetching the image must delay the `load` event.

⚠**Warning!** ***This, unfortunately, can be used to perform a rudimentary port scan of the user's local network (especially in conjunction with scripting, though scripting isn't actually necessary to carry out such an attack). User agents may implement cross-origin access control policies that mitigate this attack.***

Once the resource has been fetched, if the image is a valid image, the user agent must fire a `load` event on the `img` element (this happens after `complete` starts returning true). 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 provides access to a server-side image map. This affects how events are handled on the corresponding `a` element.

The `ismap` attribute is a boolean attribute. The attribute must not be specified on an element that does not have an ancestor `a` element with an `href` attribute.

The `img` element supports dimension attributes.

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

The DOM attributes **`width`** and **`height`** must return the rendered width and height of the image, in CSS pixels, if the image is being rendered, and is being rendered to a visual medium; or else the intrinsic with and height of the image, in CSS pixels, if the image is available but not being rendered to a visual medium; or else 0, if the image is not available or its dimensions are not known. [CSS21]

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, and false otherwise.

Note: The value of `complete` can change while a script is executing.

Three constructors are provided for creating `HTMLImageElement` objects (in addition to the factory methods from DOM Core such as `createElement()`): **`Image()`**, **`Image(`*`width`*`)`**, and **`Image(`*`width`*`, `*`height`*`)`**. When invoked as constructors, these must return a new `HTMLImageElement` object (a new `img` element). If the *width* argument is present, the new object's `width` content attribute must be set to *width*. If the *height* argument is also present, the new object's `height` content attribute must be set to *height*.

> A single image can have different appropriate alternative text depending on the context.
>
> In each of the following cases, the same image is used, yet the `alt` text is different each time. The image is the coat of arms of the Canton Geneva in Switzerland.
>
> Here it is used as a supplementary icon:
>
> ```
> I lived in <img src="carouge.svg" alt=""> Carouge.
> ```
>
> Here it is used as an icon representing the town:

```
<p>Home town: <img src="carouge.svg" alt="Carouge"></p>
```

Here it is used as part of a text on the town:

```
<p>Carouge has a coat of arms.</p>
<p><img src="carouge.svg" alt="The coat of arms depicts a lion, sitting in
front of a tree."></p>
<p>It is used as decoration all over the town.</p>
```

Here it is used as a way to support a similar text where the description is given as well as, instead of as an alternative to, the image:

```
<p>Carouge has a coat of arms.</p>
<p><img src="carouge.svg" alt=""></p>
<p>The coat of arms depicts a lion, sitting in front of a tree.
It is used as decoration all over the town.</p>
```

Here it is used as part of a story:

```
<p>He picked up the folder and a piece of paper fell out.</p>
<p><img src="carouge.svg" alt="Shaped like a shield, the paper had a
red background, a green tree, and a yellow lion with its tongue
hanging out and whose tail was shaped like an S."></p>
<p>He stared at the folder. S! The answer he had been looking for all
this time was simply the letter S! How had he not seen that before? It all
came together now. The phone call where Hector had referred to a lion's
tail,
the time Marco had stuck his tongue out...</p>
```

Here are some more examples showing the same picture used in different contexts, with different appropriate alternate texts each time.

```
<article>
 <h1>My cats</h1>
 <h2>Fluffy</h2>
 <p>Fluffy is my favourite.</p>
 <img src="fluffy.jpg" alt="She likes playing with a ball of yarn.">
 <p>She's just too cute.</p>
 <h2>Miles</h2>
 <p>My other cat, Miles just eats and sleeps.</p>
</article>
```

```
<article>
 <h1>Photography</h1>
 <h2>Shooting moving targets indoors</h2>
 <p>The trick here is to know how to anticipate; to know at what speed and
 what distance the subject will pass by.</p>
 <img src="fluffy.jpg" alt="A cat flying by, chasing a ball of yarn, can
be
 photographed quite nicely using this technique.">
 <h2>Nature by night</h2>
 <p>To achieve this, you'll need either an extremely sensitive film, or
```

```
 immense flash lights.</p>
</article>

<article>
 <h1>About me</h1>
 <h2>My pets</h2>
 <p>I've got a cat named Fluffy and a dog named Miles.</p>
 <img src="fluffy.jpg" alt="Fluffy, my cat, tends to keep itself busy.">
 <p>My dog Miles and I like go on long walks together.</p>
 <h2>music</h2>
 <p>After our walks, having emptied my mind, I like listening to Bach.</p>
</article>

<article>
 <h1>Fluffy and the Yarn</h1>
 <p>Fluffy was a cat who liked to play with yarn. He also liked to
jump.</p>
 <aside><img src="fluffy.jpg" alt="" title="Fluffy"></aside>
 <p>He would play in the morning, he would play in the evening.</p>
</article>
```

4.7.3 The `iframe` element

Categories

Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Text that conforms to the requirements given in the prose.

Element-specific attributes:

`src`
`name`
`sandbox`
`seamless`
`width`
`height`

DOM interface:

```
interface HTMLIFrameElement : HTMLElement {
           attribute DOMString src;
           attribute DOMString name;
           attribute DOMString sandbox;
           attribute boolean seamless;
           attribute long width;
           attribute long height;
};
```

> 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 gives the address of a page that the nested browsing context is to contain. The attribute, if present, must be a valid URL. When the browsing context is created, if the attribute is present, the user agent must navigate the element's browsing context to the given URL, with replacement enabled, and with the `iframe` element's document's browsing context as the source browsing context. If the user navigates away from this page, the `iframe`'s corresponding `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 URL given by that attribute's value, with the `iframe` element's document's browsing context as the source browsing context.

If the `src` attribute is not set when the element is created, the browsing context will remain at the initial `about:blank` page.

The **`name`** attribute, if present, must be a valid browsing context name. When the browsing context is created, if the attribute is present, the browsing context name must be set to the value of this attribute; otherwise, the browsing context name must be set to the empty string.

Whenever the `name` attribute is set, the nested browsing context's name must be changed to the new value. If the attribute is removed, the browsing context name must be set to the empty string.

When content loads in an `iframe`, after any `load` events are fired within the content itself, the user agent must fire a `load` event at the `iframe` element. When content fails to load (e.g. due to a network error), then the user agent must fire an `error` event at the element instead.

When there is an active parser in the `iframe`, and when anything in the `iframe` that is delaying the `load` event in the `iframe`'s browsing context, the `iframe` must delay the `load` event.

> ***Note: If, during the handling of the `load` event, the browsing context in the `iframe` is again navigated, that will further delay the `load` event.***

The **`sandbox`** attribute, when specified, enables a set of extra restrictions on any content hosted by the `iframe`. Its value must be an unordered set of unique space-separated tokens. The allowed values are `allow-same-origin`, `allow-forms`, and `allow-scripts`.

While the `sandbox` attribute is specified, the `iframe` element's nested browsing context, and all the browsing contexts nested within it (either directly or indirectly through other nested browsing contexts) must have the following flags set:

The *sandboxed navigation browsing context flag*

> This flag prevents content from navigating browsing contexts other than the sandboxed browsing context itself (or browsing contexts further nested inside it).
>
> This flag also prevents content from creating new auxiliary browsing contexts, e.g. using the `target` attribute or the `window.open()` method.

The *sandboxed plugins browsing context flag*

This flag prevents content from instantiating plugins, whether using the `embed` element, the `object` element, the `applet` element, or through navigation of a nested browsing context.

The *sandboxed annoyances browsing context flag*

This flag prevents content from showing notifications outside of the nested browsing context.

The *sandboxed origin browsing context flag*, unless the `sandbox` attribute's value, when split on spaces, is found to have the *`allow-same-origin`* keyword set

This flag forces content into a unique origin for the purposes of the same-origin policy.

This flag also prevents script from reading the `document.cookies` DOM attribute.

The `allow-same-origin` attribute is intended for two cases.

First, it can be used to allow content from the same site to be sandboxed to disable scripting, while still allowing access to the DOM of the sandboxed content.

Second, it can be used to embed content from a third-party site, sandboxed to prevent that site from opening popup windows, etc, without preventing the embedded page from communicating back to its originating site, using the database APIs to store data, etc.

The *sandboxed forms browsing context flag*, unless the `sandbox` attribute's value, when split on spaces, is found to have the *`allow-forms`* keyword set

This flag blocks form submission.

The *sandboxed scripts browsing context flag*, unless the `sandbox` attribute's value, when split on spaces, is found to have the *`allow-scripts`* keyword set

This flag blocks script execution.

These flags must not be set unless the conditions listed above define them as being set.

> In this example, some completely-unknown, potentially hostile, user-provided HTML content is embedded in a page. Because it is sandboxed, it is treated by the user agent as being from a unique origin, despite the content being served from the same site. Thus it is affected by all the normal cross-site restrictions. In addition, the embedded page has scripting disabled, plugins disabled, forms disabled, and it cannot navigate any frames or windows other than itself (or any frames or windows it itself embeds).
>
> ```
> We're not scared of you! Here is your content, unedited:
> <iframe sandbox src="getusercontent.cgi?id=12193"></iframe>
> ```
>
> Note that cookies are still send to the server in the `getusercontent.cgi` request, though they are not visible in the `document.cookies` DOM attribute.

> In this example, a gadget from another site is embedded. The gadget has scripting and forms enabled, and the origin sandbox restrictions are lifted, allowing the gadget to communicate with its originating server. The sandbox is still useful, however, as it disables plugins and popups, thus reducing the risk of the user being exposed to malware and other annoyances.
>
> ```
> <iframe sandbox="allow-same-origin allow-forms allow-scripts"
> ```

```
src="http://maps.example.com/embedded.html"></iframe>
```

The **`seamless`** attribute is a boolean attribute. When specified, it indicates that the `iframe` element's browsing context is to be rendered in a manner that makes it appear to be part of the containing document (seamlessly included in the parent document). Specifically, when the attribute is set on an element and while the browsing context's active document has the same origin as the `iframe` element's document, or the browsing context's active document's *address* has the same origin as the `iframe` element's document, the following requirements apply:

- The user agent must set the **seamless browsing context flag** to true for that browsing context. This will cause links to open in the parent browsing context.
- In a CSS-supporting user agent: the user agent must add all the style sheets that apply to the `iframe` element to the cascade of the active document of the `iframe` element's nested browsing context, at the appropriate cascade levels, before any style sheets specified by the document itself.
- In a CSS-supporting user agent: the user agent must, for the purpose of CSS property inheritance only, treat the root element of the active document of the `iframe` element's nested browsing context as being a child of the `iframe` element. (Thus inherited properties on the root element of the document in the `iframe` will inherit the computed values of those properties on the `iframe` element instead of taking their initial values.)
- In visual media, in a CSS-supporting user agent: the user agent should set the intrinsic width of the `iframe` to the width that the element would have if it was a non-replaced block-level element with 'width: auto'.
- In visual media, in a CSS-supporting user agent: the user agent should set the intrinsic height of the `iframe` to the height of the bounding box around the content rendered in the `iframe` at its current width.
- In visual media, in a CSS-supporting user agent: the user agent must force the height of the initial containing block of the active document of the nested browsing context of the `iframe` to zero.

  Note: This is intended to get around the otherwise circular dependency of percentage dimensions that depend on the height of the containing block, thus affecting the height of the document's bounding box, thus affecting the height of the viewport, thus affecting the size of the initial containing block.

- In speech media, the user agent should render the nested browsing context without announcing that it is a separate document.
- User agents should, in general, act as if the active document of the `iframe`'s nested browsing context was part of the document that the `iframe` is in.

  > For example if the user agent supports listing all the links in a document, links in "seamlessly" nested documents would be included in that list without being significantly distinguished from links in the document itself.

Parts of the above might get moved into the rendering section at some point.

If the attribute is not specified, or if the origin conditions listed above are not met, then the user agent should render the nested browsing context in a manner that is clearly distinguishable as a separate browsing context, and the seamless browsing context flag must be set to false for that browsing context.

⚠Warning! ***It is important that user agents recheck the above conditions whenever the active document of the nested browsing context of the `iframe` changes, such that the seamless browsing context flag gets unset if the nested browsing context is navigated to another origin.***

> In this example, the site's navigation is embedded using a client-side include using an `iframe`. Any links in the `iframe` will, in new user agents, be automatically opened in the `iframe`'s parent browsing context; for legacy user agents, the site could also include a `base` element with a `target` attribute with the value `_parent`. Similarly, in new user agents the styles of the parent page will be automatically applied to the contents of the frame, but to support legacy user agents authors might wish to include the styles explicitly.
>
> ```
> <nav><iframe seamless src="nav.include.html"></iframe></nav>
> ```

The `iframe` element supports dimension attributes for cases where the embedded content has specific dimensions (e.g. ad units have well-defined dimensions).

An `iframe` element never has fallback content, as it will always create a nested browsing context, regardless of whether the specified initial contents are successfully used.

Descendants of `iframe` elements represent nothing. (In legacy user agents that do not support `iframe` elements, the contents would be parsed as markup that could act as fallback content.)

The content model of `iframe` elements is text, except that the text must be such that ... anyone have any bright ideas?

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

The DOM attributes **`src`**, **`name`**, **`sandbox`**, and **`seamless`** must reflect the content attributes of the same name.

4.7.4 The `embed` element

Categories

Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Empty.

Element-specific attributes:

`src`

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

DOM interface:

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

Depending on the type of content instantiated by the `embed` element, the node may also support other interfaces.

The `embed` element represents an integration point for an external (typically non-HTML) application or interactive content.

The **`src`** attribute gives the address of the resource being embedded. The attribute must be present and contain a valid URL.

If the `src` attribute is missing, then the `embed` element must be ignored (it represents nothing).

If the sandboxed plugins browsing context flag is set on the browsing context for which the `embed` element's document is the active document, then the user agent must render the `embed` element in a manner that conveys that the plugin was disabled. The user agent may offer the user the option to override the sandbox and instantiate the plugin anyway; if the user invokes such an option, the user agent must act as if the sandboxed plugins browsing context flag was not set for the purposes of this element.

⚠Warning! Plugins are disabled in sandboxed browsing contexts because they might not honor the restrictions imposed by the sandbox (e.g. they might allow scripting even when scripting in the sandbox is disabled). User agents should convey the danger of overriding the sandbox to the user if an option to do so is provided.

When the element is created with a `src` attribute, and whenever the `src` attribute is subsequently set, if the element is not in a sandboxed browsing context, user agents should fetch the specified resource, find an appropriate plugin it based on the content's type, and hand that plugin the content of the resource. If the plugin supports a scriptable interface, the `HTMLEmbedElement` object representing the element should expose that interfaces.

Fetching the resource must delay the `load` event.

Any (namespace-less) attribute may be specified on the `embed` element, so long as its name is XML-compatible.

The user agent should pass the names and values of all the attributes of the `embed` element that have no namespace to the plugin used.

The **`type`** attribute, if present, gives the MIME type of the linked resource. The value must be a valid MIME

type, optionally with parameters. If the attribute is present, its value must specify the same type as the explicit Content-Type metadata of the resource given by the `src` attribute. [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 <path> component of the URL of the specified resource matches a pattern that a plugin supports, then the content's type is the type that that plugin can handle.

   > For example, a plugin might say that it can handle resources with <path> components that end with the four character string "`.swf`".

   It would be better if browsers didn't do extension sniffing like this, and only based their decision on the actual contents of the resource. Couldn't we just apply the sniffed type of a resource steps?
3. Otherwise, if the specified resource has explicit Content-Type metadata, then that is the content's type.
4. Otherwise, the content has no type and there can be no appropriate plugin for it.

Whether the resource is fetched successfully or not must be ignored when determining the resource's type and when handing the resource to the plugin.

> ***Note: This allows servers to return data for plugins even with error responses (e.g. HTTP 500 Internal Server Error codes can still contain plugin data).***

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 plugin for the content. (This default could be as simple as saying "Unsupported Format", of course.)

The `embed` element supports dimension attributes.

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

4.7.5 The `object` element

Categories

Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Zero or more `param` elements, then, transparent.

Element-specific attributes:

`data`
`type`
`name`
`usemap`

`width`
`height`

DOM interface:

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

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

Depending on the type of content instantiated by the `object` element, the node may also support other interfaces.

The `object` element can represent an external resource, which, depending on the type of the resource, will either be treated as an image, as a nested browsing context, or as an external resource to be processed by a plugin.

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

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

One or both of the `data` and `type` attributes must be present.

The **`name`** attribute, if present, must be a valid browsing context name.

When the element is created, and subsequently whenever the `classid` attribute changes, or, if the `classid` attribute is not present, whenever the `data` attribute changes, or, if neither `classid` attribute nor the `data` attribute are present, whenever the `type` attribute changes, the user agent must run the following steps to determine what the `object` element represents:

1. If the `classid` attribute is present, and has a value that isn't the empty string, then: if the user agent can find a plugin suitable according to the value of the `classid` attribute, and plugins aren't being sandboxed, then that plugin should be used, and the value of the `data` attribute, if any, should be passed to the plugin. If no suitable plugin can be found, or if the plugin reports an error, jump to the last step in the overall set of steps (fallback).
2. If the `data` attribute is present, then:
    1. If the `type` attribute is present and its value is not a type that the user agent supports, and is not a type that the user agent can find a plugin for, then the user agent may jump to the last step in the overall set of steps (fallback) without downloading the content to examine its real type.

2. Fetch the resource specified by the `data` attribute.

   The download of the resource must delay the `load` event.

3. If the resource is not yet available (e.g. because the resource was not available in the cache, so that loading the resource required making a request over the network), then jump to the last step in the overall set of steps (fallback). When the resource becomes available, or if the load fails, restart this algorithm from this step. Resources can load incrementally; user agents may opt to consider a resource "available" whenever enough data has been obtained to begin processing the resource.

4. If the load failed (e.g. an HTTP 404 error, a DNS error), fire an `error` event at the element, then jump to the last step in the overall set of steps (fallback).

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

   1. Let the *resource type* be unknown.

   2. If the resource has associated Content-Type metadata, then let the *resource type* be the type specified in the resource's Content-Type metadata.

   3. If the *resource type* is unknown or "`application/octet-stream`" and there is a `type` attribute present on the `object` element, then change the *resource type* to instead be the type specified in that `type` attribute.

   4. If the *resource type* is still unknown, then change the *resource type* to instead be the sniffed type of the resource.

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

   ↪ **If the *resource type* can be handled by a plugin and plugins aren't being sandboxed**

   The user agent should use that plugin and pass the content of the resource to that plugin. If the plugin reports an error, then jump to the last step in the overall set of steps (fallback).

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

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

   ↪ **If the *resource type* does not start with "`image/`"**

   The `object` element must be associated with a nested browsing context, if it does not already have one. The element's nested browsing context must then be navigated to the given resource, with replacement enabled, and with the `object` element's document's browsing context as the source browsing context. (The `data` attribute of the `object` element doesn't get updated if the browsing context gets further navigated to other locations.)

   If the `name` attribute is present, the browsing context name must be set to the value of this attribute; otherwise, the browsing context name must be set to the empty string.

   navigation might end up treating it as something else, because it can do sniffing. how should we handle that?

↪ **If the *resource type* starts with "`image/`", and support for images has not been disabled**

Apply the image sniffing rules to determine the type of the image.

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

If the image cannot be rendered, e.g. because it is malformed or in an unsupported format, jump to the last step in the overall set of steps (fallback).

↪ **Otherwise**

The given *resource type* is not supported. Jump to the last step in the overall set of steps (fallback).

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

8. Once the resource is completely loaded, fire a `load` event at the element.

3. If the `data` attribute is absent but the `type` attribute is present, plugins aren't being sandboxed, and the user agent can find a plugin suitable according to the value of the `type` attribute, then that plugin should be used. If no suitable plugin can be found, or if the plugin reports an error, jump to the next step (fallback).

4. (Fallback.) The `object` element represents what the element's contents represent, ignoring any leading `param` element children. This is the element's fallback content.

When the algorithm above instantiates a plugin, the user agent should pass the names and values of all the parameters given by `param` elements that are children of the `object` element to the plugin used. If the plugin supports a scriptable interface, the `HTMLObjectElement` object representing the element should expose that interface. The plugin is not a nested browsing context.

If the sandboxed plugins browsing context flag is set on the browsing context for which the `object` element's document is the active document, then the steps above must always act as if they had failed to find a plugin, even if one would otherwise have been used.

Due to the algorithm above, the contents of `object` elements act as fallback content, used only when referenced resources can't be shown (e.g. because it returned a 404 error). This allows multiple `object` elements to be nested inside each other, targeting multiple user agents with different capabilities, with the user agent picking the first one it supports.

Whenever the `name` attribute is set, if the `object` element has a nested browsing context, its name must be changed to the new value. If the attribute is removed, if the `object` element has a browsing context, the browsing context name must be set to the empty string.

The `usemap` attribute, if present while the `object` element represents an image, can indicate that the object has an associated image map. The attribute must be ignored if the `object` element doesn't represent an image.

The `object` element supports dimension attributes.

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

In the following example, a Java applet is embedded in a page using the `object` element. (Generally speaking, it is better to avoid using applets like these and instead use native JavaScript and HTML to provide the functionality, since that way the application will work on all Web browsers without requiring a third-party plugin. Many devices, especially embedded devices, do not support third-party technologies like Java.)

```
<figure>
 <object type="application/x-java-applet">
  <param name="code" value="MyJavaClass">
  <p>You do not have Java available, or it is disabled.</p>
 </object>
 <legend>My Java Clock</legend>
</figure>
```

In this example, an HTML page is embedded in another using the `object` element.

```
<figure>
 <object data="clock.html"></object>
 <legend>My HTML Clock</legend>
</figure>
```

4.7.6 The `param` element

Categories

None.

Contexts in which this element may be used:

As a child of an `object` element, before any flow content.

Content model:

Empty.

Element-specific attributes:

`name`
`value`

DOM interface:

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

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

The **`name`** attribute gives the name of the parameter.

The **`value`** attribute gives the value of the parameter.

Both attributes must be present. They may have any value.

If both attributes are present, and if the parent element of the `param` is an `object` element, then the element defines a **parameter** with the given name/value pair.

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

4.7.7 The `video` element

Categories

Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

If the element has a `src` attribute: transparent.
If the element does not have a `src` attribute: one or more `source` elements, then, transparent.

Element-specific attributes:

`src`
`poster`
`autoplay`
`start`
`loopstart`
`loopend`
`end`
`playcount`
`controls`
`width`
`height`

DOM interface:

```
interface HTMLVideoElement : HTMLMediaElement {
           attribute long width;
           attribute long height;
  readonly attribute unsigned long videoWidth;
  readonly attribute unsigned long videoHeight;
           attribute DOMString poster;
};
```

A `video` element represents a video or movie.

Content may be provided inside the `video` element. User agents should not show this content to the user; it is intended for older Web browsers which do not support `video`, so that legacy video plugins can be tried, or to show text to the users of these older browser informing them of how to access the video contents.

Note: In particular, this content is not fallback content intended to address accessibility concerns. To make video content accessible to the blind, deaf, and those with other physical

or cognitive disabilities, authors are expected to provide alternative media streams and/or to embed accessibility aids (such as caption or subtitle tracks) into their media streams.

The `video` element is a media element whose media data is ostensibly video data, possibly with associated audio data.

The `src`, `autoplay`, `start`, `loopstart`, `loopend`, `end`, `playcount`, and `controls` attributes are the attributes common to all media elements.

The **`poster`** attribute gives the address of an image file that the user agent can show while no video data is available. The attribute, if present, must contain a valid URL. If the specified resource is to be used, it must be fetched when the element is created or when the `poster` attribute is set. The **poster frame** is then the image obtained from that resource, if any.

Note: The image given by the `poster` attribute, the poster frame, is intended to be a representative frame of the video (typically one of the first non-blank frames) that gives the user an idea of what the video is like.

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

When no video data is available (the element's `networkState` attribute is either `EMPTY`, `LOADING`, or `LOADED_METADATA`), `video` elements represent either the poster frame, or nothing.

When a `video` element is paused and the current playback position is the first frame of video, the element represents either the frame of video corresponding to the current playback position or the poster frame, at the discretion of the user agent.

Notwithstanding the above, the poster frame should be preferred over nothing, but the poster frame should not be shown again after a frame of video has been shown.

When a `video` element is paused at any other position, the element represents the frame of video corresponding to the current playback position, or, if that is not yet available (e.g. because the video is seeking or buffering), the last frame of the video to have been rendered.

When a `video` element is actively playing, it represents the frame of video at the continuously increasing "current" position. When the current playback position changes such that the last frame rendered is no longer the frame corresponding to the current playback position in the video, the new frame must be rendered. Similarly, any audio associated with the video must, if played, be played synchronized with the current playback position, at the specified volume with the specified mute state.

When a `video` element is neither actively playing nor paused (e.g. when seeking or stalled), the element represents the last frame of the video to have been rendered.

Note: Which frame in a video stream corresponds to a particular playback position is defined by the video stream's format.

In addition to the above, the user agent may provide messages to the user (such as "buffering", "no video loaded", "error", or more detailed information) by overlaying text or icons on the video or other areas of the element's playback area, or in another appropriate manner.

User agents that cannot render the video may instead make the element represent a link to an external video playback utility or to the video data itself.

The intrinsic width and height of the video are the aspect-ratio corrected dimensions given by the video data itself: the **intrinsic width** is the number of pixels per line of the video data multiplied by the pixel ratio given by the resource, multiplied by the resolution of the resource; the **intrinsic height** is the number of pixels per column of the video data multiplied by the resolution of the resource. The **resolution of the resource** is the physical distance intended for each pixel of video data, and assumes square pixels, with the resource's pixel ratio then adjusting the width of the pixels to the actual aspect-ratio-corrected width. In the absence of resolution information defining the mapping of pixels in the video to physical dimensions, user agents should assume that one pixel in the video corresponds to one CSS pixel. The **pixel ratio of the resource** is the corrected aspect ratio of the video divided by the ratio of the number of pixels per line to the number of pixels per column. In the absence of pixel ratio information in the resource, user agents should assume a default of 1.0 (square pixels).

The **`videoWidth`** DOM attribute must return the intrinsic width of the video in CSS pixels. The **`videoHeight`** DOM attribute must return the intrinsic height of the video in CSS pixels. If no video data is available, then the attributes must return 0.

If the video's pixel ratio override's is *none*, then the video's **adjusted width** is the same as the video's intrinsic width. If the video has a pixel ratio override other than *none*, then the adjusted width of the video is the number of pixels per line of the video data multiplied by the video's pixel ratio override, multiplied by the resolution of the resource; the pixel ratio of the resource is thus ignored.

The video's **adjusted height** is the same as the video's intrinsic height.

The **adjusted aspect ratio** of a video is the ratio of its adjusted width to its adjusted height.

User agents may adjust the adjusted width and height of the video to ensure that each pixel of video data corresponds to at least one device pixel, so long as this doesn't affect the adjusted aspect ratio (this is especially relevant for pixel ratios that are less than 1.0).

The `video` element supports dimension attributes.

Video content should be rendered inside the element's playback area such that the video content is shown centered in the playback area at the largest possible size that fits completely within it, with the video content's adjusted aspect ratio being preserved. Thus, if the aspect ratio of the playback area does not match the adjusted aspect ratio of the video, the video will be shown letterboxed. Areas of the element's playback area that do not contain the video represent nothing.

The intrinsic width of a `video` element's playback area is the adjusted width of the video resource, if that is available; otherwise it is the intrinsic width of the poster frame, if that is available; otherwise it is 300 CSS pixels.

The intrinsic height of a `video` element's playback area is the intrinsic height of the video resource, if that is available; otherwise it is the intrinsic height of the poster frame, if that is available; otherwise it is 150 CSS pixels.

Note: The poster frame is not affected by the pixel ratio conversions.

User agents should provide controls to enable or disable the display of closed captions associated with the video stream, though such features should, again, not interfere with the page's normal rendering.

User agents may allow users to view the video content in manners more suitable to the user (e.g. full-screen or in an independent resizable window). As for the other user interface features, controls to enable this should not interfere with the page's normal rendering unless the user agent is exposing a user interface. In such an independent context, however, user agents may make full user interfaces visible, with, e.g., play, pause, seeking, and volume controls, even if the `controls` attribute is absent.

User agents may allow video playback to affect system features that could interfere with the user's experience; for example, user agents could disable screensavers while video playback is in progress.

⚠**Warning!** ***User agents should not provide a public API to cause videos to be shown full-screen. A script, combined with a carefully crafted video file, could trick the user into thinking a system-modal dialog had been shown, and prompt the user for a password. There is also the danger of "mere" annoyance, with pages launching full-screen videos when links are clicked or pages navigated. Instead, user-agent specific interface features may be provided to easily allow the user to obtain a full-screen playback mode.***

The spec does not currently define the interaction of the "controls" attribute with the "height" and "width" attributes. This will likely be defined in the rendering section based on implementation experience. So far, browsers seem to be making the controls overlay-only, thus somewhat sidestepping the issue.

4.7.7.1. Video and audio codecs for `video` elements

User agents may support any video and audio codecs and container formats.

It would be helpful for interoperability if all browsers could support the same codecs. However, there are no known codecs that satisfy all the current players: we need a codec that is known to not require per-unit or per-distributor licensing, that is compatible with the open source development model, that is of sufficient quality as to be usable, and that is not an additional submarine patent risk for large companies. This is an ongoing issue and this section will be updated once more information is available.

Note: Certain user agents might support no codecs at all, e.g. text browsers running over SSH connections.

4.7.8 The `audio` element

Categories

Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

If the element has a `src` attribute: transparent.
If the element does not have a `src` attribute: one or more `source` elements, then, transparent.

Element-specific attributes:

`src`
`autoplay`
`start`
`loopstart`
`loopend`
`end`
`playcount`
`controls`

DOM interface:

```
[NamedConstructor=Audio(),
 NamedConstructor=Audio(in DOMString url)]
interface HTMLAudioElement : HTMLMediaElement {
  // no members
};
```

An `audio` element represents a sound or audio stream.

Content may be provided inside the `audio` element. User agents should not show this content to the user; it is intended for older Web browsers which do not support `audio`, so that legacy audio plugins can be tried, or to show text to the users of these older browser informing them of how to access the audio contents.

Note: In particular, this content is not fallback content intended to address accessibility concerns. To make audio content accessible to the deaf or to those with other physical or cognitive disabilities, authors are expected to provide alternative media streams and/or to embed accessibility aids (such as transcriptions) into their media streams.

The `audio` element is a media element whose media data is ostensibly audio data.

The `src`, `autoplay`, `start`, `loopstart`, `loopend`, `end`, `playcount`, and `controls` attributes are the attributes common to all media elements.

When an `audio` element is actively playing, it must have its audio data played synchronized with the current playback position, at the specified volume with the specified mute state.

When an `audio` element is not actively playing, audio must not play for the element.

Two constructors are provided for creating `HTMLAudioElement` objects (in addition to the factory methods from DOM Core such as `createElement()`): **`Audio()`** and **`Audio(`*`url`*`)`**. When invoked as constructors, these must return a new `HTMLAudioElement` object (a new `audio` element). If the *src* argument is present, the object created must have its `src` content attribute set to the provided value, and the user agent must invoke the `load()` method on the object before returning.

4.7.8.1. Audio codecs for `audio` elements

User agents may support any audio codecs and container formats.

User agents must support the WAVE container format with audio encoded using the PCM format.

4.7.9 The `source` element

Categories

None.

Contexts in which this element may be used:

As a child of a media element, before any flow content.

Content model:

Empty.

Element-specific attributes:

`src`
`type`
`media`
`pixelratio`

DOM interface:

```
interface HTMLSourceElement : HTMLElement {
           attribute DOMString src;
           attribute DOMString type;
           attribute DOMString media;
           attribute float pixelRatio;
};
```

The `source` element allows authors to specify multiple media resources for media elements.

The **`src`** attribute gives the address of the media resource. The value must be a valid URL. This attribute must be present.

The **`type`** attribute gives the type of the media resource, to help the user agent determine if it can play this media resource before fetching it. Its value must be a MIME type. The `codecs` parameter may be specified and might be necessary to specify exactly how the resource is encoded. [RFC2046] [RFC4281]

> The following list shows some examples of how to use the `codecs=` MIME parameter in the `type` attribute.
>
> **H.264 Simple baseline profile video (main and extended video compatible) level 3 and Low-Complexity AAC audio in MP4 container**
>
> ```
> <source src="video.mp4" type="video/mp4; codecs="avc1.42E01E,
> mp4a.40.2"">
> ```
>
> **H.264 Extended profile video (baseline-compatible) level 3 and Low-Complexity AAC audio in MP4 container**
>
> ```
> <source src="video.mp4" type="video/mp4; codecs="avc1.58A01E,
> ```

```
mp4a.40.2&amp;quot;">
```

H.264 Main profile video level 3 and Low-Complexity AAC audio in MP4 container

```
<source src="video.mp4" type="video/mp4; codecs=&amp;quot;avc1.4D401E,
mp4a.40.2&amp;quot;">
```

H.264 "High" profile video (incompatible with main, baseline, or extended profiles) level 3 and Low-Complexity AAC audio in MP4 container

```
<source src="video.mp4" type="video/mp4; codecs=&amp;quot;avc1.64001E,
mp4a.40.2&amp;quot;">
```

MPEG-4 Visual Simple Profile Level 0 video and Low-Complexity AAC audio in MP4 container

```
<source src="video.mp4" type="video/mp4; codecs=&amp;quot;mp4v.20.8,
mp4a.40.2&amp;quot;">
```

MPEG-4 Advanced Simple Profile Level 0 video and Low-Complexity AAC audio in MP4 container

```
<source src="video.mp4" type="video/mp4; codecs=&amp;quot;mp4v.20.240,
mp4a.40.2&amp;quot;">
```

MPEG-4 Visual Simple Profile Level 0 video and AMR audio in 3GPP container

```
<source src="video.3gp" type="video/3gpp; codecs=&amp;quot;mp4v.20.8,
samr&amp;quot;">
```

Theora video and Vorbis audio in Ogg container

```
<source src="video.ogv" type="video/ogg; codecs=&amp;quot;theora,
vorbis&amp;quot;">
```

Theora video and Speex audio in Ogg container

```
<source src="video.ogv" type="video/ogg; codecs=&amp;quot;theora,
speex&amp;quot;">
```

Vorbis audio alone in Ogg container

```
<source src="audio.ogg" type="audio/ogg; codecs=vorbis">
```

Speex audio alone in Ogg container

```
<source src="audio.spx" type="audio/ogg; codecs=speex">
```

FLAC audio alone in Ogg container

```
<source src="audio.oga" type="audio/ogg; codecs=flac">
```

Dirac video and Vorbis audio in Ogg container

```
<source src="video.ogv" type="video/ogg; codecs=&amp;quot;dirac,
vorbis&amp;quot;">
```

Theora video and Vorbis audio in Matroska container

```
<source src="video.mkv" type="video/x-matroska; codecs=&quot;theora, vorbis&quot;">
```

The **media** attribute gives the intended media type of the media resource, to help the user agent determine if this media resource is useful to the user before downloading it. Its value must be a valid media query. [MQ]

Either the `type` attribute, the `media` attribute or both, must be specified, unless this is the last `source` element child of the parent element.

The **pixelratio** attribute allows the author to specify the pixel ratio of anamorphic media resources that do not self-describe their pixel ratio. The attribute value, if specified, must be a valid floating point number giving the ratio of the correct rendered width of each pixel to the actual height of each pixel in the image. The default value, if the attribute is omitted or cannot be parsed, is 1.0.

> ***Note: The only way this default is used is in deciding what number the `pixelRatio` DOM attribute will return if the content attribute is omitted or cannot be parsed. If the content attribute is omitted or cannot be parsed, then the user agent doesn't adjust the intrinsic width of the video at all; the intrinsic dimensions and the pixel ratio of the video are honoured.***

If a `source` element is inserted into a media element that is already in a document and whose `networkState` is in the `EMPTY` state, the user agent must implicitly invoke the `load()` method on the media element as soon as all other scripts have finished executing. Any exceptions raised must be ignored.

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

The DOM attribute **pixelRatio** must reflect the `pixelratio` content attribute.

4.7.10 Media elements

Media elements implement the following interface:

```
interface HTMLMediaElement : HTMLElement {

  // error state
  readonly attribute MediaError error;

  // network state
           attribute DOMString src;
  readonly attribute DOMString currentSrc;
  const unsigned short EMPTY = 0;
  const unsigned short LOADING = 1;
  const unsigned short LOADED_METADATA = 2;
  const unsigned short LOADED_FIRST_FRAME = 3;
  const unsigned short LOADED = 4;
  readonly attribute unsigned short networkState;
  readonly attribute float bufferingRate;
  readonly attribute boolean bufferingThrottled;
  readonly attribute TimeRanges buffered;
  readonly attribute ByteRanges bufferedBytes;
```

```
  readonly attribute unsigned long totalBytes;
  void load();

  // ready state
  const unsigned short DATA_UNAVAILABLE = 0;
  const unsigned short CAN_SHOW_CURRENT_FRAME = 1;
  const unsigned short CAN_PLAY = 2;
  const unsigned short CAN_PLAY_THROUGH = 3;
  readonly attribute unsigned short readyState;
  readonly attribute boolean seeking;

  // playback state
           attribute float currentTime;
  readonly attribute float duration;
  readonly attribute boolean paused;
           attribute float defaultPlaybackRate;
           attribute float playbackRate;
  readonly attribute TimeRanges played;
  readonly attribute TimeRanges seekable;
  readonly attribute boolean ended;
           attribute boolean autoplay;
  void play();
  void pause();

  // looping
           attribute float start;
           attribute float end;
           attribute float loopStart;
           attribute float loopEnd;
           attribute unsigned long playCount;
           attribute unsigned long currentLoop;

  // cue ranges
  void addCueRange(in DOMString className, in DOMString id, in float start,
in float end, in boolean pauseOnExit, in CueRangeCallback enterCallback, in
CueRangeCallback exitCallback);
  void removeCueRanges(in DOMString className);

  // controls
           attribute boolean controls;
           attribute float volume;
           attribute boolean muted;
};
```

The **media element attributes**, `src`, `autoplay`, `start`, `loopstart`, `loopend`, `end`, `playcount`, and `controls`, apply to all media elements. They are defined in this section.

Media elements are used to present audio data, or video and audio data, to the user. This is referred to as **media data** in this section, since this section applies equally to media elements for audio or for video. The term **media resource** is used to refer to the complete set of media data, e.g. the complete video file, or

complete audio file.

4.7.10.1. Error codes

All media elements have an associated error status, which records the last error the element encountered since the `load()` method was last invoked. The **`error`** attribute, on getting, must return the `MediaError` object created for this last error, or null if there has not been an error.

```
interface MediaError {
  const unsigned short MEDIA_ERR_ABORTED = 1;
  const unsigned short MEDIA_ERR_NETWORK = 2;
  const unsigned short MEDIA_ERR_DECODE = 3;
  readonly attribute unsigned short code;
};
```

The **`code`** attribute of a `MediaError` object must return the code for the error, which must be one of the following:

`MEDIA_ERR_ABORTED` **(numeric value 1)**

The download of the media resource was aborted by the user agent at the user's request.

`MEDIA_ERR_NETWORK` **(numeric value 2)**

A network error of some description caused the user agent to stop downloading the media resource.

`MEDIA_ERR_DECODE` **(numeric value 3)**

An error of some description occurred while decoding the media resource.

4.7.10.2. Location of the media resource

The **`src`** content attribute on media elements gives the address of the media resource (video, audio) to show. The attribute, if present, must contain a valid URL.

If the `src` attribute of a media element that is already in a document and whose `networkState` is in the `EMPTY` state is added, changed, or removed, the user agent must implicitly invoke the `load()` method on the media element as soon as all other scripts have finished executing. Any exceptions raised must be ignored.

> ***Note: If a `src` attribute is specified, the resource it specifies is the media resource that will be used. Otherwise, the resource specified by the first suitable `source` element child of the media element is the one used.***

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

To **pick a media resource** for a media element, a user agent must use the following steps:

1. Let the *chosen resource's pixel ratio override* be *none*.
2. If the media element has a `src` attribute, then resolve the URL given in that attribute. If that is successful, then the resulting absolute URL is the address of the media resource; jump to the last step.
3. Otherwise, let *candidate* be the first `source` element child in the media element, or null if there is no

such child.

4. *Loop*: this is the start of the loop that looks at the `source` elements.

5. If *candidate* is not null and it has a `pixelratio` attribute, and the result of applying the rules for parsing floating point number values to the value of that attribute is not an error, then let the *chosen resource's pixel ratio override* be that result; otherwise, reset it back to *none*.

6. If either:

   - *candidate* is null, or
   - the *candidate* element has no `src` attribute, or
   - resolving the URL given by the *candidate* element's `src` attribute fails, or
   - the *candidate* element has a `type` attribute and that attribute's value, when parsed as a MIME type, does not represent a type that the user agent can render (including any codecs described by the `codec` parameter), or [RFC2046] [RFC4281]
   - the *candidate* element has a `media` attribute and that attribute's value, when processed according to the rules for media queries, does not match the current environment, [MQ]

   ...then the *candidate* is not suitable; go to the next step.

   Otherwise, the result of resolving the URL given in that *candidate* element's `src` attribute is the address of the media resource; jump to the last step.

7. Let *candidate* be the next `source` element child in the media element, or null if there are no more such children.

8. If *candidate* is not null, return to the step labeled *loop*.

9. There is no media resource. Abort these steps.

10. Let the address of the **chosen media resource** be the absolute URL that was found before jumping to this step, and let its **pixel ratio override** be the value of the *chosen resource's pixel ratio override*.

The **`currentSrc`** DOM attribute must return the empty string if the media element's `networkState` has the value EMPTY, and the absolute URL that is the address of the chosen media resource otherwise.

4.7.10.3. Network states

As media elements interact with the network, they go through several states. The **`networkState`** attribute, on getting, must return the current network state of the element, which must be one of the following values:

`EMPTY` **(numeric value 0)**

The element has not yet been initialized. All attributes are in their initial states.

`LOADING` **(numeric value 1)**

The element has picked a media resource (the chosen media resource is available from the `currentSrc` attribute), but none of the metadata has yet been obtained and therefore all the other attributes are still in their initial states.

***LOADED_METADATA* (numeric value 2)**

Enough of the resource has been obtained that the metadata attributes are initialized (e.g. the length is known). The API will no longer raise exceptions when used.

***LOADED_FIRST_FRAME* (numeric value 3)**

Actual media data has been obtained. In the case of video, this specifically means that a frame of video is available and can be shown.

***LOADED* (numeric value 4)**

The entire media resource has been obtained and is available to the user agent locally. Network connectivity could be lost without affecting the media playback.

The algorithm for the `load()` method defined below describes exactly when the `networkState` attribute changes value.

4.7.10.4. Loading the media resource

All media elements have a **begun flag**, which must begin in the false state, a **loaded-first-frame flag**, which must begin in the false state, and an **autoplaying flag**, which must begin in the true state.

When the **`load()`** method on a media element is invoked, the user agent must run the following steps. Note that this algorithm might get aborted, e.g. if the `load()` method itself is invoked again.

1. Any already-running instance of this algorithm for this element must be aborted. If those method calls have not yet returned, they must finish the step they are on, and then immediately return. This is not blocking; this algorithm must not wait for the earlier instances to abort before continuing.
2. If the element's begun flag is true, then the begun flag must be set to false, the `error` attribute must be set to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_ABORTED`, and the user agent must synchronously fire a progress event called `abort` at the media element.
3. The `error` attribute must be set to null, the loaded-first-frame flag must be set to false, and the autoplaying flag must be set to true.
4. The `playbackRate` attribute must be set to the value of the `defaultPlaybackRate` attribute.
5. If the media element's `networkState` is not set to EMPTY, then the following substeps must be followed:
    1. The `networkState` attribute must be set to EMPTY.
    2. If `readyState` is not set to `DATA_UNAVAILABLE`, it must be set to that state.
    3. If the `paused` attribute is false, it must be set to true.
    4. If `seeking` is true, it must be set to false.
    5. The current playback position must be set to 0.
    6. The `currentLoop` DOM attribute must be set to 0.
    7. The user agent must synchronously fire a simple event called `emptied` at the media element.

6. The user agent must pick a media resource for the media element. If that fails, the method must raise an `INVALID_STATE_ERR` exception, and abort these steps.

7. The `networkState` attribute must be set to LOADING.

8. ***Note: The `currentSrc` attribute starts returning the new value.***

9. The user agent must then set the begun flag to true and synchronously fire a progress event called `loadstart` at the media element.

10. The method must return, but these steps must continue.

11. ***Note: Playback of any previously playing media resource for this element stops.***

12. If a download is in progress for the media element, the user agent should stop the download.

13. The user agent must then begin to download the chosen media resource. The rate of the download may be throttled, however, in response to user preferences (including throttling it to zero until the user indicates that the download can start), or to balance the download with other connections sharing the same bandwidth.

14. While the download is progressing, the user agent must fire a progress event called `progress` at the element every 350ms (±200ms) or for every byte received, whichever is *least* frequent.

    If at any point the user agent has received no data for more than about three seconds, the user agent must fire a progress event called `stalled` at the element.

    User agents may allow users to selectively block or slow media data downloads. When a media element's download has been blocked, the user agent must act as if it was stalled (as opposed to acting as if the connection was closed).

    The user agent may use whatever means necessary to download the resource (within the constraints put forward by this and other specifications); for example, reconnecting to the server in the face of network errors, using HTTP partial range requests, or switching to a streaming protocol. The user agent must consider a resource erroneous only if it has given up trying to download it.

    ↪ **If the media data cannot be downloaded at all, due to network errors, causing the user agent to give up trying to download the resource**

    DNS errors and HTTP 4xx and 5xx errors (and equivalents in other protocols) must cause the user agent to execute the following steps. User agents may also follow these steps in response to other network errors of similar severity.

    1. The user agent should cancel the download.

    2. The `error` attribute must be set to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_NETWORK`.

    3. The begun flag must be set to false and the user agent must fire a progress event called `error` at the media element.

    4. The element's `networkState` attribute must be switched to the EMPTY value and the user agent must fire a simple event called `emptied` at the element.

5. These steps must be aborted.

↪ **If the media data can be downloaded but is in an unsupported format, or can otherwise not be rendered at all**

The server returning a file of the wrong kind (e.g. one that that turns out to not be pure audio when the media element is an `audio` element), or the file using unsupported codecs for all the data, must cause the user agent to execute the following steps. User agents may also execute these steps in response to other codec-related fatal errors, such as the file requiring more resources to process than the user agent can provide in real time.

1. The user agent should cancel the download.
2. The `error` attribute must be set to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_DECODE`.
3. The begun flag must be set to false and the user agent must fire a progress event called `error` at the media element.
4. The element's `networkState` attribute must be switched to the EMPTY value and the user agent must fire a simple event called `emptied` at the element.
5. These steps must be aborted.

↪ **If the media data download is aborted by the user**

The download is aborted by the user, e.g. because the user navigated the browsing context to another page, the user agent must execute the following steps. These steps are not followed if the `load()` method itself is reinvoked, as the steps above handle that particular kind of abort.

1. The user agent should cancel the download.
2. The `error` attribute must be set to a new `MediaError` object whose `code` attribute is set to `MEDIA_ERR_ABORT`.
3. The begun flag must be set to false and the user agent must fire a progress event called `abort` at the media element.
4. If the media element's `networkState` attribute has the value `LOADING`, the element's `networkState` attribute must be switched to the EMPTY value and the user agent must fire a simple event called `emptied` at the element. (If the `networkState` attribute has a value greater than `LOADING`, then this doesn't happen; the available data, if any, will be playable.)
5. These steps must be aborted.

↪ **If the media data can be downloaded but has non-fatal errors or uses, in part, codecs that are unsupported, preventing the user agent from rendering the content completely correctly but not preventing playback altogether**

The server returning data that is partially usable but cannot be optimally rendered must cause the user agent to execute the following steps.

1. Should we fire a 'warning' event? Set the 'error' flag to

'MEDIA_ERR_SUBOPTIMAL' or something?

↪ **Once enough of the media data has been downloaded to determine the duration of the media resource, its dimensions, and other metadata**

The user agent must follow these substeps:

1. The current playback position must be set to the *effective start*.
2. The `networkState` attribute must be set to `LOADED_METADATA`.
3. ***Note: A number of attributes, including `duration`, `buffered`, and `played`, become available.***
4. ***Note: The user agent will fire a simple event called `durationchange` at the element at this point.***
5. The user agent must fire a simple event called `loadedmetadata` at the element.

↪ **Once enough of the media data has been downloaded to enable the user agent to display the frame at the effective start of the media resource**

The user agent must follow these substeps:

1. The `networkState` attribute must be set to `LOADED_FIRST_FRAME`.
2. The `readyState` attribute must change to `CAN_SHOW_CURRENT_FRAME`.
3. The loaded-first-frame flag must be set to true.
4. The user agent must fire a simple event called `loadedfirstframe` at the element.
5. The user agent must fire a simple event called `canshowcurrentframe` at the element.

When the user agent has completed the download of the entire media resource, it must move on to the next step.

15. If the download completes without errors, the begun flag must be set to false, the `networkState` attribute must be set to `LOADED`, and the user agent must fire a progress event called `load` at the element.

If a media element whose `networkState` has the value `EMPTY` is inserted into a document, user agents must implicitly invoke the `load()` method on the media element as soon as all other scripts have finished executing. Any exceptions raised must be ignored.

The **`bufferingRate`** attribute must return the average number of bits received per second for the current download over the past few seconds. If there is no download in progress, the attribute must return 0.

The **`bufferingThrottled`** attribute must return true if the user agent is intentionally throttling the bandwidth used by the download (including when throttling to zero to pause the download altogether), and false otherwise.

The **buffered** attribute must return a static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent has buffered, at the time the attribute is evaluated.

> ***Note: Typically this will be a single range anchored at the zero point, but if, e.g. the user agent uses HTTP range requests in response to seeking, then there could be multiple ranges.***

The **bufferedBytes** attribute must return a static normalized `ByteRanges` object that represents the ranges of the media resource, if any, that the user agent has buffered, at the time the attribute is evaluated.

The **totalBytes** attribute must return the length of the media resource, in bytes, if it is known and finite. If it is not known, is infinite (e.g. streaming radio), or if no media data is available, the attribute must return 0.

User agents may discard previously buffered data.

> ***Note: Thus, a time or byte position included within a range of the objects return by the `buffered` or `bufferedBytes` attributes at one time can end up being not included in the range(s) of objects returned by the same attributes at a later time.***

4.7.10.5. Offsets into the media resource

The **duration** attribute must return the length of the media resource, in seconds. If no media data is available, then the attributes must return 0. If media data is available but the length is not known, the attribute must return the Not-a-Number (NaN) value. If the media resource is known to be unbounded (e.g. a streaming radio), then the attribute must return the positive Infinity value.

When the length of the media resource changes (e.g. from being unknown to known, or from indeterminate to known, or from a previously established length to a new length) the user agent must, once any running scripts have finished, fire a simple event called `durationchange` at the media element.

Media elements have a **current playback position**, which must initially be zero. The current position is a time.

The **currentTime** attribute must, on getting, return the current playback position, expressed in seconds. On setting, the user agent must seek to the new value (which might raise an exception).

The **start** content attribute gives the offset into the media resource at which playback is to begin. The default value is the default start position of the media resource, or 0 if not enough media data has been obtained yet to determine the default start position or if the resource doesn't specify a default start position.

The ***effective start*** is the smaller of the `start` DOM attribute and the end of the media resource.

The **loopstart** content attribute gives the offset into the media resource at which playback is to begin when looping a clip. The default value of the `loopstart` content attribute is the value of the `start` DOM attribute.

The ***effective loop start*** is the smaller of the `loopStart` DOM attribute and the end of the media resource.

The **loopend** content attribute gives an offset into the media resource at which playback is to jump back to the `loopstart`, when looping the clip. The default value of the `loopend` content attribute is the value of the `end` DOM attribute.

The ***effective loop end*** is the greater of the `start`, `loopStart`, and `loopEnd` DOM attributes, except if that

is greater than the end of the media resource, in which case that's its value.

The **end** content attribute gives an offset into the media resource at which playback is to end. The default value is infinity.

The ***effective end*** is the greater of the `start`, `loopStart`, and `end` DOM attributes, except if that is greater than the end of the media resource, in which case that's its value.

The `start`, `loopstart`, `loopend`, and `end` attributes must, if specified, contain value time offsets. To get the time values they represent, user agents must use the rules for parsing time offsets.

The **start**, **loopStart**, **loopEnd**, and **end** DOM attributes must reflect the `start`, `loopstart`, `loopend`, and `end` content attributes on the media element respectively.

The **playcount** content attribute gives the number of times to play the clip. The default value is 1.

The **playCount** DOM attribute must reflect the `playcount` content attribute on the media element. The value must be limited to only positive non-zero numbers.

The **currentLoop** attribute must initially have the value 0. It gives the index of the current loop. It is changed during playback as described below.

When any of the `start`, `loopStart`, `loopEnd`, `end`, `playCount`, and `currentLoop` DOM attributes change value (either through content attribute mutations reflecting into the DOM attribute, if applicable, or through direct mutations of the DOM attribute), the user agent must apply the following steps:

1. If the `playCount` DOM attribute's value is less than or equal to the `currentLoop` DOM attribute's value, then the `currentLoop` DOM attribute's value must be set to `playCount`-1 (which will make the current loop the last loop).
2. If the media element's `networkState` is in the `EMPTY` state or the `LOADING` state, then the user agent must at this point abort these steps.
3. If the `currentLoop` is zero, and the current playback position is before the *effective start*, the user agent must seek to the *effective start*.
4. If the `currentLoop` is greater than zero, and the current playback position is before the *effective loop start*, the user agent must seek to the *effective loop start*.
5. If the `currentLoop` is less than `playCount`-1, and the current playback position is after the *effective loop end*, the user agent must seek to the *effective loop start*, and increase `currentLoop` by 1.
6. If the `currentLoop` is equal to `playCount`-1, and the current playback position is after the *effective end*, the user agent must seek to the *effective end* and then the looping will end.

4.7.10.6. The ready states

Media elements have a *ready state*, which describes to what degree they are ready to be rendered at the current playback position. The possible values are as follows; the ready state of a media element at any particular time is the greatest value describing the state of the element:

DATA_UNAVAILABLE **(numeric value 0)**

No data for the current playback position is available. Media elements whose `networkState` attribute is less than `LOADED_FIRST_FRAME` are always in the `DATA_UNAVAILABLE` state.

CAN_SHOW_CURRENT_FRAME **(numeric value 1)**

Data for the immediate current playback position is available, but not enough data is available that the user agent could successfully advance the current playback position at all without immediately reverting to the `DATA_UNAVAILABLE` state. In video, this corresponds to the user agent having data from the current frame, but not the next frame. In audio, this corresponds to the user agent only having audio up to the current playback position, but no further.

CAN_PLAY **(numeric value 2)**

Data for the immediate current playback position is available, as well as enough data for the user agent to advance the current playback position at least a little without immediately reverting to the `DATA_UNAVAILABLE` state. In video, this corresponds to the user agent having data for the current frame and the next frame. In audio, this corresponds to the user agent having data beyond the current playback position.

CAN_PLAY_THROUGH **(numeric value 3)**

Data for the immediate current playback position is available, as well as enough data for the user agent to advance the current playback position at least a little without immediately reverting to the `DATA_UNAVAILABLE` state, and, in addition, the user agent estimates that data is being downloaded at a rate where the current playback position, if it were to advance at the rate given by the `defaultPlaybackRate` attribute, would not overtake the available data before playback reaches the effective end of the media resource on the last loop.

When the ready state of a media element whose `networkState` is not `EMPTY` changes, the user agent must follow the steps given below:

↪ **If the new ready state is `DATA_UNAVAILABLE`**

The user agent must fire a simple event called `dataunavailable` at the element.

↪ **If the new ready state is `CAN_SHOW_CURRENT_FRAME`**

If the element's loaded-first-frame flag is true, the user agent must fire a simple event called `canshowcurrentframe` event.

Note: The first time the `networkState` attribute switches to this value, the loaded-first-frame flag is false, and the event is fired by the algorithm described above for the `load()` method, in conjunction with other steps.

↪ **If the new ready state is `CAN_PLAY`**

The user agent must fire a simple event called `canplay`.

↪ **If the new ready state is `CAN_PLAY_THROUGH`**

The user agent must fire a simple event called `canplaythrough` event. If the autoplaying flag is true, and the `paused` attribute is true, and the media element has an `autoplay` attribute specified, then the user agent must also set the `paused` attribute to false and fire a simple event called `play`.

Note: It is possible for the ready state of a media element to jump between these states

> ***discontinuously. For example, the state of a media element whose loaded-first-frame flag is false can jump straight from `DATA_UNAVAILABLE` to `CAN_PLAY_THROUGH` without passing through the `CAN_SHOW_CURRENT_FRAME` and `CAN_PLAY` states, and thus without firing the `canshowcurrentframe` and `canplay` events. The only state that is guaranteed to be reached is the `CAN_SHOW_CURRENT_FRAME` state, which is reached as part of the `load()` method's processing.***

The **`readyState`** DOM attribute must, on getting, return the value described above that describes the current ready state of the media element.

The **`autoplay`** attribute is a boolean attribute. When present, the algorithm described herein will cause the user agent to automatically begin playback of the media resource as soon as it can do so without stopping.

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

4.7.10.7. Playing the media resource

The **`paused`** attribute represents whether the media element is paused or not. The attribute must initially be true.

A media element is said to be **actively playing** when its `paused` attribute is false, the `readyState` attribute is either `CAN_PLAY` or `CAN_PLAY_THROUGH`, the element has not ended playback, playback has not stopped due to errors, and the element has not paused for user interaction.

A media element is said to have **ended playback** when the element's `networkState` attribute is `LOADED_METADATA` or greater, the current playback position is equal to the *effective end* of the media resource, and the `currentLoop` attribute is equal to `playCount`-1.

A media element is said to have **stopped due to errors** when the element's `networkState` attribute is `LOADED_METADATA` or greater, and the user agent encounters a non-fatal error during the processing of the media data, and due to that error, is not able to play the content at the current playback position.

A media element is said to have **paused for user interaction** when its `paused` attribute is false, the `readyState` attribute is either `CAN_PLAY` or `CAN_PLAY_THROUGH` and the user agent has reached a point in the media resource where the user has to make a selection for the resource to continue.

It is possible for a media element to have both ended playback and paused for user interaction at the same time.

When a media element is actively playing and its owner `Document` is an active document, its current playback position must increase monotonically at `playbackRate` units of media time per unit time of wall clock time. If this value is not 1, the user agent may apply pitch adjustments to any audio component of the media resource.

> ***Note: This specification doesn't define how the user agent achieves the appropriate playback rate — depending on the protocol and media available, it is plausible that the user agent could negotiate with the server to have the server provide the media data at the appropriate rate, so that (except for the period between when the rate is changed and when the server updates the stream's playback rate) the client doesn't actually have to drop or interpolate any frames.***

Media resources might be internally scripted or interactive. Thus, a media element could play in a non-linear

fashion. If this happens, the user agent must act as if the algorithm for seeking was used whenever the current playback position changes in a discontinuous fashion (so that the relevant events fire).

When a media element that is actively playing stops playing because its `readyState` attribute changes to a value lower than `CAN_PLAY`, without the element having ended playback, or playback having stopped due to errors, or playback having paused for user interaction, or the seeking algorithm being invoked, the user agent must fire a simple event called `timeupdate` at the element, and then must fire a simple event called `waiting` at the element.

When a media element that is actively playing stops playing because it has paused for user interaction, the user agent must fire a simple event called `timeupdate` at the element.

When `currentLoop` is less than `playCount`-1 and the current playback position reaches the *effective loop end*, then the user agent must seek to the *effective loop start*, increase `currentLoop` by 1, and fire a simple event called `timeupdate`.

When `currentLoop` is equal to the `playCount`-1 and the current playback position reaches the *effective end*, then the user agent must follow these steps:

1. The user agent must stop playback.
2. ***Note: The `ended` attribute becomes true.***
3. The user agent must fire a simple event called `timeupdate` at the element.
4. The user agent must fire a simple event called `ended` at the element.

The **`defaultPlaybackRate`** attribute gives the desired speed at which the media resource is to play, as a multiple of its intrinsic speed. The attribute is mutable, but on setting, if the new value is 0.0, a `NOT_SUPPORTED_ERR` exception must be raised instead of the value being changed. It must initially have the value 1.0.

The **`playbackRate`** attribute gives the speed at which the media resource plays, as a multiple of its intrinsic speed. If it is not equal to the `defaultPlaybackRate`, then the implication is that the user is using a feature such as fast forward or slow motion playback. The attribute is mutable, but on setting, if the new value is 0.0, a `NOT_SUPPORTED_ERR` exception must be raised instead of the value being changed. Otherwise, the playback must change speed (if the element is actively playing). It must initially have the value 1.0.

When the `defaultPlaybackRate` or `playbackRate` attributes change value (either by being set by script or by being changed directly by the user agent, e.g. in response to user control) the user agent must, once any running scripts have finished, fire a simple event called `ratechange` at the media element.

When the **`play()`** method on a media element is invoked, the user agent must run the following steps.

1. If the media element's `networkState` attribute has the value EMPTY, then the user agent must invoke the `load()` method and wait for it to return. If that raises an exception, that exception must be reraised by the `play()` method.
2. If the playback has ended, then the user agent must set `currentLoop` to zero and seek to the *effective start*.

Note: If this involved a seek, the user agent will fire a simple event called `timeupdate` at the media element.

3. The `playbackRate` attribute must be set to the value of the `defaultPlaybackRate` attribute.

   Note: If this caused the `playbackRate` attribute to change value, the user agent will fire a simple event called `ratechange` at the media element.

4. If the media element's `paused` attribute is true, it must be set to false.
5. The media element's autoplaying flag must be set to false.
6. The method must then return.
7. If the fourth step above changed the value of `paused`, the user agent must, after any running scripts have finished executing, and after any other events triggered by this algorithm (specifically `timeupdate` and `ratechange`) have fired, fire a simple event called `play` at the element.

When the **`pause()`** method is invoked, the user agent must run the following steps:

1. If the media element's `networkState` attribute has the value EMPTY, then the user agent must invoke the `load()` method and wait for it to return. If that raises an exception, that exception must be reraised by the `pause()` method.
2. If the media element's `paused` attribute is false, it must be set to true.
3. The media element's autoplaying flag must be set to false.
4. The method must then return.
5. If the second step above changed the value of `paused`, then, after any running scripts have finished executing, the user agent must first fire a simple event called `timeupdate` at the element, and then fire a simple event called `pause` at the element.

When a media element is removed from a `Document`, if the media element's `networkState` attribute has a value other than EMPTY then the user agent must act as if the `pause()` method had been invoked.

Media elements that are actively playing while not in a `Document` must not play any video, but should play any audio component. Media elements must not stop playing just because all references to them have been removed; only once a media element to which no references exist has reached a point where no further audio remains to be played for that element (e.g. because the element is paused or because the end of the clip has been reached) may the element be garbage collected.

Note: If the media element's `ownerDocument` stops being an active document, then the playback will stop until the document is active again.

The **`ended`** attribute must return true if the media element has ended playback, and false otherwise.

The **`played`** attribute must return a static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent has so far rendered, at the time the attribute is evaluated.

4.7.10.8. Seeking

The **`seeking`** attribute must initially have the value false.

When the user agent is required to **seek** to a particular *new playback position* in the media resource, it means that the user agent must run the following steps:

1. If the media element's `networkState` is less than `LOADED_METADATA`, then the user agent must raise an `INVALID_STATE_ERR` exception (if the seek was in response to a DOM method call or setting of a DOM attribute), and abort these steps.
2. If `currentLoop` is 0, let *min* be the *effective start*. Otherwise, let it be the *effective loop start*.
3. If `currentLoop` is equal to `playCount`-1, let *max* be the *effective end*. Otherwise, let it be the *effective loop end*.
4. If the *new playback position* is more than *max*, let it be *max*.
5. If the *new playback position* is less than *min*, let it be *min*.
6. If the (possibly now changed) *new playback position* is not in one of the ranges given in the `seekable` attribute, then the user agent must raise an `INDEX_SIZE_ERR` exception (if the seek was in response to a DOM method call or setting of a DOM attribute), and abort these steps.
7. The current playback position must be set to the given *new playback position*.
8. The `seeking` DOM attribute must be set to true.
9. If the seek was in response to a DOM method call or setting of a DOM attribute, then continue the script. The remainder of these steps must be run asynchronously.
10. Once any running scripts have finished executing, the user agent must fire a simple event called `timeupdate` at the element.
11. If the media element was actively playing immediately before it started seeking, but seeking caused its `readyState` attribute to change to a value lower than `CAN_PLAY`, the user agent must fire a simple event called `waiting` at the element.
12. If, when it reaches this step, the user agent has still not established whether or not the media data for the *new playback position* is available, and, if it is, decoded enough data to play back that position, the user agent must fire a simple event called `seeking` at the element.
13. The user agent must wait until it has established whether or not the media data for the *new playback position* is available, and, if it is, until it has decoded enough data to play back that position.
14. The `seeking` DOM attribute must be set to false.
15. Once any running scripts have finished executing, the user agent must fire a simple event called `seeked` at the element.

The **`seekable`** attribute must return a static normalized `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent is able to seek to, at the time the attribute is evaluated, notwithstanding the looping attributes (i.e. the *effective start* and *effective end*, etc, don't affect the `seekable` attribute).

Note: If the user agent can seek to anywhere in the media resource, e.g. because it a simple movie file and the user agent and the server support HTTP Range requests, then the attribute would return an object with one range, whose start is the time of the first frame (typically zero), and whose end is the same as the time of the first frame plus the `duration` attribute's value (which would equal the time of the last frame).

4.7.10.9. Cue ranges

Media elements have a set of **cue ranges**. Each cue range is made up of the following information:

A class name

A group of related ranges can be given the same class name so that they can all be removed at the same time.

An identifier

A string can be assigned to each cue range for identification by script. The string need not be unique and can contain any value.

A start time

An end time

The actual time range, using the same timeline as the media resource itself.

A "pause" boolean

A flag indicating whether to pause playback on exit.

An "enter" callback

A callback that is called when the current playback position enters the range.

An "exit" callback

A callback that is called when the current playback position exits the range.

An "active" boolean

A flag indicating whether the range is active or not.

The **`addCueRange(`*`className`*`, `*`id`*`, `*`start`*`, `*`end`*`, `*`pauseOnExit`*`, `*`enterCallback`*`, `*`exitCallback`*`)`** method must, when called, add a cue range to the media element, that cue range having the class name *className*, the identifier *id*, the start time *start* (in seconds), the end time *end* (in seconds), the "pause" boolean with the same value as *pauseOnExit*, the "enter" callback *enterCallback*, the "exit" callback *exitCallback*, and an "active" boolean that is true if the current playback position is equal to or greater than the start time and less than the end time, and false otherwise.

The **`removeCueRanges(`*`className`*`)`** method must, when called, remove all the cue ranges of the media element which have the class name *className*.

When the current playback position of a media element changes (e.g. due to playback or seeking), the user agent must run the following steps. If the current playback position changes while the steps are running, then the user agent must wait for the steps to complete, and then must immediately rerun the steps. (These steps are thus run as often as possible or needed — if one iteration takes a long time, this can cause certain ranges to be skipped over as the user agent rushes ahead to "catch up".)

1. Let *current ranges* be an ordered list of cue ranges, initialized to contain all the cue ranges of the media

element whose start times are less than or equal to the current playback position and whose end times are greater than the current playback position, in the order they were added to the element.

2. Let *other ranges* be an ordered list of cue ranges, initialized to contain all the cue ranges of the media element that are not present in *current ranges*, in the order they were added to the element.

3. If none of the cue ranges in *current ranges* have their "active" boolean set to "false" (inactive) and none of the cue ranges in *other ranges* have their "active" boolean set to "true" (active), then abort these steps.

4. If the time was reached through the usual monotonic increase of the current playback position during normal playback, the user agent must then fire a simple event called `timeupdate` at the element. (In the other cases, such as explicit seeks, relevant events get fired as part of the overall process of changing the current playback position.)

5. If the time was reached through the usual monotonic increase of the current playback position during normal playback, and there are cue ranges in *other ranges* that have both their "active" boolean and their "pause" boolean set to "true", then immediately act as if the element's `pause()` method had been invoked. (In the other cases, such as explicit seeks, playback is not paused by exiting a cue range, even if that cue range has its "pause" boolean set to "true".)

6. Invoke all the non-null "exit" callbacks for all of the cue ranges in *other ranges* that have their "active" boolean set to "true" (active), in list order, passing their identifier as the callback's only argument.

7. Invoke all the non-null "enter" callbacks for all of the cue ranges in *current ranges* that have their "active" boolean set to "false" (inactive), in list order, passing their identifier as the callback's only argument.

8. Set the "active" boolean of all the cue ranges in the *current ranges* list to "true" (active), and the "active" boolean of all the cue ranges in the *other ranges* list to "false" (inactive).

Invoking a callback (an object implementing one of the following two interfaces) means calling its `handleEvent()` method.

```
interface VoidCallback {
  void handleEvent();
};

interface CueRangeCallback {
  void handleEvent(in DOMString id);
};
```

The **`handleEvent`** method of objects implementing these interfaces is the entry point for the callback represented by the object.

4.7.10.10. User interface

The **`controls`** attribute is a boolean attribute. If the attribute is present, or if the media element is without script, then the user agent should **expose a user interface to the user**. This user interface should include features to begin playback, pause playback, seek to an arbitrary position in the content (if the content supports arbitrary seeking), change the volume, and show the media content in manners more suitable to the user (e.g.

full-screen video or in an independent resizable window). Other controls may also be made available.

If the attribute is absent, then the user agent should avoid making a user interface available that could conflict with an author-provided user interface. User agents may make the following features available, however, even when the attribute is absent:

User agents may provide controls to affect playback of the media resource (e.g. play, pause, seeking, and volume controls), but such features should not interfere with the page's normal rendering. For example, such features could be exposed in the media element's context menu.

Where possible (specifically, for starting, stopping, pausing, and unpausing playback, for muting or changing the volume of the audio, and for seeking), user interface features exposed by the user agent must be implemented in terms of the DOM API described above, so that, e.g., all the same events fire.

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

The **`volume`** attribute must return the playback volume of any audio portions of the media element, in the range 0.0 (silent) to 1.0 (loudest). Initially, the volume must be 1.0, but user agents may remember the last set value across sessions, on a per-site basis or otherwise, so the volume may start at other values. On setting, if the new value is in the range 0.0 to 1.0 inclusive, the attribute must be set to the new value and the playback volume must be correspondingly adjusted as soon as possible after setting the attribute, with 0.0 being silent, and 1.0 being the loudest setting, values in between increasing in loudness. The range need not be linear. The loudest setting may be lower than the system's loudest possible setting; for example the user could have set a maximum volume. If the new value is outside the range 0.0 to 1.0 inclusive, then, on setting, an `INDEX_SIZE_ERR` exception must be raised instead.

The **`muted`** attribute must return true if the audio channels are muted and false otherwise. On setting, the attribute must be set to the new value; if the new value is true, audio playback for this media resource must then be muted, and if false, audio playback must then be enabled.

Whenever either the `muted` or `volume` attributes are changed, after any running scripts have finished executing, the user agent must fire a simple event called `volumechange` at the media element.

4.7.10.11. Time ranges

Objects implementing the `TimeRanges` interface represent a list of ranges (periods) of time.

```
interface TimeRanges {
  readonly attribute unsigned long length;
  float start(in unsigned long index);
  float end(in unsigned long index);
};
```

The **`length`** DOM attribute must return the number of ranges represented by the object.

The **`start(`*`index`*`)`** method must return the position of the start of the *index*th range represented by the object, in seconds measured from the start of the timeline that the object covers.

The **`end(`*`index`*`)`** method must return the position of the end of the *index*th range represented by the object, in seconds measured from the start of the timeline that the object covers.

These methods must raise `INDEX_SIZE_ERR` exceptions if called with an *index* argument greater than or equal to the number of ranges represented by the object.

When a `TimeRanges` object is said to be a **normalized `TimeRanges` object**, the ranges it represents must obey the following criteria:

- The start of a range must be greater than the end of all earlier ranges.
- The start of a range must be less than the end of that same range.

In other words, the ranges in such an object are ordered, don't overlap, aren't empty, and don't touch (adjacent ranges are folded into one bigger range).

The timelines used by the objects returned by the `buffered`, `seekable` and `played` DOM attributes of media elements must be the same as that element's media resource's timeline.

4.7.10.12. Byte ranges

Objects implementing the `ByteRanges` interface represent a list of ranges of bytes.

```
interface ByteRanges {
  readonly attribute unsigned long length;
  unsigned long start(in unsigned long index);
  unsigned long end(in unsigned long index);
};
```

The **`length`** DOM attribute must return the number of ranges represented by the object.

The **`start(`*`index`*`)`** method must return the position of the first byte of the *index*th range represented by the object.

The **`end(`*`index`*`)`** method must return the position of the byte immediately after the last byte of the *index*th range represented by the object. (The byte position returned by this method is not in the range itself. If the first byte of the range is the byte at position 0, and the entire stream of bytes is in the range, then the value of the position of the byte returned by this method for that range will be the same as the number of bytes in the stream.)

These methods must raise `INDEX_SIZE_ERR` exceptions if called with an *index* argument greater than or equal to the number of ranges represented by the object.

When a `ByteRanges` object is said to be a **normalized `ByteRanges` object**, the ranges it represents must obey the following criteria:

- The start of a range must be greater than the end of all earlier ranges.
- The start of a range must be less than the end of that same range.

In other words, the ranges in such an object are ordered, don't overlap, aren't empty, and don't touch (adjacent ranges are folded into one bigger range).

4.7.10.13. Event summary

The following events fire on media elements as part of the processing model described above:

Event name	Interface	Dispatched when...	Preconditions
loadstart	ProgressEvent [PROGRESS]	The user agent begins fetching the media data, synchronously during the load() method call.	networkState equals LOADING
progress	ProgressEvent [PROGRESS]	The user agent is fetching media data.	networkState is more than EMPTY and less than LOADED
loadedmetadata	Event	The user agent is fetching media data, and the media resource's metadata has just been received.	networkState equals LOADED_METADATA
loadedfirstframe	Event	The user agent is fetching media data, and the media resource's first frame has just been received.	networkState equals LOADED_FIRST_FRAME
load	ProgressEvent [PROGRESS]	The user agent finishes downloading the entire media resource.	networkState equals LOADED
abort	ProgressEvent [PROGRESS]	The user agent stops fetching the media data before it is completely downloaded. This can be fired synchronously during the load() method call.	error is an object with the code MEDIA_ERR_ABORTED. networkState equals either EMPTY or LOADED, depending on when the download was aborted.
error	ProgressEvent [PROGRESS]	An error occurs while fetching the media data.	error is an object with the code MEDIA_ERR_NETWORK_ERROR or higher. networkState equals either EMPTY or LOADED, depending on when the download was aborted.
emptied	Event	A media element whose networkState was previously not in the EMPTY state has just switched to that state (either because of a fatal error during load that's about to be reported, or because the load() method was reinvoked, in which case it is fired synchronously during the load() method call).	networkState is EMPTY; all the DOM attributes are in their initial states.
stalled	ProgressEvent	The user agent is trying to fetch media data, but data is unexpectedly not forthcoming.	
play	Event	Playback has begun. Fired after the play method has returned.	paused is newly false.
pause	Event	Playback has been paused. Fired after the pause method has returned.	paused is newly true.
waiting	Event	Playback has stopped because the next frame is not available, but the user agent expects that frame to become available in due course.	readyState is either DATA_UNAVAILABLE or CAN_SHOW_CURRENT_FRAME, and paused is false. Either seeking is true, or the current playback position is not contained in any of the ranges in buffered. It is possible for playback to stop for two other reasons without paused being false, but those two reasons do not fire this event: maybe playback ended, or playback stopped due to errors.
seeking	Event	The seeking DOM attribute changed to true and the seek operation is taking long enough that the user agent has time to fire the event.	
seeked	Event	The seeking DOM attribute changed to false.	

Event name	Interface	Dispatched when...	Preconditions
timeupdate	Event	The current playback position changed in an interesting way, for example discontinuously.	
ended	Event	Playback has stopped because the end of the media resource was reached.	`currentTime` equals the *effective end*; `ended` is true.
dataunavailable	Event	The user agent cannot render the data at the current playback position because data for the current frame is not immediately available.	The `readyState` attribute is newly equal to `DATA_UNAVAILABLE`.
canshowcurrentframe	Event	The user agent cannot render the data after the current playback position because data for the next frame is not immediately available.	The `readyState` attribute is newly equal to `CAN_SHOW_CURRENT_FRAME`.
canplay	Event	The user agent can resume playback of the media data, but estimates that if playback were to be started now, the media resource could not be rendered at the current playback rate up to its end without having to stop for further buffering of content.	The `readyState` attribute is newly equal to `CAN_PLAY`.
canplaythrough	Event	The user agent estimates that if playback were to be started now, the media resource could be rendered at the current playback rate all the way to its end without having to stop for further buffering.	The `readyState` attribute is newly equal to `CAN_PLAY_THROUGH`.
ratechange	Event	Either the `defaultPlaybackRate` or the `playbackRate` attribute has just been updated.	
durationchange	Event	The `duration` attribute has just been updated.	
volumechange	Event	Either the `volume` attribute or the `muted` attribute has changed. Fired after the relevant attribute's setter has returned.	

4.7.10.14. Security and privacy considerations

Talk about making sure interactive media files (e.g. SVG) don't have access to the container DOM (XSS potential); talk about not exposing any sensitive data like metadata from tracks in the media files (intranet snooping risk)

4.7.11 The `canvas` element

Categories

Embedded content.

Contexts in which this element may be used:

Where embedded content is expected.

Content model:

Transparent.

Element-specific attributes:

`width`
`height`

DOM interface:

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

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

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

The `canvas` element represents a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, or other visual images on the fly.

Authors should not use the `canvas` element in a document when a more suitable element is available. For example, it is inappropriate to use a `canvas` element to render a page heading: if the desired presentation of the heading is graphically intense, it should be marked up using appropriate elements (typically `h1`) and then styled using CSS and supporting technologies such as XBL.

When authors use the `canvas` element, they 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, if the `canvas` element is with script, the `canvas` element represents an embedded element with a dynamically created image.

In non-interactive, static, visual media, if the `canvas` element has been previously painted on (e.g. if the page was viewed in an interactive visual medium and is now being printed, or if some script that ran during the page layout process painted on the element), then the `canvas` element represents embedded content with the current image and size. Otherwise, the element represents its fallback content instead.

In non-visual media, and in visual media if the `canvas` element is without script, the `canvas` element represents its fallback content instead.

The `canvas` element has two attributes to control the size of the coordinate space: **`width`** and **`height`**. These attributes, when specified, must have values that are valid non-negative integers. The rules for parsing non-negative integers must be used to obtain their numeric values. If an attribute is missing, or if parsing its value returns an error, then the default value must be used instead. The `width` attribute defaults to 300, and the `height` attribute defaults to 150.

The intrinsic dimensions of the `canvas` element equal the size of the coordinate space, with the numbers interpreted in CSS pixels. However, the element can be sized arbitrarily by a style sheet. During rendering, the image is scaled to fit this layout size.

The size of the coordinate space does not necessarily represent the size of the actual bitmap that the user agent will use internally or during rendering. On high-definition displays, for instance, the user agent may internally use a bitmap with two device pixels per unit in the coordinate space, so that the rendering remains at high quality throughout.

Whenever the `width` and `height` attributes are set (whether to a new value or to the previous value), the bitmap and any associated contexts must be cleared back to their initial state and reinitialized with the newly specified coordinate space dimensions.

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

> Only one square appears to be drawn in the following example:
>
> ```
> // canvas is a reference to a <canvas> element
> var context = canvas.getContext('2d');
> context.fillRect(0,0,50,50);
> canvas.setAttribute('width', '300'); // clears the canvas
> context.fillRect(0,100,50,50);
> canvas.width = canvas.width; // clears the canvas
> context.fillRect(100,0,50,50); // only this square remains
> ```

When the canvas is initialized it must be set to fully transparent black.

To draw on the canvas, authors must first obtain a reference to a **context** using the **`getContext(`*`contextId`*`)`** method of the `canvas` element.

This specification only defines one context, with the name "`2d`". If `getContext()` is called with that exact string for its *contextId* argument, then the UA must return a reference to an object implementing `CanvasRenderingContext2D`. Other specifications may define their own contexts, which would return different objects.

Vendors may also define experimental contexts using the syntax *`vendorname-context`*, for example, `moz-3d`.

When the UA is passed an empty string or a string specifying a context that it does not support, then it must return null. String comparisons must be literal and case-sensitive.

Arguments other than the *contextId* 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).

> ***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:` URL containing a representation of the image as a PNG file. [PNG].

If the canvas has no pixels (i.e. either its horizontal dimension or its vertical dimension is zero) then the method must return the string "`data:,`". (This is the shortest `data:` URL; it represents the empty string in a `text/plain` resource.)

The **`toDataURL(`*`type`*`)`** method (when called with one *or more* arguments) must return a `data:` URL containing a representation of the image in the format given by *type*. The possible values are MIME types with

no parameters, for example `image/png`, `image/jpeg`, or even maybe `image/svg+xml` if the implementation actually keeps enough information to reliably render an SVG image from the canvas.

Only support for `image/png` is required. User agents may support other types. If the user agent does not support the requested type, it must return the image using the PNG format.

User agents must convert the provided type to lower case before establishing if they support that type and before creating the `data:` URL.

> ***Note: When trying to use types other than `image/png`, authors can check if the image was really returned in the requested format by checking to see if the returned string starts with one the exact strings "`data:image/png,`" or "`data:image/png;`". If it does, the image is PNG, and thus the requested type was not supported. (The one exception to this is if the canvas has either no height or no width, in which case the result might simply be "`data:,`".)***

If the method is invoked with the first argument giving a type corresponding to one of the types given in the first column of the following table, and the user agent supports that type, then the subsequent arguments, if any, must be treated as described in the second cell of that row.

Type	Other arguments
image/jpeg	The second argument, if it is a number between 0.0 and 1.0, must be treated as the desired quality level.

Other arguments 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.

4.7.11.1. The 2D context

When the `getContext()` method of a `canvas` element is invoked with **`2d`** as the argument, a `CanvasRenderingContext2D` object is returned.

There is only one `CanvasRenderingContext2D` object per canvas, so calling the `getContext()` method with the `2d` argument a second time must return the same object.

The 2D context represents a flat Cartesian surface whose origin (0,0) is at the top left corner, with the coordinate space having *x* values increasing when going right, and *y* values increasing when going down.

```
interface CanvasRenderingContext2D {

  // back-reference to the canvas
  readonly attribute HTMLCanvasElement canvas;

  // state
  void save(); // push state on state stack
  void restore(); // pop state stack and restore state

  // transformations (default transform is the identity matrix)
  void scale(in float x, in float y);
  void rotate(in float angle);
```

```
  void translate(in float x, in float y);
  void transform(in float m11, in float m12, in float m21, in float m22, in
float dx, in float dy);
  void setTransform(in float m11, in float m12, in float m21, in float m22,
in float dx, in float dy);

  // compositing
           attribute float globalAlpha; // (default 1.0)
           attribute DOMString globalCompositeOperation; // (default source-
over)

  // colors and styles
           attribute DOMObject strokeStyle; // (default black)
           attribute DOMObject fillStyle; // (default black)
  CanvasGradient createLinearGradient(in float x0, in float y0, in float x1,
in float y1);
  CanvasGradient createRadialGradient(in float x0, in float y0, in float r0,
in float x1, in float y1, in float r1);
  CanvasPattern createPattern(in HTMLImageElement image, in DOMString
repetition);
  CanvasPattern createPattern(in HTMLCanvasElement image, in DOMString
repetition);

  // line caps/joins
           attribute float lineWidth; // (default 1)
           attribute DOMString lineCap; // "butt", "round", "square"
(default "butt")
           attribute DOMString lineJoin; // "round", "bevel", "miter"
(default "miter")
           attribute float miterLimit; // (default 10)

  // shadows
           attribute float shadowOffsetX; // (default 0)
           attribute float shadowOffsetY; // (default 0)
           attribute float shadowBlur; // (default 0)
           attribute DOMString shadowColor; // (default transparent black)

  // rects
  void clearRect(in float x, in float y, in float w, in float h);
  void fillRect(in float x, in float y, in float w, in float h);
  void strokeRect(in float x, in float y, in float w, in float h);

  // path API
  void beginPath();
  void closePath();
  void moveTo(in float x, in float y);
  void lineTo(in float x, in float y);
  void quadraticCurveTo(in float cpx, in float cpy, in float x, in float y);
  void bezierCurveTo(in float cp1x, in float cp1y, in float cp2x, in float
```

```
cp2y, in float x, in float y);
  void arcTo(in float x1, in float y1, in float x2, in float y2, in float
radius);
  void rect(in float x, in float y, in float w, in float h);
  void arc(in float x, in float y, in float radius, in float startAngle, in
float endAngle, in boolean anticlockwise);
  void fill();
  void stroke();
  void clip();
  boolean isPointInPath(in float x, in float y);

  // text
           attribute DOMString font; // (default 10px sans-serif)
           attribute DOMString textAlign; // "start", "end", "left",
"right", "center" (default: "start")
           attribute DOMString textBaseline; // "top", "hanging", "middle",
"alphabetic", "ideographic", "bottom" (default: "alphabetic")
  void fillText(in DOMString text, in float x, in float y);
  void fillText(in DOMString text, in float x, in float y, in float
maxWidth);
  void strokeText(in DOMString text, in float x, in float y);
  void strokeText(in DOMString text, in float x, in float y, in float
maxWidth);
  TextMetrics measureText(in DOMString text);

  // drawing images
  void drawImage(in HTMLImageElement image, in float dx, in float dy);
  void drawImage(in HTMLImageElement image, in float dx, in float dy, in
float dw, in float dh);
  void drawImage(in HTMLImageElement image, in float sx, in float sy, in
float sw, in float sh, in float dx, in float dy, in float dw, in float dh);
  void drawImage(in HTMLCanvasElement image, in float dx, in float dy);
  void drawImage(in HTMLCanvasElement image, in float dx, in float dy, in
float dw, in float dh);
  void drawImage(in HTMLCanvasElement image, in float sx, in float sy, in
float sw, in float sh, in float dx, in float dy, in float dw, in float dh);

  // pixel manipulation
  ImageData createImageData(in float sw, in float sh);
  ImageData getImageData(in float sx, in float sy, in float sw, in float
sh);
  void putImageData(in ImageData imagedata, in float dx, in float dy);
  void putImageData(in ImageData imagedata, in float dx, in float dy, in
float dirtyX, in float dirtyY, in float dirtyWidth, in float dirtyHeight);
};

interface CanvasGradient {
  // opaque object
  void addColorStop(in float offset, in DOMString color);
```

```
};

interface CanvasPattern {
  // opaque object
};

interface TextMetrics {
  readonly attribute float width;
};

interface ImageData {
  readonly attribute long int width;
  readonly attribute long int height;
  readonly attribute CanvasPixelArray data;
};

interface CanvasPixelArray {
  readonly attribute unsigned long length;
  [IndexGetter] float XXX5(in unsigned long index);
  [IndexSetter] void XXX6(in unsigned long index, in float value);
};
```

The **`canvas`** attribute must return the `canvas` element that the context paints on.

Unless otherwise stated, for the 2D context interface, any method call with a numeric argument whose value is infinite or a NaN value must be ignored.

Whenever the CSS value `currentColor` is used as a color in this API, the "computed value of the 'color' property" for the purposes of determining the computed value of the `currentColor` keyword is the computed value of the 'color' property on the element in question at the time that the color is specified (e.g. when the appropriate attribute is set, or when the method is called; not when the color is rendered or otherwise used). If the computed value of the 'color' property is undefined for a particular case (e.g. because the element is not in a document), then the "computed value of the 'color' property" for the purposes of determining the computed value of the `currentColor` keyword is fully opaque black. [CSS3COLOR]

4.7.11.1.1. The canvas state

Each context maintains a stack of drawing states. **Drawing states** consist of:

- The current transformation matrix.
- The current clipping region.
- The current values of the following attributes: `strokeStyle`, `fillStyle`, `globalAlpha`, `lineWidth`, `lineCap`, `lineJoin`, `miterLimit`, `shadowOffsetX`, `shadowOffsetY`, `shadowBlur`, `shadowColor`, `globalCompositeOperation`, `font`, `textAlign`, `textBaseline`.

Note: The current path and the current bitmap are not part of the drawing state. The current path is persistent, and can only be reset using the `beginPath()` method. The current bitmap is a property of the canvas, not the context.

The **`save()`** method must push a copy of the current drawing state onto the drawing state stack.

The **`restore()`** method must pop the top entry in the drawing state stack, and reset the drawing state it describes. If there is no saved state, the method must do nothing.

4.7.11.1.2. Transformations

The transformation matrix is applied to coordinates when creating shapes and paths.

When the context is created, the transformation matrix must initially be the identity transform. It may then be adjusted using the transformation methods.

The transformations must be performed in reverse order. For instance, if a scale transformation that doubles the width is applied, followed by a rotation transformation that rotates drawing operations by a quarter turn, and a rectangle twice as wide as it is tall is then drawn on the canvas, the actual result will be a square.

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. If the *angle* argument is infinite, the method call must be ignored.

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	*m21*	*dx*
m12	*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.

4.7.11.1.3. 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, *A*, is the shape or image being rendered, and the destination image, *B*, is the current state of the bitmap.

source-atop

A atop *B*. Display the source image wherever both images are opaque. Display the destination image wherever the destination image is opaque but the source image is transparent. Display transparency elsewhere.

source-in

A in *B*. Display the source image wherever both the source image and destination image are opaque. Display transparency elsewhere.

source-out

A out *B*. Display the source image wherever the source image is opaque and the destination image is transparent. Display transparency elsewhere.

source-over **(default)**

A over *B*. Display the source image wherever the source image is opaque. Display the destination image elsewhere.

destination-atop

B atop *A*. Same as `source-atop` but using the destination image instead of the source image and vice versa.

destination-in

B in *A*. Same as `source-in` but using the destination image instead of the source image and vice versa.

destination-out

B out *A*. Same as `source-out` but using the destination image instead of the source image and vice versa.

destination-over

B over *A*. Same as `source-over` but using the destination image instead of the source image and vice versa.

lighter

A plus *B*. Display the sum of the source image and destination image, with color values approaching 1 as a limit.

copy

A (*B* is ignored). Display the source image instead of the destination image.

xor

A xor *B*. Exclusive OR of the source image and destination image.

vendorName-operationName

Vendor-specific extensions to the list of composition operators should use this syntax.

These values are all case-sensitive — they must be used exactly as shown. User agents must not recognize values that do not exactly match the values given above.

The operators in the above list must be treated as described by the Porter-Duff operator given at the start of their description (e.g. *A* over *B*). [PORTERDUFF]

On setting, if the user agent does not recognize the specified value, it must be ignored, leaving the value of `globalCompositeOperation` unaffected.

When the context is created, the `globalCompositeOperation` attribute must initially have the value `source-over`.

4.7.11.1.4. 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 must be parsed as CSS <color> values and the color assigned, and `CanvasGradient` and `CanvasPattern` objects must be assigned themselves. [CSS3COLOR] If the value is a string but is not a valid color, or is neither a string, a `CanvasGradient`, nor a `CanvasPattern`, then it must be ignored, and the attribute must retain its previous value.

On getting, if the value is a color, then the serialization of the color must be returned. Otherwise, if it is not a color but a `CanvasGradient` or `CanvasPattern`, then the respective object must be returned. (Such objects are opaque and therefore only useful for assigning to other attributes or for comparison to other gradients or patterns.)

The **serialization of a color** for a color value is a string, computed as follows: if it has alpha equal to 1.0, then the string is a lowercase six-digit hex value, prefixed with a "#" character (U+0023 NUMBER SIGN), with the first two digits representing the red component, the next two digits representing the green component, and the last two digits representing the blue component, the digits being in the range 0-9 a-f (U+0030 to U+0039 and U+0061 to U+0066). Otherwise, the color value has alpha less than 1.0, and the string is the color value in the CSS `rgba()` functional-notation format: the literal string `rgba` (U+0072 U+0067 U+0062 U+0061) followed by a U+0028 LEFT PARENTHESIS, a base-ten integer in the range 0-255 representing the red component (using digits 0-9, U+0030 to U+0039, in the shortest form possible), a literal U+002C COMMA and U+0020 SPACE, an integer for the green component, a comma and a space, an integer for the blue component, another comma and space, a U+0030 DIGIT ZERO, a U+002E FULL STOP (representing the decimal point), one or more digits in the range 0-9 (U+0030 to U+0039) representing the fractional part of the alpha value, and finally a U+0029 RIGHT PARENTHESIS.

When the context is created, the `strokeStyle` and `fillStyle` attributes must initially have the string value `#000000`.

There are two types of gradients, linear gradients and radial gradients, both represented by objects implementing the opaque `CanvasGradient` interface.

Once a gradient has been created (see below), stops are placed along it to define how the colors are distributed along the gradient. The color of the gradient at each stop is the color specified for that stop. Between each such stop, the colors and the alpha component must be linearly interpolated over the RGBA space without premultiplying the alpha value to find the color to use at that offset. Before the first stop, the color must be the color of the first stop. After the last stop, the color must be the color of the last stop. When there are no stops, the gradient is transparent black.

The **`addColorStop(`*`offset, color`*`)`** method on the `CanvasGradient` interface adds a new stop to a gradient. If the *offset* is less than 0, greater than 1, infinite, or NaN, then an `INDEX_SIZE_ERR` exception must be raised. If the *color* cannot be parsed as a CSS color, then a `SYNTAX_ERR` exception must be raised.

Otherwise, the gradient must have a new stop placed, at offset *offset* relative to the whole gradient, and with the color obtained by parsing *color* as a CSS <color> value. If multiple stops are added at the same offset on a gradient, they must be placed in the order added, with the first one closest to the start of the gradient, and each subsequent one infinitesimally further along towards the end point (in effect causing all but the first and last stop added at each point to be ignored).

The **`createLinearGradient(x0, y0, x1, y1)`** method takes four arguments that represent the start point (*x0*, *y0*) and end point (*x1*, *y1*) of the gradient. If any of the arguments to `createLinearGradient()` are infinite or NaN, the method must raise a `NOT_SUPPORTED_ERR` exception. Otherwise, the method must return a linear `CanvasGradient` initialized with the specified line.

Linear gradients must be rendered such that all points on a line perpendicular to the line that crosses the start and end points have the color at the point where those two lines cross (with the colors coming from the interpolation and extrapolation described above). The points in the linear gradient must be transformed as described by the current transformation matrix when rendering.

If $x_0 = x_1$ and $y_0 = y_1$, then the linear gradient must paint nothing.

The **`createRadialGradient(x0, y0, r0, x1, y1, r1)`** method takes six arguments, the first three representing the start circle with origin (*x0*, *y0*) and radius *r0*, and the last three representing the end circle with origin (*x1*, *y1*) and radius *r1*. The values are in coordinate space units. If any of the arguments are infinite or NaN, a `NOT_SUPPORTED_ERR` exception must be raised. If either of *r0* or *r1* are negative, an `INDEX_SIZE_ERR` exception must be raised. Otherwise, the method must return a radial `CanvasGradient` initialized with the two specified circles.

Radial gradients must be rendered by following these steps:

1. If $x_0 = x_1$ and $y_0 = y_1$ and $r_0 = r_1$, then the radial gradient must paint nothing. Abort these steps.

2. Let $x(\omega) = (x_1-x_0)\omega + x_0$

   Let $y(\omega) = (y_1-y_0)\omega + y_0$

   Let $r(\omega) = (r_1-r_0)\omega + r_0$

   Let the color at ω be the color at that position on the gradient (with the colors coming from the interpolation and extrapolation described above).

3. For all values of ω where $r(\omega) > 0$, starting with the value of ω nearest to positive infinity and ending with the value of ω nearest to negative infinity, draw the circumference of the circle with radius $r(\omega)$ at position $(x(\omega), y(\omega))$, with the color at ω, but only painting on the parts of the canvas that have not yet been painted on by earlier circles in this step for this rendering of the gradient.

Note: This effectively creates a cone, touched by the two circles defined in the creation of the gradient, with the part of the cone before the start circle (0.0) using the color of the first offset, the part of the cone after the end circle (1.0) using the color of the last offset, and areas outside the cone untouched by the gradient (transparent black).

Gradients must be painted only where the relevant stroking or filling effects requires that they be drawn.

The points in the radial gradient must be transformed as described by the current transformation matrix when rendering.

Patterns are represented by objects implementing the opaque `CanvasPattern` interface.

To create objects of this type, the **`createPattern(`*`image, repetition`*`)`** method is used. The first argument gives the image to use as the pattern (either an `HTMLImageElement` or an `HTMLCanvasElement`). Modifying this image after calling the `createPattern()` method must not affect the pattern. The second argument must be a string with one of the following values: `repeat`, `repeat-x`, `repeat-y`, `no-repeat`. If the empty string or null is specified, `repeat` must be assumed. If an unrecognized value is given, then the user agent must raise a `SYNTAX_ERR` exception. User agents must recognize the four values described above exactly (e.g. they must not do case folding). The method must return a `CanvasPattern` object suitably initialized.

The *image* argument must be an instance of an `HTMLImageElement` or `HTMLCanvasElement`. If the *image* is of the wrong type or null, the implementation must raise a `TYPE_MISMATCH_ERR` exception.

If the *image* argument is an `HTMLImageElement` object whose `complete` attribute is false, then the implementation must raise an `INVALID_STATE_ERR` exception.

If the *image* argument is an `HTMLCanvasElement` object with either a horizontal dimension or a vertical dimension equal to zero, then the implementation must raise an `INVALID_STATE_ERR` exception.

Patterns must be painted so that the top left of the first image is anchored at the origin of the coordinate space, and images are then repeated horizontally to the left and right (if the `repeat-x` string was specified) or vertically up and down (if the `repeat-y` string was specified) or in all four directions all over the canvas (if the `repeat` string was specified). The images are not scaled by this process; one CSS pixel of the image must be painted on one coordinate space unit. Of course, patterns must actually be painted only where the stroking or filling effect requires that they be drawn, and are affected by the current transformation matrix.

When the `createPattern()` method is passed, as its *image* argument, an animated image, the poster frame of the animation, or the first frame of the animation if there is no poster frame, must be used.

Support for patterns is optional. If the user agent doesn't support patterns, then `createPattern()` must return null.

4.7.11.1.5. Line styles

The **`lineWidth`** attribute gives the width of lines, in coordinate space units. On setting, zero, negative, infinite, and NaN values must be ignored, leaving the value unchanged.

When the context is created, the `lineWidth` attribute must initially have the value `1.0`.

The **`lineCap`** attribute defines the type of endings that UAs will place on the end of lines. The three valid values are `butt`, `round`, and `square`. The `butt` value means that the end of each line has a flat edge perpendicular to the direction of the line (and that no additional line cap is added). The `round` value means that a semi-circle with the diameter equal to the width of the line must then be added on to the end of the line. The `square` value means that a rectangle with the length of the line width and the width of half the line width, placed flat against the edge perpendicular to the direction of the line, must be added at the end of each line. On setting, any other value than the literal strings `butt`, `round`, and `square` must be ignored, leaving the value unchanged.

When the context is created, the `lineCap` attribute must initially have the value `butt`.

The **lineJoin** attribute defines the type of corners that UAs will place where two lines meet. The three valid values are `bevel`, `round`, and `miter`.

On setting, any other value than the literal strings `bevel`, `round`, and `miter` must be ignored, leaving the value unchanged.

When the context is created, the `lineJoin` attribute must initially have the value `miter`.

A join exists at any point in a subpath shared by two consecutive lines. When a subpath is closed, then a join also exists at its first point (equivalent to its last point) connecting the first and last lines in the subpath.

In addition to the point where the join occurs, two additional points are relevant to each join, one for each line: the two corners found half the line width away from the join point, one perpendicular to each line, each on the side furthest from the other line.

A filled triangle connecting these two opposite corners with a straight line, with the third point of the triangle being the join point, must be rendered at all joins. The `lineJoin` attribute controls whether anything else is rendered. The three aforementioned values have the following meanings:

The `bevel` value means that this is all that is rendered at joins.

The `round` value means that a filled arc connecting the two aforementioned corners of the join, abutting (and not overlapping) the aforementioned triangle, with the diameter equal to the line width and the origin at the point of the join, must be rendered at joins.

The `miter` value means that a second filled triangle must (if it can given the miter length) be rendered at the join, with one line being the line between the two aforementioned corners, abutting the first triangle, and the other two being continuations of the outside edges of the two joining lines, as long as required to intersect without going over the miter length.

The miter length is the distance from the point where the lines touch on the inside of the join to the intersection of the line edges on the outside of the join. The miter limit ratio is the maximum allowed ratio of the miter length to half the line width. If the miter length would cause the miter limit ratio to be exceeded, this second triangle must not be rendered.

The miter limit ratio can be explicitly set using the **miterLimit** attribute. On setting, zero, negative, infinite, and NaN values must be ignored, leaving the value unchanged.

When the context is created, the `miterLimit` attribute must initially have the value `10.0`.

4.7.11.1.6. Shadows

All drawing operations are affected by the four global shadow attributes.

The **shadowColor** attribute sets the color of the shadow.

When the context is created, the `shadowColor` attribute initially must be fully-transparent black.

On getting, the serialization of the color must be returned.

On setting, the new value must be parsed as a CSS <color> value and the color assigned. If the value is not a valid color, then it must be ignored, and the attribute must retain its previous value. [CSS3COLOR]

The **shadowOffsetX** and **shadowOffsetY** attributes specify the distance that the shadow will be offset in the positive horizontal and positive vertical distance respectively. Their values are in coordinate space units. They are not affected by the current transformation matrix.

When the context is created, the shadow offset attributes must initially have the value `0`.

On getting, they must return their current value. On setting, the attribute being set must be set to the new value, except if the value is infinite or NaN, in which case the new value must be ignored.

The **shadowBlur** attribute specifies the size of the blurring effect. (The units do not map to coordinate space units, and are not affected by the current transformation matrix.)

When the context is created, the `shadowBlur` attribute must initially have the value `0`.

On getting, the attribute must return its current value. On setting the attribute must be set to the new value, except if the value is negative, infinite or NaN, in which case the new value must be ignored.

When shadows are drawn, they must be rendered as follows:

1. Let *A* be the source image for which a shadow is being created.
2. Let *B* be an infinite transparent black bitmap, with a coordinate space and an origin identical to *A*.
3. Copy the alpha channel of *A* to *B*, offset by `shadowOffsetX` in the positive *x* direction, and `shadowOffsetY` in the positive *y* direction.
4. If `shadowBlur` is greater than 0:
    1. If `shadowBlur` is less than 8, let *σ* be half the value of `shadowBlur`; otherwise, let *σ* be the square root of multiplying the value of `shadowBlur` by 2.
    2. Perform a 2D Gaussian Blur on *B*, using *σ* as the standard deviation.

    User agents may limit values of *σ* to an implementation-specific maximum value to avoid exceeding hardware limitations during the Gaussian blur operation.
5. Set the red, green, and blue components of every pixel in *B* to the red, green, and blue components (respectively) of the color of `shadowColor`.
6. Multiply the alpha component of every pixel in *B* by the alpha component of the color of `shadowColor`.
7. The shadow is in the bitmap *B*, and is rendered as part of the drawing model described below.

4.7.11.1.7. Simple shapes (rectangles)

There are three methods that immediately draw rectangles to the bitmap. They each take four arguments; the first two give the *x* and *y* coordinates of the top left of the rectangle, and the second two give the width *w* and height *h* of the rectangle, respectively.

The current transformation matrix must be applied to the following four coordinates, which form the path that must then be closed to get the specified rectangle: (*x*, *y*), (*x*+*w*, *y*), (*x*+*w*, *y*+*h*), (*x*, *y*+*h*).

Shapes are painted without affecting the current path, and are subject to the clipping region, and, with the

exception of `clearRect()`, also shadow effects, global alpha, and global composition operators.

The **`clearRect(x, y, w, h)`** method must clear the pixels in the specified rectangle that also intersect the current clipping region to a fully transparent black, erasing any previous image. If either height or width are zero, this method has no effect.

The **`fillRect(x, y, w, h)`** method must paint the specified rectangular area using the `fillStyle`. If either height or width are zero, this method has no effect.

The **`strokeRect(x, y, w, h)`** method must stroke the specified rectangle's path using the `strokeStyle`, `lineWidth`, `lineJoin`, and (if appropriate) `miterLimit` attributes. If both height and width are zero, this method has no effect, since there is no path to stroke (it's a point). If only one of the two is zero, then the method will draw a line instead (the path for the outline is just a straight line along the non-zero dimension).

4.7.11.1.8. Complex shapes (paths)

The context always has a current path. There is only one current path, it is not part of the drawing state.

A **path** has a list of zero or more subpaths. Each subpath consists of a list of one or more points, connected by straight or curved lines, and a flag indicating whether the subpath is closed or not. A closed subpath is one where the last point of the subpath is connected to the first point of the subpath by a straight line. Subpaths with fewer than two points are ignored when painting the path.

Initially, the context's path must have zero subpaths.

The points and lines added to the path by these methods must be transformed according to the current transformation matrix as they are added.

The **`beginPath()`** method must empty the list of subpaths so that the context once again has zero subpaths.

The **`moveTo(x, y)`** method must create a new subpath with the specified point as its first (and only) point.

The **`closePath()`** method must do nothing if the context has no subpaths. Otherwise, it must mark the last subpath as closed, create a new subpath whose first point is the same as the previous subpath's first point, and finally add this new subpath to the path. (If the last subpath had more than one point in its list of points, then this is equivalent to adding a straight line connecting the last point back to the first point, thus "closing" the shape, and then repeating the last `moveTo()` call.)

New points and the lines connecting them are added to subpaths using the methods described below. In all cases, the methods only modify the last subpath in the context's paths.

The **`lineTo(x, y)`** method must do nothing if the context has no subpaths. Otherwise, it must connect the last point in the subpath to the given point (*x*, *y*) using a straight line, and must then add the given point (*x*, *y*) to the subpath.

The **`quadraticCurveTo(cpx, cpy, x, y)`** method must do nothing if the context has no subpaths. Otherwise it must connect the last point in the subpath to the given point (*x*, *y*) using a quadratic Bézier curve with control point (*cpx*, *cpy*), and must then add the given point (*x*, *y*) to the subpath. [BEZIER]

The **`bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)`** method must do nothing if the context has no subpaths. Otherwise, it must connect the last point in the subpath to the given point (*x*, *y*) using a cubic Bézier

curve with control points (*cp1x*, *cp1y*) and (*cp2x*, *cp2y*). Then, it must add the point (*x*, *y*) to the subpath. [BEZIER]

The **`arcTo(x1, y1, x2, y2, radius)`** method must do nothing if the context has no subpaths. If the context *does* have a subpath, then the behavior depends on the arguments and the last point in the subpath.

Negative values for *radius* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

Let the point (*x0*, *y0*) be the last point in the subpath.

If the point (*x0*, *y0*) is equal to the point (*x1*, *y1*), or if the point (*x1*, *y1*) is equal to the point (*x2*, *y2*), or if the radius *radius* is zero, then the method must add the point (*x1*, *y1*) to the subpath, and connect that point to the previous point (*x0*, *y0*) by a straight line.

Otherwise, if the points (*x0*, *y0*), (*x1*, *y1*), and (*x2*, *y2*) all lie on a single straight line, then: if the direction from (*x0*, *y0*) to (*x1*, *y1*) is the same as the direction from (*x1*, *y1*) to (*x2*, *y2*), then the method must add the point (*x1*, *y1*) to the subpath, and connect that point to the previous point (*x0*, *y0*) by a straight line; otherwise, the direction from (*x0*, *y0*) to (*x1*, *y1*) is the opposite of the direction from (*x1*, *y1*) to (*x2*, *y2*), and the method must add a point (x_∞, y_∞) to the subpath, and connect that point to the previous point (*x0*, *y0*) by a straight line, where (x_∞, y_∞) is the point that is infinitely far away from (*x1*, *y1*), that lies on the same line as (*x0*, *y0*), (*x1*, *y1*), and (*x2*, *y2*), and that is on the same side of (*x1*, *y1*) on that line as (*x2*, *y2*).

Otherwise, let *The Arc* be the shortest arc given by circumference of the circle that has radius *radius*, and that has one point tangent to the half-infinite line that crosses the point (*x0*, *y0*) and ends at the point (*x1*, *y1*), and that has a different point tangent to the half-infinite line that ends at the point (*x1*, *y1*) and crosses the point (*x2*, *y2*). The points at which this circle touches these two lines are called the start and end tangent points respectively.

The method must connect the point (*x0*, *y0*) to the start tangent point by a straight line, adding the start tangent point to the subpath, and then must connect the start tangent point to the end tangent point by *The Arc*, adding the end tangent point to the subpath.

The **`arc(x, y, radius, startAngle, endAngle, anticlockwise)`** method draws an arc. If the context has any subpaths, then the method must add a straight line from the last point in the subpath to the start point of the arc. In any case, it must draw the arc between the start point of the arc and the end point of the arc, and add the start and end points of the arc to the subpath. The arc and its start and end points are defined as follows:

Consider a circle that has its origin at (*x*, *y*) and that has radius *radius*. The points at *startAngle* and *endAngle* along this circle's circumference, measured in radians clockwise from the positive x-axis, are the start and end points respectively.

If the *anticlockwise* argument is false and *endAngle*-*startAngle* is equal to or greater than 2π, or, if the *anticlockwise* argument is *true* and *startAngle*-*endAngle* is equal to or greater than 2π, then the arc is the whole circumference of this circle.

Otherwise, the arc is the path along the circumference of this circle from the start point to the end point, going anti-clockwise if the *anticlockwise* argument is true, and clockwise otherwise. Since the points are on the circle, as opposed to being simply angles from zero, the arc can never cover an angle greater than 2π radians. If the two points are the same, or if the radius is zero, then the arc is defined as being of zero length in both directions.

Negative values for *radius* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`rect(x, y, w, h)`** method must create a new subpath containing just the four points (*x*, *y*), (*x*+*w*, *y*), (*x*+*w*, *y*+*h*), (*x*, *y*+*h*), with those four points connected by straight lines, and must then mark the subpath as closed. It must then create a new subpath with the point (*x*, *y*) as the only point in the subpath.

The **`fill()`** method must fill all the subpaths of the current path, using `fillStyle`, and using the non-zero winding number rule. Open subpaths must be implicitly closed when being filled (without affecting the actual subpaths).

> ***Note: Thus, if two overlapping but otherwise independent subpaths have opposite windings, they cancel out and result in no fill. If they have the same winding, that area just gets painted once.***

The **`stroke()`** method must calculate the strokes of all the subpaths of the current path, using the `lineWidth`, `lineCap`, `lineJoin`, and (if appropriate) `miterLimit` attributes, and then fill the combined stroke area using the `strokeStyle`, attribute.

> ***Note: Since the subpaths are all stroked as one, overlapping parts of the paths in one stroke operation are treated as if their union was what was painted.***

Paths, when filled or stroked, must be painted without affecting the current path, and must be subject to shadow effects, global alpha, the clipping region, and global composition operators. (Transformations affect the path when the path is created, not when it is painted, though the stroke *style* is still affected by the transformation during painting.)

Zero-length line segments must be pruned before stroking a path. Empty subpaths must be ignored.

The **`clip()`** method must create a new **clipping region** by calculating the intersection of the current clipping region and the area described by the current path, using the non-zero winding number rule. Open subpaths must be implicitly closed when computing the clipping region, without affecting the actual subpaths. The new clipping region replaces the current clipping region.

When the context is initialized, the clipping region must be set to the rectangle with the top left corner at (0,0) and the width and height of the coordinate space.

The **`isPointInPath(x, y)`** method must return true if the point given by the *x* and *y* coordinates passed to the method, when treated as coordinates in the canvas coordinate space unaffected by the current transformation, is inside the current path; and must return false otherwise. Points on the path itself are considered to be inside the path. If either of the arguments is infinite or NaN, then the method must return false.

4.7.11.1.9. Text

The **`font`** DOM attribute, on setting, must be parsed the same way as the 'font' property of CSS (but without supporting property-independent stylesheet syntax like 'inherit'), and the resulting font must be assigned to the context, with the 'line-height' component forced to 'normal'. [CSS]

Font names must be interpreted in the context of the `canvas` element's stylesheets; any fonts embedded using `@font-face` must therefore be available. [CSSWEBFONTS]

Only vector fonts should be used by the user agent; if a user agent were to use bitmap fonts then

transformations would likely make the font look very ugly.

On getting, the `font` attribute must return the serialized form of the current font of the context. [CSSOM]

When the context is created, the font of the context must be set to 10px sans-serif. When the 'font-size' component is set to lengths using percentages, 'em' or 'ex' units, or the 'larger' or 'smaller' keywords, these must be interpreted relative to the computed value of the 'font-size' property of the corresponding `canvas` element at the time that the attribute is set. When the 'font-weight' component is set to the relative values 'bolder' and 'lighter', these must be interpreted relative to the computed value of the 'font-weight' property of the corresponding `canvas` element at the time that the attribute is set. If the computed values are undefined for a particular case (e.g. because the `canvas` element is not in a document), then the relative keywords must be interpreted relative to the normal-weight 10px sans-serif default.

The **`textAlign`** DOM attribute, on getting, must return the current value. On setting, if the value is one of `start`, `end`, `left`, `right`, or `center`, then the value must be changed to the new value. Otherwise, the new value must be ignored. When the context is created, the `textAlign` attribute must initially have the value `start`.

The **`textBaseline`** DOM attribute, on getting, must return the current value. On setting, if the value is one of `top`, `hanging`, `middle`, `alphabetic`, `ideographic`, or `bottom`, then the value must be changed to the new value. Otherwise, the new value must be ignored. When the context is created, the `textBaseline` attribute must initially have the value `alphabetic`.

The `textBaseline` attribute's allowed keywords correspond to alignment points in the font:

The keywords map to these alignment points as follows:

`top`

The top of the em square

`hanging`

The hanging baseline

`middle`

The middle of the em square

alphabetic

The alphabetic baseline

ideographic

The ideographic baseline

bottom

The bottom of the em square

The **`fillText()`** and **`strokeText()`** methods take three or four arguments, *text*, *x*, *y*, and optionally *maxWidth*, and render the given *text* at the given (*x*, *y*) coordinates ensuring that the text isn't wider than *maxWidth* if specified, using the current `font`, `textAlign`, and `textBaseline` values. Specifically, when the methods are called, the user agent must run the following steps:

1. Let *font* be the current font of the browsing context, as given by the `font` attribute.

2. Replace all the space characters in *text* with U+0020 SPACE characters.

3. Form a hypothetical infinitely wide CSS line box containing a single inline box containing the text *text*, with all the properties at their initial values except the 'font' property of the inline element set to *font* and the 'direction' property of the inline element set to the 'direction' property of the `canvas` element. [CSS]

4. If the *maxWidth* argument was specified and the hypothetical width of the inline box in the hypothetical line box is greater than *maxWidth* CSS pixels, then change *font* to have a more condensed font (if one is available or if a reasonably readable one can be synthesized by applying a horizontal scale factor to the font) or a smaller font, and return to the previous step.

5. Let the *anchor point* be a point on the inline box, determined by the `textAlign` and `textBaseline` values, as follows:

   Horizontal position:

   If `textAlign` is `left`

   If `textAlign` is `start` and the 'direction' property on the `canvas` element has a computed value of 'ltr'

   If `textAlign` is `end` and the 'direction' property on the `canvas` element has a computed value of 'rtl'

   Let the *anchor point*'s horizontal position be the left edge of the inline box.

   If `textAlign` is `right`

   If `textAlign` is `end` and the 'direction' property on the `canvas` element has a computed value of 'ltr'

   If `textAlign` is `start` and the 'direction' property on the `canvas` element has a computed value of 'rtl'

   Let the *anchor point*'s horizontal position be the right edge of the inline box.

   If `textAlign` is `center`

   Let the *anchor point*'s horizontal position be half way between the left and right edges of the inline box.

Vertical position:

If `textBaseline` is `top`

Let the *anchor point*'s vertical position be the top of the em box of the first available font of the inline box.

If `textBaseline` is `hanging`

Let the *anchor point*'s vertical position be the hanging baseline of the first available font of the inline box.

If `textBaseline` is `middle`

Let the *anchor point*'s vertical position be half way between the bottom and the top of the em box of the first available font of the inline box.

If `textBaseline` is `alphabetic`

Let the *anchor point*'s vertical position be the alphabetic baseline of the first available font of the inline box.

If `textBaseline` is `ideographic`

Let the *anchor point*'s vertical position be the ideographic baseline of the first available font of the inline box.

If `textBaseline` is `bottom`

Let the *anchor point*'s vertical position be the bottom of the em box of the first available font of the inline box.

6. Paint the hypothetical inline box as the shape given by the text's glyphs, as transformed by the current transformation matrix, and anchored and sized so that before applying the current transformation matrix, the *anchor point* is at (*x*, *y*) and each CSS pixel is mapped to one coordinate space unit.

   For `fillText()` `fillStyle` must be applied to the glyphs and `strokeStyle` must be ignored. For `strokeText()` the reverse holds and `strokeStyle` must be applied to the glyph outlines and `fillStyle` must be ignored.

   Text is painted without affecting the current path, and is subject to shadow effects, global alpha, the clipping region, and global composition operators.

The **`measureText()`** method takes one argument, *text*. When the method is invoked, the user agent must replace all the space characters in *text* with U+0020 SPACE characters, and then must form a hypothetical infinitely wide CSS line box containing a single inline box containing the text *text*, with all the properties at their initial values except the 'font' property of the inline element set to the current font of the browsing context, as given by the `font` attribute, and must then return a new `TextMetrics` object with its `width` attribute set to the width of that inline box, in CSS pixels. [CSS]

The `TextMetrics` interface is used for the objects returned from `measureText()`. It has one attribute, **`width`**, which is set by the `measureText()` method.

Note: Glyphs rendered using `fillText()` and `strokeText()` can spill out of the box given by the font size (the em square size) and the width returned by `measureText()` (the text width). This version of the specification does not provide a way to obtain the bounding box dimensions of the text. If the text is to be rendered and removed, care needs to be taken to

replace the entire area of the canvas that the clipping region covers, not just the box given by the em square height and measured text width.

Note: A future version of the 2D context API may provide a way to render fragments of documents, rendered using CSS, straight to the canvas. This would be provided in preference to a dedicated way of doing multiline layout.

4.7.11.1.10. 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 must default to the values of *sw* and *sh*, interpreted such that one CSS pixel in the image is treated as one unit in the canvas coordinate space. If the *sx*, *sy*, *sw*, and *sh* arguments are omitted, they must default to 0, 0, the image's intrinsic width in image pixels, and the image's intrinsic height in image pixels, respectively.

The *image* argument must be an instance of an `HTMLImageElement` or `HTMLCanvasElement`. If the *image* is of the wrong type or null, the implementation must raise a `TYPE_MISMATCH_ERR` exception.

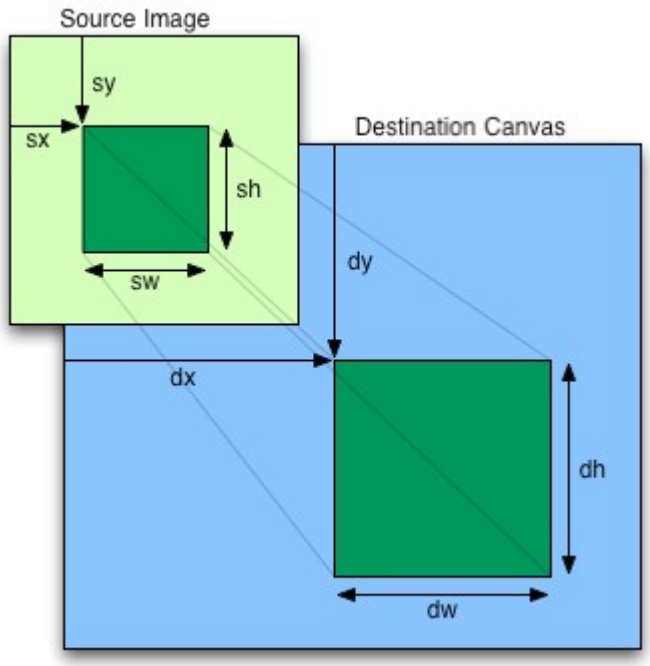
If the *image* argument is an `HTMLImageElement` object whose `complete` attribute is false, then the implementation must raise an `INVALID_STATE_ERR` exception.

The source rectangle is the rectangle whose corners are the four points (*sx*, *sy*), (*sx*+*sw*, *sy*), (*sx*+*sw*, *sy*+*sh*), (*sx*, *sy*+*sh*).

If the source rectangle is not entirely within the source image, or if one of the *sw* or *sh* arguments is zero, the implementation must raise an `INDEX_SIZE_ERR` exception.

The destination rectangle is the rectangle whose corners are the four points (*dx*, *dy*), (*dx*+*dw*, *dy*), (*dx*+*dw*, *dy*+*dh*), (*dx*, *dy*+*dh*).

When `drawImage()` is invoked, the region of the image specified by the source rectangle must be painted on the region of the canvas specified by the destination rectangle, after applying the current transformation matrix to the points of the destination rectangle.



Note: When a canvas is drawn onto itself, the drawing model requires the source to be copied before the image is drawn back onto the canvas, so it is possible to copy parts of a canvas onto overlapping parts of itself.

When the `drawImage()` method is passed, as its *image* argument, an animated image, the poster frame of the animation, or the first frame of the animation if there is no poster frame, must be used.

Images are painted without affecting the current path, and are subject to shadow effects, global alpha, the clipping region, and global composition operators.

4.7.11.1.11. Pixel manipulation

The **`createImageData(sw, sh)`** method must return an `ImageData` object representing a rectangle with a width in CSS pixels equal to the absolute magnitude of *sw* and a height in CSS pixels equal to the absolute magnitude of *sh*, filled with transparent black.

The **`getImageData(sx, sy, sw, sh)`** method must return an `ImageData` object representing the underlying pixel data for the area of the canvas denoted by the rectangle whose corners are the four points (*sx*, *sy*), (*sx*+*sw*, *sy*), (*sx*+*sw*, *sy*+*sh*), (*sx*, *sy*+*sh*), in canvas coordinate space units. Pixels outside the canvas must be returned as transparent black. Pixels must be returned as non-premultiplied alpha values.

If any of the arguments to `createImageData()` or `getImageData()` are infinite or NaN, the method must instead raise a `NOT_SUPPORTED_ERR` exception. If either the *sw* or *sh* arguments are zero, the method must instead raise an `INDEX_SIZE_ERR` exception.

`ImageData` objects must be initialized so that their **`width`** attribute is set to *w*, the number of physical device pixels per row in the image data, their **`height`** attribute is set to *h*, the number of rows in the image data, and their **`data`** attribute is initialized to a `CanvasPixelArray` object holding the image data. At least one pixel's worth of image data must be returned.

The `CanvasPixelArray` object provides ordered, indexed access to the color components of each pixel of the image data. The data must be represented in left-to-right order, row by row top to bottom, starting with the top left, with each pixel's red, green, blue, and alpha components being given in that order for each pixel. Each component of each device pixel represented in this array must be in the range 0..255, representing the 8 bit value for that component. The components must be assigned consecutive indices starting with 0 for the top left pixel's red component.

The `CanvasPixelArray` object thus represents *h*×*w*×4 integers. The **`length`** attribute of a `CanvasPixelArray` object must return this number.

The **`XXX5(index)`** method must return the value of the *index*th component in the array.

The **`XXX6(index, value)`** method must set the value of the *index*th component in the array to *value*. JS `undefined` values must be converted to zero. Other values must first be converted to numbers using JavaScript's ToNumber algorithm, and if the result is a NaN value, then the value be must converted to zero. If the result is less than 0, it must be clamped to zero. If the result is more than 255, it must be clamped to 255. If the number is not an integer, it must be rounded to the nearest integer using the IEEE 754r *convertToIntegerTiesToEven* rounding mode. [ECMA262] [IEEE754R]

> ***Note: The width and height (w and h) might be different from the sw and sh arguments to the above methods, e.g. if the canvas is backed by a high-resolution bitmap, or if the sw and sh arguments are negative.***

The **`putImageData(imagedata, dx, dy)`** and **`putImageData(imagedata, dx, dy, dirtyX, dirtyY, dirtyWidth, dirtyHeight)`** methods write data from `ImageData` structures back to the canvas.

If any of the arguments to the method are infinite or NaN, the method must raise an `NOT_SUPPORTED_ERR` exception.

If the first argument to the method is null or not an `ImageData` object then the `putImageData()` method must raise a `TYPE_MISMATCH_ERR` exception.

When the last four arguments are omitted, they must be assumed to have the values 0, 0, the `width` member of the *imagedata* structure, and the `height` member of the *imagedata* structure, respectively.

When invoked with arguments that do not, per the last few paragraphs, cause an exception to be raised, the `putImageData()` method must act as follows:

1. Let dx_{device} be the x-coordinate of the device pixel in the underlying pixel data of the canvas corresponding to the *dx* coordinate in the canvas coordinate space.

   Let dy_{device} be the y-coordinate of the device pixel in the underlying pixel data of the canvas corresponding to the *dy* coordinate in the canvas coordinate space.

2. If *dirtyWidth* is negative, let *dirtyX* be *dirtyX*+*dirtyWidth*, and let *dirtyWidth* be equal to the absolute magnitude of *dirtyWidth*.

   If *dirtyHeight* is negative, let *dirtyY* be *dirtyY*+*dirtyHeight*, and let *dirtyHeight* be equal to the absolute magnitude of *dirtyHeight*.

3. If *dirtyX* is negative, let *dirtyWidth* be *dirtyWidth*+*dirtyX*, and let *dirtyX* be zero.

If *dirtyY* is negative, let *dirtyHeight* be *dirtyHeight*+*dirtyY*, and let *dirtyY* be zero.

4. If *dirtyX*+*dirtyWidth* is greater than the `width` attribute of the *imagedata* argument, let *dirtyWidth* be the value of that `width` attribute, minus the value of *dirtyX*.

   If *dirtyY*+*dirtyHeight* is greater than the `height` attribute of the *imagedata* argument, let *dirtyHeight* be the value of that `height` attribute, minus the value of *dirtyY*.

5. If, after those changes, either *dirtyWidth* or *dirtyHeight* is negative or zero, stop these steps without affecting the canvas.

6. Otherwise, for all integer values of *x* and *y* where *dirtyX* ≤ *x* < *dirtyX*+*dirtyWidth* and *dirtyY* ≤ *y* < *dirtyY*+*dirtyHeight*, copy the four channels of the pixel with coordinate (*x*, *y*) in the *imagedata* data structure to the pixel with coordinate (dx_{device}+*x*, dy_{device}+*y*) in the underlying pixel data of the canvas.

The handling of pixel rounding when the specified coordinates do not exactly map to the device coordinate space is not defined by this specification, except that the following must result in no visible changes to the rendering:

```
context.putImageData(context.getImageData(x, y, w, h), x, y);
```

...for any value of *x*, *y*, *w*, and *h*, and the following two calls:

```
context.createImageData(w, h);
context.getImageData(0, 0, w, h);
```

...must return `ImageData` objects with the same dimensions, for any value of *w* and *h*. In other words, while user agents may round the arguments of these methods so that they map to device pixel boundaries, any rounding performed must be performed consistently for all of the `createImageData()`, `getImageData()` and `putImageData()` operations.

The current path, transformation matrix, shadow attributes, global alpha, the clipping region, and global composition operator must not affect the `getImageData()` and `putImageData()` methods.

> The data returned by `getImageData()` is at the resolution of the canvas backing store, which is likely to not be one device pixel to each CSS pixel if the display used is a high resolution display.
>
> In the following example, the script generates an `ImageData` object so that it can draw onto it.
>
> ```
> // canvas is a reference to a <canvas> element
> var context = canvas.getContext('2d');
>
> // create a blank slate
> var data = context.createImageData(canvas.width, canvas.height);
>
> // create some plasma
> FillPlasma(data, 'green'); // green plasma
>
> // add a cloud to the plasma
> AddCloud(data, data.width/2, data.height/2); // put a cloud in the middle
>
> // paint the plasma+cloud on the canvas
> ```

```
context.putImageData(data, 0, 0);

// support methods
function FillPlasma(data, color) { ... }
function AddCloud(data, x, y) { ... }
```

Here is an example of using `getImageData()` and `putImageData()` to implement an edge detection filter.

```
<!DOCTYPE HTML>
<html>
 <head>
  <title>Edge detection demo</title>
  <script>
   var image = new Image();
   function init() {
     image.onload = demo;
     image.src = "image.jpeg";
   }
   function demo() {
     var canvas = document.getElementsByTagName('canvas')[0];
     var context = canvas.getContext('2d');

     // draw the image onto the canvas
     context.drawImage(image, 0, 0);

     // get the image data to manipulate
     var input = context.getImageData(0, 0, canvas.width, canvas.height);

     // get an empty slate to put the data into
     var output = context.crateImageData(canvas.width, canvas.height);

     // alias some variables for convenience
     // notice that we are using input.width and input.height here
     // as they might not be the same as canvas.width and canvas.height
     // (in particular, they might be different on high-res displays)
     var w = input.width, h = input.height;
     var inputData = input.data;
     var outputData = output.data;

     // edge detection
     for (var y = 1; y < h-1; y += 1) {
       for (var x = 1; x < w-1; x += 1) {
         for (var c = 0; c < 3; c += 1) {
           var i = (y*w + x)*4 + c;
           outputData[i] = 127 + -inputData[i - w*4 - 4] -   inputData[i -
w*4] - inputData[i - w*4 + 4] +
                                 -inputData[i - 4]       + 8*inputData[i]
- inputData[i + 4] +
                                 -inputData[i + w*4 - 4] -   inputData[i +
```

```
    w*4] - inputData[i + w*4 + 4];
              }
              outputData[(y*w + x)*4 + 3] = 255; // alpha
            }
          }

          // put the image data back after manipulation
          context.putImageData(output, 0, 0);
        }
      </script>
     </head>
     <body onload="init()">
      <canvas></canvas>
     </body>
    </html>
```

4.7.11.1.12. Drawing model

When a shape or image is painted, user agents must follow these steps, in the order given (or act as if they do):

1. Render the shape or image, creating image *A*, as described in the previous sections. For shapes, the current fill, stroke, and line styles must be honored, and the stroke must itself also be subjected to the current transformation matrix.
2. Render the shadow from image *A*, using the current shadow styles, creating image *B*.
3. Multiply the alpha component of every pixel in *B* by `globalAlpha`.
4. Within the clipping region, composite *B* over the current canvas bitmap using the current composition operator.
5. Multiply the alpha component of every pixel in *A* by `globalAlpha`.
6. Within the clipping region, composite *A* over the current canvas bitmap using the current composition operator.

4.7.11.2. Color spaces and color correction

The `canvas` APIs must perform color correction at only two points: when rendering images with their own gamma correction and color space information onto the canvas, to convert the image to the color space used by the canvas (e.g. using the `drawImage()` method with an `HTMLImageElement` object), and when rendering the actual canvas bitmap to the output device.

> ***Note: Thus, in the 2D context, colors used to draw shapes onto the canvas will exactly match colors obtained through the `getImageData()` method.***

The `toDataURL()` method must not include color space information in the resource returned. Where the output format allows it, the color of pixels in resources created by `toDataURL()` must match those returned by the `getImageData()` method.

In user agents that support CSS, the color space used by a `canvas` element must match the color space used for processing any colors for that element in CSS.

The gamma correction and color space information of images must be handled in such a way that an image rendered directly using an `img` element would use the same colors as one painted on a `canvas` element that is then itself rendered. Furthermore, the rendering of images that have no color correction information (such as those returned by the `toDataURL()` method) must be rendered with no color correction.

Note: Thus, in the 2D context, calling the `drawImage()` method to render the output of the `toDataURL()` method to the canvas, given the appropriate dimensions, has no visible effect.

4.7.11.3. Security with `canvas` elements

Information leakage can occur if scripts from one origin can access information (e.g. read pixels) from images from another origin (one that isn't the same).

To mitigate this, `canvas` elements are defined to have a flag indicating whether they are *origin-clean*. All `canvas` elements must start with their *origin-clean* set to true. The flag must be set to false if any of the following actions occur:

- The element's 2D context's `drawImage()` method is called with an `HTMLImageElement` whose origin is not the same as that of the `Document` object that owns the `canvas` element.
- The element's 2D context's `drawImage()` method is called with an `HTMLCanvasElement` whose *origin-clean* flag is false.
- The element's 2D context's `fillStyle` attribute is set to a `CanvasPattern` object that was created from an `HTMLImageElement` whose origin was not the same as that of the `Document` object that owns the `canvas` element when the pattern was created.
- The element's 2D context's `fillStyle` attribute is set to a `CanvasPattern` object that was created from an `HTMLCanvasElement` whose *origin-clean* flag was false when the pattern was created.
- The element's 2D context's `strokeStyle` attribute is set to a `CanvasPattern` object that was created from an `HTMLImageElement` whose origin was not the same as that of the `Document` object that owns the `canvas` element when the pattern was created.
- The element's 2D context's `strokeStyle` attribute is set to a `CanvasPattern` object that was created from an `HTMLCanvasElement` whose *origin-clean* flag was false when the pattern was created.

Whenever the `toDataURL()` method of a `canvas` element whose *origin-clean* flag is set to false is called, the method must raise a security exception.

Whenever the `getImageData()` method of the 2D context of a `canvas` element whose *origin-clean* flag is set to false is called with otherwise correct arguments, the method must raise a security exception.

Note: Even resetting the canvas state by changing its `width` or `height` attributes doesn't reset the origin-clean flag.

4.7.12 The `map` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Element-specific attributes:

`name`

DOM interface:

```
interface HTMLMapElement : HTMLElement {
           attribute DOMString name;
  readonly attribute HTMLCollection areas;
  readonly attribute HTMLCollection images;
};
```

The `map` element, in conjunction with any `area` element descendants, defines an image map.

The **`name`** attribute gives the map a name so that it can be referenced. The attribute must be present and must have a non-empty value. Whitespace is significant in this attribute's value. If the `id` attribute is also specified, both attributes must have the same value.

The **`areas`** attribute must return an `HTMLCollection` rooted at the `map` element, whose filter matches only `area` elements.

The **`images`** attribute must return an `HTMLCollection` rooted at the `Document` node, whose filter matches only `img` and `object` elements that are associated with this `map` element according to the image map processing model.

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

4.7.13 The `area` element

Categories

Phrasing content.

Contexts in which this element may be used:

Where phrasing content is expected, but only if there is a `map` element ancestor.

Content model:

Empty.

Element-specific attributes:

`alt`

coords
shape
href
target
ping
rel
media
hreflang
type

DOM interface:

```
interface HTMLAreaElement : HTMLElement {
           attribute DOMString alt;
           attribute DOMString coords;
           attribute DOMString shape;
           attribute DOMString href;
           attribute DOMString target;
           attribute DOMString ping;
           attribute DOMString rel;
  readonly attribute DOMTokenList relList;
           attribute DOMString media;
           attribute DOMString hreflang;
           attribute DOMString type;
};
```

The `area` element represents either a hyperlink with some text and a corresponding area on an image map, or a dead area on an image map.

If the `area` element has an `href` attribute, then the `area` element represents a hyperlink; the **`alt`** attribute, which must then be present, specifies the text.

However, if the `area` element has no `href` attribute, then the area represented by the element cannot be selected, and the `alt` attribute must be omitted.

In both cases, the `shape` and `coords` attributes specify the area.

The **`shape`** attribute is an enumerated attribute. The following table lists the keywords defined for this attribute. The states given in the first cell of the rows with keywords give the states to which those keywords map. Some of the keywords are non-conforming, as noted in the last column.

State	Keywords	Notes
Circle state	`circ`	Non-conforming
	`circle`	
Default state	`default`	
Polygon state	`poly`	
	`polygon`	Non-conforming
Rectangle state	`rect`	
	`rectangle`	Non-conforming

The attribute may be omitted. The *missing value default* is the rectangle state.

The **coords** attribute must, if specified, contain a valid list of integers. This attribute gives the coordinates for the shape described by the shape attribute. The processing for this attribute is described as part of the image map processing model.

In the circle state, area elements must have a coords attribute present, with three integers, the last of which must be non-negative. The first integer must be the distance in CSS pixels from the left edge of the image to the center of the circle, the second integer must be the distance in CSS pixels from the top edge of the image to the center of the circle, and the third integer must be the radius of the circle, again in CSS pixels.

In the default state state, area elements must not have a coords attribute.

In the polygon state, area elements must have a coords attribute with at least six integers, and the number of integers must be even. Each pair of integers must represent a coordinate given as the distances from the left and the top of the image in CSS pixels respectively, and all the coordinates together must represent the points of the polygon, in order.

In the rectangle state, area elements must have a coords attribute with exactly four integers, the first of which must be less than the third, and the second of which must be less than the fourth. The four points must represent, respectively, the distance from the left edge of the image to the top left side of the rectangle, the distance from the top edge to the top side, the distance from the left edge to the right side, and the distance from the top edge to the bottom side, all in CSS pixels.

When user agents allow users to follow hyperlinks created using the area element, as described in the next section, the href, target and ping attributes decide how the link is followed. The rel, media, hreflang, and type attributes may be used to indicate to the user the likely nature of the target resource before the user follows the link.

The target, ping, rel, media, hreflang, and type attributes must be omitted if the href attribute is not present.

The activation behavior of area elements is to run the following steps:

1. If the DOMActivate event in question is not trusted (i.e. a click() method call was the reason for the event being dispatched), and the area element's target attribute is ... then raise an INVALID_ACCESS_ERR exception.
2. Otherwise, the user agent must follow the hyperlink defined by the area element, if any.

Note: One way that a user agent can enable users to follow hyperlinks is by allowing area elements to be clicked, or focussed and activated by the keyboard. This will cause the aforementioned activation behavior to be invoked.

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

The DOM attribute **shape** must reflect the shape content attribute, limited to only known values.

The DOM attribute **relList** must reflect the rel content attribute.

4.7.14 Image maps

An **image map** allows geometric areas on an image to be associated with hyperlinks.

An image, in the form of an `img` element or an `object` element representing an image, may be associated with an image map (in the form of a `map` element) by specifying a **`usemap`** attribute on the `img` or `object` element. The `usemap` attribute, if specified, must be a valid hash-name reference to a `map` element.

If an `img` element or an `object` element representing an image has a `usemap` attribute specified, user agents must process it as follows:

1. First, rules for parsing a hash-name reference to a `map` element must be followed. This will return either an element (the *map*) or null.
2. If that returned null, then abort these steps. The image is not associated with an image map after all.
3. Otherwise, the user agent must collect all the `area` elements that are descendants of the *map*. Let those be the *areas*.

Having obtained the list of `area` elements that form the image map (the *areas*), interactive user agents must process the list in one of two ways.

If the user agent intends to show the text that the `img` element represents, then it must use the following steps.

> ***Note: In user agents that do not support images, or that have images disabled, `object` elements cannot represent images, and thus this section never applies (the fallback content is shown instead). The following steps therefore only apply to `img` elements.***

1. Remove all the `area` elements in *areas* that have no `href` attribute.
2. Remove all the `area` elements in *areas* that have no `alt` attribute, or whose `alt` attribute's value is the empty string, *if* there is another `area` element in *areas* with the same value in the `href` attribute and with a non-empty `alt` attribute.
3. Each remaining `area` element in *areas* represents a hyperlink. Those hyperlinks should all be made available to the user in a manner associated with the text of the `img`.

   In this context, user agents may represent `area` and `img` elements with no specified `alt` attributes, or whose `alt` attributes are the empty string or some other non-visible text, in a user-agent-defined fashion intended to indicate the lack of suitable author-provided text.

If the user agent intends to show the image and allow interaction with the image to select hyperlinks, then the image must be associated with a set of layered shapes, taken from the `area` elements in *areas*, in reverse tree order (so the last specified `area` element in the *map* is the bottom-most shape, and the first element in the *map*, in tree order, is the top-most shape).

Each `area` element in *areas* must be processed as follows to obtain a shape to layer onto the image:

1. Find the state that the element's `shape` attribute represents.
2. Use the rules for parsing a list of integers to parse the element's `coords` attribute, if it is present, and

let the result be the *coords* list. If the attribute is absent, let the *coords* list be the empty list.

3. If the number of items in the *coords* list is less than the minimum number given for the `area` element's current state, as per the following table, then the shape is empty; abort these steps.

State	Minimum number of items
Circle state	3
Default state	0
Polygon state	6
Rectangle state	4

4. Check for excess items in the *coords* list as per the entry in the following list corresponding to the `shape` attribute's state:

   ↪ **Circle state**

   Drop any items in the list beyond the third.

   ↪ **Default state**

   Drop all items in the list.

   ↪ **Polygon state**

   Drop the last item if there's an odd number of items.

   ↪ **Rectangle state**

   Drop any items in the list beyond the fourth.

5. If the `shape` attribute represents the rectangle state, and the first number in the list is numerically less than the third number in the list, then swap those two numbers around.

6. If the `shape` attribute represents the rectangle state, and the second number in the list is numerically less than the fourth number in the list, then swap those two numbers around.

7. If the `shape` attribute represents the circle state, and the third number in the list is less than or equal to zero, then the shape is empty; abort these steps.

8. Now, the shape represented by the element is the one described for the entry in the list below corresponding to the state of the `shape` attribute:

   ↪ **Circle state**

   Let *x* be the first number in *coords*, *y* be the second number, and *r* be the third number.

   The shape is a circle whose center is *x* CSS pixels from the left edge of the image and *x* CSS pixels from the top edge of the image, and whose radius is *r* pixels.

   ↪ **Default state**

   The shape is a rectangle that exactly covers the entire image.

   ↪ **Polygon state**

   Let x_i be the (2*i*)th entry in *coords*, and y_i be the (2*i*+1)th entry in *coords* (the first entry in *coords* being the one with index 0).

   Let *the coordinates* be (x_i, y_i), interpreted in CSS pixels measured from the top left of the image, for all integer values of *i* from 0 to (*N*/2)-1, where *N* is the number of items in *coords*.

The shape is a polygon whose vertices are given by *the coordinates*, and whose interior is established using the even-odd rule. [GRAPHICS]

↪ **Rectangle state**

Let *x1* be the first number in *coords*, *y1* be the second number, *x2* be the third number, and *y2* be the fourth number.

The shape is a rectangle whose top-left corner is given by the coordinate (*x1*, *y1*) and whose bottom right corner is given by the coordinate (*x2*, *y2*), those coordinates being interpreted as CSS pixels from the top left corner of the image.

For historical reasons, the coordinates must be interpreted relative to the *displayed* image, even if it stretched using CSS or the image element's `width` and `height` attributes.

Mouse clicks on an image associated with a set of layered shapes per the above algorithm must be dispatched to the top-most shape covering the point that the pointing device indicated (if any), and then, must be dispatched again (with a new `Event` object) to the image element itself. User agents may also allow individual `area` elements representing hyperlinks to be selected and activated (e.g. using a keyboard); events from this are not also propagated to the image.

Note: Because a `map` element (and its `area` elements) can be associated with multiple `img` and `object` elements, it is possible for an `area` element to correspond to multiple focusable areas of the document.

Image maps are *live*; if the DOM is mutated, then the user agent must act as if it had rerun the algorithms for image maps.

4.7.15 MathML

The `math` element from the MathML namespace falls into the embedded content category for the purposes of the content models in this specification.

User agents must handle text other than inter-element whitespace found in MathML elements whose content models do not allow raw text by pretending for the purposes of MathML content models, layout, and rendering that that text is actually wrapped in an `mtext` element in the MathML namespace. (Such text is not, however, conforming.)

User agents must act as if any MathML element whose contents does not match the element's content model was replaced, for the purposes of MathML layout and rendering, by an `merror` element in the MathML namespace containing some appropriate error message.

To enable authors to use MathML tools that only accept MathML in its XML form, interactive HTML user agents are encouraged to provide a way to export any MathML fragment as a namespace-well-formed XML fragment.

4.7.16 SVG

The `svg` element from the SVG namespace falls into the embedded content category for the purposes of the content models in this specification.

To enable authors to use SVG tools that only accept SVG in its XML form, interactive HTML user agents are encouraged to provide a way to export any SVG fragment as a namespace-well-formed XML fragment.

4.7.17 Dimension attributes

The **`width`** and **`height`** attributes on `img`, `embed`, `object`, and `video` elements may be specified to give the dimensions of the visual content of the element (the width and height respectively, relative to the nominal direction of the output medium), in CSS pixels. The attributes, if specified, must have values that are valid positive non-zero integers.

The specified dimensions given may differ from the dimensions specified in the resource itself, since the resource may have a resolution that differs from the CSS pixel resolution. (On screens, CSS pixels have a resolution of 96ppi, but in general the CSS pixel resolution depends on the reading distance.) If both attributes are specified, then the ratio of the specified width to the specified height must be the same as the ratio of the intrinsic width to the intrinsic height in the resource, or alternatively, in the case of the `video` element, the same as the adjusted ratio. The two attributes must be omitted if the resource in question does not have both an intrinsic width and an intrinsic height.

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. The user agent requirements for processing the values obtained from parsing these attributes are described in the rendering section. If one of these attributes, when parsing, returns no value, it must be treated, for the purposes of those requirements, as if it was not specified.

The **`width`** and **`height`** DOM attributes on the `embed`, `object`, and `video` elements must reflect the content attributes of the same name.

4.8 Tabular data

4.8.1 Introduction

This section is non-normative.

...examples, how to write tables accessibly, a brief mention of the table model, etc...

4.8.2 The `table` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

In this order: optionally a `caption` element, followed by either zero or more `colgroup` elements, followed optionally by a `thead` element, followed optionally by a `tfoot` element, followed by either zero or more `tbody` elements *or* one or more `tr` elements, followed optionally by a `tfoot` element (but there can only be one `tfoot` element child in total).

Element-specific attributes:

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;
  HTMLElement createTBody();
  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).

we need some editorial text on how layout tables are bad practice and non-conforming

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. ***(Only expressible in the XML serialization.)***
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, if any, or null otherwise. On setting, if the new value is a `caption` element, the first `caption` element child of the `table` element, if any, must be removed, and the new value must be inserted as the first node of the `table` element. If the new value is not a `caption` element, then a `HIERARCHY_REQUEST_ERR` DOM

exception must be raised instead.

The **`createCaption()`** method must return the first `caption` element child of the `table` element, if any; otherwise a new `caption` element must be created, inserted as the first node of the `table` element, and then returned.

The **`deleteCaption()`** method must remove the first `caption` element child of the `table` element, if any.

The **`tHead`** DOM attribute must return, on getting, the first `thead` element child of the `table` element, if any, or null otherwise. On setting, if the new value is a `thead` element, the first `thead` element child of the `table` element, if any, must be removed, and the new value must be inserted immediately before the first element in the `table` element that is neither a `caption` element nor a `colgroup` element, if any, or at the end of the table otherwise. If the new value is not a `thead` element, then a `HIERARCHY_REQUEST_ERR` DOM exception must be raised instead.

The **`createTHead()`** method must return the first `thead` element child of the `table` element, if any; otherwise a new `thead` element must be created and inserted immediately before the first element in the `table` element that is neither a `caption` element nor a `colgroup` element, if any, or at the end of the table otherwise, and then that new element must be returned.

The **`deleteTHead()`** method must remove the first `thead` element child of the `table` element, if any.

The **`tFoot`** DOM attribute must return, on getting, the first `tfoot` element child of the `table` element, if any, or null otherwise. On setting, if the new value is a `tfoot` element, the first `tfoot` element child of the `table` element, if any, must be removed, and the new value must be inserted immediately before the first element in the `table` element that is neither a `caption` element, a `colgroup` element, nor a `thead` element, if any, or at the end of the table if there are no such elements. If the new value is not a `tfoot` element, then a `HIERARCHY_REQUEST_ERR` DOM exception must be raised instead.

The **`createTFoot()`** method must return the first `tfoot` element child of the `table` element, if any; otherwise a new `tfoot` element must be created and inserted immediately before the first element in the `table` element that is neither a `caption` element, a `colgroup` element, nor a `thead` element, if any, or at the end of the table if there are no such elements, and then that new element must be returned.

The **`deleteTFoot()`** method must remove the first `tfoot` element child of the `table` element, if any.

The **`tBodies`** attribute must return an `HTMLCollection` rooted at the `table` node, whose filter matches only `tbody` elements that are children of the `table` element.

The **`createTBody()`** method must create a new `tbody` element, insert it immediately after the last `tbody` element in the `table` element, if any, or at the end of the `table` element if the `table` element has no `tbody` element children, and then must return the new `tbody` element.

The **`rows`** attribute must return an `HTMLCollection` rooted at the `table` node, whose filter matches only `tr` elements that are either children of the `table` element, or children of `thead`, `tbody`, or `tfoot` elements that are themselves children of the `table` element. The elements in the collection must be ordered such that those elements whose parent is a `thead` are included first, in tree order, followed by those elements whose parent is either a `table` or `tbody` element, again in tree order, followed finally by those elements whose parent is a `tfoot` element, still in tree order.

The behavior of the **`insertRow(`*`index`*`)`** method depends on the state of the table. When it is called, the

method must act as required by the first item in the following list of conditions that describes the state of the table and the *index* argument:

↪ **If *index* is less than −1 or greater than the number of elements in `rows` collection:**

The method must raise an `INDEX_SIZE_ERR` exception.

↪ **If the `rows` collection has zero elements in it, and the `table` has no `tbody` elements in it:**

The method must create a `tbody` element, then create a `tr` element, then append the `tr` element to the `tbody` element, then append the `tbody` element to the `table` element, and finally return the `tr` element.

↪ **If the `rows` collection has zero elements in it:**

The method must create a `tr` element, append it to the last `tbody` element in the table, and return the `tr` element.

↪ **If *index* is equal to −1 or equal to the number of items in `rows` collection:**

The method must create a `tr` element, and append it to the parent of the last `tr` element in the `rows` collection. Then, the newly created `tr` element must be returned.

↪ **Otherwise:**

The method must create a `tr` element, insert it immediately before the *index*th `tr` element in the `rows` collection, in the same parent, and finally must return the newly created `tr` element.

When the **`deleteRow(`*`index`*`)`** method is called, the user agent must run the following steps:

1. If *index* is equal to −1, then *index* must be set to the number if items in the `rows` collection, minus one.
2. Now, if *index* is less than zero, or greater than or equal to the number of elements in the `rows` collection, the method must instead raise an `INDEX_SIZE_ERR` exception, and these steps must be aborted.
3. Otherwise, the method must remove the *index*th element in the `rows` collection from its parent.

4.8.3 The `caption` element

Categories

None.

Contexts in which this element may be used:

As the first element child of a `table` element.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `caption` element represents the title of the `table` that is its parent, if it has a parent and that is a `table`

element.

The `caption` element takes part in the table model.

4.8.4 The `colgroup` element

Categories

None.

Contexts in which this element may be used:

As a child of a `table` element, after any `caption` elements and before any `thead`, `tbody`, `tfoot`, and `tr` elements.

Content model:

Zero or more `col` elements.

Element-specific attributes:

`span`

DOM interface:

```
interface HTMLTableColElement : HTMLElement {
           attribute unsigned long span;
};
```

The `colgroup` element represents a group of one or more columns in the `table` that is its parent, if it has a parent and that is a `table` element.

If the `colgroup` element contains no `col` elements, then the element may have a **`span`** content attribute specified, whose value must be a valid non-negative integer greater than zero.

The `colgroup` element and its `span` attribute take part in the table model.

The **`span`** DOM attribute must reflect the content attribute of the same name. The value must be limited to only positive non-zero numbers.

4.8.5 The `col` element

Categories

None.

Contexts in which this element may be used:

As a child of a `colgroup` element that doesn't have a `span` attribute.

Content model:

Empty.

Element-specific attributes:

`span`

DOM interface:

`HTMLTableColElement`, same as for `colgroup` elements. This interface defines one member, `span`.

If a `col` element has a parent and that is a `colgroup` element that itself has a parent that is a `table` element, then the `col` element represents one or more columns in the column group represented by that `colgroup`.

The element may have a **`span`** content attribute specified, whose value must be a valid non-negative integer greater than zero.

The `col` element and its `span` attribute take part in the table model.

The **`span`** DOM attribute must reflect the content attribute of the same name. The value must be limited to only positive non-zero numbers.

4.8.6 The `tbody` element

Categories

None.

Contexts in which this element may be used:

As a child of a `table` element, after any `caption`, `colgroup`, and `thead` elements, but only if there are no `tr` elements that are children of the `table` element.

Content model:

Zero or more `tr` elements

Element-specific attributes:

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 `tbody` 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.

4.8.7 The `thead` element

Categories

None.

Contexts in which this element may be used:

As a child of a `table` element, after any `caption`, and `colgroup` elements and before any `tbody`, `tfoot`, and `tr` elements, but only if there are no other `thead` elements that are children of the `table` element.

Content model:

Zero or more `tr` elements

Element-specific attributes:

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.

4.8.8 The `tfoot` element

Categories

None.

Contexts in which this element may be used:

As a child of a `table` element, after any `caption`, `colgroup`, and `thead` elements and before any `tbody` and `tr` elements, but only if there are no other `tfoot` elements that are children of the `table` element.

As a child of a `table` element, after any `caption`, `colgroup`, `thead`, `tbody`, and `tr` elements, but

only if there are no other `tfoot` elements that are children of the `table` element.

Content model:

Zero or more `tr` elements

Element-specific attributes:

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 `tfoot` element has a parent and it is a `table`.

The `tfoot` element takes part in the table model.

4.8.9 The `tr` element

Categories

None.

Contexts in which this element may be used:

As a child of a `thead` element.
As a child of a `tbody` element.
As a child of a `tfoot` element.
As a child of a `table` element, after any `caption`, `colgroup`, and `thead` elements, but only if there are no `tbody` elements that are children of the `table` element.

Content model:

Zero or more `td` or `th` elements

Element-specific attributes:

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`** attribute must, if the element has a parent `table` element, or a parent `tbody`, `thead`, or

`tfoot` element and a *grandparent* `table` element, return the index of the `tr` element in that `table` element's `rows` collection. If there is no such `table` element, then the attribute must return −1.

The **`sectionRowIndex`** attribute must, if the element has a parent `table`, `tbody`, `thead`, or `tfoot` element, return the index of the `tr` element in the parent element's `rows` collection (for tables, that's the `rows` collection; for table sections, that's the `rows` collection). If there is no such parent element, then the attribute must return −1.

The **`cells`** attribute must return an `HTMLCollection` rooted at the `tr` element, whose filter matches only `td` and `th` elements that are children of the `tr` element.

The **`insertCell(`*`index`*`)`** method must act as follows:

If *index* is less than −1 or greater than the number of elements in the `cells` collection, the method must raise an `INDEX_SIZE_ERR` exception.

If *index* is equal to −1 or equal to the number of items in `cells` collection, the method must create a `td` element, append it to the `tr` element, and return the newly created `td` element.

Otherwise, the method must create a `td` element, insert it as a child of the `tr` element, immediately before the *index*th `td` or `th` element in the `cells` collection, and finally must return the newly created `td` element.

The **`deleteCell(`*`index`*`)`** method must remove the *index*th element in the `cells` collection from its parent. If *index* is less than zero or greater than or equal to the number of elements in the `cells` collection, the method must instead raise an `INDEX_SIZE_ERR` exception.

4.8.10 The `td` element

Categories

Sectioning root.

Contexts in which this element may be used:

As a child of a `tr` element.

Content model:

Flow content.

Element-specific attributes:

`colspan`
`rowspan`
`headers`

DOM interface:

```
interface HTMLTableDataCellElement : HTMLTableCellElement {
           attribute DOMString headers;
};
```

The `td` element represents a data cell in a table.

The `td` element may have a **`headers`** content attribute specified. The `headers` attribute, if specified, must contain a string consisting of an unordered set of unique space-separated tokens, each of which must have the value of an ID of a `th` element taking part in the same table as the `td` element (as defined by the table model).

The exact effect of the attribute is described in detail in the algorithm for assigning header cells to data cells, which user agents must apply to determine the relationships between data cells and header cells.

The `td` element and its `colspan` and `rowspan` attributes take part in the table model.

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

4.8.11 The `th` element

Categories

None.

Contexts in which this element may be used:

As a child of a `tr` element.

Content model:

Phrasing content.

Element-specific attributes:

`colspan`
`rowspan`
`scope`

DOM interface:

```
interface HTMLTableHeaderCellElement : HTMLTableCellElement {
           attribute DOMString scope;
};
```

The `th` element represents a header cell in a table.

The `th` element may have a **`scope`** content attribute specified. The `scope` attribute is an enumerated attribute with five states, four of which have explicit keywords:

The *`row`* keyword, which maps to the *row* state

The *row* state means the header cell applies to all the remaining cells in the row.

The *`col`* keyword, which maps to the *column* state

The *column* state means the header cell applies to all the remaining cells in the column.

The *`rowgroup`* keyword, which maps to the *row group* state

The *row group* state means the header cell applies to all the remaining cells in the row group.

The *`colgroup`* keyword, which maps to the *column group* state

The *column group* state means the header cell applies to all the remaining cells in the column group.

The *auto* state

The *auto* state makes the header cell apply to a set of cells selected based on context.

The `scope` attribute's *missing value default* is the *auto* state.

The exact effect of these values is described in detail in the algorithm for assigning header cells to data cells, which user agents must apply to determine the relationships between data cells and header cells.

The `th` element and its `colspan` and `rowspan` attributes take part in the table model.

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

4.8.12 Attributes common to `td` and `th` elements

The `td` and `th` elements may have a **`colspan`** content attribute specified, whose value must be a valid non-negative integer greater than zero.

The `td` and `th` elements may also have a **`rowspan`** content attribute specified, whose value must be a valid non-negative integer.

The `td` and `th` elements implement interfaces that inherit from the `HTMLTableCellElement` interface:

```
interface HTMLTableCellElement : HTMLElement {
           attribute long colSpan;
           attribute long rowSpan;
  readonly attribute long cellIndex;
};
```

The **`colSpan`** DOM attribute must reflect the content attribute of the same name. The value must be limited to only positive non-zero numbers.

The **`rowSpan`** DOM attribute must reflect the content attribute of the same name. Its default value, which must be used if parsing the attribute as a non-negative integer returns an error, is also 1.

The **`cellIndex`** DOM attribute must, if the element has a parent `tr` element, return the index of the cell's element in the parent element's `cells` collection. If there is no such parent element, then the attribute must return 0.

4.8.13 Processing model

The various table elements and their content attributes together define the **table model**.

A **table** consists of cells aligned on a two-dimensional grid of **slots** with coordinates (x, y). The grid is finite, and is either empty or has one or more slots. If the grid has one or more slots, then the x coordinates are always in the range $0 \le x < x_{width}$, and the y coordinates are always in the range $0 \le y < y_{height}$. If one or both of x_{width} and y_{height} are zero, then the table is empty (has no slots). Tables correspond to `table` elements.

A **cell** is a set of slots anchored at a slot ($cell_x$, $cell_y$), and with a particular *width* and *height* such that the cell covers all the slots with coordinates (x, y) where $cell_x \le x < cell_x + width$ and $cell_y \le y < cell_y + height$. Cells can either be *data cells* or *header cells*. Data cells correspond to `td` elements, and have zero or more associated

header cells. Header cells correspond to `th` elements.

A **row** is a complete set of slots from x=0 to x=x_{width}-1, for a particular value of y. Rows correspond to `tr` elements.

A **column** is a complete set of slots from y=0 to y=y_{height}-1, for a particular value of x. Columns can correspond to `col` elements, but in the absence of `col` elements are implied.

A **row group** is a set of rows anchored at a slot (0, $group_y$) with a particular *height* such that the row group covers all the slots with coordinates (x, y) where $0 \le x < x_{width}$ and $group_y \le y < group_y+height$. Row groups correspond to `tbody`, `thead`, and `tfoot` elements. Not every row is necessarily in a row group.

A **column group** is a set of columns anchored at a slot ($group_x$, 0) with a particular *width* such that the column group covers all the slots with coordinates (x, y) where $group_x \le x < group_x+width$ and $0 \le y < y_{height}$. Column groups correspond to `colgroup` elements. Not every column is necessarily in a column group.

Row groups cannot overlap each other. Similarly, column groups cannot overlap each other.

A cell cannot cover slots that are from two or more row groups. It is, however, possible for a cell to be in multiple column groups. All the slots that form part of one cell are part of zero or one row groups and zero or more column groups.

In addition to cells, columns, rows, row groups, and column groups, tables can have a `caption` element associated with them. This gives the table a heading, or legend.

A **table model error** is an error with the data represented by `table` elements and their descendants. Documents must not have table model errors.

4.8.13.1. Forming a table

To determine which elements correspond to which slots in a table associated with a `table` element, to determine the dimensions of the table (x_{width} and y_{height}), and to determine if there are any table model errors, user agents must use the following algorithm:

1. Let x_{width} be zero.
2. Let y_{height} be zero.
3. Let *pending `tfoot` elements* be a list of `tfoot` elements, initially empty.
4. Let *the table* be the table represented by the `table` element. The x_{width} and y_{height} variables give *the table*'s dimensions. *The table* is initially empty.
5. If the `table` element has no children elements, then return *the table* (which will be empty), and abort these steps.
6. Associate the first `caption` element child of the `table` element with *the table*. If there are no such children, then it has no associated `caption` element.
7. Let the *current element* be the first element child of the `table` element.

   If a step in this algorithm ever requires the *current element* to be **advanced to the next child of the `table`** when there is no such next child, then the user agent must jump to the step labeled *end*, near

the end of this algorithm.

8. While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:

   - `colgroup`
   - `thead`
   - `tbody`
   - `tfoot`
   - `tr`

9. If the *current element* is a `colgroup`, follow these substeps:

   1. *Column groups*: Process the *current element* according to the appropriate case below:

      ↪ If the *current element* has any `col` element children

      Follow these steps:

      1. Let x_{start} have the value of x_{width}.
      2. Let the *current column* be the first `col` element child of the `colgroup` element.
      3. *Columns*: If the *current column* `col` element has a `span` attribute, then parse its value using the rules for parsing non-negative integers.

         If the result of parsing the value is not an error or zero, then let *span* be that value.

         Otherwise, if the `col` element has no `span` attribute, or if trying to parse the attribute's value resulted in an error, then let *span* be 1.
      4. Increase x_{width} by *span*.
      5. Let the last *span* columns in *the table* correspond to the *current column* `col` element.
      6. If *current column* is not the last `col` element child of the `colgroup` element, then let the *current column* be the next `col` element child of the `colgroup` element, and return to the step labeled *columns*.
      7. Let all the last columns in *the table* from x=x_{start} to x=x_{width}-1 form a new column group, anchored at the slot (x_{start}, 0), with width x_{width}-x_{start}, corresponding to the `colgroup` element.

      ↪ If the *current element* has no `col` element children

      1. If the `colgroup` element has a `span` attribute, then parse its value using the rules for parsing non-negative integers.

         If the result of parsing the value is not an error or zero, then let *span* be that value.

         Otherwise, if the `colgroup` element has no `span` attribute, or if trying to parse

the attribute's value resulted in an error, then let *span* be 1.

2. Increase x_{width} by *span*.
3. Let the last *span* columns in *the table* form a new column group, anchored at the slot (x_{width}-*span*, 0), with width *span*, corresponding to the `colgroup` element.

2. Advance the *current element* to the next child of the `table`.
3. While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:
   - `colgroup`
   - `thead`
   - `tbody`
   - `tfoot`
   - `tr`
4. If the *current element* is a `colgroup` element, jump to the step labeled *column groups* above.

10. Let $y_{current}$ be zero.
11. Let the *list of downward-growing cells* be an empty list.
12. *Rows*: While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:
    - `thead`
    - `tbody`
    - `tfoot`
    - `tr`
13. If the *current element* is a `tr`, then run the algorithm for processing rows, advance the *current element* to the next child of the `table`, and return to the step labeled *rows*.
14. Run the algorithm for ending a row group.
15. If the *current element* is a `tfoot`, then add that element to the list of *pending `tfoot` elements*, advance the *current element* to the next child of the `table`, and return to the step labeled *rows*.
16. The *current element* is either a `thead` or a `tbody`.

    Run the algorithm for processing row groups.
17. Advance the *current element* to the next child of the `table`.
18. Return to the step labeled *rows*.
19. *End*: For each `tfoot` element in the list of *pending `tfoot` elements*, in tree order, run the algorithm for processing row groups.
20. If there exists a row or column in the table *the table* containing only slots that do not have a cell anchored to them, then this is a table model error.

21. Return *the table*.

The **algorithm for processing row groups**, which is invoked by the set of steps above for processing `thead`, `tbody`, and `tfoot` elements, is:

1. Let y_{start} have the value of y_{height}.
2. For each `tr` element that is a child of the element being processed, in tree order, run the algorithm for processing rows.
3. If $y_{height} > y_{start}$, then let all the last rows in *the table* from y=y_{start} to y=y_{height}-1 form a new row group, anchored at the slot with coordinate (0, y_{start}), with height y_{height}-y_{start}, corresponding to the *current element*.
4. Run the algorithm for ending a row group.

The **algorithm for ending a row group**, which is invoked by the set of steps above when starting and ending a block of rows, is:

1. While $y_{current}$ is less than y_{height}, follow these steps:
    1. Run the algorithm for growing downward-growing cells.
    2. Increase $y_{current}$ by 1.
2. Empty the *list of downward-growing cells*.

The **algorithm for processing rows**, which is invoked by the set of steps above for processing `tr` elements, is:

1. If y_{height} is equal to $y_{current}$, then increase y_{height} by 1. ($y_{current}$ is never *greater* than y_{height}.)
2. Let $x_{current}$ be 0.
3. Let *current cell* be the first `td` or `th` element in the `tr` element being processed.
4. Run the algorithm for growing downward-growing cells.
5. *Cells*: While $x_{current}$ is less than x_{width} and the slot with coordinate ($x_{current}$, $y_{current}$) already has a cell assigned to it, increase $x_{current}$ by 1.
6. If $x_{current}$ is equal to x_{width}, increase x_{width} by 1. ($x_{current}$ is never *greater* than x_{width}.)
7. 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.
8. 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.
9. If *rowspan* is zero, then let *cell grows downward* be true, and set *rowspan* to 1. Otherwise, let *cell grows downward* be false.

10. If $x_{width} < x_{current}+colspan$, then let x_{width} be $x_{current}+colspan$.

11. If $y_{height} < y_{current}+rowspan$, then let y_{height} be $y_{current}+rowspan$.

12. 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.

13. If *cell grows downward* is true, then add the tuple {*c*, $x_{current}$, *colspan*} to the *list of downward-growing cells*.

14. Increase $x_{current}$ by *colspan*.

15. If *current cell* is the last `td` or `th` element in the `tr` element being processed, then increase $y_{current}$ by 1, abort this set of steps, and return to the algorithm above.

16. Let *current cell* be the next `td` or `th` element in the `tr` element being processed.

17. Return to step 5 (cells).

When the algorithms above require the user agent to run the **algorithm for growing downward-growing cells**, the user agent must, for each {*cell*, $cell_x$, *width*} tuple in the *list of downward-growing cells*, if any, extend the cell *cell* so that it also covers the slots with coordinates (x, $y_{current}$), where $cell_x \le x < cell_x+width$.

4.8.13.2. Forming relationships between data cells and header cells

Each data cell can be assigned zero or more header cells. The **algorithm for assigning header cells to data cells** is as follows.

1. For each header cell in the table, in tree order, run these substeps:

    1. Let ($header_x$, $header_y$) be the coordinate of the slot to which the header cell is anchored.

    2. Let $header_{width}$ be the width of the header cell.

    3. Let $header_{height}$ be the height of the header cell.

    4. Let *data cells* be a list of data cells, initially empty.

    5. Examine the `scope` attribute of the `th` element corresponding to the header cell, and, based on its state, apply the appropriate substep:

        ↪ **If it is in the *row* state**

            Add all the data cells that cover slots with coordinates ($slot_x$, $slot_y$), where $header_x+header_{width} \le slot_x < x_{width}$ and $header_y \le slot_y < header_y+header_{height}$, to the *data cells* list.

↪ **If it is in the *column* state**

Add all the data cells that cover slots with coordinates ($slot_x$, $slot_y$), where $header_x \leq slot_x < header_x+header_{width}$ and $header_y+header_{height} \leq slot_y < y_{height}$, to the *data cells* list.

↪ **If it is in the *row group* state**

If the header cell is not in a row group, then do nothing.

Otherwise, let (0, $group_y$) be the slot at which the row group is anchored, let *height* be the number of rows in the row group, and add all the data cells that cover slots with coordinates ($slot_x$, $slot_y$), where $header_x \leq slot_x < x_{width}$ and $header_y \leq slot_y < group_y+height$, to the *data cells* list.

↪ **If it is in the *column group* state**

If the header cell is not anchored in a column group, then do nothing.

Otherwise, let ($group_x$, 0) be the slot at which that column group is anchored, let *width* be the number of columns in the column group, and add all the data cells that cover slots with coordinates ($slot_x$, $slot_y$), where $header_x \leq slot_x < group_x+width$ and $header_y \leq slot_y < y_{height}$, to the *data cells* list.

↪ **Otherwise, it is in the *auto* state**

Run these steps:

1. If the header cell is equivalent to a wide cell, let $header_{width}$ equal x_{width}-$header_x$. [UNICODE]
2. Let x equal $header_x+header_{width}$.
3. *Horizontal*: If x is equal to x_{width}, then jump down to the step below labeled *vertical*.
4. If there is a header cell anchored at (x, $header_y$) with height $header_{height}$, then jump down to the step below labeled *vertical*.
5. Add all the data cells that cover slots with coordinates ($slot_x$, $slot_y$), where $slot_x = x$ and $header_y \leq slot_y < header_y+header_{height}$, to the *data cells* list.
6. Increase x by 1.
7. Jump up to the step above labeled *horizontal*.
8. *Vertical*: Let y equal $header_y+header_{height}$.
9. If y is equal to y_{height}, then jump to the step below labeled *end*.
10. If there is a header cell *cell* anchored at ($header_x$, y), then follow these substeps:
    1. If the header cell *cell* is equivalent to a wide cell, then let *width* be x_{width}-$header_x$. Otherwise, let *width* be the width of the header cell *cell*.
    2. If *width* is equal to $header_{width}$, then jump to the step below labeled *end*.

11. Add all the data cells that cover slots with coordinates ($slot_x$, $slot_y$), where $header_x \leq slot_x < header_x+header_{width}$ and $slot_y = y$, to the *data cells* list.

12. Increase *y* by 1.

13. Jump up to the step above labeled *vertical*.

14. *End*: Coalesce all the duplicate entries in the *data cells* list, so that each data cell is only present once, in tree order.

6. Assign the header cell to all the data cells in the *data cells* list that correspond to `td` elements that do not have a `headers` attribute specified.

2. For each data cell in the table, in tree order, run these substeps:

    1. If the data cell corresponds to a `td` element that does not have a `headers` attribute specified, then skip these substeps and move on to the next data cell (if any).

    2. Otherwise, take the value of the `headers` attribute and split it on spaces, letting *id list* be the list of tokens obtained.

    3. For each token in the *id list*, run the following steps:

        1. Let *id* be the token.

        2. If there is a header cell in the table whose corresponding `th` element has an ID that is equal to the value of *id*, then assign that header cell to the data cell.

A header cell anchored at ($header_x$, $header_y$) with width $header_{width}$ and height $header_{height}$ is said to be **equivalent to a wide cell** if all the slots with coordinates ($slot_x$, $slot_y$), where $header_x+header_{width} \leq slot_x < x_{width}$ and $header_y \leq slot_y < header_y+header_{height}$, are all either empty or covered by empty data cells.

A data cell is said to be an **empty data cell** if it contains no elements and its text content, if any, consists only of characters in the Unicode character class Zs. [UNICODE]

User agents may remove empty data cells when analyzing data in a table.

4.9 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.

4.9.1 The `form` element

4.9.2 The `fieldset` element

4.9.3 The `input` element

4.9.4 The `button` element

4.9.5 The `label` element

4.9.6 The `select` element

4.9.7 The `datalist` element

4.9.8 The `optgroup` element

4.9.9 The `option` element

4.9.10 Constructors

All `Window` objects must provide the following constructors:

`Option()`

`Option(in DOMString name)`

`Option(in DOMString name, in DOMString value)`

When invoked as constructors, these must return a new `HTMLOptionElement` object (a new `option` element). need to define argument processing

4.9.11 The `textarea` element

4.9.12 The `output` element

4.9.13 Processing model

See WF2 for now

4.9.13.1. Form submission

See WF2 for now

If a form is in a browsing context whose sandboxed forms browsing context flag is set, it must not be submitted.

4.10 Scripting

Scripts allow authors to add interactivity to their documents.

Authors are encouraged to use declarative alternatives to scripting where possible, as declarative mechanisms are often more maintainable, and many users disable scripting.

> For example, instead of using script to show or hide a section to show more details, the `details` element could be used.

Authors are also encouraged to make their applications degrade gracefully in the absence of scripting support.

> For example, if an author provides a link in a table header to dynamically resort the table, the link could also be made to function without scripts by requesting the sorted table from the server.

4.10.1 The `script` element

Categories

Metadata content.
Phrasing content.

Contexts in which this element may be used:

Where metadata content is expected.
Where phrasing content is expected.

Content model:

If there is no `src` attribute, depends on the value of the `type` attribute.
If there *is* a `src` attribute, the element must be empty.

Element-specific attributes:

`src`
`async`
`defer`
`type`
`charset`

DOM interface:

```
interface HTMLScriptElement : HTMLElement {
           attribute DOMString src;
           attribute boolean async;
           attribute boolean defer;
           attribute DOMString type;
           attribute DOMString charset;
           attribute DOMString text;
```

```
};
```

The `script` element allows authors to include dynamic script and script data in their documents.

When used to include dynamic scripts, the scripts may either be embedded inline or may be imported from an external file using the `src` attribute. If the language is not that described by "`text/javascript`", then the type of the script's language must be given using the `type` attribute.

When used to include script data, the script data must be embedded inline, the format of the data must be given using the `type` attribute, and the `src` attribute must not be specified.

The **`type`** attribute gives the language of the script or format of the script data. If the attribute is present, its value must be a valid MIME type, optionally with parameters. The `charset` parameter must not be specified. (The default, which is used if the attribute is absent, is "`text/javascript`".) [RFC2046]

The **`src`** attribute, if specified, gives the address of the external script resource to use. The value of the attribute must be a valid URL identifying a script resource of the type given by the `type` attribute, if the attribute is present, or of the type "`text/javascript`", if the attribute is absent.

The **`charset`** attribute gives the character encoding of the external script resource. The attribute must not be specified if the `src` attribute is not present. If the attribute is set, its value must be a valid character encoding name, and must be the preferred name for that encoding. [IANACHARSET]

The encoding specified must be the encoding used by the script resource. If the `charset` attribute is omitted, the character encoding of the document will be used. If the script resource uses a different encoding than the document, then the attribute must be specified.

The **`async`** and **`defer`** attributes are boolean attributes that indicate how the script should be executed.

There are three possible modes that can be selected using these attributes. If the `async` attribute is present, then the script will be executed asynchronously, as soon as it is available. If the `async` attribute is not present but the `defer` attribute is present, then the script is executed when the page has finished parsing. If neither attribute is present, then the script is downloaded and executed immediately, before the user agent continues parsing the page. The exact processing details for these attributes is described below.

The `defer` attribute may be specified even if the `async` attribute is specified, to cause legacy Web browsers that only support `defer` (and not `async`) to fall back to the `defer` behavior instead of the synchronous blocking behavior that is the default.

Changing the `src`, `type`, `charset`, `async`, and `defer` attributes dynamically has no direct effect; these attribute are only used at specific times described below (namely, when the element is inserted into the document).

`script` elements have four associated pieces of metadata. The first is a flag indicating whether or not the script block has been **"already executed"**. Initially, `script` elements must have this flag unset (script blocks, when created, are not "already executed"). When a `script` element is cloned, the "already executed" flag, if set, must be propagated to the clone when it is created. The second is a flag indicating whether the element was **"parser-inserted"**. This flag is set by the HTML parser and is used to handle `document.write()` calls. The third and fourth pieces of metadata are ***the script's type*** and ***the script's character encoding***. They are determined when the script is run, based on the attributes on the element at that time.

Running a script: When a script block is inserted into a document, the user agent must act as follows:

1. If the `script` element has a `type` attribute and its value is the empty string, or if the `script` element has no `type` attribute but it has a `language` attribute and *that* attribute's value is the empty string, or if the `script` element has neither a `type` attribute nor a `language` attribute, let *the script's type* for this `script` element be "`text/javascript`".

   Otherwise, if the `script` element has a `type` attribute, let *the script's type* for this `script` element be the value of that attribute.

   Otherwise, the element has a `language` attribute; let *the script's type* for this `script` element be the concatenation of the string "`text/`" followed by the value of the `language` attribute.

2. If the `script` element has a `charset` attribute, then let *the script's character encoding* for this `script` element be the encoding given by the `charset` attribute.

   Otherwise, let *the script's character encoding* for this `script` element be the same as the encoding of the document itself.

3. If the `script` element is without script, or if the `script` element was created by an XML parser that itself was created as part of the processing of the `innerHTML` attribute's setter, or if the user agent does not support the scripting language given by *the script's type* for this `script` element, or if the `script` element has its "already executed" flag set, then the user agent must abort these steps at this point. The script is not executed.

4. The user agent must set the element's "already executed" flag.

5. 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.

6. Then, the first of the following options that describes the situation must be followed:

   ↪ **If the document is still being parsed, and the element has a `defer` attribute, and the element does not have an `async` attribute**

   The element must be added to the end of the list of scripts that will execute when the document has finished parsing. 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 pending external script. (There can only be one such script at a time.)

↪ **If the element has a `src` attribute**

The element must be added to the end of the list of scripts that will execute as soon as possible. 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 run 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 *pending external script*:**

The script will be handled when the parser resumes.

The download of an external script must delay the `load` event.

Executing a script block: When the steps above require that the script be executed, the user agent must act as follows:

↪ **If the load resulted in an error (for example a DNS error, or an HTTP 404 error)**

Executing the script must just consist of firing an `error` event at the element.

↪ **If the load was successful**

1. If the `script` element's `Document` is the active document in its browsing context, the user agent must execute the script:

   ↪ **If the script is from an external file**

   That file must be used as the file to execute.

   The file must be interpreted using the character encoding given by *the script's character encoding*, regardless of any metadata given by the file's Content-Type metadata.

   This means that a UTF-16 document will always assume external scripts are UTF-16...? This applies, e.g., to document's created using createDocument()... It also means changing document.charSet will affect the character encoding used to interpret scripts, is that really what happens?

   ↪ **If the script is inline**

   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 defined by the language associated with *the script's type* (see the scripting languages section below).

   The script execution context of the script must be the `Window` object of that browsing context.

   The script document context of the script must be the `Document` object that owns the `script` element.

   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.

2. Then, the user agent must fire a `load` event at the `script` element.

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

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.

4.10.1.1. Scripting languages

A user agent is said to **support the scripting language** if *the script's type* matches the MIME type of a scripting language that the user agent implements.

The following lists some MIME types and the languages to which they refer:

`text/javascript`

ECMAScript. [ECMA262]

`text/javascript;e4x=1`

ECMAScript with ECMAScript for XML. [ECMA357]

User agents may support other MIME types and other languages.

When examining types to determine if they support the language, user agents must not ignore unknown MIME parameters — types with unknown parameters must be assumed to be unsupported.

4.10.2 The `noscript` element

Categories

Metadata content.
Phrasing content.

Contexts in which this element may be used:

In a `head` element of an HTML document, if there are no ancestor `noscript` elements.
Where phrasing content is expected in HTML documents, if there are no ancestor `noscript` elements.

Content model:

Without script, in a `head` element: in any order, zero or more `link` elements, zero or more `style` elements, and zero or more `meta` elements.
Without script, not in a `head` element: transparent, but there must be no `noscript` element descendants.
With script: text that conforms to the requirements given in the prose.

Element-specific attributes:

None.

DOM interface:

Uses `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.

Note: The `noscript` element is only effective in the HTML serialization, it has no effect in the XML serialization.

When used in HTML documents, the allowed content model is as follows:

In a `head` element, if the `noscript` element is without script, then the content model of a `noscript` element must contain only `link`, `style`, and `meta` elements. If the `noscript` element is with script, then the content model of a `noscript` element is text, except that invoking the HTML fragment parsing algorithm with the `noscript` element as the *context* element and the text contents as the *input* must result in a list of nodes that consists only of `link`, `style`, and `meta` elements.

Outside of `head` elements, if the `noscript` element is without script, then the content model of a `noscript` element is transparent, with the additional restriction that a `noscript` element must not have a `noscript` element as an ancestor (that is, `noscript` can't be nested).

Outside of `head` elements, if the `noscript` element is with script, 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 the element is with script.

Note: All these contortions are required because, for historical reasons, the `noscript` element is handled differently by the HTML parser based on whether scripting was enabled or not when the parser was invoked. The element is not allowed in XML, because in XML the parser is not affected by such state, and thus the element would not have the desired effect.

Note: The `noscript` element interacts poorly with the `designMode` feature. Authors are encouraged to not use `noscript` elements on pages that will have `designMode` enabled.

4.10.3 The `eventsource` element

Categories

Metadata content.
Phrasing content.

Contexts in which this element may be used:

Where metadata content is expected.
Where phrasing content is expected.

Content model:

Empty.

Element-specific attributes:

`src`

DOM interface:

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

The `eventsource` element represents a target for events generated by a remote server.

The **`src`** attribute, if specified, must give a valid URL identifying a resource that uses the `text/event-stream` format.

When an `eventsource` element with a `src` attribute specified is inserted into the document, and when an `eventsource` element that is already in the document has a `src` attribute added, the user agent must run the add declared event source algorithm.

While an `eventsource` element is in a document, if its `src` attribute is mutated, the user agent must must run the remove declared event source algorithm followed by the add declared event source algorithm.

When an `eventsource` element with a `src` attribute specified is removed from a document, and when an `eventsource` element that is in a document with a `src` attribute specified has its `src` attribute removed, the user agent must run the remove declared event source algorithm.

When it is created, an `eventsource` element must have its *current declared event source* set to "undefined".

The **add declared event source** algorithm is as follows:

1. Resolve the URL specified by the `eventsource` element's `src` attribute.
2. If that fails, then set the element's *current declared event source* to "undefined" and abort these steps.
3. Otherwise, act as if the `addEventSource()` method on the `eventsource` element had been invoked with the resulting absolute URL.
4. Let the element's *current declared event source* be that absolute URL.

The **remove declared event source** algorithm is as follows:

1. If the element's *current declared event source* is "undefined", abort these steps.
2. Otherwise, act as if the `removeEventSource()` method on the `eventsource` element had been invoked with the element's *current declared event source*.
3. Let the element's *current declared event source* be "undefined".

There can be more than one `eventsource` element per document, but authors should take care to avoid opening multiple connections to the same server as HTTP recommends a limit to the number of simultaneous connections that a user agent can open per server.

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

4.11 Interactive elements

4.11.1 The `details` element

Categories

Flow content.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

One `legend` element followed by flow content.

Element-specific attributes:

`open`

DOM interface:

```
interface HTMLDetailsElement : HTMLElement {
           attribute boolean open;
};
```

The `details` element represents additional information or controls which the user can obtain on demand.

The first element child of a `details` element, if it is a `legend` element, represents the summary of the details.

If the first element is not a `legend` element, the UA should provide its own legend (e.g. "Details").

The **`open`** content attribute is a boolean attribute. If present, it indicates that the details should be shown to the user. If the attribute is absent, the details should not be shown.

If the attribute is removed, then the details should be hidden. If the attribute is added, the details should be shown.

The user 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?

4.11.2 The `datagrid` element

Categories

Flow content.
Interactive element.
Sectioning root.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Either: Nothing.
Or: Flow content, but where the first element child node, if any, is not a `table`, `select`, or `datalist` element.
Or: A single `table` element.
Or: A single `select` element.
Or: A single `datalist` element.

Element-specific attributes:

`multiple`
`disabled`

DOM interface:

```
interface HTMLDataGridElement : HTMLElement {
           attribute DataGridDataProvider data;
  readonly attribute DataGridSelection selection;
           attribute boolean multiple;
           attribute boolean disabled;
  void updateEverything();
  void updateRowsChanged(in RowSpecification row, in unsigned long
count);
  void updateRowsInserted(in RowSpecification row, in unsigned long
count);
```

```
  void updateRowsRemoved(in RowSpecification row, in unsigned long
count);
  void updateRowChanged(in RowSpecification row);
  void updateColumnChanged(in unsigned long column);
  void updateCellChanged(in RowSpecification row, in unsigned long
column);
};
```

One possible thing to be added is a way to detect when a row/selection has been deleted, activated, etc, by the user (delete key, enter key, etc).

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` and `disabled` attributes are boolean attributes. Their effects are described in the processing model sections below.

The **`multiple`** and **`disabled`** DOM attributes must reflect the `multiple` and `disabled` content attributes respectively.

4.11.2.1. The `datagrid` data model

This section is non-normative.

In the `datagrid` data model, data is structured as a set of rows representing a tree, each row being split into a number of columns. The columns are always present in the data model, although individual columns may be hidden in the presentation.

Each row can have child rows. Child rows may be hidden or shown, by closing or opening (respectively) the parent row.

Rows are referred to by the path along the tree that one would take to reach the row, using zero-based indices. Thus, the first row of a list is row "0", the second row is row "1"; the first child row of the first row is row "0,0", the second child row of the first row is row "0,1"; the fourth child of the seventh child of the third child of the tenth row is "9,2,6,3", etc.

The columns can have captions. Those captions are not considered a row in their own right, they are obtained separately.

Selection of data in a `datagrid` operates at the row level. If the `multiple` attribute is present, multiple rows can be selected at once, otherwise the user can only select one row at a time.

The `datagrid` element can be disabled entirely by setting the `disabled` attribute.

Columns, rows, and cells can each have specific flags, known as classes, applied to them by the data provider. These classes affect the functionality of the `datagrid` element, and are also passed to the style system. They are similar in concept to the `class` attribute, except that they are not specified on elements but are given by scripted data providers.

4.11.2.2. How rows are identified

The chains of numbers that give a row's path, or identifier, are represented by objects that implement the RowSpecification interface.

```
[NoInterfaceObject] interface RowSpecification {
  // binding-specific interface
};
```

In ECMAScript, two classes of objects are said to implement this interface: Numbers representing non-negative integers, and homogeneous arrays of Numbers representing non-negative integers. Thus, `[1,0,9]` is a RowSpecification, as is `1` on its own. However, `[1,0.2,9]` is not a RowSpecification object, since its second value is not an integer.

User agents must always represent `RowSpecification`s in ECMAScript by using arrays, even if the path only has one number.

The root of the tree is represented by the empty path; in ECMAScript, this is the empty array (`[]`). Only the `getRowCount()` and `GetChildAtPosition()` methods ever get called with the empty path.

4.11.2.3. The data provider interface

The conformance criteria in this section apply to any implementation of the `DataGridDataProvider`, including (and most commonly) the content author's implementation(s).

```
// To be implemented by Web authors as a JS object
[NoInterfaceObject] interface DataGridDataProvider {
  void initialize(in HTMLDataGridElement datagrid);
  unsigned long getRowCount(in RowSpecification row);
  unsigned long getChildAtPosition(in RowSpecification parentRow, in
unsigned long position);
  unsigned long getColumnCount();
  DOMString getCaptionText(in unsigned long column);
  void getCaptionClasses(in unsigned long column, in DOMTokenList classes);
  DOMString getRowImage(in RowSpecification row);
  HTMLMenuElement getRowMenu(in RowSpecification row);
  void getRowClasses(in RowSpecification row, in DOMTokenList classes);
  DOMString getCellData(in RowSpecification row, in unsigned long column);
  void getCellClasses(in RowSpecification row, in unsigned long column, in
DOMTokenList classes);
  void toggleColumnSortState(in unsigned long column);
```

```
  void setCellCheckedState(in RowSpecification row, in unsigned long column,
in long state);
  void cycleCell(in RowSpecification row, in unsigned long column);
  void editCell(in RowSpecification row, in unsigned long column, in
DOMString data);
};
```

The `DataGridDataProvider` interface represents the interface that objects must implement to be used as custom data views for `datagrid` elements.

Not all the methods are required. The minimum number of methods that must be implemented in a useful view is two: the `getRowCount()` and `getCellData()` methods.

Once the object is written, it must be hooked up to the `datagrid` using the **`data`** DOM attribute.

The following methods may be usefully implemented:

`initialize(datagrid)`

Called by the `datagrid` element (the one given by the *datagrid* argument) after it has first populated itself. This would typically be used to set the initial selection of the `datagrid` element when it is first loaded. The data provider could also use this method call to register a `select` event handler on the `datagrid` in order to monitor selection changes.

`getRowCount(row)`

Must return the number of rows that are children of the specified *row*, including rows that are off-screen. If *row* is empty, then the number of rows at the top level must be returned. If the value that this method would return for a given *row* changes, the relevant update methods on the `datagrid` must be called first. Otherwise, this method must always return the same number. For a list (as opposed to a tree), this method must return 0 whenever it is called with a *row* identifier that is not empty.

`getChildAtPosition(parentRow, position)`

Must return the index of the row that is a child of *parentRow* and that is to be positioned as the *position*th row under *parentRow* when rendering the children of *parentRow*. If *parentRow* is empty, then *position* refers to the *position*th row at the top level of the data grid. May be omitted if the rows are always to be sorted in the natural order. (The natural order is the one where the method always returns *position*.) For a given *parentRow*, this method must never return the same value for different values of *position*. The returned value *x* must be in the range $0 \leq x < n$, where *n* is the value returned by `getRowCount(`*`parentRow`*`)`.

`getColumnCount()`

Must return the number of columns currently in the data model (including columns that might be hidden). May be omitted if there is only one column. If the value that this method would return changes, the `datagrid`'s `updateEverything()` method must be called.

`getCaptionText(column)`

Must return the caption, or label, for column *column*. May be omitted if the columns have no captions. If the value that this method would return changes, the `datagrid`'s `updateColumnChanged()` method must be called with the appropriate column index.

`getCaptionClasses(column, classes)`

Must add the classes that apply to column *column* to the *classes* object. May be omitted if the columns

have no special classes. If the classes that this method would add changes, the `datagrid`'s `updateColumnChanged()` method must be called with the appropriate column index. Some classes have predefined meanings.

getRowImage(row)

Must return a URL giving the address of an image that represents row *row*, or the empty string if there is no applicable image. May be omitted if no rows have associated images. If the value that this method would return changes, the `datagrid`'s update methods must be called to update the row in question.

getRowMenu(row)

Must return an `HTMLMenuElement` object that is to be used as a context menu for row *row*, or null if there is no particular context menu. May be omitted if none of the rows have a special context menu. As this method is called immediately before showing the menu in question, no precautions need to be taken if the return value of this method changes.

getRowClasses(row, classes)

Must add the classes that apply to row *row* to the *classes* object. May be omitted if the rows have no special classes. If the classes that this method would add changes, the `datagrid`'s update methods must be called to update the row in question. Some classes have predefined meanings.

getCellData(row, column)

Must return the value of the cell on row *row* in column *column*. For text cells, this must be the text to show for that cell. For progress bar cells, this must be either a floating point number in the range 0.0 to 1.0 (converted to a string representation), indicating the fraction of the progress bar to show as full (1.0 meaning complete), or the empty string, indicating an indeterminate progress bar. If the value that this method would return changes, the `datagrid`'s update methods must be called to update the rows that changed. If only one cell changed, the `updateCellChanged()` method may be used.

getCellClasses(row, column, classes)

Must add the classes that apply to the cell on row *row* in column *column* to the *classes* object. May be omitted if the cells have no special classes. If the classes that this method would add changes, the `datagrid`'s update methods must be called to update the rows or cells in question. Some classes have predefined meanings.

toggleColumnSortState(column)

Called by the `datagrid` when the user tries to sort the data using a particular column *column*. The data provider must update its state so that the `GetChildAtPosition()` method returns the new order, and the classes of the columns returned by `getCaptionClasses()` represent the new sort status. There is no need to tell the `datagrid` that it the data has changed, as the `datagrid` automatically assumes that the entire data model will need updating.

setCellCheckedState(row, column, state)

Called by the `datagrid` when the user changes the state of a checkbox cell on row *row*, column *column*. The checkbox should be toggled to the state given by *state*, which is a positive integer (1) if the checkbox is to be checked, zero (0) if it is to be unchecked, and a negative number (−1) if it is to be set to the indeterminate state. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

cycleCell(row, column)

Called by the `datagrid` when the user changes the state of a cyclable cell on row *row*, column *column*. The data provider should change the state of the cell to the new state, as appropriate. There is

no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

editCell(row, column, data)

Called by the `datagrid` when the user edits the cell on row *row*, column *column*. The new value of the cell is given by *data*. The data provider should update the cell accordingly. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

The following classes (for rows, columns, and cells) may be usefully used in conjunction with this interface:

Class name	Applies to	Description
checked	Cells	The cell has a checkbox and it is checked. (The `cyclable` and `progress` classes override this, though.)
cyclable	Cells	The cell can be cycled through multiple values. (The `progress` class overrides this, though.)
editable	Cells	The cell can be edited. (The `cyclable`, `progress`, `checked`, `unchecked` and `indeterminate` classes override this, though.)
header	Rows	The row is a heading, not a data row.
indeterminate	Cells	The cell has a checkbox, and it can be set to an indeterminate state. If neither the `checked` nor `unchecked` classes are present, then the checkbox is in that state, too. (The `cyclable` and `progress` classes override this, though.)
initially-hidden	Columns	The column will not be shown when the `datagrid` is initially rendered. If this class is not present on the column when the `datagrid` is initially rendered, the column will be visible if space allows.
initially-closed	Rows	The row will be closed when the `datagrid` is initially rendered. If neither this class nor the `initially-open` class is present on the row when the `datagrid` is initially rendered, the initial state will depend on platform conventions.
initially-open	Rows	The row will be opened when the `datagrid` is initially rendered. If neither this class nor the `initially-closed` class is present on the row when the `datagrid` is initially rendered, the initial state will depend on platform conventions.
progress	Cells	The cell is a progress bar.
reversed	Columns	If the cell is sorted, the sort direction is descending, instead of ascending.
selectable-separator	Rows	The row is a normal, selectable, data row, except that instead of having data, it only has a separator. (The `header` and `separator` classes override this, though.)
separator	Rows	The row is a separator row, not a data row. (The `header` class overrides this, though.)
sortable	Columns	The data can be sorted by this column.
sorted	Columns	The data is sorted by this column. Unless the `reversed` class is also present, the sort direction is ascending.
unchecked	Cells	The cell has a checkbox and, unless the `checked` class is present as well, it is unchecked. (The `cyclable` and `progress` classes override this, though.)

4.11.2.4. The default data provider

The user agent must supply a default data provider for the case where the `datagrid`'s `data` attribute is null. It must act as described in this section.

The behavior of the default data provider depends on the nature of the first element child of the `datagrid`.

↳ While the first element child is a `table` element

`getRowCount(`*`row`*`)` : The number of rows returned by the default data provider for the root of the tree (when *row* is empty) must be the total number of `tr` elements that are children of `tbody` elements that are children of the `table`, if there are any such child `tbody` elements. If there are no such `tbody` elements then the number of rows returned for the root must be the number of `tr`

elements that are children of the `table`.

When *row* is not empty, the number of rows returned must be zero.

Note: The `table`-based default data provider cannot represent a tree.

Note: Rows in `thead` elements do not contribute to the number of rows returned, although they do affect the columns and column captions. Rows in `tfoot` elements are ignored completely by this algorithm.

`getChildAtPosition(row, i)`: The default data provider must return the mapping appropriate to the current sort order.

`getColumnCount()`: The number of columns returned must be the number of `td` element children in the first `tr` element child of the first `tbody` element child of the `table`, if there are any such `tbody` elements. If there are no such `tbody` elements, then it must be the number of `td` element children in the first `tr` element child of the `table`, if any, or otherwise 1. If the number that would be returned by these rules is 0, then 1 must be returned instead.

`getCaptionText(i)`: If the `table` has no `thead` element child, or if its first `thead` element child has no `tr` element child, the default data provider must return the empty string for all captions. Otherwise, the value of the `textContent` attribute of the *i*th `th` element child of the first `tr` element child of the first `thead` element child of the `table` element must be returned. If there is no such `th` element, the empty string must be returned.

`getCaptionClasses(i, classes)`: If the `table` has no `thead` element child, or if its first `thead` element child has no `tr` element child, the default data provider must not add any classes for any of the captions. Otherwise, each class in the `class` attribute of the *i*th `th` element child of the first `tr` element child of the first `thead` element child of the `table` element must be added to the *classes*. If there is no such `th` element, no classes must be added. The user agent must then:

1. Remove the `sorted` and `reversed` classes.
2. If the `table` element has a `class` attribute that includes the `sortable` class, add the `sortable` class.
3. If the column is the one currently being used to sort the data, add the `sorted` class.
4. If the column is the one currently being used to sort the data, and it is sorted in descending order, add the `reversed` class as well.

The various row- and cell- related methods operate relative to a particular element, the element of the row or cell specified by their arguments.

For rows: Since the default data provider for a `table` always returns 0 as the number of children for any row other than the root, the path to the row passed to these methods will always consist of a single number. In the prose below, this number is referred to as *i*.

If the `table` has `tbody` element children, the element for the *i*th row is the *i*th `tr` element that is a child of a `tbody` element that is a child of the `table` element. If the `table` does not have `tbody`

element children, then the element for the *i*th real row is the *i*th `tr` element that is a child of the `table` element.

For cells: Given a row and its element, the row's *i*th cell's element is the *i*th `td` element child of the row element.

Note: The `colspan` and `rowspan` attributes are ignored by this algorithm.

`getRowImage(i)`: The URL of the row's image is the absolute URL obtained by resolving the value of the `src` attribute of the first `img` element child of the row's first cell's element, if there is one and resolving its attribute is successful. Otherwise, the URL of the row's image is the empty string.

`getRowMenu(i)`: If the row's first cell's element has a `menu` element child, then the row's menu is the first `menu` element child of the row's first cell's element. Otherwise, the row has no menu.

`getRowClasses(i, classes)`: The default data provider must never add a class to the row's classes.

`toggleColumnSortState(i)`: If the data is already being sorted on the given column, then the user agent must change the current sort mapping to be the inverse of the current sort mapping; if the sort order was ascending before, it is now descending, otherwise it is now ascending. Otherwise, if the current sort column is another column, or the data model is currently not sorted, the user agent must create a new mapping, which maps rows in the data model to rows in the DOM so that the rows in the data model are sorted by the specified column, in ascending order. (Which sort comparison operator to use is left up to the UA to decide.)

When the sort mapping is changed, the values returned by the `getChildAtPosition()` method for the default data provider will change appropriately.

`getCellData(i, j)`, `getCellClasses(i, j, classes)`, `getCellCheckedState(i, j, state)`, `cycleCell(i, j)`, and `editCell(i, j, data)`: See the common definitions below.

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate. For example, if a `tr` is removed, then the `updateRowsRemoved()` methods would probably need to be invoked, and any change to a cell or its descendants must cause the cell to be updated. If the `table` element stops being the first child of the `datagrid`, then the data provider must call the `updateEverything()` method on the `datagrid`. Any change to a cell that is in the column that the data provider is currently using as its sort column must also cause the sort to be reperformed, with a call to `updateEverything()` if the change did affect the sort order.

↪ **While the first element child is a `select` or `datalist` element**

The default data provider must return 1 for the column count, the empty string for the column's caption, and must not add any classes to the column's classes.

For the rows, assume the existence of a node filter view of the descendants of the first element child of the `datagrid` element (the `select` or `datalist` element), that skips all nodes other than `optgroup` and `option` elements, as well as any 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 values
> removed
>        to make the underlying structure clearer -->
>   <optgroup>
>    <option/>
>    <option/>
>   </optgroup>
>   <optgroup>
>    <option/>
>    <optgroup id="a">
>     <option/>
>     <option/>
>     <bogus/>
>     <option id="b"/>
>    </optgroup>
>    <option/>
>   </optgroup>
>  </select>
> </datagrid>
> ```
>
> The path "1,1,2" would select the element with ID "b". In the filtered view, the text nodes, comment nodes, and bogus elements are ignored; so for instance, the element with ID "a" (path "1,1") has only 3 child nodes in the view.

`getRowCount(`*`row`*`)` must drill through the view to find the element corresponding to the method's argument, and return the number of child nodes in the filtered view that the corresponding element has. (If the *row* is empty, the corresponding element is the `select` element at the root of the filtered view.)

`getChildAtPosition(`*`row`*`, `*`position`*`)` must return *position*. (The `select`/`datalist` default data provider does not support sorting the data grid.)

`getRowImage(`*`i`*`)` must return the empty string, `getRowMenu(`*`i`*`)` must return null.

`getRowClasses(`*`row`*`, `*`classes`*`)` must add the classes from the following list to *classes* when their condition is met:

- If the *row*'s corresponding element is an `optgroup` element: `header`
- If the *row*'s corresponding element contains other elements that are also in the view, and the element's `class` attribute contains the `closed` class: `initially-closed`
- If the *row*'s corresponding element contains other elements that are also in the view, and the element's `class` attribute contains the `open` class: `initially-open`

The `getCellData(`*`row`*`, `*`cell`*`)` method must return the value of the `label` attribute if the *row*'s

corresponding element is an `optgroup` element, otherwise, if the *row*'s corresponding element is an `option`element, its `label` attribute if it has one, otherwise the value of its `textContent` DOM attribute.

The `getCellClasses(`*`row`*`, `*`cell`*`, `*`classes`*`)` method must add no classes.

autoselect some rows when initialized, reflect the selection in the select, reflect the multiple attribute somehow.

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate.

↪ While the first element child is another element

The default data provider must return 1 for the column count, the empty string for the column's caption, and must not add any classes to the column's classes.

For the rows, assume the existence of a node filter view of the descendants of the `datagrid` that skips all nodes other than `li`, `h1`–`h6`, and `hr` elements, and skips any descendants of `menu` elements.

Given this view, each element in the view represents a row in the data model. The element corresponding to a path *row* is the one obtained by drilling into the view, taking the child given by the path each time. The element of the row of a particular method call is the element given by drilling into the view along the path given by the method's arguments.

`getRowCount(`*`row`*`)` must return the number of child elements in this view for the given row, or the number of elements at the root of the view if the *row* is empty.

> In the following example, the elements are identified by the paths given by their child text nodes:
>
> ```
> <datagrid>
>  <ol>
>   <li> row 0 </li>
>   <li> row 1
>    <ol>
>     <li> row 1,0 </li>
>    </ol>
>   </li>
>   <li> row 2 </li>
>  </ol>
> </datagrid>
> ```
>
> In this example, only the `li` elements actually appear in the data grid; the `ol` element does not affect the data grid's processing model.

`getChildAtPosition(`*`row`*`, `*`position`*`)` must return *position*. (The generic default data provider does not support sorting the data grid.)

`getRowImage(`*`i`*`)` must return the absolute URL obtained from resolving the value of the `src` attribute of the first `img` element descendant (in the real DOM) of the row's element, that is not also

a descendant of another element in the filtered view that is a descendant of the row's element, if such an element exists and resolving its attribute is successful. Otherwise, it must return the empty string.

> In the following example, the row with path "1,0" returns "http://example.com/a" as its image URL, and the other rows (including the row with path "1") return the empty string:
>
> ```
> <datagrid>
>  <ol>
>   <li> row 0 </li>
>   <li> row 1
>    <ol>
>     <li> row 1,0 <img src="http://example.com/a" alt=""> </li>
>    </ol>
>   </li>
>   <li> row 2 </li>
>  </ol>
> </datagrid>
> ```

`getRowMenu(i)` must return the first `menu` element descendant (in the real DOM) of the row's element, that is not also a descendant of another element in the filtered view that is a descendant of the row's element. (This is analogous to the image case above.)

`getRowClasses(i, classes)` must add the classes from the following list to *classes* when their condition is met:

- If the row's element contains other elements that are also in the view, and the element's `class` attribute contains the `closed` class: `initially-closed`
- If the row's element contains other elements that are also in the view, and the element's `class` attribute contains the `open` class: `initially-open`
- If the row's element is an `h1`–`h6` element: `header`
- If the row's element is an `hr` element: `separator`

The `getCellData(i, j)`, `getCellClasses(i, j, classes)`, `getCellCheckedState(i, j, state)`, `cycleCell(i, j)`, and `editCell(i, j, data)` methods must act as described in the common definitions below, treating the row's element as being the cell's element.

selection handling?

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate.

↪ **Otherwise, while there is no element child**

The data provider must return 0 for the number of rows, 1 for the number of columns, the empty string for the first column's caption, and must add no classes when asked for that column's classes. If the `datagrid`'s child list changes such that there is a first element child, then the data provider must call the `updateEverything()` method on the `datagrid`.

4.11.2.4.1. Common default data provider method definitions for cells

These definitions are used for the cell-specific methods of the default data providers (other than in the `select`/`datalist` case). How they behave is based on the contents of an element that represents the cell given by their first two arguments. Which element that is is defined in the previous section.

Cyclable cells

If the first element child of a cell's element is a `select` element that has a no `multiple` attribute and has at least one `option` element 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.

4.11.2.5. Populating the `datagrid` element

A `datagrid` must be disabled until its end tag has been parsed (in the case of a `datagrid` element in the original document markup) or until it has been inserted into the document (in the case of a dynamically created element). After that point, the element must fire a single `load` event at itself, which doesn't bubble and cannot be canceled.

The end-tag parsing thing should be moved to the parsing section.

The `datagrid` must then populate itself using the data provided by the data provider assigned to the `data` DOM attribute. After the view is populated (using the methods described below), the `datagrid` must invoke the `initialize()` method on the data provider specified by the `data` attribute, passing itself (the `HTMLDataGridElement` object) as the only argument.

When the `data` attribute is null, the `datagrid` must use the default data provider described in the previous section.

To obtain data from the data provider, the element must invoke methods on the data provider object in the following ways:

To determine the total number of columns

Invoke the `getColumnCount()` method with no arguments. The return value is the number of columns. If the return value is zero or negative, not an integer, or simply not a numeric type, or if the method is not defined, then 1 must be used instead.

To get the captions to use for the columns

Invoke the `getCaptionText()` method with the index of the column in question. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of columns. The return value is the string to use when referring to that column. If the method returns null or the empty string, the column has no caption. If the method is not defined, then none of the columns have any captions.

To establish what classes apply to a column

Invoke the `getCaptionClasses()` method with the index of the column in question, and an object implementing the `DOMTokenList` interface, associated with an anonymous empty string. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of columns. The tokens contained in the string underlying `DOMTokenList` object when the method returns represent the classes that apply to the given column. If the method is not defined, no classes apply to the column.

To establish whether a column should be initially included in the visible columns

Check whether the `initially-hidden` class applies to the column. If it does, then the column should not be initially included; if it does not, then the column should be initially included.

To establish whether the data can be sorted relative to a particular column

Check whether the `sortable` class applies to the column. If it does, then the user 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 identity transform; the data is rendered in its natural order).

To establish what classes apply to a row

Invoke the `getRowClasses()` method with a `RowSpecification` object representing the row in question, and a `DOMTokenList` associated with an empty string. The tokens contained in the `DOMTokenList` object's underlying string when the method returns represent the classes that apply to the row in question. If the method is not defined, no classes apply to the row.

To establish whether a row is a data row or a special row

Examine the classes that apply to the row. If the `header` class applies to the row, then it is not a data row, it is a subheading. The data from the first cell of the row is the text of the subheading, the rest of the cells must be ignored. Otherwise, if the `separator` class applies to the row, then in the place of the row, a separator should be shown. Otherwise, if the `selectable-separator` class applies to the row, then the row should be a data row, but represented as a separator. (The difference between a `separator` and a `selectable-separator` is that the former is not an item that can be actually selected, whereas the second can be selected and thus has a context menu that applies to it, and so forth.) For both kinds of separator rows, the data of the rows' cells must all be ignored. If none of those three classes apply then the row is a simple data row.

To establish whether a row is openable

Determine the number of child rows for that row. If there are one or more child rows, then the row is openable.

To establish whether a row should be initially open or closed

If the row is openable, examine the classes that apply to the row. If the `initially-open` class applies to the row, then it should be initially open. Otherwise, if the `initially-closed` class applies to the row, then it must be initially closed. Otherwise, if neither class applies to the row, or if the row is not openable, then the initial state of the row should be based on platform conventions.

To obtain a URL identifying an image representing a row

Invoke the `getRowImage()` method with a `RowSpecification` object representing the row in question. The return value is a URL. Immediately resolve that URL as if it came from an attribute of the `datagrid` element to obtain an absolute URL identifying the image that represents the row. If the method returns the empty string, null, or if the method is not defined, then the row has no associated image.

To obtain a context menu appropriate for a particular row

Invoke the `getRowMenu()` method with a `RowSpecification` object representing the row in question. The return value is a reference to an object implementing the `HTMLMenuElement` interface, i.e. a `menu` element DOM node. (This element must then be interpreted as described in the section on context menus to obtain the actual context menu to use.) If the method returns something that is not an `HTMLMenuElement`, or if the method is not defined, then the row has no associated context menu. User agents may provide their own default context menu, and may add items to the author-provided context menu. For example, such a menu could allow the user to change the presentation of the `datagrid` element.

To establish the value of a particular cell

Invoke the `getCellData()` method with the first argument being a `RowSpecification` object representing the row of the cell in question and the second argument being the index of the cell's column. The second argument must be a non-negative integer less than the total number of columns. The return value is the value of the cell. If the return value is null or the empty string, or if the method is not defined, then the cell has no data. (For progress bar cells, the cell's value must be further interpreted, as described below.)

To establish what classes apply to a cell

Invoke the `getCellClasses()` method with the first argument being a `RowSpecification` object representing the row of the cell in question, the second argument being the index of the cell's column, and the third being an object implementing the `DOMTokenList` interface, associated with an empty string. The second argument must be a non-negative integer less than the total number of columns. The tokens contained in the `DOMTokenList` object's underlying string when the method returns represent the classes that apply to that cell. If the method is not defined, no classes apply to the cell.

To establish 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 applies to the cell and the cell's checkbox is in an indeterminate state. When the `indeterminate` class applies to the cell, the checkbox is a tristate checkbox, and the user can set it to the indeterminate state. Otherwise, only the `checked` and/or `unchecked` classes apply to the cell, and the cell can only be toggled between those two states.

If the data provider ever raises an exception while the `datagrid` is invoking one of its methods, the `datagrid` must act, for the purposes of that particular method call, as if the relevant method had not been defined.

A `RowSpecification` object p with n path components passed to a method of the data provider must fulfill the constraint $0 \leq p_i < m\text{-}1$ for all integer values of i in the range $0 \leq i < n\text{-}1$, where m is the value that was last returned by the `getRowCount()` method when it was passed the `RowSpecification` object q with i-1 items, where $p_i = q_i$ for all integer values of i in the range $0 \leq i < n$-1, with any changes implied by the update methods taken into account.

The data model is considered stable: user agents may assume that subsequent calls to the data provider methods will return the same data, until one of the update methods is called on the `datagrid` element. If a user agent is returned inconsistent data, for example if the number of rows returned by `getRowCount()` varies in ways that do not match the calls made to the update methods, the user agent may disable the `datagrid`. User agents that do not disable the `datagrid` in inconsistent cases must honor the most recently returned values.

User agents may cache returned values so that the data provider is never asked for data that could contradict earlier data. User agents must not cache the return value of the `getRowMenu` method.

The exact algorithm used to populate the data grid is not defined here, since it will differ based on the presentation used. However, the behavior of user agents must be consistent with the descriptions above. For example, it would be non-conformant for a user agent to make cells have both a checkbox and be editable, as the descriptions above state that cells that have a checkbox cannot be edited.

4.11.2.6. Updating the `datagrid`

Whenever the `data` attribute is set to a new value, the `datagrid` must clear the current selection, remove all the displayed rows, and plan to repopulate itself using the information from the new data provider at the earliest opportunity.

There are a number of update methods that can be invoked on the `datagrid` element to cause it to refresh itself in slightly less drastic ways:

When the **`updateEverything()`** method is called, the user agent must repopulate the entire `datagrid`. If

the number of rows decreased, the selection must be updated appropriately. If the number of rows increased, the new rows should be left unselected.

When the **updateRowsChanged(*row, count*)** method is called, the user agent must refresh the rendering of the rows starting from the row specified by *row*, and including the *count* next siblings of the row (or as many next siblings as it has, if that is less than *count*), including all descendant rows.

When the **updateRowsInserted(*row, count*)** method is called, the user agent must assume that *count* new rows have been inserted, such that the first new row is identified by *row*. The user agent must update its rendering and the selection accordingly. The new rows should not be selected.

When the **updateRowsRemoved(*row, count*)** method is called, the user agent must assume that *count* rows have been removed starting from the row that used to be identifier by *row*. The user agent must update its rendering and the selection accordingly.

The **updateRowChanged(*row*)** method must be exactly equivalent to calling `updateRowsChanged(`*row*`, 1)`.

When the **updateColumnChanged(*column*)** method is called, the user agent must refresh the rendering of the specified column *column*, for all rows.

When the **updateCellChanged(*row, column*)** method is called, the user agent must refresh the rendering of the cell on row *row*, in column *column*.

Any effects the update methods have on the `datagrid`'s selection is not considered a change to the selection, and must therefore not fire the `select` event.

These update methods should be called only by the data provider, or code acting on behalf of the data provider. In particular, calling the `updateRowsInserted()` and `updateRowsRemoved()` methods without actually inserting or removing rows from the data provider is likely to result in inconsistent renderings, and the user agent is likely to disable the data grid.

4.11.2.7. Requirements for interactive user agents

This section only applies to interactive user agents.

If the `datagrid` element has a **disabled** attribute, then the user agent must disable the `datagrid`, preventing the user from interacting with it. The `datagrid` element should still continue to update itself when the data provider signals changes to the data, though. Obviously, conformance requirements stating that `datagrid` elements must react to users in particular ways do not apply when one is disabled.

If a row is openable, then the user 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" behavior of a cell, then the `datagrid` must invoke the data provider's `cycleCell()` method, with a `RowSpecification` object representing the cell's row as the first argument and the cell's column index as the second. The `datagrid` must 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 be possible only 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.

4.11.2.8. The selection

This section only applies to interactive user agents. For other user agents, the `selection` attribute must return null.

```
interface DataGridSelection {
  readonly attribute unsigned long length;
  [IndexGetter] RowSpecification item(in unsigned long index);
  boolean isSelected(in RowSpecification row);
  void setSelected(in RowSpecification row, in boolean selected);

  void selectAll();
  void invert();
  void clear();
};
```

Each `datagrid` element must keep track of which rows are currently selected. Initially no rows are selected, but this can be changed via the methods described in this section.

The selection of a `datagrid` is represented by its **`selection`** DOM attribute, which must be a `DataGridSelection` object.

`DataGridSelection` objects represent the rows in the selection. In the selection the rows must be ordered in the natural order of the data provider (and not, e.g., the rendered order). Rows that are not rendered because one of their ancestors is closed must share the same selection state as their nearest rendered ancestor. Such rows are not considered part of the selection for the purposes of iterating over the selection.

> ***Note: This selection API doesn't allow for hidden rows to be selected because it is trivial to create a data provider that has infinite depth, which would then require the selection to be infinite if every row, including every hidden row, was selected.***

The **`length`** attribute must return the number of rows currently present in the selection. 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 not be able to select more than a single row at a time, and selecting another one must unselect all the other rows.

> ***Note: This only applies to the user. Scripts can select multiple rows even when the `multiple` attribute is absent.***

Whenever the selection of a `datagrid` changes, whether due to the user interacting with the element, or as a result of calls to methods of the `selection` object, a **`select`** event that bubbles but is not cancelable must be fired on the `datagrid` element. If changes are made to the selection via calls to the object's methods during the execution of a script, then the `select` events must be coalesced into one, which must then be fired when the script execution has completed.

> ***Note: The `DataGridSelection` interface has no relation to the `Selection` interface.***

4.11.2.9. Columns and captions

This section only applies to interactive user agents.

Each `datagrid` element must keep track of which columns are currently being rendered. User agents should initially show all the columns except those with the `initially-hidden` class, but may allow users to hide or show columns. User agents should initially display the columns in the order given by the data provider, but may allow this order to be changed by the user.

If columns are not being used, as might be the case if the data grid is being presented in an icon view, or if an overview of data is being read in an aural context, then the text of the first column of each row should be used to represent the row.

If none of the columns have any captions (i.e. if the data provider does not provide a `getCaptionText()` method), then user agents may avoid showing the column headers at all. This may prevent the user from performing actions on the columns (such as reordering them, changing the sort column, and so on).

Note: Whatever the order used for rendering, and irrespective of what columns are being shown or hidden, the "first column" as referred to in this specification is always the column with index zero, and the "last column" is always the column with the index one less than the value returned by the `getColumnCount()` method of the data provider.

If a column is sortable, then the user 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.

4.11.3 The `command` element

Categories

Metadata content.
Phrasing content.

Contexts in which this element may be used:

Where metadata content is expected.
Where phrasing content is expected.

Content model:

Empty.

Element-specific attributes:

`type`
`label`
`icon`
`hidden`
`disabled`
`checked`
`radiogroup`
`default`
Also, the `title` attribute has special semantics on this element.

DOM interface:

```
interface HTMLCommandElement : HTMLElement {
           attribute DOMString type;
           attribute DOMString label;
           attribute DOMString icon;
           attribute boolean 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 valid URL.

The **`hidden`** attribute is a boolean attribute that, if present, indicates that the command is not relevant and is to be hidden.

The **`disabled`** attribute is a boolean attribute that, if present, indicates that the command is not available in the current state.

> ***Note: The distinction between Disabled 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 is a boolean attribute that, if present, indicates that the command is selected.

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. The `default` attribute is a boolean attribute.

> 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 the content attributes of the same name.

The **`click()`** method's behavior depends on the value of the `type` attribute of the element, as follows:

↪ **If the `type` attribute 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 triggered 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.

4.11.4 The `bb` element

Categories

Phrasing content.
Interactive content.

Contexts in which this element may be used:

Where phrasing content is expected.

Phrasing content.

Empty.

Element-specific attributes:

`type`

DOM interface:

```
interface HTMLBrowserButtonElement : HTMLElement {
           attribute DOMString type;
  readonly attribute boolean supported;
```

```
  readonly attribute boolean disabled;
};
```

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

The `bb` element represents a user agent command that the user can invoke.

The **`type`** attribute indicates the kind of command. The `type` attribute is an enumerated attribute. The following table lists the keywords and states for the attribute — the keywords in the left column map to the states listed in the cell in the second column on the same row as the keyword.

Keyword	State
`makeapp`	*make application*

The missing value default state is the *null* state.

Each state has an *action* and a *relevance*, defined in the following sections.

When the attribute is in the ***null*** state, the *action* is to not do anything, and the *relevance* is unconditionally false.

A `bb` element whose `type` attribute is in a state whose *relevance* is true must be enabled. Conversely, a `bb` element whose `type` attribute is in a state whose *relevance* is false must be disabled.

If a `bb` element is enabled, it must match the `:enabled` pseudo-class; otherwise, it must match the `:disabled` pseudo-class.

User agents should allow users to invoke `bb` elements when they are enabled. When a user invokes a `bb` element, its `type` attribute's state's *action* must be invoked.

When the element has no descendant element children and has no descendant text node children of non-zero length, the element represents a browser button with a user-agent-defined icon or text representing the `type` attribute's state's *action* and *relevance* (enabled vs disabled). Otherwise, the element represents its descendants.

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

The **`supported`** DOM attribute must return true if the `type` attribute is in a state other than the *null* state and the user agent supports that state's *action* (i.e. when the attribute's value is one that the user agent recognises and supports), and false otherwise.

The **`disabled`** DOM attribute must return true if the element is disabled, and false otherwise (i.e. it returns the opposite of the `type` attribute's state's *relevance*).

4.11.4.1. Browser button types

4.11.4.1.1. The make application state

Some user agents support making sites accessible as independent applications, as if they were not Web sites at all. The *make application* state exists to allow Web pages to offer themselves to the user as targets for this mode of operation.

The *action* of the *make application* state is to confirm the user's intent to use the current site in a standalone fashion, and, provided the user's intent is confirmed, offer the user a way to make the resource identified by the document's address available in such a fashion.

⚠Warning! ***The confirmation is needed because it is relatively easy to trick users into activating buttons. The confirmation could, e.g. take the form of asking the user where to "save" the application, or non-modal information panel that is clearly from the user agent and gives the user the opportunity to drag an icon to their system's application launcher.***

The *relevance* of the *make application* state is false if the user agent is already handling the site in such a fashion, or if the user agent doesn't support making the site available in that fashion, and true otherwise.

> In the following example, a few links are listed on an application's page, to allow the user perform certain actions, including making the application standalone:
>
> ```
> <menu>
>  <li><a href="settings.html"
> onclick="panels.show('settings')">Settings</a>
>  <li><bb type="makeapp">Download standalone application</a>
>  <li><a href="help.html" onclick="panels.show('help')">Help</a>
>  <li><a href="logout.html" onclick="panels.show('logout')">Sign out</a>
> </menu>
> ```
>
> With the following stylesheet, it could be make to look like a single line of text with vertical bars separating the options, with the "make app" option disappearing when it's not supported or relevant:
>
> ```
> menu li { display: none; }
> menu li:enabled { display: inline; }
> menu li:not(:first-child)::before { content: ' | '; }
> ```
>
> This could look like this:
>
> 
>

> The following example shows another way to do the same thing as the previous one, this time not relying on CSS support to hide the "make app" link if it doesn't apply:
>
> ```
> <menu>
>  <a href="settings.html" onclick="panels.show('settings')">Settings</a> |
>  <bb type="makeapp" id="makeapp"> </bb>
>  <a href="help.html" onclick="panels.show('help')">Help</a> |
>  <a href="logout.html" onclick="panels.show('logout')">Sign out</a>
> </menu>
> <script>
>  var bb = document.getElementById('makeapp');
>  if (bb.supported && bb.enabled) {
>    bb.parentNode.nextSibling.textContent = ' | ';
>    bb.textContent = 'Download standalone application';
>  } else {
>    bb.parentNode.removeChild(bb);
>  }
> </script>
> ```

4.11.5 The `menu` element

Categories

Flow content.

If there is a `menu` element ancestor: phrasing content.

Contexts in which this element may be used:

Where flow content is expected.

If there is a `menu` element ancestor: where phrasing content is expected.

Content model:

Either: Zero or more `li` elements.

Or: Phrasing content.

Element-specific attributes:

`type`
`label`
`autosubmit`

DOM interface:

```
interface HTMLMenuElement : HTMLElement {
           attribute DOMString type;
           attribute DOMString label;
           attribute boolean autosubmit;
};
```

The `menu` element represents a list of commands.

The **`type`** attribute is an enumerated attribute indicating the kind of menu being declared. The attribute has three states. The `context` keyword maps to the **context menu** state, in which the element is declaring a context menu. The `toolbar` keyword maps to the **tool bar** state, in which the element is declaring a tool bar. The attribute may also be omitted. The *missing value default* is the **list** state, which indicates that the element is merely a list of commands that is neither declaring a context menu nor defining a tool bar.

If a `menu` element's `type` attribute is in the context menu state, then the element represents the commands of a context menu, and the user can only interact with the commands if that context menu is activated.

If a `menu` element's `type` attribute is in the tool bar state, then the element represents a list of active commands that the user can immediately interact with.

If a `menu` element's `type` attribute is in the list state, then the element either represents an unordered list of items (each represented by an `li` element), each of which represents a command that the user may perform or activate, or, if the element has no `li` element children, flow content describing available commands.

The **`label`** attribute gives the label of the menu. It is used by user agents to display nested menus in the UI. For example, a context menu containing another menu would use the nested menu's `label` attribute for the submenu's menu label.

The **`autosubmit`** attribute is a boolean attribute that, if present, indicates that selections made to form

controls in this menu are to result in the control's form being immediately submitted.

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 **`type`**, **`label`**, and **`autosubmit`** DOM attributes must reflect the content attributes of the same name.

4.11.5.1. Introduction

This section is non-normative.

...

4.11.5.2. Building menus and tool bars

A menu (or tool bar) consists of a list of zero or more of the following components:

- Commands, which can be marked as default commands
- Separators
- Other menus (which allows the list to be nested)

The list corresponding to a particular `menu` element is built by iterating over its child nodes. For each child node in tree order, the required behavior depends on what the node is, as follows:

↪ **An element that defines a command**

Append the command to the menu. 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.

4.11.5.3. Context menus

The **`contextmenu`** attribute gives the element's context menu. The value must be the ID of a `menu` element in the DOM. If the node that would be obtained by the invoking the `getElementById()` method using the attribute's value as the only argument is null or not a `menu` element, then the element has no assigned context menu. Otherwise, the element's assigned context menu is the element so identified.

When an element's context menu is requested (e.g. by the user right-clicking the element, or pressing a context menu key), the UA must fire a `contextmenu` event on the element for which the menu was requested.

> ***Note: Typically, therefore, the firing of the `contextmenu` event will be the default action of a `mouseup` or `keyup` event. The exact sequence of events is UA-dependent, as it will vary based on platform conventions.***

The default action of the `contextmenu` event depends on whether the element has a context menu assigned (using the `contextmenu` attribute) or not. If it does not, the default action must be for the user agent to show its default context menu, if it has one.

Context menus should inherit (so clicking on a span in a paragraph with a context menu should show the menu).

If the element *does* have a context menu assigned, then the user agent must fire a `show` event on the relevant `menu` element.

The default action of *this* event is that the user agent must show a context menu built from the `menu` element.

The user agent may also provide access to its default context menu, if any, with the context menu shown. For example, it could merge the menu items from the two menus together, or provide the page's context menu as a submenu of the default menu.

If the user dismisses the menu without making a selection, nothing in particular happens.

If the user selects a menu item that represents a command, then the UA must invoke that command's Action.

Context menus must not, while being shown, reflect changes in the DOM; they are constructed as the default

action of the `show` event and must remain like that until dismissed.

User agents may provide means for bypassing the context menu processing model, ensuring that the user can always access the UA's default context menus. For example, the user agent could handle right-clicks that have the Shift key depressed in such a way that it does not fire the `contextmenu` event and instead always shows the default context menu.

The **`contextMenu`** attribute must reflect the `contextmenu` content attribute.

4.11.5.4. 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.

4.11.6 Commands

A **command** is the abstraction behind menu items, buttons, and links. Once a command is defined, other parts of the interface can refer to the same command, allowing many access points to a single feature to share aspects such as the disabled state.

Commands are defined to have the following *facets*:

Type

The kind of command: "command", meaning it is a normal command; "radio", meaning that triggering the command will, amongst other things, set the Checked State to true (and probably uncheck some other commands); or "checkbox", meaning that triggering the command will, amongst other things, toggle the value of the Checked State.

ID

The name of the command, for referring to the command from the markup or from script. If a command has no ID, it is an **anonymous command**.

Label

The name of the command as seen by the user.

Hint

A helpful or descriptive string that can be shown to the user.

Icon

An absolute URL identifying a graphical image that represents the action. A command might not have an Icon.

Hidden State

Whether the command is hidden or not (basically, whether it should be shown in menus).

Disabled State

Whether the command 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 URL 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:

Actually even better would be to just mix it straight into those interfaces somehow.

```
[NoInterfaceObject] interface Command {
  readonly attribute DOMString commandType;
  readonly attribute DOMString id;
  readonly attribute DOMString label;
  readonly attribute DOMString title;
  readonly attribute DOMString icon;
  readonly attribute boolean hidden;
  readonly attribute boolean disabled;
  readonly attribute boolean checked;
  void click();
  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 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 the absolute URL of the command's Icon. If the element does not specify an icon, or if the element does not define a command, then the attribute must return null. This attribute will be shadowed by the `icon` DOM attribute on `command` elements.

The **`hidden`** attribute must return true if the command's Hidden State is that the command is hidden, and false if it is that the command is not hidden. If the element does not define a command, the attribute must return false. This attribute will be shadowed by the `hidden` DOM attribute on `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`.

4.11.6.1. Using the `a` element to define a command

An `a` element with an `href` attribute defines a command.

The Type of the command is "command".

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the string given by the element's `textContent` DOM attribute.

The Hint of the command is the value of the `title` attribute of the element. If the attribute is not present, the Hint is the empty string.

The Icon of the command is the absolute URL obtained from resolving the value of the `src` attribute of the first `img` element descendant of the element, if there is such an element and resolving its attribute is successful. Otherwise, there is no Icon for the command.

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.

4.11.6.2. Using the `button` element to define a command

A `button` element always defines a command.

The Type, ID, Label, Hint, Icon, Hidden State, Checked State, and Action facets of the command are determined as for `a` elements (see the previous section).

The Disabled State of the command mirrors the disabled state of the button. 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).

4.11.6.3. 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.

4.11.6.4. Using the `option` element to define a command

An `option` element with an ancestor `select` element and either no `value` attribute or a `value` attribute that is not the empty string defines a command.

The Type of the command is "radio" if the `option`'s nearest ancestor `select` element has no `multiple` attribute, and "checkbox" if it does.

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the value of the `option` element's `label` attribute, if there is one, or the value of the `option` element's `textContent` DOM attribute if 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.

4.11.6.5. Using the `command` element to define a command

A `command` element defines a command.

The Type of the command is "radio" if the `command`'s `type` attribute is "`radio`", "checkbox" if the attribute's value is "`checkbox`", and "command" otherwise.

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the value of the element's `label` attribute, if there is one, or the empty string if it doesn't.

The Hint of the command is the string given by the element's `title` attribute, if any, and the empty string if the attribute is absent.

The Icon for the command is the absolute URL obtained from resolving the value of the element's `icon` attribute, if it has such an attribute and resolving it is successful. Otherwise, there is no Icon for the command.

The Hidden State of the command is true (hidden) if the element has a `hidden` attribute, and false otherwise.

The Disabled State of the command is true (disabled) if the element has 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 behavior described in the definition of the `click()` method of the `HTMLCommandElement` interface.

4.11.6.6. Using the `bb` element to define a command

A `bb` element always defines a command.

The Type of the command is "command".

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the string given by the element's `textContent` DOM attribute, if that is not the empty string, or a user-agent-defined string appropriate for the `bb` element's `type` attribute's state.

The Hint of the command is the value of the `title` attribute of the element. If the attribute is not present, the Hint is a user-agent-defined string appropriate for the `bb` element's `type` attribute's state.

The Icon of the command is the absolute URL obtained from resolving the value of the `src` attribute of the first `img` element descendant of the element, if there is such an element and resolving its attribute is successful. Otherwise, the Icon is a user-agent-defined image appropriate for the `bb` element's `type` attribute's state.

The Hidden State facet of the command is true if the `bb` element's `type` attribute's state is not *null*.

The Disabled State facet of the command is true if the `bb` element's `type` attribute's state's *relevance* is false, and true otherwise.

The Checked State of the command is always false. (The command is never checked.)

The Action of the command is to perform the *action* of the `bb` element's `type` attribute's state.

4.12 Data Templates

4.12.1 Introduction

This section is non-normative.

...examples...

4.12.2 The `datatemplate` element

Categories

Metadata content.
Flow content.

Contexts in which this element may be used:

As the root element of an XML document.
Where metadata content is expected.
Where flow content is expected.

Content model:

Zero or more `rule` elements.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `datatemplate` element brings together the various rules that form a data template. The element doesn't itself do anything exciting.

4.12.3 The `rule` element

Categories

None.

Contexts in which this element may be used:

As a child of a `datatemplate` element.

Content model:

Anything, regardless of the children's required contexts (but see prose).

Element-specific attributes:

`condition`
`mode`

DOM interface:

```
interface HTMLRuleElement : HTMLElement {
           attribute DOMString condition;
           attribute DOMString mode;
  readonly attribute DOMTokenString modeList;
};
```

The `rule` element represents a template of content that is to be used for elements when updating an element's generated content.

The **`condition`** attribute, if specified, must contain a valid selector. It specifies which nodes in the data tree

will have the condition's template applied. [SELECTORS]

If the `condition` attribute is not specified, then the condition applies to all elements, text nodes, `CDATASection` nodes, and processing instructions.

The **`mode`** attribute, if specified, must have a value that is an unordered set of unique space-separated tokens representing the various modes for which the rule applies. When, and only when, the `mode` attribute is omitted, the rule applies if and only if the mode is the empty string. A mode is invoked by the `nest` element; for the first node (the root node) of the data tree, the mode is the empty string.

The contents of `rule` elements form a template, and may be anything that, when the parent `datatemplate` is applied to some conforming data, results in a conforming DOM tree.

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

The **`mode`** and **`modeList`** DOM attributes must reflect the `mode` content attribute.

4.12.4 The `nest` element

Categories

None.

Contexts in which this element may be used:

As a child of a descendant of a `rule` element, regardless of the element's content model.

Content model:

Empty.

Element-specific attributes:

`filter`
`mode`

DOM interface:

```
interface HTMLNestElement : HTMLElement {
           attribute DOMString filter;
           attribute DOMString mode;
};
```

The `nest` element represents a point in a template where the user agent should recurse and start inserting the children of the data node that matches the `rule` in which the `nest` element finds itself.

The **`filter`** attribute, if specified, must contain a valid selector. It specifies which of the child nodes in the data tree will be examined for further processing at this point. [SELECTORS]

If the `filter` attribute is not specified, then all elements, text nodes, `CDATASection` nodes, and processing instructions are processed.

The **`mode`** attribute, if specified, must have a value that is a word token consisting of one or more characters, none of which are space characters. It gives the mode which will be in effect when looking at the rules in the

data template.

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

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

4.12.5 Global attributes for data templates

The **template** attribute may be added to an element to indicate that the template processing model is to be applied to that element.

The `template` attribute, when specified, must be a valid URL to an XML or HTML document, or a fragment identifier pointing at another part of the document. If there is a fragment identifier present, then the element with that ID in the target document must be a `datatemplate` element, otherwise, the root element must be a `datatemplate` element.

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

The **ref** attribute may be specified on any element on which the `template` attribute is specified. If it is specified, it must be a valid URL to an XML or HTML document, or a fragment identifier pointing at another part of the document.

When an element has a `template` attribute but no `ref` attribute, the element may, instead of its usual content model, have a single element of any kind. That element is then used as the root node of the data for the template.

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

The **registrationmark** attribute may be specified on any element that is a descendant of a `rule` element, except `nest` elements. Its value may be any string, including the empty string (which is the value that is assumed if the attribute is omitted). This attribute performs a role similar to registration marks in printing presses: when the generated content is regenerated, elements with the same `registrationmark` are lined up. This allows the author to disambiguate how elements should be moved around when generated content is regenerated in the face of changes to the data tree.

The **registrationMark** DOM attribute must reflect the `registrationmark` content attribute.

4.12.6 Processing model

4.12.6.1. The `originalContent` DOM attribute

The **originalContent** is set to a `DocumentFragment` to hold the original children of an element that has been replaced by content generated for a data template. Initially, it must be null. Its value is set when the `template` attribute is set to a usable value, and is unset when the attribute is removed.

> ***Note: The `originalContent` DOM attribute can thus be used as an indicator of whether a template is currently being applied, just as the `templateElement` DOM attribute can.***

4.12.6.2. The `template` attribute

Setting: When an HTML element without a `template` attribute has its `template` attribute set, the user agent must fetch the specified file and parse it (without a browsing context) to obtain a DOM. If the URL, when resolved, is the same as the document's address, then the current document's DOM must be assumed to be that parsed DOM. While this loading and parsing is in progress, the element is said to be *busy loading the template rules or data*.

If the resource specified by the `template` attribute is not the current document and does not have an XML MIME type, or if an XML parse error is found while parsing the resource, then the resource cannot be successfully parsed, and the user agent must jump to the failed to parse steps below.

Once the DOM in question has been parsed, assuming that it indeed can be parsed and does so successfully, the user agent must wait for no scripts to be executing, and as soon as that opportunity arises, run the following algorithm:

1. If the `template` attribute's value has a fragment identifier, and, in the DOM in question, it identifies a `datatemplate` element, then set the `templateElement` DOM attribute to that element.

   Otherwise, if the `template` attribute value does not have a fragment identifier, and the root element of the DOM in question is a `datatemplate` element, then set the `templateElement` DOM attribute to that element.

   Otherwise, jump to the failed to parse steps below.

2. Create a new `DocumentFragment` and move all the nodes that are children of the element to that `DocumentFragment` object. Set the `originalContent` DOM attribute on the element to this new `DocumentFragment` object.

3. Jump to the steps below for updating the generated content.

If the resource has **failed to parse**, the user agent must fire a simple event with the name `error` at the element on which the `template` attribute was found.

Unsetting: When an HTML element with a `template` attribute has its `template` attribute removed or dynamically changed from one value to another, the user agent must run the following algorithm:

1. Set the `templateElement` DOM attribute to null.

2. If the `originalContent` DOM attribute of the element is not null, run these substeps:

   1. Remove all the nodes that are children of the element.

   2. Append the nodes in the `originalContent` `DocumentFragment` to the element.

   3. Set `originalContent` to null.

   (If the `originalContent` DOM attribute of the element is null, then either there was an error loading or parsing the previous template, or the previous template never finished loading; in either case, there is nothing to undo.)

3. If the `template` attribute was changed (as opposed to simply removed), then act as if it was now set to its new value (fetching the specified page, etc, as described above).

The **`templateElement`** DOM attribute is updated by the above algorithm to point to the currently active

`datatemplate` element. Initially, the attribute must have the value null.

4.12.6.3. The `ref` attribute

Setting: When an HTML element without a `ref` attribute has its `ref` attribute set, the user agent must fetch the specified file and parse it (without a browsing context) to obtain a DOM. If the URL, when resolved, is the same as the document's address, then the current document's DOM must be assumed to be that parsed DOM. While this loading and parsing is in progress, the element is said to be *busy loading the template rules or data*.

If the resource specified by the `ref` attribute is not the current document and does not have an XML MIME type, or if an XML parse error is found while parsing the resource, then the resource cannot be successfully parsed, and the user agent must jump to the failed to parse steps below.

Once the DOM in question has been parsed, assuming that it indeed can be parsed and does so successfully, the user agent must wait for no scripts to be executing, and as soon as that opportunity arises, run the following algorithm:

1. If the `ref` attribute value does not have a fragment identifier, then set the `refNode` DOM attribute to the `Document` node of that DOM.

   Otherwise, if the `ref` attribute's value has a fragment identifier, and, in the DOM in question, that fragment identifier identifies an element, then set the `refNode` DOM attribute to that element.

   Otherwise, jump to the failed to parse steps below.

2. Jump to the steps below for updating the generated content.

If the resource has **failed to parse**, the user agent must fire a simple event with the name `error` at the element on which the `ref` attribute was found, and must then jump to the steps below for updating the generated content (the contents of the element will be used instead of the specified resource).

Unsetting: When an HTML element with a `ref` attribute has its `ref` attribute removed or dynamically changed from one value to another, the user agent must run the following algorithm:

1. Set the `refNode` DOM attribute to null.

2. If the `ref` attribute was changed (as opposed to simply removed), then act as if it was now set to its new value (fetching the specified page, etc, as described above). Otherwise, jump to the steps below for updating the generated content.

The **`refNode`** DOM attribute is updated by the above algorithm to point to the current data tree, if one is specified explicitly. If it is null, then the data tree is given by the `originalContent` DOM attribute, unless that is also null, in which case no template is currently being applied. Initially, the attribute must have the value null.

4.12.6.4. The `NodeDataTemplate` interface

All objects that implement the `Node` interface must also implement the `NodeDataTemplate` interface, whose members must be accessible using binding-specific casting mechanisms.

```
interface NodeDataTemplate {
  readonly attribute Node dataNode;
};
```

The **dataNode** DOM attribute returns the node for which *this* node was generated. It must initially be null. It is set on the nodes that form the content generated during the algorithm for updating the generated content of elements that are using the data template feature.

4.12.6.5. Mutations

An element with a non-null `templateElement` is said to be a **data tree user** of the node identified by the element's `refNode` attribute, as well as all of that node's children, or, if that attribute is null, of the node identified by the element's `originalContent`, as well as all *that* node's children.

Nodes that have one or more data tree users associated with them (as per the previous paragraph) are themselves termed **data tree component nodes**.

Whenever a data tree component node changes its name or value, or has one of its attributes change name or value, or has an attribute added or removed, or has a child added or removed, the user agent must update the generated content of all of that node's data tree users.

An element with a non-null `templateElement` is also said to be a **template tree user** of the node identified by the element's `templateElement` attribute, as well as all of that node's children.

Nodes that have one or more template tree users associated with them (as per the previous paragraph) are themselves termed **template tree component nodes**.

Whenever a template tree component node changes its name or value, or has one of its attributes change name or value, or has an attribute added or removed, or has a child added or removed, the user agent must update the generated content of all of that node's template tree users.

> ***Note: In other words, user agents update the content generated from a template whenever either the backing data changes or the template itself changes.***

4.12.6.6. Updating the generated content

When the user agent is to **update the generated content** of an element that uses a template, the user agent must run the following steps:

1. Let *destination* be the element whose generated content is being updated.
2. If the *destination* element is *busy loading the template rules or data*, then abort these steps. Either the steps will be invoked again once the loading has completed, or the loading will fail and the generated content will be removed at that point.
3. Let *template tree* be the element given by *destination*'s `templateElement` DOM attribute. If it is null, then abort these steps. There are no rules to apply.
4. Let *data tree* be the node given by *destination*'s `refNode` DOM attribute. If it is null, then let *data tree* be the node given by the `originalContent` DOM node.

5. Let *existing nodes* be a set of ordered lists of nodes, each list being identified by a tuple consisting of a node, a node type and name, and a registration mark (a string).

6. For each node *node* that is a descendant of *destination*, if any, add *node* to the list identified by the tuple given by: *node*'s `dataNode` DOM attribute; the *node*'s node type and, if it's an element, its qualified name (that is, its namespace and local name), or, if it's a processing instruction, its target name, and the value of the *node*'s `registrationmark` attribute, if it has one, or the empty string otherwise.

7. Remove all the child nodes of *destination*, so that its child node list is empty.

8. Run the Levenberg data node algorithm (described below) using *destination* as the destination node, *data tree* as the source node, *template tree* as the rule container, the empty string as the mode, and the *existing nodes* lists as the lists of existing nodes.

The Levenberg algorithm consists of two algorithms that invoke each other recursively, the Levenberg data node algorithm and the Levenberg template node algorithm. These algorithms use the data structures initialized by the set of steps described above.

The **Levenberg data node algorithm** is as follows. It is always invoked with three DOM nodes, one string, and a set of lists as arguments: the *destination node*, the *source node*, the *rule container*, the *mode string*, and the *existing nodes lists* respectively.

1. Let *condition* be the first `rule` element child of the *rule container* element, or null if there aren't any.

2. If *condition* is null, follow these substeps:

    1. If the *source node* is an element, then, for each child *child node* of the *source node* element, in tree order, invoke the Levenberg data node algorithm recursively, with *destination node*, *child node*, *rule container*, the empty string, and *existing nodes lists* as the five arguments respectively.

    2. Abort the current instance of the Levenberg data node algorithm, returning to whatever algorithm invoked it.

3. Let *matches* be a boolean with the value true.

4. If the *condition* element has a `mode` attribute, but the value of that attribute is not a mode match for the current mode string, then let *matches* be false.

5. If the *condition* element has a `condition` attribute, and the attribute's value, when evaluated as a selector, does not match the current *source node*, then let *matches* be false.

6. If *matches* is true, then follow these substeps:

    1. For each child *child node* of the *condition* element, in tree order, invoke the Levenberg template node algorithm recursively, with the five arguments being *destination node*, *source node*, *rule container*, *child node*, and *existing nodes lists* respectively.

    2. Abort the current instance of the Levenberg data node algorithm, returning to whatever algorithm invoked it.

7. Let *condition* be the next `rule` element that is a child of the *rule container* element, after the *condition* element itself, or null if there are no more `rule` elements.

8. Jump to step 2 in this set of steps.

The **Levenberg template node algorithm** is as follows. It is always invoked with four DOM nodes and a set of lists as arguments: the *destination node*, the *source node*, the *rule container*, the *template node*, and the *existing nodes lists* respectively.

1. If *template node* is a comment node, abort the current instance of the Levenberg template node algorithm, returning to whatever algorithm invoked it.
2. If *template node* is a `nest` element, then run these substeps:
    1. If *source node* is not an element, then abort the current instance of the Levenberg template node algorithm, returning to whatever algorithm invoked it.
    2. If the *template node* has a `mode` attribute, then let *mode* be the value of that attribute; otherwise, let *mode* be the empty string.
    3. Let *child node* be the first child of the *source node* element, or null if *source node* has no children.
    4. If *child node* is null, abort the current instance of the Levenberg template node algorithm, returning to whatever algorithm invoked it.
    5. If the *template node* element has a `filter` attribute, and the attribute's value, when evaluated as a selector, matches *child node*, then invoke the Levenberg data node algorithm recursively, with *destination node*, *child node*, *rule container*, *mode*, and *existing nodes lists* as the five arguments respectively.
    6. Let *child node* be *child node*'s next sibling, or null if *child node* was the last node of *source node*.
    7. Return to step 4 in this set of substeps.
3. If *template node* is an element, and that element has a `registrationmark` attribute, then let *registration mark* have the value of that attribute. Otherwise, let *registration mark* be the empty string.
4. If there is a list in the *existing nodes lists* corresponding to the tuple (*source node*, the node type and name of *template node*, *registration mark*), and that list is not empty, then run the following substeps. (For an element node, the name of the node is its qualified tag name, i.e. its namespace and local name. For a processing instruction, its name is the target. For other types of nodes, there is no name.)
    1. Let *new node* be the first node in that list.
    2. Remove *new node* from that list.
    3. If *new node* is an element, remove all the child nodes of *new node*, so that its child node list is empty.

    Otherwise, if there is no matching list, or there was, but it is now empty, then run these steps instead:

    1. Let *new node* be a shallow clone of *template node*.
    2. Let *new node*'s `dataNode` DOM attribute be *source node*.
5. If *new node* is an element, run these substeps:

1. For each attribute on *new node*, if an attribute with the same qualified name is not present on *template node*, remove that attribute.

2. For each attribute *attribute* on *template node*, run these substeps:

    1. Let *expanded* be the result of passing the value of *attribute* to the text expansion algorithm for templates along with *source node*.

    2. If an attribute with the same qualified name as *attribute* is already present on *new node*, then: if its value is different from *expanded*, replace its value with *expanded*.

    3. Otherwise, if there is no attribute with the same qualified name as *attribute* on *new node*, then add an attribute with the same namespace, prefix, and local name as *attribute*, with its value set to *expanded*'s.

Otherwise, the *new node* is a text node, `CDATASection` node, or processing instruction. Run these substeps instead:

1. Let *expanded* be the result of passing the node value of *template node* (the content of the text node, `CDATASection` node, or processing instruction) to the text expansion algorithm for templates along with *source node*.

2. If the value of the *new node* is different from *expanded*, then set the value of *new node* to *expanded*.

6. Append *new node* to *destination*.

7. If *template node* is an element, then, for each child *child node* of the *template node* element, in tree order, invoke the Levenberg template node algorithm recursively, with the five arguments being *new child*, *source node*, *rule container*, *child node*, and *existing nodes lists* respectively.

Define: **evaluated as a selector**

Define: **text expansion algorithm for templates**

4.13 Miscellaneous elements

4.13.1 The `legend` element

Categories

None.

Contexts in which this element may be used:

As the first child of a `fieldset` element.
As the first child of a `details` element.
As a child of a `figure` element, if there are no other `legend` element children of that element.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

Uses `HTMLElement`.

The `legend` element represents a title or explanatory caption for the rest of the contents of the `legend` element's parent element.

4.13.2 The `div` element

Categories

None.

Contexts in which this element may be used:

Where flow content is expected.

Content model:

Flow content.

Element-specific attributes:

None.

DOM interface:

Uses `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.

Allowing `div` elements to contain phrasing content makes it easy for authors to abuse `div`, using it with the `class=""` attribute to the point of not having any other elements in the markup. This is a disaster from an accessibility point of view, and it would be nice if we could somehow make such pages non-compliant without preventing people from using `div`s as the extension mechanism that they are, to handle things the spec can't otherwise do (like making new widgets).

5. Web browsers

This section describes features that apply most directly to Web browsers. Having said that, unless specified elsewhere, the requirements defined in this section *do* apply to all user agents, whether they are Web browsers or not.

5.1 Browsing contexts

A **browsing context** is a collection of one or more `Document` objects, and one or more views.

At any one time, one of the `Document`s in a browsing context is the **active document**. The collection of `Document`s is the browsing context's session history.

A **view** is a user agent interface tied to a particular media used for the presentation of `Document` objects in some media. A view may be interactive. Each view is represented by an `AbstractView` object. Each view belongs to a browsing context. [DOM2VIEWS]

> ***Note: The `document` attribute of an `AbstractView` object representing a view gives the `Document` object of the view's browsing context's active document. [DOM2VIEWS]***

> ***Note: Events that use the `UIEvent` interface are related to a specific view (the view in which the event happened); the `AbstractView` of that view is given in the event object's `view` attribute. [DOM3EVENTS]***

> ***Note: A typical Web browser has one obvious view per browsing context: the browser's window (screen media). If a page is printed, however, a second view becomes evident, that of the print media. The two views always share the same underlying `Document`, but they have a different presentation of that document. A speech browser also establishes a browsing context, one with a view in the speech media.***

> ***Note: A `Document` does not necessarily have a browsing context associated with it. In particular, data mining tools are likely to never instantiate browsing contexts.***

The main view through which a user primarily interacts with a user agent is the **default view**.

> ***Note: The default view of a `Document` is given by the `defaultView` attribute on the `Document` object's `DocumentView` interface. [DOM3VIEWS]***

When a browsing context is first created, it must be created with a single `Document` in its session history, whose address is `about:blank`, which is marked as being an HTML document, and whose character encoding is UTF-8. The `Document` must have a single child `html` node, which itself has a single child `body` node. If the browsing context is created specifically to be immediately navigated, then that initial navigation will have replacement enabled.

The origin of the `about:blank` `Document` is set when the `Document` is created, in a manner dependent on whether the browsing context created is a nested browsing context, as follows:

↪ **If the new browsing context is a nested browsing context**

The origin of the `about:blank` `Document` is the origin of the active document of the new browsing context's parent browsing context at the time of its creation.

↪ **If the new browsing context is an auxiliary browsing context**

The origin of the `about:blank` `Document` is the origin of the active document of the new browsing context's opener browsing context at the time of the new browsing context's creation.

↪ **Otherwise**

The origin of the `about:blank` `Document` is a globally unique identifier assigned when the new browsing context is created.

5.1.1 Nested browsing contexts

Certain elements (for example, `iframe` elements) can instantiate further browsing contexts. These are called **nested browsing contexts**. If a browsing context *P* has an element in one of its `Document`s *D* that nests another browsing context *C* inside it, then *P* is said to be the **parent browsing context** of *C*, *C* is said to be a **child browsing context** of *P*, and *C* is said to be **nested through** *D*.

A browsing context *A* is said to be an ancestor of a browsing context *B* if there exists a browsing context *A'* that is a child browsing context of *A* and that is itself an ancestor of *B*, or if there is a browsing context *P* that is a child browsing context of *A* and that is the parent browsing context of *B*.

The browsing context with no parent browsing context is the **top-level browsing context** of all the browsing contexts nested within it (either directly or indirectly through other nested browsing contexts).

The transitive closure of parent browsing contexts for a nested browsing context gives the list of **ancestor browsing contexts**.

A `Document` is said to be **fully active** when it is the active document of its browsing context, and either its browsing context is a top-level browsing context, or the `Document` through which that browsing context is nested is itself fully active.

Because they are nested through an element, child browsing contexts are always tied to a specific `Document` in their parent browsing context. User agents must not allow the user to interact with child browsing contexts of elements that are in `Document`s that are not themselves fully active.

A nested browsing context can have a seamless browsing context flag set, if it is embedded through an `iframe` element with a `seamless` attribute.

The **`parent`** DOM attribute on the `Window` object of a browsing context *b* must return the `Window` object of the parent browsing context, if there is one (i.e. if *b* is a child browsing context), or the `Window` object of the browsing context *b* itself, otherwise (i.e. if it is a top-level browsing context).

5.1.2 Auxiliary browsing contexts

It is possible to create new browsing contexts that are related to a top level browsing context without being nested through an element. Such browsing contexts are called **auxiliary browsing contexts**. Auxiliary browsing contexts are always top-level browsing contexts.

An auxiliary browsing context has an **opener browsing context**, which is the browsing context from which the auxiliary browsing context was created, and it has a **furthest ancestor browsing context**, which is the top-level browsing context of the opener browsing context when the auxiliary browsing context was created.

The **`opener`** DOM attribute on the `Window` object must return the `Window` object of the browsing context from which the current browsing context was created (its opener browsing context), if there is one and it is still available.

5.1.3 Secondary browsing contexts

User agents may support **secondary browsing contexts**, which are browsing contexts that form part of the user agent's interface, apart from the main content area.

5.1.4 Security

A browsing context *A* is **allowed to navigate** a second browsing context *B* if one of the following conditions is true:

- Either the origin of the active document of *A* is the same as the origin of the active document of *B*, or
- The browsing context *B* is an auxiliary browsing context and either its opener browsing context is *A* or *A* is allowed to navigate *B*'s opener browsing context, or
- The browsing context *B* is not a top-level browsing context, but there exists an ancestor browsing context of *B* whose active document has the same origin as the active document of *A* (possibly in fact being *A* itself).

5.1.5 Threads

Each browsing context is defined as having a list of zero or more **directly reachable browsing contexts**. These are:

- All the browsing context's child browsing contexts.
- The browsing context's parent browsing context.
- All the browsing contexts that have the browsing context as their opener browsing context.
- The browsing context's opener browsing context.

The transitive closure of all the browsing contexts that are directly reachable browsing contexts forms a **unit of related browsing contexts**.

All the executable code in a unit of related browsing contexts must execute on a single conceptual thread. The dispatch of events fired by the user agent (e.g. in response to user actions or network activity) and the execution of any scripts associated with timers must be serialized so that for each unit of related browsing contexts there is only one script being executed at a time.

5.1.6 Browsing context names

Browsing contexts can have a **browsing context name**. By default, a browsing context has no name (its name is not set).

A **valid browsing context name** is any string with at least one character that does not start with a U+005F LOW LINE character. (Names starting with an underscore are reserved for special keywords.)

A **valid browsing context name or keyword** is any string that is either a valid browsing context name or that case-insensitively matches one of: `_blank`, `_self`, `_parent`, or `_top`.

The rules for choosing a browsing context given a browsing context name are as follows. The rules assume that they are being applied in the context of a browsing context.

1. If the given browsing context name is the empty string or `_self`, then the chosen browsing context must be the current one.

2. If the given browsing context name is `_parent`, then the chosen browsing context must be the *parent* browsing context of the current one, unless there isn't one, in which case the chosen browsing context must be the current browsing context.

3. If the given browsing context name is `_top`, then the chosen browsing context must be the most top-level browsing context of the current one.

4. If the given browsing context name is not `_blank` and there exists a browsing context whose name is the same as the given browsing context name, and the current browsing context is allowed to navigate that browsing context, and the user agent determines that the two browsing contexts are related enough that it is ok if they reach each other, then that browsing context must be the chosen one. If there are multiple matching browsing contexts, the user agent should select one in some arbitrary consistent manner, such as the most recently opened, most recently focused, or more closely related.

5. Otherwise, a new browsing context is being requested, and what happens depends on the user agent's configuration and/or abilities:

   If the current browsing context has the sandboxed navigation browsing context flag set.

   The user agent may offer to create a new top-level browsing context or reuse an existing top-level browsing context. If the user picks one of those options, then the designated browsing context must be the chosen one (the browsing context's name isn't set to the given browsing context name). Otherwise (if the user agent doesn't offer the option to the user, or if the user declines to allow a browsing context to be used) there must not be a chosen browsing context.

   If the user agent has been configured such that in this instance it will create a new browsing context

   A new auxiliary browsing context must be created, with the opener browsing context being the current one. If the given browsing context name is not `_blank`, then the new auxiliary browsing context's name must be the given browsing context name (otherwise, it has no name). The chosen browsing context must be this new browsing context. If it is immediately navigated, then the navigation will be done with replacement enabled.

   If the user agent has been configured such that in this instance it will reuse the current browsing context

   The chosen browsing context is the current browsing context.

   If the user agent has been configured such that in this instance it will not find a browsing context

There must not be a chosen browsing context.

User agent implementors are encouraged to provide a way for users to configure the user agent to always reuse the current browsing context.

5.2 The default view

The `AbstractView` object of default views must also implement the `Window`, `WindowBrowsingContext`, and `EventTarget` interfaces.

```
[NoInterfaceObject] interface Window {
  // self-reference
  readonly attribute Window window;
  readonly attribute Window self;

  // the user agent
  readonly attribute Storage localStorage;
  Database openDatabase(in DOMString name, in DOMString version, in
DOMString displayName, in unsigned long estimatedSize);

  // user prompts
  void showNotification(in DOMString title, in DOMString subtitle, in
DOMString description);
  void showNotification(in DOMString title, in DOMString subtitle, in
DOMString description, in VoidCallback onclick);
};

[NoInterfaceObject] interface WindowBrowsingContext {
  // the current browsing context
           attribute DOMString name;
  [PutForwards=href] readonly attribute Location location;
  readonly attribute History history;
  readonly attribute UndoManager undoManager;
  Selection getSelection();

  // the user agent
  readonly attribute ClientInformation navigator;
  readonly attribute Storage sessionStorage;

  // user prompts
  void alert(in DOMString message);
  boolean confirm(in DOMString message);
  DOMString prompt(in DOMString message);
  DOMString prompt(in DOMString message, in DOMString default);
  void print();
  any showModalDialog(in DOMString url);
  any showModalDialog(in DOMString url, in any arguments);
```

```
  // other browsing contexts
  readonly attribute Window frames;
  readonly attribute unsigned long length;
  [IndexGetter] Window XXX4(in unsigned long index);
  readonly attribute Window opener;
  readonly attribute Window parent;
  Window open();
  Window open(in DOMString url);
  Window open(in DOMString url, in DOMString target);
  Window open(in DOMString url, in DOMString target, in DOMString features);
  Window open(in DOMString url, in DOMString target, in DOMString features,
in DOMString replace);

  // cross-document messaging
  void postMessage(in DOMString message, in DOMString targetOrigin);
  void postMessage(in DOMString message, in MessagePort messagePort, in
DOMString targetOrigin);

  // event handler DOM attributes
           attribute EventListener onabort;
           attribute EventListener onbeforeunload;
           attribute EventListener onblur;
           attribute EventListener onchange;
           attribute EventListener onclick;
           attribute EventListener oncontextmenu;
           attribute EventListener ondblclick;
           attribute EventListener ondrag;
           attribute EventListener ondragend;
           attribute EventListener ondragenter;
           attribute EventListener ondragleave;
           attribute EventListener ondragover;
           attribute EventListener ondragstart;
           attribute EventListener ondrop;
           attribute EventListener onerror;
           attribute EventListener onfocus;
           attribute EventListener onkeydown;
           attribute EventListener onkeypress;
           attribute EventListener onkeyup;
           attribute EventListener onload;
           attribute EventListener onmessage;
           attribute EventListener onmousedown;
           attribute EventListener onmousemove;
           attribute EventListener onmouseout;
           attribute EventListener onmouseover;
           attribute EventListener onmouseup;
           attribute EventListener onmousewheel;
           attribute EventListener onresize;
           attribute EventListener onscroll;
           attribute EventListener onselect;
```

```
            attribute EventListener onstorage;
            attribute EventListener onsubmit;
            attribute EventListener onunload;
};
```

The **`window`**, **`frames`**, and **`self`** DOM attributes must all return the `Window` object itself.

The `Window` object also provides the scope for script execution. Each `Document` in a browsing context has an associated **list of added properties** that, when a document is active, are available on the `Document`'s default view's `Window` object. A `Document` object's list of added properties must be empty when the `Document` object is created.

Each `Document` in a browsing context also has an associated **list of message ports**, which must be initially empty. The list is used during history traversal in much the same way as the list of added properties, to keep track of message ports that need to be reactivated if the `Document` is made the active document again.

5.2.1 Security

User agents must raise a security exception whenever any of the members of a `Window` object are accessed by scripts whose effective script origin is not the same as the `Window` object's browsing context's active document's effective script origin, with the following exceptions:

- The `location` object
- The `postMessage()` method with two arguments
- The `postMessage()` method with three arguments
- The `frames` attribute
- The `XXX4` method

User agents must not allow scripts to override the `location` object's setter.

5.2.2 APIs for creating and navigating browsing contexts by name

The **`open()`** method on `Window` objects provides a mechanism for navigating an existing browsing context or opening and navigating an auxiliary browsing context.

The method has four arguments, though they are all optional.

The first argument, *url*, must be a valid URL for a page to load in the browsing context. If no arguments are provided, or if the first argument is the empty string, then the *url* argument defaults to "`about:blank`". The argument must be resolved to an absolute URL (or an error) when the method is invoked.

The second argument, *target*, specifies the name of the browsing context that is to be navigated. It must be a valid browsing context name or keyword. If fewer than two arguments are provided, then the *name* argument defaults to the value "`_blank`".

The third argument, *features*, has no effect and is supported for historical reasons only.

The fourth argument, *replace*, specifies whether or not the new page will replace the page currently loaded in the browsing context, when *target* identifies an existing browsing context (as opposed to leaving the current page in the browsing context's session history). When three or fewer arguments are provided, *replace* defaults to false.

When the method is invoked, the user agent must first select a browsing context to navigate by applying the rules for choosing a browsing context given a browsing context name using the *target* argument as the name and the browsing context of the script as the context in which the algorithm is executed, unless the user has indicated a preference, in which case the browsing context to navigate may instead be the one indicated by the user.

> For example, suppose there is a user agent that supports control-clicking a link to open it in a new tab. If a user clicks in that user agent on an element whose `onclick` handler uses the `window.open()` API to open a page in an iframe, but, while doing so, holds the control key down, the user agent could override the selection of the target browsing context to instead target a new tab.

Then, the user agent must navigate the selected browsing context to the absolute URL (or error) obtained from resolving *url*. If the *replace* is true, then replacement must be enabled; otherwise, it must not be enabled unless the browsing context was just created as part of the the rules for choosing a browsing context given a browsing context name. The navigation must be done with the script browsing context of the script that invoked the method as the source browsing context.

The method must return the `Window` object of the default view of the browsing context that was navigated, or null if no browsing context was navigated.

The **`name`** attribute of the `Window` object must, on getting, return the current name of the browsing context, and, on setting, set the name of the browsing context to the new value.

> ***Note: The name gets reset when the browsing context is navigated to another domain.***

5.2.3 Accessing other browsing contexts

The **`length`** DOM attribute on the `Window` interface must return the number of child browsing contexts of the active `Document`.

The **`XXX4(`*`index`*`)`** method must return the *index*th child browsing context of the active `Document`, sorted in document order of the elements nesting those browsing contexts.

5.3 Origin

The **origin** of a resource and the **effective script origin** of a resource are both either opaque identifiers or tuples consisting of a scheme component, a host component, a port component, and optionally extra data.

> ***Note: The extra data could include the certificate of the site when using encrypted connections, to ensure that if the site's secure certificate changes, the origin is considered to change as well.***

These characteristics are defined as follows:

For URLs

The origin and effective script origin of the URL is whatever is returned by the following algorithm:

1. Let *url* be the URL for which the origin is being determined.
2. Parse *url*.
3. If *url* does not use a server-based naming authority, or if parsing *url* failed, or if *url* is not an absolute URL, then return a new globally unique identifier.
4. Let *scheme* be the <scheme> component of the URI, converted to lowercase.
5. If the UA doesn't support the protocol given by *scheme*, then return a new globally unique identifier.
6. If *scheme* is "`file`", then the user agent may return a UA-specific value.
7. Let *host* be the <host> component of *url*.
8. Apply the IDNA ToASCII algorithm to *host*, with both the AllowUnassigned and UseSTD3ASCIIRules flags set. Let *host* be the result of the ToASCII algorithm.

   If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, then return a new globally unique identifier. [RFC3490]
9. Let *host* be the result of converting *host* to lowercase.
10. If there is no <port> component, then let *port* be the default port for the protocol given by *scheme*. Otherwise, let *port* be the <port> component of *url*.
11. Return the tuple (*scheme*, *host*, *port*).

In addition, if the URL is in fact associated with a `Document` object that was created by parsing the resource obtained from fetching URL, and this was done over a secure connection, then the server's secure certificate may be added to the origin as additional data.

For scripts

The origin and effective script origin of a script are determined from another resource, called the *owner*:

- **If a script is in a `script` element**
  The owner is the `Document` to which the `script` element belongs.
- **If a script is in an event handler content attribute**
  The owner is the `Document` to which the attribute node belongs.
- **If a script is a function or other code reference created by another script**
  The owner is the script that created it.
- **If a script is a `javascript:` URL that was returned as the location of an HTTP redirect (or equivalent in other protocols)**
  The owner is the URL that redirected to the `javascript:` URL.
- **If a script is a `javascript:` URL in an attribute**
  The owner is the `Document` of the element on which the attribute is found.
- **If a script is a `javascript:` URL in a style sheet**
  The owner is the URL of the style sheet.

- **If a script is a `javascript:` URL to which a browsing context is being navigated, the URL having been provided by the user (e.g. by using a *bookmarklet*)**

  The owner is the `Document` of the browsing context's active document.

- **If a script is a `javascript:` URL to which a browsing context is being navigated, the URL having been declared in markup**

  The owner is the `Document` of the element (e.g. an `a` or `area` element) that declared the URL.

- **If a script is a `javascript:` URL to which a browsing context is being navigated, the URL having been provided by script**

  The owner is the script that provided the URL.

The origin of the script is then equal to the origin of the owner, and the effective script origin of the script is equal to the effective script origin of the owner.

For `Document` objects and images

- **If a `Document` is in a browsing context whose sandboxed origin browsing context flag is set**

  The origin is a globally unique identifier assigned when the `Document` is created.

- **If a `Document` or image was returned by the `XMLHttpRequest` API**

  The origin and effective script origin are equal to the origin and effective script origin of the `Document` object that was the active document of the browsing context of the `Window` object from which the `XMLHttpRequest` constructor was invoked. (That is, they track the `Document` to which the `XMLHttpRequest` object's `Document` pointer pointed when it was created.) [XHR]

- **If a `Document` or image was generated from a `javascript:` URL**

  The origin is equal to the origin of the script of that `javascript:` URL.

- **If a `Document` or image was served over the network and has an address that uses a URL scheme with a server-based naming authority**

  The origin is the origin of the address of the `Document` or image.

- **If a `Document` or image was generated from a `data:` URL that was returned as the location of an HTTP redirect (or equivalent in other protocols)**

  The origin is the origin of the URL that redirected to the `data:` URL.

- **If a `Document` or image was generated from a `data:` URL found in another `Document` or in a script**

  The origin is the origin of the `Document` or script in which the `data:` URL was found.

- **If a `Document` has the address "`about:blank`"**

  The origin of the `Document` is the origin it was assigned when its browsing context was created.

- **If a `Document` or image was obtained in some other manner (e.g. a `data:` URL typed in by the user, a `Document` created using the `createDocument()` API, a `data:` URL returned as the location of an HTTP redirect, etc)**

  The origin is a globally unique identifier assigned when the `Document` or image is created.

When a `Document` is created, unless stated otherwise above, its effective script origin is initialized to the origin of the `Document`. However, the `document.domain` attribute can be used to change it.

The **Unicode serialization of an origin** is the string obtained by applying the following algorithm to the given origin:

1. If the origin in question is not a scheme/host/port tuple, then return the empty string and abort these steps.

2. Otherwise, let *result* be the scheme part of the origin tuple.

3. Append the string "`://`" to *result*.

4. Apply the IDNA ToUnicode algorithm to each component of the host part of the origin tuple, and append the results — each component, in the same order, separated by U+002E FULL STOP characters (".") — to *result*.

5. If the port part of the origin tuple gives a port that is different from the default port for the protocol given by the scheme part of the origin tuple, then append a U+003A COLON character (":") and the given port, in base ten, to *result*.

6. Return *result*.

The **ASCII serialization of an origin** is the string obtained by applying the following algorithm to the given origin:

1. If the origin in question is not a scheme/host/port tuple, then return the empty string and abort these steps.

2. Otherwise, let *result* be the scheme part of the origin tuple.

3. Append the string "`://`" to *result*.

4. Apply the IDNA ToASCII algorithm the host part of the origin tuple, with both the AllowUnassigned and UseSTD3ASCIIRules flags set, and append the results *result*.

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

5. If the port part of the origin tuple gives a port that is different from the default port for the protocol given by the scheme part of the origin tuple, then append a U+003A COLON character (":") and the given port, in base ten, to *result*.

6. Return *result*.

Two origins are said to be the **same origin** if the following algorithm returns true:

1. Let *A* be the first origin being compared, and *B* be the second origin being compared.

2. If *A* and *B* are both opaque identifiers, and their value is equal, then return true.

3. Otherwise, if either *A* or *B* or both are opaque identifiers, return false.

4. If *A* and *B* have scheme components that are not identical, return false.

5. If *A* and *B* have host components that are not identical, return false.

6. If *A* and *B* have port components that are not identical, return false.

7. If either *A* or *B* have additional data, but that data is not identical for both, return false.

8. Return true.

5.3.1 Relaxing the same-origin restriction

The **`domain`** attribute on `Document` objects must be initialized to the document's domain, if it has one, and the empty string otherwise. On getting, the attribute must return its current value, unless the document was created by `XMLHttpRequest`, in which case it must throw an `INVALID_ACCESS_ERR` exception. On setting, the user agent must run the following algorithm:

1. If the document was created by `XMLHttpRequest`, throw an `INVALID_ACCESS_ERR` exception and abort these steps.
2. Apply the IDNA ToASCII algorithm to the new value, with both the AllowUnassigned and UseSTD3ASCIIRules flags set. Let *new value* be the result of the ToASCII algorithm.

   If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, then throw a security exception and abort these steps. [RFC3490]
3. If *new value* is not exactly equal to the current value of the `document.domain` attribute, then run these substeps:
   1. If the current value is an IP address, throw a security exception and abort these steps.
   2. If *new value*, prefixed by a U+002E FULL STOP ("."), does not exactly match the end of the current value, throw a security exception and abort these steps.
4. Set the attribute's value to *new value*.
5. Set the host part of the effective script origin tuple of the `Document` to *new value*.
6. Set the port part of the effective script origin tuple of the `Document` to "manual override" (a value that, for the purposes of comparing origins, is identical to "manual override" but not identical to any other value).

The **domain** of a `Document` is the host part of the document's origin, if that is a scheme/host/port tuple. If it isn't, then the document does not have a domain.

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

5.4 Scripting

Various mechanisms can cause author-provided executable code to run in the context of a document. These mechanisms include, but are probably not limited to:

- Processing of `script` elements.
- Processing of inline `javascript:` URLs (e.g. the `src` attribute of `img` elements, or an `@import` rule in a CSS `style` element block).

- Event handlers, whether registered through the DOM using `addEventListener()`, by explicit event handler content attributes, by event handler DOM attributes, or otherwise.
- Processing of technologies like XBL or SVG that have their own scripting features.

When a script is created, it is associated with a script execution context, a script browsing context, and a script document context.

5.4.1 Script execution contexts

The **script execution context** of a script is defined when that script is created. It is either a `Window` object or an empty object.

When the script execution context of a script is an empty object, it can't do anything that interacts with the environment.

A script execution context always has an associated browsing context, known as the **script browsing context**. If the script execution context is a `WindowBrowsingContext` object, then that object's browsing context is it. Otherwise, the script execution context is associated explicitly with a browsing context when it is created.

It is said that **scripting is disabled** in a script execution context when any of the following conditions are true:

- The user agent does not support scripting.
- The user has disabled scripting for this script execution context. (User agents may provide users with the option to disable scripting globally, on a per-origin basis, or in other ways down to the granularity of individual script execution contexts.)
- The script execution context's associated browsing context's active document has `designMode` enabled.
- The script execution context's associated browsing context has the sandboxed scripts browsing context flag set.

A node is said to be **without script** if either the `Document` object of the node (the node itself, if it is itself a `Document` object) does not have an associated browsing context, or scripting is disabled in that browsing context.

A node is said to be **with script** if it is not without script.

If you can find a better pair of terms than "with script" and "without script" let me know. The only things I can find that are less confusing are also way, way longer.

When a script is to be executed in a script execution context in which scripting is disabled, the script must do nothing and return nothing (a void return value).

> ***Note: Thus, for instance, enabling `designMode` will disable any event handler attributes, event listeners, timeouts, etc, that were set by scripts in the document.***

Every script whose script execution context is a `Window` object is also associated with a `Document` object,

known as its **script document context**. It is used to resolve URLs. The document is assigned when the script is created, as with the script browsing context.

5.4.2 Security exceptions

Define **security exception**.

5.4.3 The `javascript:` protocol

A URL using the `javascript:` protocol must, if and when dereferenced, be evaluated by executing the script obtained using the content retrieval operation defined for `javascript:` URLs. [JSURL]

When a browsing context is navigated to a `javascript:` URL, and the active document of that browsing context has the same origin as the script given by that URL, the script execution context must be the `Window` object of the browsing context being navigated, and the script document context must be that active document.

When a browsing context is navigated to a `javascript:` URL, and the active document of that browsing context has an origin that is *not* the same as that of the script given by the URL, the script execution context must be an empty object, and the script browsing context must be the browsing context being navigated.

Otherwise, the script execution context must be an empty object, and the script execution context's associated browsing context must be the browsing context of the `Document` object of the element, attribute, or style sheet from which the `javascript:` URL was reached.

If the result of executing the script is void (there is no return value), then the URL must be treated in a manner equivalent to an HTTP resource with an HTTP 204 No Content response.

Otherwise, the URL must be treated in a manner equivalent to an HTTP resource with a 200 OK response whose Content-Type metadata is `text/html` and whose response body is the return value converted to a string value.

Note: Certain contexts, in particular `img` elements, ignore the Content-Type metadata.

> So for example a `javascript:` URL for a `src` attribute of an `img` element would be evaluated in the context of an empty object as soon as the attribute is set; it would then be sniffed to determine the image type and decoded as an image.
>
> A `javascript:` URL in an `href` attribute of an `a` element would only be evaluated when the link was followed.
>
> The `src` attribute of an `iframe` element would be evaluated in the context of the `iframe`'s own browsing context; once evaluated, its return value (if it was not void) would replace that browsing context's document, thus changing the variables visible in that browsing context.

Note: The rules for handling script execution in a script execution context include making the script not execute (and just return void) in certain cases, e.g. in a sandbox or when the user has disabled scripting altogether.

5.4.4 Events

We need to define how to handle events that are to be fired on a Document that is no longer the active document of its browsing context, and for Documents that have no browsing context. Do the events fire? Do the handlers in that document not fire? Do we just define scripting to be disabled when the document isn't active, with events still running as is? See also the `script` element section, which says scripts don't run when the document isn't active.

5.4.4.1. 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.

Objects other than `Element` objects, in particular `Window`, only have event handler DOM attribute (since they have no content attributes).

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 `Window` object of the browsing context of that `Document`. 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]***

The script execution context of the event handler must be the `Window` object at the end of the scope chain. The script document context of the event handler must be the `Document` object that owns the event handler content attribute that was set.

How do we allow non-JS event handlers?

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, and on `Window` objects, as DOM attributes:

onabort

Must be invoked whenever an `abort` event is targeted at or bubbles through the element.

onbeforeunload

Must be invoked whenever a `beforeunload` event is targeted at or bubbles through the element.

onblur

Must be invoked whenever a `blur` event is targeted at or bubbles through the element.

onchange

Must be invoked whenever a `change` event is targeted at or bubbles through the element.

onclick

Must be invoked whenever a `click` event is targeted at or bubbles through the element.

oncontextmenu

Must be invoked whenever a `contextmenu` event is targeted at or bubbles through the element.

ondblclick

Must be invoked whenever a `dblclick` event is targeted at or bubbles through the element.

ondrag

Must be invoked whenever a `drag` event is targeted at or bubbles through the element.

ondragend

Must be invoked whenever a `dragend` event is targeted at or bubbles through the element.

ondragenter

Must be invoked whenever a `dragenter` event is targeted at or bubbles through the element.

ondragleave

Must be invoked whenever a `dragleave` event is targeted at or bubbles through the element.

ondragover

Must be invoked whenever a `dragover` event is targeted at or bubbles through the element.

ondragstart

Must be invoked whenever a `dragstart` event is targeted at or bubbles through the element.

ondrop

Must be invoked whenever a `drop` event is targeted at or bubbles through the element.

onerror

Must be invoked whenever an `error` event is targeted at or bubbles through the element.

Note: The `onerror` handler is also used for reporting script errors.

onfocus

Must be invoked whenever a `focus` event is targeted at or bubbles through the element.

onkeydown

Must be invoked whenever a `keydown` event is targeted at or bubbles through the element.

onkeypress

Must be invoked whenever a `keypress` event is targeted at or bubbles through the element.

onkeyup

Must be invoked whenever a `keyup` event is targeted at or bubbles through the element.

onload

Must be invoked whenever a `load` event is targeted at or bubbles through the element.

onmessage

Must be invoked whenever a `message` event is targeted at or bubbles through the element.

onmousedown

Must be invoked whenever a `mousedown` event is targeted at or bubbles through the element.

onmousemove

Must be invoked whenever a `mousemove` event is targeted at or bubbles through the element.

onmouseout

Must be invoked whenever a `mouseout` event is targeted at or bubbles through the element.

onmouseover

Must be invoked whenever a `mouseover` event is targeted at or bubbles through the element.

onmouseup

Must be invoked whenever a `mouseup` event is targeted at or bubbles through the element.

onmousewheel

Must be invoked whenever a `mousewheel` event is targeted at or bubbles through the element.

onresize

Must be invoked whenever a `resize` event is targeted at or bubbles through the element.

onscroll

Must be invoked whenever a `scroll` event is targeted at or bubbles through the element.

onselect

Must be invoked whenever a `select` event is targeted at or bubbles through the element.

onstorage

Must be invoked whenever a `storage` event is targeted at or bubbles through the element.

onsubmit

Must be invoked whenever a `submit` event is targeted at or bubbles through the element.

onunload

Must be invoked whenever an `unload` event is targeted at or bubbles through the element.

When an event handler attribute is invoked, its argument must be set to the `Event` object of the event in question. If the function returns the exact boolean value 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.

All event handler attributes on an element, whether set to null or to a function, must be registered as event listeners on the element, as if the `addEventListenerNS()` method on the `Element` object's `EventTarget` interface had been invoked when the element was created, with the event type (*type* argument) equal to the type described for the event handler attribute in the list above, the namespace (*namespaceURI* argument) set to null, the listener set to be a target and bubbling phase listener (*useCapture* argument set to false), the event group set to the default group (*evtGroup* argument set to null), and the event listener itself (*listener* argument) set to do nothing while the event handler attribute is null, and set to invoke the function associated with the event handler attribute otherwise. (The *listener* argument is emphatically *not* the event handler attribute itself.)

5.4.4.2. Event firing

maybe this should be moved higher up (terminology? conformance? DOM?) Also, the whole terminology thing should be changed so that we don't define any specific events here, we only define 'simple event', 'progress event', 'mouse event', 'key event', and the like, and have the actual dispatch use those generic terms when firing events.

Certain operations and methods are defined as firing events on elements. For example, the `click()` method on the `HTMLElement` interface is defined as firing a `click` event on the element. [DOM3EVENTS]

Firing a `click` event means that a `click` event with no namespace, which bubbles and is cancelable, and which uses the `MouseEvent` interface, must be dispatched at the given 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 with no 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 with no 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*, with no namespace, which does not bubble but is cancelable (unless otherwise stated), and which uses the `Event` interface, must be dispatched at the given element.

Firing a `show` event means firing a simple event called `show`. Actually this should fire an event that has modifier information (shift/ctrl etc), as well as having a pointer to the node on which the menu was fired, and

with which the menu was associated (which could be an ancestor of the former).

Firing a `load` event means firing a simple event called `load`. **Firing an `error` event** means firing a simple event called `error`.

Firing a progress event called *e* means something that hasn't yet been defined, in the [PROGRESS] spec.

The default action of these event is to do nothing unless otherwise stated.

If you dispatch a custom "click" event at an element that would normally have default actions, should they get triggered? If so, we need to go through the entire spec and make sure that any default actions are defined in terms of *any* event of the right type on that element, not those that are dispatched in expected ways.

5.4.4.3. Events and the `Window` object

When an event is dispatched at a DOM node in a `Document` in a browsing context, if the event is not a `load` event, the user agent must also dispatch the event to the `Window`, as follows:

1. In the capture phase, the event must be dispatched to the `Window` object before being dispatched to any of the nodes.
2. In the bubble phase, the event must be dispatched to the `Window` object at the end of the phase, unless bubbling has been prevented.

5.4.4.4. Runtime script errors

This section only applies to user agents that support scripting in general and ECMAScript in particular.

Whenever a runtime script error occurs in one of the scripts associated with the document, the value of the `onerror` event handler DOM attribute of the `Window` 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 absolute URL of the resource in which the error occurred, and the third must give the line number in that resource on which the error occurred.

If the function returns false, then the error 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`.

5.5 User prompts

5.5.1 Simple dialogs

The **`alert(`*`message`*`)`** method, when invoked, must show the given *message* to the user. The user agent may make the method wait for the user to acknowledge the message before returning; if so, the user agent must pause while the method is waiting.

The **`confirm(`*`message`*`)`** method, when invoked, must show the given *message* to the user, and ask the user to respond with a positive or negative response. The user agent must then pause as the method waits for the user's response. If the user responds positively, the method must return true, and if the user responds negatively, the method must return false.

The **`prompt(`*`message`*`, `*`default`*`)`** method, when invoked, must show the given *message* to the user, and ask the user to either respond with a string value or abort. The user agent must then pause as the method waits for the user's response. The second argument is optional. If the second argument (*default*) is present, then the response must be defaulted to the value given by *default*. If the user aborts, then the method must return null; otherwise, the method must return the string that the user responded with.

5.5.2 Printing

The **`print()`** method, when invoked, must run the printing steps.

User agents should also run the printing steps whenever the user attempts to obtain a physical form (e.g. printed copy), or the representation of a physical form (e.g. PDF copy), of a document.

The **printing steps** are as follows:

1. The user agent may display a message to the user and/or may abort these steps.

   > For instance, a kiosk browser could silently ignore any invocations of the `print()` method.

   > For instance, a browser on a mobile device could detect that there are no printers in the vicinity and display a message saying so before continuing to offer a "save to PDF" option.

2. The user agent must fire a simple event called `beforeprint` at the `Window` object of the browsing context of the `Document` that is being printed, as well as any nested browsing contexts in it.

   > The `beforeprint` event can be used to annotate the printed copy, for instance adding the time at which the document was printed.

3. The user agent should offer the user the opportunity to obtain a physical form (or the representation of a physical form) of the document. The user agent may wait for the user to either accept or decline

before returning; if so, the user agent must pause while the method is waiting. Even if the user agent doesn't wait at this point, the user agent must use the state of the relevant documents as they are at this point in the algorithm if and when it eventually creates the alternate form.

4. The user agent must fire a simple event called `afterprint` at the `Window` object of the browsing context of the `Document` that is being printed, as well as any nested browsing contexts in it.

> The `afterprint` event can be used to revert annotations added in the earlier event, as well as showing post-printing UI. For instance, if a page is walking the user through the steps of applying for a home loan, the script could automatically advance to the next step after having printed a form or other.

5.5.3 Dialogs implemented using separate documents

The **`showModalDialog(`*`url`*`, `*`arguments`*`)`** method, when invoked, must cause the user agent to run the following steps:

1. If the user agent is configured such that this invocation of `showModalDialog()` is somehow disabled, then the method returns the empty string; abort these steps.

   Note: User agents are expected to disable this method in certain cases to avoid user annoyance. For instance, a user agent could require that a site be white-listed before enabling this method, or the user agent could be configured to only allow one modal dialog at a time.

2. Let *the list of background browsing contexts* be a list of all the browsing contexts that:

   - are part of the same unit of related browsing contexts as the browsing context of the `Window` object on which the `showModalDialog()` method was called, and that
   - have an active document whose origin is the same as the origin of the script that called the `showModalDialog()` method at the time the method was called,

   ...as well as any browsing contexts that are nested inside any of the browsing contexts matching those conditions.

3. Disable the user interface for all the browsing contexts in *the list of background browsing contexts*. This should prevent the user from navigating those browsing contexts, causing events to to be sent to those browsing context, or editing any content in those browsing contexts. However, it does not prevent those browsing contexts from receiving events from sources other than the user, from running scripts, from running animations, and so forth.

4. Create a new auxiliary browsing context, with the opener browsing context being the browsing context of the `Window` object on which the `showModalDialog()` method was called. The new auxiliary browsing context has no name.

   Note: This browsing context implements the `ModalWindow` interface.

5. Let the dialog arguments of the new browsing context be set to the value of *arguments*.

6. Let the dialog arguments' origin be the origin of the script that called the `showModalDialog()` method.
7. Navigate the new browsing context to *url*, with replacement enabled, and with the script browsing context of the script that invoked the method as the source browsing context.
8. Wait for the browsing context to be closed. (The user agent must allow the user to indicate that the browsing context is to be closed.)
9. Reenable the user interface for all the browsing contexts in *the list of background browsing contexts*.
10. Return the auxiliary browsing context's return value.

Browsing contexts created by the above algorithm must implement the `ModalWindow` interface:

```
[XXX] interface ModalWindow {
  readonly attribute any dialogArguments;
           attribute DOMString returnValue;
};
```

Such browsing contexts have associated **dialog arguments**, which are stored along with the **dialog arguments' origin**. These values are set by the `showModalDialog()` method in the algorithm above, when the browsing context is created, based on the arguments provided to the method.

The **`dialogArguments`** DOM attribute, on getting, must check whether its browsing context's active document's origin is the same as the dialog arguments' origin. If it is, then the browsing context's dialog arguments must be returned unchanged. Otherwise, if the dialog arguments are an object, then the empty string must be returned, and if the dialog arguments are not an object, then the stringification of the dialog arguments must be returned.

These browsing contexts also have an associated **return value**. The return value of a browsing context must be initialized to the empty string when the browsing context is created.

The **`returnValue`** DOM attribute, on getting, must return the return value of its browsing context, and on setting, must set the return value to the given new value.

5.5.4 Notifications

Notifications are short, transient messages that bring the user's attention to new information, or remind the user of scheduled events.

Since notifications can be annoying if abused, this specification defines a mechanism that scopes notifications to a site's existing rendering area unless the user explicitly indicates that the site can be trusted.

To this end, each origin can be flagged as being a **trusted notification source**. By default origins should not be flagged as such, but user agents may allow users to whitelist origins or groups of origins as being trusted notification sources. Only origins flagged as trusted in this way are allowed to show notification UI outside of their tab.

> For example, a user agent could allow a user to mark all subdomains and ports of example.org as trusted notification sources. Then, mail.example.org and calendar.example.org would both be able to show

> notifications, without the user having to flag them individually.

The **`showNotification(`*`title`*`, `*`subtitle`*`, `*`description`*`, `*`onclick`*`)`** method, when invoked, must cause the user agent to show a notification.

If the method was invoked from a script whose script browsing context has the sandboxed annoyances browsing context flag set, then the notification must be shown within that browsing context. The notification is said to be a **sandboxed notification**.

Otherwise, if the origin of the script browsing context of the script that invoked the method is *not* flagged as being a trusted notification source, then the notification should be rendered within the top-level browsing context of the script browsing context of the script that invoked the method. The notification is said to be a **normal notification**. User agents should provide a way to set the origin's trusted notification source flag from the notification, so that the user can benefit from notifications even when the user agent is not the active application.

Otherwise, the origin is flagged as a trusted notification source, and the notification should be shown using the platform conventions for system-wide notifications. The notification is said to be a **trusted notification**. User agents may provide a way to unset the origin's trusted notification source flag from within the notification, so as to allow users to easily disable notifications from sites that abuse the privilege.

> For example, if a site contains a gadget of a mail application in a sandboxed `iframe` and that frame triggers a notification upon the receipt of a new e-mail message, that notification would be displayed on top of the gadget only.
>
> However, if the user then goes to the main site of that mail application, the notification would be displayed over the entire rendering area of the tab for the site.
>
> The notification, in this case, would have a button on it to let the user indicate that he trusts the site. If the user clicked this button, the next notification would use the system-wide notification system, appearing even if the tab for the mail application was buried deep inside a minimised window.

> The style of notifications varies from platform to platform. On some, it is typically displayed as a "toast" window that slides in from the bottom right corner. In others, notifications are shown as semi-transparent white-on-grey overlays centered over the screen. Other schemes could include simulated ticker tapes, and speech-synthesis playback.

When a normal notification (but not a sandboxed notification) is shown, the user agent may bring the user's attention to the top-level browsing context of the script browsing context of the script that invoked the method, if that would be useful; but user agents should not use system-wide notification mechanisms to do so.

When a trusted notification is shown, the user agent should bring the user's attention to the notification and the script browsing context of the script that invoked the method, as per the platform conventions for attracting the user's attention to applications.

> In the case of normal notifications, typically the only attention-grabbing device that would be employed would be something like flashing the tab's caption, or making it bold, or some such.
>
> In addition, in the case of a trusted notification, the entire window could flash, or the browser's application icon could bounce or flash briefly, or a short sound effect could be played.

Notifications should include the following content:

- The *title*, *subtitle*, and *description* strings passed to the method. They may be truncated or abbreviated if necessary.
- The application name, if available, or else the document title, of the active document of the script browsing context of the script that invoked the method.
- An icon chosen from the external resource links of type `icon`, if any are available.

If a new notification from one browsing context has *title*, *subtitle*, and *description* strings that are identical to the *title*, *subtitle*, and *description* strings of an already-active notification from the same browsing context or another browsing context with the same origin, the user agent should not display the new notification, but should instead add an indicator to the already-active notification that another identical notification would otherwise have been shown.

> For instance, if a user has his mail application open in three windows, and thus the same "New Mail" notification is fired three times each time a mail is received, instead of displaying three identical notifications each time, the user agent could just show one, with the title "New Mail x3".

Notifications should have a lifetime based on the platform conventions for notifications. However, the lifetime of a notification should not begin until the user has had the opportunity to see it, so if a notification is spawned for a browsing context that is hidden, it should be shown for its complete lifetime once the user brings that browsing context into view.

User agents should support multiple notifications at once.

User agents should support user interaction with notifications, if and as appropriate given the platform conventions. If a user activates a notification, and the *onclick* callback argument was present and is not null, then the script browsing context of the function given by *onclick* should be brought to the user's attention, and the *onclick* callback should then be invoked.

5.6 Browser state

The **`navigator`** attribute of the `Window` 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 url, in
DOMString title);
  void registerContentHandler(in DOMString mimeType, in DOMString url, in
DOMString title);
};
```

5.6.1 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 URLs that they consider against the list of registered handlers.

The *protocol* value, if it contains a colon (as in "`ftp:`"), will never match anything, since schemes don't contain colons.

mimeType **(`registerContentHandler()` only)**

A MIME type, such as `model/vrml` or `text/richtext`. The MIME type must be 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.

url

The URL of the page that will handle the requests. When the user agent uses this URL, it must replace the first occurrence of the exact literal string "`%s`" with an escaped version of the URL of the content in question (as defined below), then resolve the resulting URL (using the document base URL of the script document context of the script that originally invoked the `registerContentHandler()` or `registerProtocolHandler()` method), and then fetch the resulting URL using the GET method (or equivalent for non-HTTP URLs).

To get the escaped version of the URL of the content in question, the user agent must resolve the URL, and then every character in the URL that doesn't match the <query> production defined in RFC 3986 must be replaced by the percent-encoded form of the character. [RFC3986]

> If the user had visited a site at `http://example.com/` that made the following call:
>
> ```
> navigator.registerContentHandler('application/x-soup', 'soup?url=%s',
> 'SoupWeb™')
> ```
>
> ...and then, much later, while visiting `http://www.example.net/`, clicked on a link such as:

> ```
> <a href="chickenkïwi.soup">Download our Chicken Kiwi soup!</a>
> ```
>
> ...then, assuming this `chickenkïwi.soup` file was served with the MIME type `application/x-soup`, the UA might navigate to the following URL:
>
> ```
> http://example.com/soup?url=http://www.example.net/chickenk
> %C3%AFwi.soup
> ```
>
> This site could then fetch the `chickenkïwi.soup` file and do whatever it is that it does with soup (synthesize it and ship it to the user, or whatever).

title

A descriptive title of the handler, which the UA might use to remind the user what the site in question is.

User agents should raise security exceptions if the methods are called with *protocol* or *mimeType* values that the UA deems to be "privileged". For example, a site attempting to register a handler for `http` URLs or `text/html` content in a Web browser would likely cause an exception to be raised.

User agents must raise a `SYNTAX_ERR` exception if the *url* argument passed to one of these methods does not contain the exact literal string "`%s`".

User agents must not raise any other exceptions (other than binding-specific exceptions, such as for an incorrect number of arguments in an ECMAScript implementation).

This section does not define how the pages registered by these methods are used, beyond the requirements on how to process the *url* value (see above). To some extent, the processing model for navigating across documents defines some cases where these methods are relevant, but in general UAs may use this information wherever they would otherwise consider handing content to native plugins or helper applications.

UAs must not use registered content handlers to handle content that was returned as part of a non-GET transaction (or rather, as part of any non-idempotent transaction), as the remote site would not be able to fetch the same data.

5.6.1.1. Security and privacy

These mechanisms can introduce a number of concerns, in particular privacy concerns.

Hijacking all Web usage. User agents should not allow protocols that are key to its normal operation, such as `http` or `https`, to be rerouted through third-party sites. This would allow a user's activities to be trivially tracked, and would allow user information, even in secure connections, to be collected.

Hijacking defaults. It is strongly recommended that user agents do not automatically change any defaults, as this could lead the user to send data to remote hosts that the user is not expecting. New handlers registering themselves should never automatically cause those sites to be used.

Registration spamming. User agents should consider the possibility that a site will attempt to register a large number of handlers, possibly from multiple domains (e.g. by redirecting through a series of pages each on a different domain, and each registering a handler for `video/mpeg` — analogous practices abusing other Web browser features have been used by pornography Web sites for many years). User agents should gracefully handle such hostile attempts, protecting the user.

Misleading titles. User agents should not rely wholly on the *title* argument to the methods when presenting

the registered handlers to the user, since sites could easily lie. For example, a site `hostile.example.net` could claim that it was registering the "Cuddly Bear Happy Content Handler". User agents should therefore use the handler's domain in any UI along with any title.

Hostile handler metadata. User agents should protect against typical attacks against strings embedded in their interface, for example ensuring that markup or escape characters in such strings are not executed, that null bytes are properly handled, that over-long strings do not cause crashes or buffer overruns, and so forth.

Leaking Intranet URLs. The mechanism described in this section can result in secret Intranet URLs being leaked, in the following manner:

1. The user registers a third-party content handler as the default handler for a content type.
2. The user then browses his corporate Intranet site and accesses a document that uses that content type.
3. The user agent contacts the third party and hands the third party the URI to the Intranet content.

No actual confidential file data is leaked in this manner, but the URLs themselves could contain confidential information. For example, the URL could be `http://www.corp.example.com/upcoming-aquisitions/the-sample-company.egf`, which might tell the third party that Example Corporation is intending to merge with The Sample Company. Implementors might wish to consider allowing administrators to disable this feature for certain subdomains, content types, or protocols.

Leaking secure URLs. User agents should not send HTTPS URLs to third-party sites registered as content handlers, in the same way that user agents do not send `Referer` headers from secure sites to third-party sites.

Leaking credentials. User agents must never send username or password information in the URLs that are escaped and included sent to the handler sites. User agents may even avoid attempting to pass to Web-based handlers the URLs of resources that are known to require authentication to access, as such sites would be unable to access the resources in question without prompting the user for credentials themselves (a practice that would require the user to know whether to trust the third-party handler, a decision many users are unable to make or even understand).

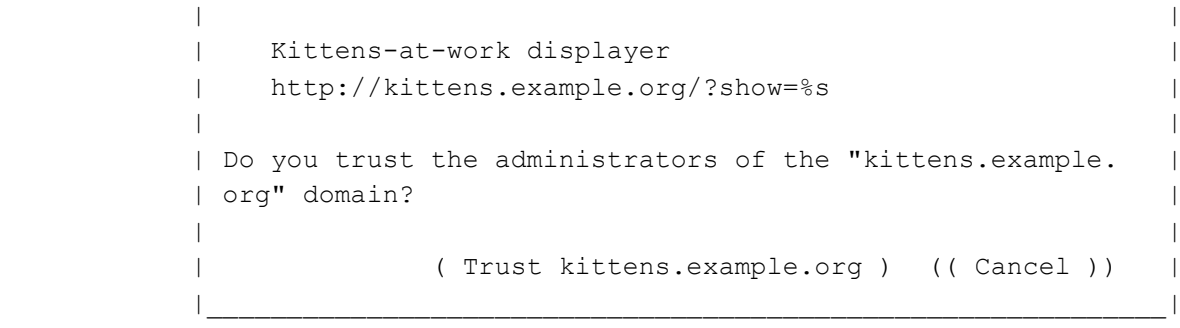
5.6.1.2. Sample user interface

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:

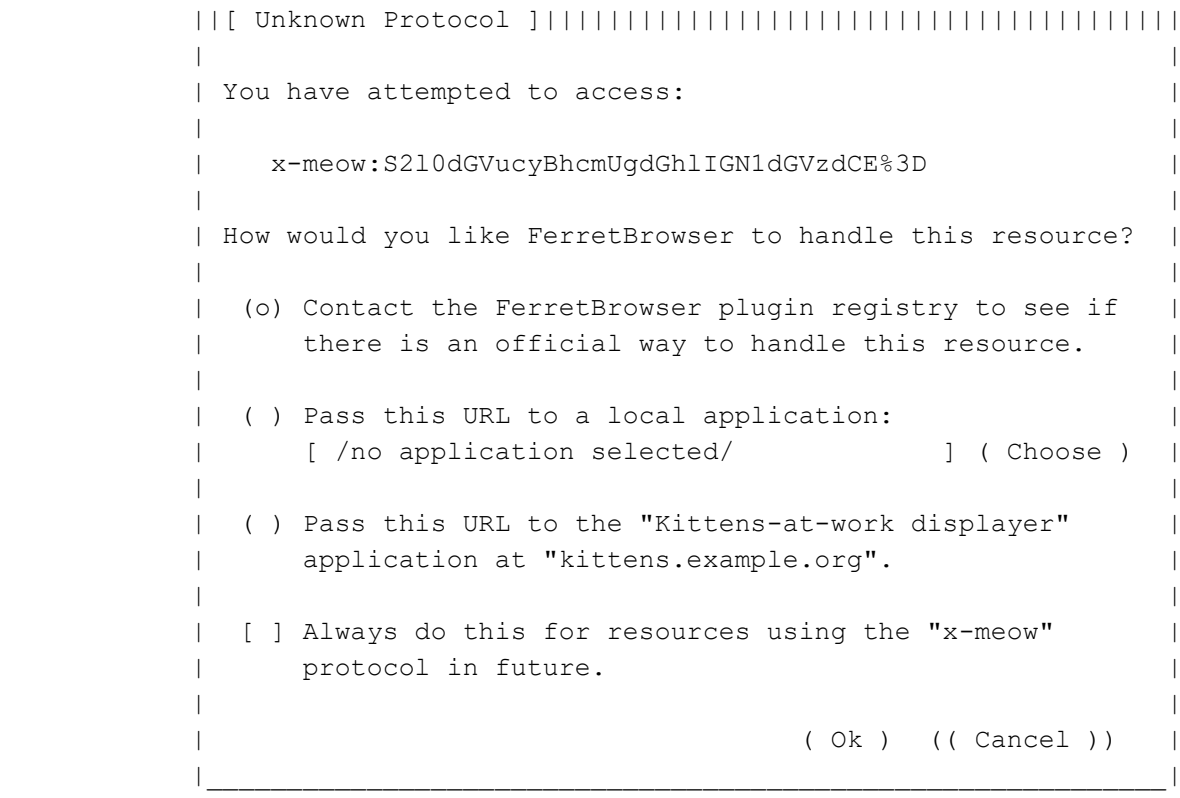
```
||[ Protocol Handler Registration ]||||||||||||||||||||||||||||
|                                                              |
| This Web page:                                               |
|                                                              |
|    Kittens at work                                           |
|    http://kittens.example.org/                               |
|                                                              |
| ...would like permission to handle the protocol "x-meow:"    |
| using the following Web-based application:                   |
```



...where "Kittens at work" is the title of the page that invoked the method, "http://kittens.example.org/" is the URL of that page, "x-meow" is the string that was passed to the `registerProtocolHandler()` method as its first argument (*protocol*), "http://kittens.example.org/?show=%s" was the second argument (*url*), and "Kittens-at-work displayer" was the third argument (*title*).

If the user clicks the Cancel button, then nothing further happens. If the user clicks the "Trust" button, then the handler is remembered.

When the user then attempts to fetch a URL that uses the "x-meow:" scheme, then it might display a dialog as follows:



...where the third option is the one that was primed by the site registering itself earlier.

If the user does select that option, then the browser, in accordance with the requirements described in the previous two sections, will redirect the user to "http://kittens.example.org/?show=x-meow%3AS2l0dGVucyBhcmUgdGhlIGN1dGVzdCE%253D".

The `registerContentHandler()` method would work equivalently, but for unknown MIME types instead of

unknown protocols.

5.7 Offline Web applications

5.7.1 Introduction

This section is non-normative.

...

5.7.2 Application caches

An **application cache** is a collection of resources. An application cache is identified by the absolute URL of a resource manifest which is used to populate the cache.

Application caches are versioned, and there can be different instances of caches for the same manifest URL, each having a different version. A cache is newer than another if it was created after the other (in other words, caches in a group have a chronological order).

Each group of application caches for the same manifest URL have a common **update status**, which is one of the following: *idle*, *checking*, *downloading*.

A browsing context can be associated with an application cache. A child browsing context is always associated with the same application cache as its parent browsing context, if any. A top-level browsing context is associated with the application cache appropriate for its active document. (A browsing context's associated cache thus can change during session history traversal.)

A `Document` initially has no appropriate cache, but steps in the parser and in the navigation sections cause cache selection to occur early in the page load process.

An application cache consists of:

- One of more resources (including their out-of-band metadata, such as HTTP headers, if any), identified by URLs, each falling into one (or more) of the following categories:

  Implicit entries

  Documents that were added to the cache because a top-level browsing context was navigated to that document and the document indicated that this was its cache, using the `manifest` attribute.

  The manifest

  The resource corresponding to the URL that was given in an implicit entry's `html` element's `manifest` attribute. The manifest is downloaded and processed during the application cache update process. All the implicit entries have the same origin as the manifest.

  Explicit entries

  Resources that were listed in the cache's manifest. Explicit entries can also be marked as **foreign**, which means that they have a `manifest` attribute but that it doesn't point at this cache's manifest.

  Fallback entries

Resources that were listed in the cache's manifest as fallback entries.

Opportunistically cached entries

Resources whose URLs matched an opportunistic caching namespace when they were fetched, and were therefore cached in the application cache.

Dynamic entries

Resources that were added to the cache by the `add()` method.

Note: A URL in the list can be flagged with multiple different types, and thus an entry can end up being categorized as multiple entries. For example, an entry can be an explicit entry and a dynamic entry at the same time.

- Zero or more **opportunistic caching namespaces**: URLs, used as prefix match patterns, each of which is mapped to a fallback entry. Each namespace URL prefix, when parsed as a URL, has the same origin as the manifest.
- Zero or more URLs that form the **online whitelist**.

Multiple application caches can contain the same resource, e.g. if their manifests all reference that resource. If the user agent is to **select an application cache** from a list of caches that contain a resource, that the user agent must use the application cache that the user most likely wants to see the resource from, taking into account the following:

- which application cache was most recently updated,
- which application cache was being used to display the resource from which the user decided to look at the new resource, and
- which application cache the user prefers.

5.7.3 The cache manifest syntax

5.7.3.1. Writing cache manifests

Manifests must be served using the `text/cache-manifest` MIME type. All resources served using the `text/cache-manifest` MIME type must follow the syntax of application cache manifests, as described in this section.

An application cache manifest is a text file, whose text is encoded using UTF-8. Data in application cache manifests is line-based. Newlines must be represented by U+000A LINE FEED (LF) characters, U+000D CARRIAGE RETURN (CR) characters, or U+000D CARRIAGE RETURN (CR) U+000A LINE FEED (LF) pairs.

Note: This is a willful double violation of RFC2046. [RFC2046]

The first line of an application cache manifest must consist of the string "CACHE", a single U+0020 SPACE character, the string "MANIFEST", and zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters. The first line may optionally be preceded by a U+FEFF BYTE ORDER MARK (BOM) character. If any other text is found on the first line, the user agent will ignore the entire file.

Subsequent lines, if any, must all be one of the following:

A blank line

Blank lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters only.

A comment

Comment lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, followed by a single U+0023 NUMBER SIGN (#) character, followed by zero or more characters other than U+000A LINE FEED (LF) and U+000D CARRIAGE RETURN (CR) characters.

Note: Comments must be on a line on their own. If they were to be included on a line with a URL, the "#" would be mistaken for part of a fragment identifier.

A section header

Section headers change the current section. There are three possible section headers:

`CACHE:`
: Switches to the explicit section.

`FALLBACK:`
: Switches to the fallback section.

`NETWORK:`
: Switches to the online whitelist section.

Section header lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, followed by one of the names above (including the U+003A COLON (:) character) followed by zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters.

Ironically, by default, the current section is the explicit section.

Data for the current section

The format that data lines must take depends on the current section.

When the current section is the explicit section or the online whitelist section, data lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, a valid URL identifying a resource other than the manifest itself, and then zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters.

When the current section is the fallback section, data lines must consist of zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, a valid URL identifying a resource other than the manifest itself, one or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, another valid URL identifying a resource other than the manifest itself, and then zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters.

Note: The URLs in data lines can't be empty strings, since those would be relative URLs to the manifest itself. Such lines would be confused with blank or invalid lines, anyway.

Manifests may contain sections more than once. Sections may be empty.

URLs that are to be fallback pages associated with opportunistic caching namespaces, and those namespaces themselves, must be given in fallback sections, with the namespace being the first URL of the data line, and the corresponding fallback page being the second URL. All the other pages to be cached must be listed in explicit sections.

Opportunistic caching namespaces must have the same origin as the manifest itself.

An opportunistic caching namespace must not be listed more than once.

URLs that the user agent is to put into the online whitelist must all be specified in online whitelist sections. (This is needed for any URL that the page is intending to use to communicate back to the server.)

URLs in the online whitelist section must not also be listed in explicit section, and must not be listed as fallback entries in the fallback section. (URLs in the online whitelist section may match opportunistic caching namespaces, however.)

Relative URLs must be given relative to the manifest's own URL.

URLs in manifests must not have fragment identifiers (i.e. the U+0023 NUMBER SIGN character isn't allowed in URLs in manifests).

5.7.3.2. Parsing cache manifests

When a user agent is to **parse a manifest**, it means that the user agent must run the following steps:

1. The user agent must decode the byte stream corresponding with the manifest to be parsed, treating it as UTF-8. Bytes or sequences of bytes that are not valid UTF-8 sequences must be interpreted as a U+FFFD REPLACEMENT CHARACTER.

2. Let *explicit URLs* be an initially empty list of explicit entries.

3. Let *fallback URLs* be an initially empty mapping of opportunistic caching namespaces to fallback entries.

4. Let *online whitelist URLs* be an initially empty list of URLs for a online whitelist.

5. Let *input* be the decoded text of the manifest's byte stream.

6. Let *position* be a pointer into *input*, initially pointing at the first character.

7. If *position* is pointing at a U+FEFF BYTE ORDER MARK (BOM) character, then advance *position* to the next character.

8. If the characters starting from *position* are "CACHE", followed by a U+0020 SPACE character, followed by "MANIFEST", then advance *position* to the next character after those. Otherwise, this isn't a cache manifest; abort this algorithm with a failure while checking for the magic signature.

9. Collect a sequence of characters that are U+0020 SPACE or U+0009 CHARACTER TABULATION (tab) characters.

10. If *position* is not past the end of *input* and the character at *position* is neither a U+000A LINE FEED (LF) characters nor a U+000D CARRIAGE RETURN (CR) character, then this isn't a cache manifest; abort this algorithm with a failure while checking for the magic signature.

11. This is a cache manifest. The algorithm cannot fail beyond this point (though bogus lines can get ignored).

12. Let *mode* be "explicit".

13. *Start of line*: If *position* is past the end of *input*, then jump to the last step. Otherwise, collect a sequence of characters that are U+000A LINE FEED (LF), U+000D CARRIAGE RETURN (CR), U+0020 SPACE, or U+0009 CHARACTER TABULATION (tab) characters.

14. Now, collect a sequence of characters that are *not* U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR) characters, and let the result be *line*.

15. Drop any trailing U+0020 SPACE, or U+0009 CHARACTER TABULATION (tab) characters at the end of *line*.

16. If *line* is the empty string, then jump back to the step labeled "start of line".

17. If the first character in *line* is a U+0023 NUMBER SIGN (#) character, then jump back to the step labeled "start of line".

18. If *line* equals "CACHE:" (the word "CACHE" followed by a U+003A COLON (:) character), then set *mode* to "explicit" and jump back to the step labeled "start of line".

19. If *line* equals "FALLBACK:" (the word "FALLBACK" followed by a U+003A COLON (:) character), then set *mode* to "fallback" and jump back to the step labeled "start of line".

20. If *line* equals "NETWORK:" (the word "NETWORK" followed by a U+003A COLON (:) character), then set *mode* to "online whitelist" and jump back to the step labeled "start of line".

21. This is either a data line or it is syntactically incorrect.

    ↪ **If *mode* is "explicit"**

    Resolve *line*.

    If this fails, then jump back to the step labeled "start of line".

    If the resulting absolute URL has a different <scheme> component than the manifest's URL (compared case-insensitively), then jump back to the step labeled "start of line".

    Drop the <fragment> component of the resulting absolute URL, if it has one.

    Add the resulting absolute URL to the *explicit URLs*.

    ↪ **If *mode* is "fallback"**

    If *line* does not contain at least one U+0020 SPACE or U+0009 CHARACTER TABULATION (tab) character, then jump back to the step labeled "start of line".

    Otherwise, let everything before the first U+0020 SPACE or U+0009 CHARACTER TABULATION (tab) character in *line* be *part one*, and let everything after the first U+0020 SPACE or U+0009 CHARACTER TABULATION (tab) character in *line* be *part two*.

    Resolve *part one* and *part two*.

    If either fails, then jump back to the step labeled "start of line".

If the absolute URL corresponding to *part one* does not have the same origin as the manifest's URL, then jump back to the step labeled "start of line".

If the resulting absolute URL for *part two* has a different <scheme> component than the manifest's URL (compared case-insensitively), then jump back to the step labeled "start of line".

Drop any the <fragment> components of the resulting absolute URLs.

If the absolute URL corresponding to *part one* is already in the *fallback URIs* mapping as an opportunistic caching namespace, then jump back to the step labeled "start of line".

Otherwise, add the absolute URL corresponding to *part one* to the *fallback URIs* mapping as an opportunistic caching namespace, mapped to the absolute URL corresponding to *part two* as the fallback entry.

↪ **If *mode* is "online whitelist"**

Resolve *line*.

If this fails, then jump back to the step labeled "start of line".

If the resulting absolute URL has a different <scheme> component than the manifest's URL (compared case-insensitively), then jump back to the step labeled "start of line".

Drop the <fragment> component of the resulting absolute URL, if it has one.

Add the resulting absolute URL to the *online whitelist URLs*.

22. Jump back to the step labeled "start of line". (That step jumps to the next, and last, step when the end of the file is reached.)

23. Return the *explicit URLs* list, the *fallback URLs* mapping, and the *online whitelist URLs*.

Note: If a resource is listed in both the online whitelist and in the explicit section, then that resource will be downloaded and cached, but when the page tries to use this resource, the user agent will ignore the cached copy and attempt to fetch the file from the network. Indeed, the cached copy will only be used if it is opened from a top-level browsing context.

5.7.4 Updating an application cache

When the user agent is required (by other parts of this specification) to start the **application cache update process**, the user agent must run the following steps:

the event stuff needs to be more consistent -- something about showing every step of the ui or no steps or something; and we need to deal with showing ui for browsing contexts that open when an update is already in progress, and we may need to give applications control over the ui the first time they cache themselves (right now the original cache is done without notifications to the browsing contexts)

1. Let *manifest URL* be the URL of the manifest of the cache to be updated.

2. Let *cache group* be the group of application caches identified by *manifest URL*.

3. Let *cache* be the most recently updated application cache identified by *manifest URL* (that is, the newest version found in *cache group*).

4. If the status of the *cache group* is either *checking* or *downloading*, then abort these steps, as an update is already in progress for them. Otherwise, set the status of this group of caches to *checking*. This entire step must be performed as one atomic operation so as to avoid race conditions.

5. If there is already a resource with the URL of *manifest URL* in *cache*, and that resource is categorized as a manifest, then this is an **upgrade attempt**. Otherwise, this is a **cache attempt**.

   Note: If this is a cache attempt, then cache is forcibly the only application cache in cache group, and it hasn't ever been populated from its manifest (i.e. this update is an attempt to download the application for the first time). It also can't have any browsing contexts associated with it.

6. Fire a simple event called `checking` at the `ApplicationCache` singleton of each top-level browsing context that is associated with a cache in *cache group*. The default action of this event should be the display of some sort of user interface indicating to the user that the user agent is checking for the availability of updates.

7. Fetch the resource from *manifest URL*, and let *manifest* be that resource.

   If the resource is labeled with the MIME type `text/cache-manifest`, parse *manifest* according to the rules for parsing manifests, obtaining a list of explicit entries, fallback entries and the opportunistic caching namespaces that map to them, and entries for the online whitelist.

8. If the previous step fails (e.g. the server returns a 4xx or 5xx response or equivalent, or there is a DNS error, or the connection times out, or the user cancels the download, or the parser for manifests fails when checking the magic signature), or if the resource is labeled with a MIME type other than `text/cache-manifest`, then run the cache failure steps.

9. If this is an upgrade attempt and the newly downloaded *manifest* is byte-for-byte identical to the manifest found in *cache*, or if the server reported it as "304 Not Modified" or equivalent, then run these substeps:

   1. Fire a simple event called `noupdate` at the `ApplicationCache` singleton of each top-level browsing context that is associated with a cache in *cache group*. The default action of this event should be the display of some sort of user interface indicating to the user that the application is up to date.

   2. If there are any pending downloads of implicit entries that are being stored in the cache, then wait for all of them to have completed. If any of these downloads fail (e.g. the server returns a 4xx or 5xx response or equivalent, or there is a DNS error, or the connection times out, or the user cancels the download), then run the cache failure steps.

   3. Let the status of the group of caches to which *cache* belongs be *idle*. If appropriate, remove any user interface indicating that an update for this cache is in progress.

   4. Abort the update process.

10. Set the status of *cache group* to *downloading*.

11. Fire a simple event called `downloading` at the `ApplicationCache` singleton of each top-level browsing context that is associated with a cache in *cache group*. The default action of this event should be the display of some sort of user interface indicating to the user that a new version is being downloaded.

12. If this is an upgrade attempt, then let *new cache* be a newly created application cache identified by manifest URL, being a new version in *cache group*. Otherwise, let *new cache* and *cache* be the same version of the application cache.

13. Let *file list* be an empty list of URLs with flags.

14. Add all the URLs in the list of explicit entries obtained by parsing *manifest* to *file list*, each flagged with "explicit entry".

15. Add all the URLs in the list of fallback entries obtained by parsing *manifest* to *file list*, each flagged with "fallback entry".

16. If this is an upgrade attempt, then add all the URLs of opportunistically cached entries in *cache* that match the opportunistic caching namespaces obtained by parsing *manifest* to *file list*, each flagged with "opportunistic entry".

17. If this is an upgrade attempt, then add all the URLs of implicit entries in *cache* to *file list*, each flagged with "implicit entry".

18. If this is an upgrade attempt, then add all the URLs of dynamic entries in *cache* to *file list*, each flagged with "dynamic entry".

19. If any URL is in *file list* more than once, then merge the entries into one entry for that URL, that entry having all the flags that the original entries had.

20. For each URL in *file list*, run the following steps. These steps may be run in parallel for two or more of the URLs at a time.

    1. Fire a simple event called `progress` at the `ApplicationCache` singleton of each top-level browsing context that is associated with a cache in *cache group*. The default action of this event should be the display of some sort of user interface indicating to the user that a file is being downloaded in preparation for updating the application.

    2. Fetch the resource. If this is an upgrade attempt, then use *cache* as an HTTP cache, and honor HTTP caching semantics (such as expiration, ETags, and so forth) with respect to that cache. User agents may also have other caches in place that are also honored. If the resource in question is already being downloaded for other reasons then the existing download process may be used for the purposes of this step.

       > An example of a resource that might already be being downloaded is a large image on a Web page that is being seen for the first time. The image would get downloaded to satisfy the `img` element on the page, as well as being listed in the cache manifest. According to the previous paragraph, that image only need be downloaded once, and it can be used both for the cache and for the rendered Web page.

    3. If the previous steps fails (e.g. the server returns a 4xx or 5xx response or equivalent, or there is a DNS error, or the connection times out, or the user cancels the download), then run the cache failure steps.

4. Otherwise, the fetching succeeded. Store the resource in the *new cache*.
5. If the URL being processed was flagged as an "explicit entry" in *file list*, then categorize the entry as an explicit entry.
6. If the URL being processed was flagged as a "fallback entry" in *file list*, then categorize the entry as a fallback entry.
7. If the URL being processed was flagged as a "opportunistic entry" in *file list*, then categorize the entry as an opportunistically cached entry.
8. If the URL being processed was flagged as an "implicit entry" in *file list*, then categorize the entry as a implicit entry.
9. If the URL being processed was flagged as an "dynamic entry" in *file list*, then categorize the entry as a dynamic entry.

21. Store *manifest* in *new cache*, if it's not there already, and categorize this entry (whether newly added or not) as the manifest.
22. Store the list of opportunistic caching namespaces, and the URLs of the fallback entries that they map to, in the new cache.
23. Store the URLs that form the new online whitelist in the new cache.
24. Wait for all pending downloads of implicit entries that are being stored in the cache to have completed.

    > For example, if the top-level browsing context's active document isn't itself listed in the cache manifest, then it might still be being downloaded.

    If any of these downloads fail (e.g. the server returns a 4xx or 5xx response or equivalent, or there is a DNS error, or the connection times out, or the user cancels the download), then run the cache failure steps.
25. If this is a cache attempt, then:

    Associate any `Document` objects that were flagged as candidates for this manifest URL's caches with *cache*.

    Fire a simple event called `cached` at the `ApplicationCache` singleton of each top-level browsing context that is associated with a cache in *cache group*. The default action of this event should be the display of some sort of user interface indicating to the user that the application has been cached and that they can now use it offline.

    Set the status of *cache group* to *idle*.
26. Otherwise, this is an upgrade attempt:

    Fire a simple event called `updateready` at the `ApplicationCache` singleton of each top-level browsing context that is associated with a cache in *cache group*. The default action of this event should be the display of some sort of user interface indicating to the user that a new version is available and that they can activate it by reloading the page.

    Set the status of *cache group* to *idle*.

The **cache failure steps** are as follows:

1. Fire a simple event called `error` at the `ApplicationCache` singleton of each top-level browsing context that is associated with a cache in *cache group*. The default action of this event should be the display of some sort of user interface indicating to the user that the user agent failed to save the application for offline use.

2. If this is a cache attempt, then discard *cache* and abort the update process.

3. Otherwise, let the status of the group of caches to which *cache* belongs be *idle*. If appropriate, remove any user interface indicating that an update for this cache is in progress. Abort the update process.

5.7.5 Processing model

The processing model of application caches for offline support in Web applications is part of the navigation model, but references the algorithms defined in this section.

A URL **matches an opportunistic caching namespace** if there exists an application cache whose manifest's URL has the same origin as the URL in question, and if that application cache has an opportunistic caching namespace with a <path> component that exactly matches the start of the <path> component of the URL being examined. If multiple opportunistic caching namespaces match the same URL, the one with the longest <path> component is the one that matches. A URL looking for an opportunistic caching namespace can match more than one application cache at a time, but only matches one namespace in each cache.

> If a manifest `http://example.com/app1/manifest` declares that `http://example.com/resources/images` should be opportunistically cached, and the user navigates to `http://example.com/resources/images/cat.png`, then the user agent will decide that the application cache identified by `http://example.com/app1/manifest` contains a namespace with a match for that URL.

When the **application cache selection algorithm** algorithm is invoked with a manifest URL, the user agent must run the first applicable set of steps from the following list:

↪ **If the resource is not being loaded as part of navigation of a top-level browsing context**

As an optimization, if the resource was loaded from an application cache, and the manifest URL of that cache doesn't match the manifest URL with which the algorithm was invoked, then the user agent should mark the entry in that application cache corresponding to the resource that was just loaded as being foreign.

Other than that, nothing special happens with respect to application caches.

↪ **If the resource being loaded was loaded from an application cache and the URL of that application cache's manifest is the same as the manifest URL with which the algorithm was invoked**

Associate the `Document` with the cache from which it was loaded. Invoke the application cache update process.

↪ **If the resource being loaded was loaded from an application cache and the URL of that application cache's manifest is *not* the same as the manifest URL with which the algorithm was invoked**

Mark the entry for this resource in the application cache from which it was loaded as foreign.

Restart the current navigation from the top of the navigation algorithm, undoing any changes that

were made as part of the initial load (changes can be avoided by ensuring that the step to update the session history with the new page is only ever completed *after* the application cache selection algorithm is run, though this is not required).

Note: The navigation will not result in the same resource being loaded, because "foreign" entries are never picked during navigation.

User agents may notify the user of the inconsistency between the cache manifest and the resource's own metadata, to aid in application development.

↪ **If the resource being loaded was not loaded from an application cache, but it was loaded using HTTP GET or equivalent**

1. If the manifest URL does not have the same origin as the resource's own URL, then invoke the application cache selection algorithm again, but without a manifest, and abort these steps.

2. If there is already an application cache identified by this manifest URL, and the most up to date version of that application cache contains a resource with the URL of the manifest, and that resource is categorized as a manifest, then: store the resource in the matching cache, categorized as an implicit entry, associate the `Document` with that cache, invoke the application cache update process, and abort these steps.

3. Flag the resource's `Document` as a candidate for this manifest URL's caches, so that it will be associated with an application cache identified by this manifest URL later, when such an application cache is ready.

4. If there is already an application cache identified by this manifest URL, then the most up to date version of that application cache does not yet contain a resource with the URL of the manifest, or it does but that resource is not yet categorized as a manifest: store the resource in that cache, categorized as an implicit entry (replacing the file's previous contents if it was already in the cache, but not removing any other categories it might have), and abort these steps. (An application cache update process is already in progress.)

5. Otherwise, there is no matching application cache: create a new application cache identified by this manifest URL, store the resource in that cache, categorized as an implicit entry, and then invoke the application cache update process.

↪ **Otherwise**

Invoke the application cache selection algorithm again, but without a manifest.

When the **application cache selection algorithm** is invoked *without* a manifest, then: if the resource is being loaded as part of navigation of a top-level browsing context, and the resource was fetched from a particular application cache, then the user agent must associate the `Document` with that application cache and invoke the application cache update process for that cache; otherwise, nothing special happens with respect to application caches.

5.7.5.1. Changes to the networking model

When a browsing context is associated with an application cache, any and all resource loads must go through the following steps instead of immediately invoking the mechanisms appropriate to that resource's scheme:

1. If the resource is not to be fetched using the HTTP GET mechanism or equivalent, then fetch the resource normally and abort these steps.

2. If the resource's URL, ignoring its fragment identifier if any, is listed in the application cache's online whitelist, then fetch the resource normally and abort these steps.

3. If the resource's URL is an implicit entry, the manifest, an explicit entry, a fallback entry, an opportunistically cached entry, or a dynamic entry in the application cache, then fetch the resource from the cache and abort these steps.

4. If the resource's URL has the same origin as the manifest's URL, and the start of the resource's URL's <path> component is exactly matched by the <path> component of an opportunistic caching namespace in the application cache, then:

   Fetch the resource normally. If this results 4xx or 5xx status codes or equivalent, or if there were network errors (but not if the user canceled the download), then instead fetch, from the cache, the resource of the fallback entry corresponding to the namespace with the longest matching <path> component. Abort these steps.

5. Fail the resource load.

Note: The above algorithm ensures that resources that are not present in the manifest will always fail to load (at least, after the cache has been primed the first time), making the testing of offline applications simpler.

5.7.6 Application cache API

```
interface ApplicationCache {

  // update status
  const unsigned short UNCACHED = 0;
  const unsigned short IDLE = 1;
  const unsigned short CHECKING = 2;
  const unsigned short DOWNLOADING = 3;
  const unsigned short UPDATEREADY = 4;
  readonly attribute unsigned short status;

  // updates
  void update();
  void swapCache();

  // dynamic entries
  readonly attribute unsigned long length;
  DOMString item(in unsigned long index);
  void add(in DOMString url);
  void remove(in DOMString url);

  // events
           attribute EventListener onchecking;
```

```
            attribute EventListener onerror;
            attribute EventListener onnoupdate;
            attribute EventListener ondownloading;
            attribute EventListener onprogress;
            attribute EventListener onupdateready;
            attribute EventListener oncached;

};
```

Objects implementing the `ApplicationCache` interface must also implement the `EventTarget` interface.

There is a one-to-one mapping from `Document` objects to `ApplicationCache` objects. The **`applicationCache`** attribute on `Window` objects must return the `ApplicationCache` object associated with the active document of the `Window`'s browsing context.

An `ApplicationCache` object might be associated with an application cache. When the `Document` object that the `ApplicationCache` object maps to is associated with an application cache, then that is the application cache with which the `ApplicationCache` object is associated. Otherwise, the `ApplicationCache` object is associated with the application cache that the `Document` object's browsing context is associated with, if any.

The **`status`** attribute, on getting, must return the current state of the application cache `ApplicationCache` object is associated with, if any. This must be the appropriate value from the following list:

***`UNCACHED`* (numeric value 0)**

The `ApplicationCache` object is not associated with an application cache at this time.

***`IDLE`* (numeric value 1)**

The `ApplicationCache` object is associated with an application cache whose group is in the *idle* update status, and that application cache is the newest cache in its group that contains a resource categorized as a manifest.

***`CHECKING`* (numeric value 2)**

The `ApplicationCache` object is associated with an application cache whose group is in the *checking* update status.

***`DOWNLOADING`* (numeric value 3)**

The `ApplicationCache` object is associated with an application cache whose group is in the *downloading* update status.

***`UPDATEREADY`* (numeric value 4)**

The `ApplicationCache` object is associated with an application cache whose group is in the *idle* update status, but that application cache is *not* the newest cache in its group that contains a resource categorized as a manifest.

The **`length`** attribute must return the number of dynamic entries in the application cache with which the `ApplicationCache` object is associated, if any, and zero if the object is not associated with any application cache.

The dynamic entries in the application cache are ordered in the same order as they were added to the cache

by the `add()` method, with the oldest entry being the zeroth entry, and the most recently added entry having the index `length`-1.

The **`item(`*`index`*`)`** method must return the absolute URL of the dynamic entry with index *index* from the application cache, if one is associated with the `ApplicationCache` object. If the object is not associated with any application cache, or if the *index* argument is lower than zero or greater than `length`-1, the method must instead raise an `INDEX_SIZE_ERR` exception.

The **`add(`*`url`*`)`** method must run the following steps:

1. If the `ApplicationCache` object is not associated with any application cache, then raise an `INVALID_STATE_ERR` exception and abort these steps.
2. Resolve the *url* argument. If this fails, raise a `SYNTAX_ERR` exception and abort these steps.
3. If there is already a resource in in the application cache with which the `ApplicationCache` object is associated that has the address *url*, then ensure that entry is categorized as a dynamic entry and return and abort these steps.
4. If *url* has a different <scheme> component than the manifest's URL, then raise a security exception.
5. Return, but do not abort these steps.
6. Fetch the resource referenced by *url*.
7. If this results 4xx or 5xx status codes or equivalent, or if there were network errors, or if the user canceled the download, then abort these steps.
8. Wait for there to be no running scripts, or at least no running scripts that can reach an `ApplicationCache` object associated with the application cache with which this `ApplicationCache` object is associated.

   Add the fetched resource to the application cache and categorize it as a dynamic entry before letting any such scripts resume.

We can make the add() API more usable (i.e. make it possible to detect progress and distinguish success from errors without polling and timeouts) if we have the method return an object that is a target of Progress Events, much like the XMLHttpRequestEventTarget interface. This would also make this far more complex to spec and implement.

The **`remove(`*`url`*`)`** method must resolve the *url* argument and, if that is successful, remove the dynamic entry categorization of any entry whose address is the resulting absolute URL in the application cache with which the `ApplicationCache` object is associated. If this removes the last categorization of an entry in that cache, then the entry must be removed entirely (such that if it is re-added, it will be loaded from the network again). If the `ApplicationCache` object is not associated with any application cache, then the method must raise an `INVALID_STATE_ERR` exception instead.

If the **`update()`** method is invoked, the user agent must invoke the application cache update process, in the background, for the application cache with which the `ApplicationCache` object is associated. If there is no such application cache, then the method must raise an `INVALID_STATE_ERR` exception instead.

If the **`swapCache()`** method is invoked, the user agent must run the following steps:

1. Let *document* be the `Document` with which the `ApplicationCache` object is associated.
2. Check that *document* is associated with an application cache. If it is not, then raise an `INVALID_STATE_ERR` exception and abort these steps.

   Note: This is not the same thing as the `ApplicationCache` object being itself associated with an application cache! In particular, the `Document` with which the `ApplicationCache` object is associated can only itself be associated with an application cache if it is in a top-level browsing context.

3. Let *cache* be the application cache with which the `ApplicationCache` object is associated. (By definition, this is the same as the one that was found in the previous step.)
4. Check that there is an application cache in the same group as *cache* which has an entry categorized as a manifest that has is newer than *cache*. If there is not, then raise an `INVALID_STATE_ERR` exception and abort these steps.
5. Let *new cache* be the newest application cache in the same group as *cache* which has an entry categorized as a manifest.
6. Unassociate *document* from *cache* and instead associate it with *new cache*.

The following are the event handler DOM attributes that must be supported by objects implementing the `ApplicationCache` interface:

onchecking

Must be invoked whenever an `checking` event is targeted at or bubbles through the `ApplicationCache` object.

onerror

Must be invoked whenever an `error` event is targeted at or bubbles through the `ApplicationCache` object.

onnoupdate

Must be invoked whenever an `noupdate` event is targeted at or bubbles through the `ApplicationCache` object.

ondownloading

Must be invoked whenever an `downloading` event is targeted at or bubbles through the `ApplicationCache` object.

onprogress

Must be invoked whenever an `progress` event is targeted at or bubbles through the `ApplicationCache` object.

onupdateready

Must be invoked whenever an `updateready` event is targeted at or bubbles through the `ApplicationCache` object.

oncached

Must be invoked whenever a `cached` event is targeted at or bubbles through the `ApplicationCache` object.

5.7.7 Browser state

The **`navigator.onLine`** attribute must return false if the user agent will not contact the network when the user follows links or when a script requests a remote page (or knows that such an attempt would fail), and must return true otherwise.

When the value that would be returned by the `navigator.onLine` attribute of the `Window` changes from true to false, the user agent must fire a simple event called **`offline`** at the body element.

On the other hand, when the value that would be returned by the `navigator.onLine` attribute of the `Window` changes from false to true, the user agent must fire a simple event called **`online`** at the body element.

5.8 Session history and navigation

5.8.1 The session history of browsing contexts

The sequence of `Document`s in a browsing context is its **session history**.

`History` objects provide a representation of the pages in the session history of browsing contexts. Each browsing context has a distinct session history.

Each `Document` object in a browsing context's session history is associated with a unique instance of the `History` object, although they all must model the same underlying session history.

The **`history`** attribute of the `Window` interface must return the object implementing the `History` interface for that `Window` object's active document.

`History` objects represent their browsing context's session history as a flat list of session history entries. Each **session history entry** consists of either a URL or a state object, or both, and may in addition have a title, a `Document` object, form data, a scroll position, and other information associated with it.

Note: This does not imply that the user interface need be linear. See the notes below.

URLs without associated state objects are added to the session history as the user (or script) navigates from page to page.

A **state object** is an object representing a user interface state.

Pages can add state objects between their entry in the session history and the next ("forward") entry. These are then returned to the script when the user (or script) goes back in the history, thus enabling authors to use the "navigation" metaphor even in one-page applications.

Every `Document` in the session history is defined to have a **last activated entry**, which is the state object entry associated with that `Document` which was most recently activated. Initially, the last activated entry of a

`Document` must be the first entry for the `Document`, representing the fact that no state object entry has yet been activated.

At any point, one of the entries in the session history is the **current entry**. This is the entry representing the active document of the browsing context. The current entry is usually an entry for the location of the `Document`. However, it can also be one of the entries for state objects added to the history by that document.

Entries that consist of state objects share the same `Document` as the entry for the page that was active when they were added.

Contiguous entries that differ just by fragment identifier also share the same `Document`.

> ***Note: All entries that share the same `Document` (and that are therefore merely different states of one particular document) are contiguous by definition.***

User agents may discard the `Document` objects of entries other than the current entry that are not referenced from any script, reloading the pages afresh when the user or script navigates back to such pages. This specification does not specify when user agents should discard `Document` objects and when they should cache them.

Entries that have had their `Document` objects discarded must, for the purposes of the algorithms given below, act as if they had not. When the user or script navigates back or forwards to a page which has no in-memory DOM objects, any other entries that shared the same `Document` object with it must share the new object as well.

When state object entries are added, a URL can be provided. This URL is used to replace the state object entry if the `Document` is evicted.

When a user agent discards the `Document` object from an entry in the session history, it must also discard all the entries that share that `Document` but do not have an associated URL (i.e. entries that only have a state object). Entries that shared that `Document` object but had a state object and have a different URL must then have their *state objects* removed. Removed entries are not recreated if the user or script navigates back to the page. If there are no state object entries for that `Document` object then no entries are removed.

5.8.2 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, in DOMString title);
  void pushState(in DOMObject data, in DOMString title, in DOMString url);
  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 traverse the history to the specified entry. The *specified entry* is the one whose index equals the index of the current entry plus *delta*.

When the user navigates through a browsing context, e.g. using a browser's back and forward buttons, the user agent must translate this action into the equivalent invocations of the `history.go(delta)` method on the various affected `window` objects.

Some of the other members of the `History` interface are defined in terms of the `go()` method, as follows:

Member	Definition
`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, title, url)`** method adds a state object to the history.

When this method is invoked, the user agent must run the following steps:

1. If a third argument is specified, run these substeps:
    1. Resolve the value of the third argument.
    2. If that fails, raise a security exception and abort the `pushState()` steps.
    3. Compare the resulting absolute URL to the document's address. If any part of these two URLs differ other than the <path>, <query>, and <fragment> components, then raise a security exception and abort the `pushState()` steps.

    For the purposes of the comparison in the above substeps, the <path> and <query> components can only be the same if the URLs use a hierarchical <scheme>.
2. Remove from the session history any entries for the `Document` from the entry after the current entry up to the last entry in the session history that references the same `Document` object, if any. If the current entry is the last entry in the session history, or if there are no entries after the current entry that reference the same `Document` object, then no entries are removed.
3. Add a state object entry to the session history, after the current entry, with the specified *data* as the state object, the given *title* as the title, and, if the third argument is present, the absolute URL that was found in the first step as the URL of the entry.
4. Set this new entry as being the last activated entry for the `Document`.
5. Update the current entry to be the this newly added entry.

Note: The title is purely advisory. User agents might use the title in the user interface.

User agents may limit the number of state objects added to the session history per page. If a page hits the UA-defined limit, user agents must remove the entry immediately after the first entry for that `Document` object in the session history after having added the new entry. (Thus the state history acts as a FIFO buffer for eviction, but as a LIFO buffer for navigation.)

The **`clearState()`** method removes all the state objects for the `Document` object from the session history.

When this method is invoked, the user agent must remove from the session history all the entries from the first state object entry for that `Document` object up to the last entry that references that same `Document` object, if any.

Then, if the current entry was removed in the previous step, the current entry must be set to the last entry for that `Document` object in the session history.

5.8.3 Activating state object entries

When an entry in the session history is activated (which happens during session traversal, as described above), the user agent must run the following steps:

1. First, the user agent must set this new entry as being the last activated entry for the `Document` to which the entry belongs.
2. If the entry is a state object entry, let *state* be that state object. Otherwise, the entry is the first entry for the `Document`; let *state* be null.
3. The user agent must then fire a **`popstate`** event in no namespace on the body element using the `PopStateEvent` interface, with the `state` attribute set to the value of *state*. 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, in
boolean cancelableArg, in DOMObject stateArg);
  void initPopStateEventNS(in DOMString namespaceURIArg, in DOMString
typeArg, in boolean canBubbleArg, in boolean cancelableArg, in DOMObject
stateArg);
};
```

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, or null, if the state represented is the initial state of the `Document`.

5.8.4 The `Location` interface

Each `Document` object in a browsing context's session history is associated with a unique instance of a `Location` object.

The **location** attribute of the HTMLDocument interface must return the Location object for that Document object.

The **location** attribute of the Window interface must return the Location object for that Window object's active document.

Location objects provide a representation of their document's address, and allow the current entry of the browsing context's session history to be changed, by adding or replacing entries in the history object.

```
interface Location {
  readonly attribute DOMString href;
  void assign(in DOMString url);
  void replace(in DOMString url);
  void reload();

    // URL decomposition attributes
           attribute DOMString protocol;
           attribute DOMString host;
           attribute DOMString hostname;
           attribute DOMString port;
           attribute DOMString pathname;
           attribute DOMString search;
           attribute DOMString hash;
};
```

The **href** attribute must return the address of the page represented by the associated Document object, as an absolute URL.

On setting, the user agent must act as if the assign() method had been called with the new value as its argument.

When the **assign(*url*)** method is invoked, the UA must navigate the browsing context to the specified *url*.

When the **replace(*url*)** method is invoked, the UA must navigate the browsing context to the specified *url* with replacement enabled.

Navigation for the assign() and replace() methods must be done with the script browsing context of the script that invoked the method as the source browsing context.

The Location interface also has the complement of URL decomposition attributes, **protocol**, **host**, **port**, **hostname**, **pathname**, **search**, and **hash**. These must follow the rules given for URL decomposition attributes, with the input being the address of the page represented by the associated Document object, as an absolute URL (same as the href attribute), and the common setter action being the same as setting the href attribute to the new output value.

5.8.4.1. Security

User agents must raise a security exception whenever any of the members of a Location object are accessed by scripts whose effective script origin is not the same as the Location object's associated Document's effective script origin, with the following exceptions:

- The `href` setter, if the script is running in a browsing context that is allowed to navigate the browsing context with which the `Location` object is associated

User agents must not allow scripts to override the `href` attribute's setter.

5.8.5 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.

5.9 Browsing the Web

5.9.1 Navigating across documents

Certain actions cause the browsing context to **navigate** to a new resource. Navigation always involves **source browsing context**, which is the browsing context which was responsible for starting the navigation.

> For example, following a hyperlink, form submission, and the `window.open()` and `location.assign()` methods can all cause a browsing context to navigate.

A user agent may provide various ways for the user to explicitly cause a browsing context to navigate, in addition to those defined in this specification.

When a browsing context is navigated to a new resource, the user agent must run the following steps:

1. If the source browsing context is not the same as the browsing context being navigated, and the source browsing context is not one of the ancestor browsing contexts of the browsing context being navigated, and the source browsing context has its sandboxed navigation browsing context flag set, then abort these steps. The user agent may offer to open the new resource in a new top-level browsing context or in the top-level browsing context of the source browsing context, at the user's option, in which case the user agent must navigate that designated top-level browsing context to the new resource as if the user

had requested it independently.

2. If the source browsing context is the same as the browsing context being navigated, and this browsing context has its seamless browsing context flag set, then find the nearest ancestor browsing context that does not have its seamless browsing context flag set, and continue these steps as if *that* browsing context was the one that was going to be navigated instead.

3. Cancel any preexisting attempt to navigate the browsing context.

4. Resolve the URL of the new resource. If that fails, the user agent may abort these steps, or may treat the URL as identifying some sort of user-agent defined error resource, which could display some sort of inline content, or could be handled using a mechanism that does not affect the browsing context.

5. *Fragment identifiers*: If the absolute URL of the new resource is the same as the address of the active document of the browsing context being navigated, ignoring any <fragment> components of those URLs, and the new resource is to be fetched using HTTP GET or equivalent, then navigate to that fragment identifier and abort these steps.

6. If the new resource is to be handled by displaying some sort of inline content, e.g. an error message because the specified scheme is not one of the supported protocols, or an inline prompt to allow the user to select a registered handler for the given scheme, then display the inline content and abort these steps.

7. If the new resource is to be handled using a mechanism that does not affect the browsing context, then abort these steps and proceed with that mechanism instead.

8. If the new resource is to be fetched using HTTP GET or equivalent, and if the browsing context being navigated is a top-level browsing context, then check if there are any application caches that have a manifest with the same origin as the URL in question, and that have this URL as one of their entries (excluding entries marked as foreign), and that already contain their manifest, categorized as a manifest. If so, then the user agent must then fetch the resource from the most appropriate application cache of those that match.

   Otherwise, start fetching the new resource in the appropriate manner (e.g. performing an HTTP GET or POST operation, or reading the file from disk, or executing script in the case of a `javascript:` URL). If this results in a redirect, return to the step labeled "fragment identifiers" with the new resource.

   If fetching the resource is instantaneous, as it should be for `javascript:` URLs and `about:blank`, then this must be synchronous, but if fetching the resource depends on external resources, as it usually does for URLs that use HTTP or other networking protocols, then at this point the user agents must yield to whatever script invoked the navigation steps, if they were invoked by script.

   > For example, imagine an HTML page with an associated application cache displaying an image and a form, where the image is also used by several other application caches. If the user right-clicks on the image and chooses "View Image", then the user agent could decide to show the image from any of those caches, but it is likely that the most useful cache for the user would be the one that was used for the aforementioned HTML page. On the other hand, if the user submits the form, and the form does a POST submission, then the user agent will not use an application cache at all; the submission will be made to the network.

9. Wait for one or more bytes to be available or for the user agent to establish that the resource in question is empty. During this time, the user agent may allow the user to cancel this navigation attempt or start other navigation attempts.

10. If the resource was not fetched from an application cache, and was to be fetched using HTTP GET or equivalent, and its URL matches the opportunistic caching namespace of one or more application caches, and the user didn't cancel the navigation attempt during the previous step, then:

    ↪ **If the browsing context being navigated is a top-level browsing context, and the navigation attempt failed (e.g. the server returned a 4xx or 5xx status code or equivalent, or there was a DNS error)**

    Let *candidate* be the fallback resource specified for the opportunistic caching namespace in question. If multiple application caches match, the user agent must use the fallback of the most appropriate application cache of those that match.

    If *candidate* is not marked as foreign, then the user agent must discard the failed load and instead continue along these steps using *candidate* as the resource.

    For the purposes of session history (and features that depend on session history, e.g. bookmarking) the user agent must use the URL of the resource that was requested (the one that matched the opportunistic caching namespace), not the fallback resource. However, the user agent may indicate to the user that the original page load failed, that the page used was a fallback resource, and what the URL of the fallback resource actually is.

    ↪ **Otherwise**

    Once the download is complete, if there were no errors and the user didn't cancel the request, the user agent must cache the resource in all the application caches that have a matching opportunistic caching namespace, categorized as opportunistically cached entries. Meanwhile, the user must continue along these steps.

11. If the document's out-of-band metadata (e.g. HTTP headers), not counting any type information (such as the Content-Type HTTP header), requires some sort of processing that will not affect the browsing context, then perform that processing and abort these steps.

    Such processing might be triggered by, amongst other things, the following:

    - ***HTTP status codes (e.g. 204 No Content or 205 Reset Content)***
    - ***HTTP Content-Disposition headers***
    - ***Network errors***

12. Let *type* be the sniffed type of the resource.

13. If the user agent has been configured to process resources of the given *type* using some mechanism other than rendering the content in a browsing context, then skip this step. Otherwise, if the *type* is one of the following types, jump to the appropriate entry in the following list, and process the resource as described there:

    ↪ **"text/html"**

    Follow the steps given in the HTML document section, and abort these steps.

    ↪ **Any type ending in "+xml"**

    ↪ **"application/xml"**

    ↪ **"text/xml"**

    Follow the steps given in the XML document section. If that section determines that the content is *not* to be displayed as a generic XML document, then proceed to the next step in

this overall set of steps. Otherwise, abort these steps.

↪ **"text/plain"**

Follow the steps given in the plain text file section, and abort these steps.

↪ **A supported image type**

Follow the steps given in the image section, and abort these steps.

↪ **A type that will use an external application to render the content in the browsing context**

Follow the steps given in the plugin section, and abort these steps.

14. *Non-document content*: If, given *type*, the new resource is to be handled by displaying some sort of inline content, e.g. a native rendering of the content, an error message because the specified type is not supported, or an inline prompt to allow the user to select a registered handler for the given type, then display the inline content and abort these steps.

15. Otherwise, the document's *type* is such that the resource will not affect the browsing context, e.g. because the resource is to be handed to an external application. Process the resource appropriately.

Some of the sections below, to which the above algorithm defers in certain cases, require the user agent to **update the session history with the new page**. When a user agent is required to do this, it must follows the set of steps given below that is appropriate for the situation at hand. From the point of view of any script, these steps must occur atomically.

1. pause for scripts
2. onbeforeunload, and if present set flag that we will kill document
3. onunload, and if present set flag that we will kill document
4. if flag is set: discard the `Document`
5. **If the navigation was initiated for *entry update* of an entry**

    1. Replace the entry being updated with a new entry representing the new resource and its `Document` object and related state. The user agent may propagate state from the old entry to the new entry (e.g. scroll position).

    2. Traverse the history to the new entry.

    Otherwise

    1. Remove all the entries after the current entry in the browsing context's `Document` object's `History` object.

        Note: This doesn't necessarily have to affect the user agent's user interface.

    2. Append a new entry at the end of the `History` object representing the new resource and

its `Document` object and related state.

3. Traverse the history to the new entry.

4. If the navigation was initiated with **replacement enabled**, remove the entry immediately before the new current entry in the session history.

5.9.2 Page load processing model for HTML files

When an HTML document is to be loaded in a browsing context, the user agent must create a `Document` object, mark it as being an HTML document, create an HTML parser, associate it with the document, and begin to use the bytes provided for the document as the input stream for that parser.

> ***Note: The input stream converts bytes into characters for use in the tokeniser. This process relies, in part, on character encoding information found in the real Content-Type metadata of the resource; the "sniffed type" is not used for this purpose.***

When no more bytes are available, an EOF character is implied, which eventually causes a `load` event to be fired.

After creating the `Document` object, but potentially before the page has finished parsing, the user agent must update the session history with the new page.

> ***Note: Application cache selection happens in the HTML parser.***

5.9.3 Page load processing model for XML files

When faced with displaying an XML file inline, user agents must first create a `Document` object, following the requirements of the XML and Namespaces in XML recommendations, RFC 3023, DOM3 Core, and other relevant specifications. [XML] [XMLNS] [RFC3023] [DOM3CORE]

The actual HTTP headers and other metadata, not the headers as mutated or implied by the algorithms given in this specification, are the ones that must be used when determining the character encoding according to the rules given in the above specifications. Once the character encoding is established, the document's character encoding must be set to that character encoding.

If the root element, as parsed according to the XML specifications cited above, is found to be an `html` element with an attribute `manifest`, then, as soon as the element is inserted into the DOM, the user agent must resolve the value of that attribute, and if that is successful, must run the application cache selection algorithm with the resulting absolute URL as the manifest URL. Otherwise, if the attribute is absent or resolving it fails, then as soon as the root element is inserted into the DOM, the user agent must run the application cache selection algorithm with no manifest.

> ***Note: Because the processing of the `manifest` attribute happens only once the root element is parsed, any URLs referenced by processing instructions before the root element (such as `<?xml-styleesheet?>` and `<?xbl?>` PIs) will be fetched from the network and cannot be cached.***

User agents may examine the namespace of the root `Element` node of this `Document` object to perform namespace-based dispatch to alternative processing tools, e.g. determining that the content is actually a syndication feed and passing it to a feed handler. If such processing is to take place, abort the steps in this section, and jump to the next step (labeled "non-document content") in the navigate steps above.

Otherwise, then, with the newly created `Document`, the user agents must update the session history with the new page. User agents may do this before the complete document has been parsed (thus achieving *incremental rendering*).

Error messages from the parse process (e.g. namespace well-formedness errors) may be reported inline by mutating the `Document`.

5.9.4 Page load processing model for text files

When a plain text document is to be loaded in a browsing context, the user agent should create a `Document` object, mark it as being an HTML document, create an HTML parser, associate it with the document, act as if the tokeniser had emitted a start tag token with the tag name "pre", set the tokenization stage's content model flag to *PLAINTEXT*, and begin to pass the stream of characters in the plain text document to that tokeniser.

The rules for how to convert the bytes of the plain text document into actual characters are defined in RFC 2046, RFC 2646, and subsequent versions thereof. [RFC2046] [RFC2646]

The document's character encoding must be set to the character encoding used to decode the document.

Upon creation of the `Document` object, the user agent must run the application cache selection algorithm with no manifest.

When no more character are available, an EOF character is implied, which eventually causes a `load` event to be fired.

After creating the `Document` object, but potentially before the page has finished parsing, the user agent must update the session history with the new page.

User agents may add content to the `head` element of the `Document`, e.g. linking to stylesheet or an XBL binding, providing script, giving the document a `title`, etc.

5.9.5 Page load processing model for images

When an image resource is to be loaded in a browsing context, the user agent should create a `Document` object, mark it as being an HTML document, append an `html` element to the `Document`, append a `head` element and a `body` element to the `html` element, append an `img` to the `body` element, and set the `src` attribute of the `img` element to the address of the image.

Then, the user agent must act as if it had stopped parsing.

Upon creation of the `Document` object, the user agent must run the application cache selection algorithm with no manifest.

After creating the `Document` object, but potentially before the page has finished fully loading, the user agent must update the session history with the new page.

User agents may add content to the `head` element of the `Document`, or attributes to the `img` element, e.g. to link to stylesheet or an XBL binding, to provide a script, to give the document a `title`, etc.

5.9.6 Page load processing model for content that uses plugins

When a resource that requires an external resource to be rendered is to be loaded in a browsing context, the user agent should create a `Document` object, mark it as being an HTML document, append an `html` element to the `Document`, append a `head` element and a `body` element to the `html` element, append an `embed` to the `body` element, and set the `src` attribute of the `img` element to the address of the image.

Then, the user agent must act as if it had stopped parsing.

Upon creation of the `Document` object, the user agent must run the application cache selection algorithm with no manifest.

After creating the `Document` object, but potentially before the page has finished fully loading, the user agent must update the session history with the new page.

User agents may add content to the `head` element of the `Document`, or attributes to the `embed` element, e.g. to link to stylesheet or an XBL binding, or to give the document a `title`.

> ***Note: If the sandboxed plugins browsing context flag is set on the browsing context, the synthesized `embed` element will fail to render the content.***

5.9.7 Page load processing model for inline content that doesn't have a DOM

When the user agent is to display a user agent page inline in a browsing context, the user agent should create a `Document` object, mark it as being an HTML document, and then either associate that `Document` with a custom rendering that is not rendered using the normal `Document` rendering rules, or mutate that `Document` until it represents the content the user agent wants to render.

Once the page has been set up, the user agent must act as if it had stopped parsing.

Upon creation of the `Document` object, the user agent must run the application cache selection algorithm with no manifest.

After creating the `Document` object, but potentially before the page has been completely set up, the user agent must update the session history with the new page.

5.9.8 Navigating to a fragment identifier

When a user agent is supposed to navigate to a fragment identifier, then the user agent must update the session history with the new page, where "the new page" has the same `Document` as before but with the URL having the newly specified fragment identifier.

Part of that algorithm involves the user agent having to scroll to the fragment identifier, which is the important part for this step.

When the user agent is required to **scroll to the fragment identifier**, it must change the scrolling position of

the document, or perform some other action, such that the indicated part of the document is brought to the user's attention. If there is no indicated part, then the user agent must not scroll anywhere.

The indicated part of the document is the one that the fragment identifier, if any, identifies. The semantics of the fragment identifier in terms of mapping it to a specific DOM Node is defined by the MIME type specification of the document's MIME Type (for example, the processing of fragment identifiers for XML MIME types is the responsibility of RFC3023).

For HTML documents (and the `text/html` MIME type), the following processing model must be followed to determine what the indicated part of the document is.

1. Parse the URL, and let *fragid* be the <fragment> component of the URL.
2. If *fragid* is the empty string, then the indicated part of the document is the top of the document.
3. If there is an element in the DOM that has an ID exactly equal to *fragid*, then the first such element in tree order is the indicated part of the document; stop the algorithm here.
4. If there is an `a` element in the DOM that has a `name` attribute whose value is exactly equal to *fragid*, then the first such element in tree order is the indicated part of the document; stop the algorithm here.
5. Otherwise, there is no indicated part of the document.

For the purposes of the interaction of HTML with Selectors' `:target` pseudo-class, the *target element* is the indicated part of the document, if that is an element; otherwise there is no *target element*. [SELECTORS]

5.9.9 History traversal

When a user agent is required to **traverse the history** to a *specified entry*, the user agent must act as follows:

1. If there is no longer a `Document` object for the entry in question, the user agent must navigate the browsing context to the location for that entry to perform an entry update of that entry, and abort these steps. The "navigate" algorithm reinvokes this "traverse" algorithm to complete the traversal, at which point there *is* a `Document` object and so this step gets skipped. The navigation must be done using the same source browsing context as was used the first time this entry was created.
2. If appropriate, update the current entry in the browsing context's `Document` object's `History` object to reflect any state that the user agent wishes to persist.

   > For example, some user agents might want to persist the scroll position, or the values of form controls.

3. If the *specified entry* has a different `Document` object than the current entry then the user agent must run the following substeps:

   1. freeze any timers, intervals, XMLHttpRequests, database transactions, etc
   2. If there are any `MessagePort` objects whose owner is the browsing context's default view's `Window` object and that are entangled with another port, the user agent must deactivate all such ports and let the `Document`'s list of message ports be a list of those ports.

3. The user agent must move any properties that have been added to the browsing context's default view's `Window` object to the active document's `Document`'s list of added properties.

4. If the browsing context is a top-level browsing context (and not an auxiliary browsing context), and the origin of the `Document` of the *specified entry* is not the same as the origin of the `Document` of the current entry, then the following sub-sub-steps must be run:
    1. The current browsing context name must be stored with all the entries in the history that are associated with `Document` objects with the same origin as the active document *and* that are contiguous with the current entry.
    2. The browsing context's browsing context name must be unset.

5. The user agent must make the *specified entry*'s `Document` object the active document of the browsing context. (If it is a top-level browsing context, this might change which application cache it is associated with.)

6. If the *specified entry* has a browsing context name stored with it, then the following sub-sub-steps must be run:
    1. The browsing context's browsing context name must be set to the name stored with the specified entry.
    2. Any browsing context name stored with the entries in the history that are associated with `Document` objects with the same origin as the new active document, and that are contiguous with the specified entry, must be cleared.

7. The user agent must move any properties that have been added to the active document's `Document`'s list of added properties to browsing context's default view's `Window` object.

8. If the active document's list of message ports is not empty, then the user agent must reactivate all the ports in that list. Empty that list of message ports.

9. unfreeze any timers, intervals, XMLHttpRequests, database transactions, etc

4. If there are any entries with state objects between the last activated entry for the `Document` of the *specified entry* and the *specified entry* itself (not inclusive), then the user agent must iterate through every entry between that last activated 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.

5. If the *specified entry* is a state object or the first entry for a `Document`, the user agent must activate that entry.

6. If the *specified entry* has a URL that differs from the current entry's only by its fragment identifier, and the two share the same `Document` object, then fire a simple event with the name `hashchanged` at the `body` element, and, if the new URL has a fragment identifier, scroll to the fragment identifier.

7. 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.

8. The current entry is now the *specified entry*.

how does the changing of the global attributes affect .watch() when seen from other Windows?

5.9.10 Closing a browsing context

Closing a browsing context and discarding it (vs closing it and keeping it around in memory).

when a browsing context is closed, all session history entries' `Document` objects must be discarded.

When a user agent is to **discard a `Document`**, any frozen timers, intervals, XMLHttpRequests, database transactions, etc, must be killed, and any ports owned by the Window object that aren't in a list of message ports, if the document is the active document, or otherwise any ports in that document's list of message ports, if the document is not the active document, must be deactivated and unentangled..

Also, `unload` events should fire.

5.10 Structured client-side storage

5.10.1 Storing name/value pairs

5.10.1.1. Introduction

This section is non-normative.

This specification introduces two related mechanisms, similar to HTTP session cookies, for storing structured data on the client side. [RFC2109] [RFC2965]

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 the same site 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 client side for performance reasons.

Again, cookies do not handle this case well, because they are transmitted with every request.

The `localStorage` DOM attribute is used to access a page's local storage area.

> 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:
>
> ```
> 
>   You have viewed this page
>   an untold number of
>   time(s).
> 
> <script>
>   if (!localStorage.pageLoadCount)
>     localStorage.pageLoadCount = 0;
>   localStorage.pageLoadCount = parseInt(localStorage.pageLoadCount, 10) +
> 1;
>   document.getElementById('count').textContent =
> localStorage.pageLoadCount;
> </script>
> ```

Each site has its own separate storage area.

Storage areas (both session storage and local storage) store strings. To store structured data in a storage area, you must first convert it to a string.

5.10.1.2. The `Storage` interface

```
interface Storage {
  readonly attribute unsigned long length;
  [IndexGetter] DOMString key(in unsigned long index);
  [NameGetter] DOMString getItem(in DOMString key);
  [NameSetter] void setItem(in DOMString key, in DOMString data);
  [XXX] void removeItem(in DOMString key);
  void clear();
};
```

Each `Storage` object provides access to a list of key/value pairs, which are sometimes called items. Keys and values are strings. Any string (including the empty string) is a valid key.

Note: To store more structured data, authors may consider using the SQL interfaces instead.

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 `localStorage` attributes. Multiple separate objects implementing the

`Storage` interface can all be associated with the same list of key/value pairs simultaneously.

The **`length`** attribute must return the number of key/value pairs currently present in the list associated with the object.

The **`key(n)`** method must return the name of the *n*th 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 current value associated with the given *key*. If the given *key* does not exist in the list associated with the object then this method must return null.

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*.

If the given *key does* exist in the list, then it must have its value updated to the value given in the *value* argument.

If it couldn't set the new value, the method must raise an `INVALID_ACCESS_ERR` exception. (Setting could fail if, e.g., the user has disabled storage for the domain, or if the quota has been exceeded.)

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. If no item with that key exists, the method must do nothing.

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.

The **`clear()`** method must atomically cause the list associated with the object to be emptied of all key/value pairs.

When the `setItem()`, `removeItem()`, and `clear()` methods are invoked, events are fired on other `HTMLDocument` objects that can access the newly stored or removed data, as defined in the sections on the `sessionStorage` and `localStorage` attributes.

5.10.1.3. The `sessionStorage` attribute

The **`sessionStorage`** attribute represents the set of storage areas specific to the current top-level browsing context.

Each top-level browsing context has a unique set of session storage areas, one for each origin.

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 origin. If it has not, a new storage area for that document's origin must be created.

The `Storage` object for the document's associated `Window` object's `sessionStorage` attribute must then be associated with that origin's session storage area for that top-level browsing context.

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 the session storage area of the origin of that `HTMLDocument` must be copied into the new browsing context when it is created. From that point on, however, the two session storage areas must be considered separate, not affecting each other in any way.

When the `setItem()`, `removeItem()`, and `clear()` methods are called on a `Storage` object *x* that is associated with a session storage area, then in every `HTMLDocument` object whose `Window` 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.

5.10.1.4. The `localStorage` attribute

The **`localStorage`** object provides a `Storage` object for an origin.

User agents must have a set of local storage areas, one for each origin.

User agents should expire data from the local storage areas only 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 local storage areas should be considered potentially user-critical. It is expected that Web applications will use the local storage areas for storing user-written documents.

When the `localStorage` attribute is accessed, the user agent must check to see if it has allocated local storage area for the origin of the browsing context within which the script is running. If it has not, a new storage area for that origin must be created.

The user agent must then create a `Storage` object associated with that origin's local storage area, and return it.

When the `setItem()`, `removeItem()`, and `clear()` methods are called on a `Storage` object *x* that is associated with a local storage area, then in every `HTMLDocument` object whose `Window` object's `localStorage` attribute's `Storage` object is associated with the same storage area, other than *x*, a `storage` event must be fired, as described below.

5.10.1.5. 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 local storage).

When this happens, the user agent must dispatch an event with the name `storage`, with no namespace, which does not bubble but is cancelable, and which uses the `StorageEvent`, at the body element of each active `HTMLDocument` object affected.

If the event is being fired due to an invocation of the `setItem()` or `removeItem()` methods, the event must have its `key` attribute set to the name of the key in question, its `oldValue` attribute set to the old value of the key in question, or null if the key is newly added, and its `newValue` attribute set to the new value of the key in question, or null if the key was removed.

Otherwise, if the event is being fired due to an invocation of the `clear()` method, the event must have its `key`, `oldValue`, and `newValue` attributes set to null.

In addition, the event must have its `url` attribute set to the address of the page whose `Storage` object was affected, and its `source` attribute set to the `Window` object of the browsing context that that document is in, if the two documents are in the same unit of related browsing contexts, or null otherwise.

5.10.1.5.1. Event definition

```
interface StorageEvent : Event {
  readonly attribute DOMString key;
  readonly attribute DOMString oldValue;
  readonly attribute DOMString newValue;
  readonly attribute DOMString url;
  readonly attribute Window source;
  void initStorageEvent(in DOMString typeArg, in boolean canBubbleArg, in
boolean cancelableArg, in DOMString keyArg, in DOMString oldValueArg, in
DOMString newValueArg, in DOMString urlArg, in Window sourceArg);
  void initStorageEventNS(in DOMString namespaceURI, in DOMString typeArg,
in boolean canBubbleArg, in boolean cancelableArg, in DOMString keyArg, in
DOMString oldValueArg, in DOMString newValueArg, in DOMString urlArg, in
Window sourceArg);
};
```

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 **`key`** attribute represents the key being changed.

The **`oldValue`** attribute represents the old value of the key being changed.

The **`newValue`** attribute represents the new value of the key being changed.

The **`url`** attribute represents the address of the document that changed the key.

The **`source`** attribute represents the `Window` that changed the key.

5.10.1.6. Threads

Multiple browsing contexts must be able to access the local storage areas simultaneously in a predictable manner. Scripts must not be able to detect any concurrent script execution.

This is required to guarantee 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 local storage area, the UA blocks scripts in other browsing contexts when they try to access the local storage area for the same origin 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 origin 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.

5.10.2 Database storage

5.10.2.1. Introduction

This section is non-normative.

...

5.10.2.2. Databases

Each *origin* has an associated set of databases. Each database has a name and a current version. There is no way to enumerate or delete the databases available for a domain from this API.

> ***Note: Each database has one version at a time, a database can't exist in multiple versions at once. Versions are intended to allow authors to manage schema changes incrementally and non-destructively, and without running the risk of old code (e.g. in another browser window) trying to write to a database with incorrect assumptions.***

The **`openDatabase()`** method returns a `Database` object. The method takes four arguments: a database name, a database version, a display name, and an estimated size, in bytes, of the data that will be stored in the database.

The `openDatabase()` method must use and create databases from the origin of the active document of the browsing context of the `Window` object on which the method was invoked.

If the database version provided is not the empty string, and the database already exists but has a different version, then the method must raise an `INVALID_STATE_ERR` exception.

The user agent may also raise a security exception in case the request violates a policy decision (e.g. if the user agent is configured to not allow the page to open databases).

Otherwise, if the database version provided is the empty string, or if the database doesn't yet exist, or if the database exists and the version provided to the `openDatabase()` method is the same as the current version

associated with the database, then the method must return a `Database` object representing the database that has the name that was given. If no such database exists, it must be created first.

All strings including the empty string are valid database names. Database names are case-sensitive.

> ***Note: Implementations can support this even in environments that only support a subset of all strings as database names by mapping database names (e.g. using a hashing algorithm) to the supported set of names.***

User agents are expected to use the display name and the estimated database size to optimize the user experience. For example, a user agent could use the estimated size to suggest an initial quota to the user. This allows a site that is aware that it will try to use hundreds of megabytes to declare this upfront, instead of the user agent prompting the user for permission to increase the quota every five megabytes.

```
interface Database {
  void transaction(in SQLTransactionCallback callback);
  void transaction(in SQLTransactionCallback callback, in
SQLTransactionErrorCallback errorCallback);
  void transaction(in SQLTransactionCallback callback, in
SQLTransactionErrorCallback errorCallback, in VoidCallback successCallback);

  readonly attribute DOMString version;
  void changeVersion(in DOMString oldVersion, in DOMString newVersion, in
SQLTransactionCallback callback, in SQLTransactionErrorCallback
errorCallback, in VoidCallback successCallback);
};

interface SQLTransactionCallback {
  void handleEvent(in SQLTransaction transaction);
};

interface SQLTransactionErrorCallback {
  void handleEvent(in SQLError error);
};
```

The **`transaction()`** method takes one or two arguments. When called, the method must immediately return and then asynchronously run the transaction steps with the *transaction callback* being the first argument, the *error callback* being the second argument, if any, the *success callback* being the third argument, if any, and with no *preflight operation* or *postflight operation*.

The version that the database was opened with is the **expected version** of this `Database` object. It can be the empty string, in which case there is no expected version — any version is fine.

On getting, the **`version`** attribute must return the current version of the database (as opposed to the expected version of the `Database` object).

The **`changeVersion()`** method allows scripts to atomically verify the version number and change it at the same time as doing a schema update. When the method is invoked, it must immediately return, and then asynchronously run the transaction steps with the *transaction callback* being the third argument, the *error callback* being the fourth argument, the *success callback* being the fifth argument, the *preflight operation* being

the following:

1. Check that the value of the first argument to the `changeVersion()` method exactly matches the database's actual version. If it does not, then the *preflight operation* fails.

...and the *postflight operation* being the following:

1. Change the database's actual version to the value of the second argument to the `changeVersion()` method.
2. Change the `Database` object's expected version to the value of the second argument to the `changeVersion()` method.

5.10.2.3. Executing SQL statements

The `transaction()` and `changeVersion()` methods invoke callbacks with `SQLTransaction` objects.

```
typedef sequence<Object> ObjectArray;

interface SQLTransaction {
  void executeSql(in DOMString sqlStatement);
  void executeSql(in DOMString sqlStatement, in ObjectArray arguments);
  void executeSql(in DOMString sqlStatement, in ObjectArray arguments, in
SQLStatementCallback callback);
  void executeSql(in DOMString sqlStatement, in ObjectArray arguments, in
SQLStatementCallback callback, in SQLStatementErrorCallback errorCallback);
};

interface SQLStatementCallback {
  void handleEvent(in SQLTransaction transaction, in SQLResultSet
resultSet);
};

interface SQLStatementErrorCallback {
  boolean handleEvent(in SQLTransaction transaction, in SQLError error);
};
```

When the **`executeSql(`*`sqlStatement`*`, `*`arguments`*`, `*`callback`*`, `*`errorCallback`*`)`** method is invoked, the user agent must run the following algorithm. (This algorithm is relatively simple and doesn't actually execute any SQL — the bulk of the work is actually done as part of the transaction steps.)

1. If the method was not invoked during the execution of a `SQLTransactionCallback`, `SQLStatementCallback`, or `SQLStatementErrorCallback` then raise an `INVALID_STATE_ERR` exception. (Calls from inside a `SQLTransactionErrorCallback` thus raise an exception. The `SQLTransactionErrorCallback` handler is only called once a transaction has failed, and no SQL statements can be added to a failed transaction.)
2. Parse the first argument to the method (*sqlStatement*) as an SQL statement, with the exception that `?` characters can be used in place of literals in the statement. [SQL]

3. Replace each `?` placeholder with the value of the argument in the *arguments* array with the same position. (So the first `?` placeholder gets replaced by the first value in the *arguments* array, and generally the *n*th `?` placeholder gets replaced by the *n*th value in the *arguments* array.)

   If the second argument is omitted or null, then treat the *arguments* array as empty.

   The result is *the statement*.

   Implementation feedback is requested on what to do with arguments that are of types that are not supported by the underlying SQL backend. For example, SQLite doesn't support booleans, so what should the UA do if passed a boolean? The Gears team suggests failing, not silently converting types.

4. If the syntax of *sqlStatement* is not valid (except for the use of `?` characters in the place of literals), or the statement uses features that are not supported (e.g. due to security reasons), or the number of items in the *arguments* array is not equal to the number of `?` placeholders in the statement, or the statement cannot be parsed for some other reason, then mark *the statement* as bogus.

5. If the `Database` object that the `SQLTransaction` object was created from has an expected version that is neither the empty string nor the actual version of the database, then mark *the statement* as bogus. (Error code 2.)

6. Queue up *the statement* in the transaction, along with the third argument (if any) as the statement's result set callback and the fourth argument (if any) as the error callback.

The user agent must act as if the database was hosted in an otherwise completely empty environment with no resources. For example, attempts to read from or write to the file system will fail.

SQL inherently supports multiple concurrent connections. Authors should make appropriate use of the transaction features to handle the case of multiple scripts interacting with the same database simultaneously (as could happen if the same page was opened in two different browsing contexts).

User agents must consider statements that use the `BEGIN`, `COMMIT`, and `ROLLBACK` SQL features as being unsupported (and thus will mark them as bogus), so as to not let these statements interfere with the explicit transactions managed by the database API itself.

Note: A future version of this specification will probably define the exact SQL subset required in more detail.

5.10.2.4. Database query results

The `executeSql()` method invokes its callback with a `SQLResultSet` object as an argument.

```
interface SQLResultSet {
  readonly attribute int insertId;
  readonly attribute int rowsAffected;
  readonly attribute SQLResultSetRowList rows;
};
```

The **`insertId`** attribute must return the row ID of the row that the `SQLResultSet` object's SQL statement inserted into the database, if the statement inserted a row. If the statement inserted multiple rows, the ID of the last row must be the one returned. If the statement did not insert a row, then the attribute must instead raise an `INVALID_ACCESS_ERR` exception.

The **`rowsAffected`** attribute must return the number of rows that were affected by the SQL statement. If the statement did not affected any rows, then the attribute must return zero. For "SELECT" statements, this returns zero (querying the database doesn't affect any rows).

The **`rows`** attribute must return a `SQLResultSetRowList` representing the rows returned, in the order returned by the database. If no rows were returned, then the object will be empty (its `length` will be zero).

```
interface SQLResultSetRowList {
  readonly attribute unsigned long length;
  [IndexGetter] DOMObject item(in unsigned long index);
};
```

`SQLResultSetRowList` objects have a **`length`** attribute that must return the number of rows it represents (the number of rows returned by the database).

The **`item(`*`index`*`)`** attribute must return the row with the given index *index*. If there is no such row, then the method must raise an `INDEX_SIZE_ERR` exception.

Each row must be represented by a native ordered dictionary data type. In the ECMAScript binding, this must be `Object`. Each row object must have one property (or dictionary entry) per column, with those properties enumerating in the order that these columns were returned by the database. Each property must have the name of the column and the value of the cell, as they were returned by the database.

5.10.2.5. Errors

Errors in the database API are reported using callbacks that have a `SQLError` object as one of their arguments.

```
interface SQLError {
  readonly attribute unsigned int code;
  readonly attribute DOMString message;
};
```

The **`code`** DOM attribute must return the most appropriate code from the following table:

Code	Situation
0	The transaction failed for reasons unrelated to the database itself and not covered by any other error code.
1	The statement failed for database reasons not covered by any other error code.
2	The statement failed because the expected version of the database didn't match the actual database version.
3	The statement failed because the data returned from the database was too large. The SQL "LIMIT" modifier might be useful to reduce the size of the result set.
4	The statement failed because there was not enough remaining storage space, or the storage quota was reached and the user declined to give more space to the database.
5	The statement failed because the transaction's first statement was a read-only statement, and a subsequent statement in the same transaction tried to modify the database, but the transaction failed to obtain a write lock before another transaction obtained a write lock and changed a part of the database that the former transaction was depending upon.

Code	Situation
6	An `INSERT`, `UPDATE`, or `REPLACE` statement failed due to a constraint failure. For example, because a row was being inserted and the value given for the primary key column duplicated the value of an existing row.

We should define a more thorough list of codes. Implementation feedback is requested to determine what codes are needed.

The **`message`** DOM attribute must return an error message describing the error encountered. The message should be localized to the user's language.

5.10.2.6. Processing model

The **transaction steps** are as follows. These steps must be run asynchronously. These steps are invoked with a *transaction callback*, optionally an *error callback*, optionally a *success callback*, optionally a *preflight operation*, and optionally a *postflight operation*.

1. Open a new SQL transaction to the database, and create a `SQLTransaction` object that represents that transaction.
2. If an error occurred in the opening of the transaction, jump to the last step.
3. If a *preflight operation* was defined for this instance of the transaction steps, run that. If it fails, then jump to the last step. (This is basically a hook for the `changeVersion()` method.)
4. Invoke the *transaction callback* with the aforementioned `SQLTransaction` object as its only argument.
5. If the callback couldn't be called (e.g. it was null), or if the callback was invoked and raised an exception, jump to the last step.
6. While there are any statements queued up in the transaction, perform the following steps for each queued up statement in the transaction, oldest first. Each statement has a statement, optionally a result set callback, and optionally an error callback.
    1. If the statement is marked as bogus, jump to the "in case of error" steps below.
    2. Execute the statement in the context of the transaction. [SQL]
    3. If the statement failed, jump to the "in case of error" steps below.
    4. Create a `SQLResultSet` object that represents the result of the statement.
    5. If the statement has a result set callback, invoke it with the `SQLTransaction` object as its first argument and the new `SQLResultSet` object as its second argument.
    6. If the callback was invoked and raised an exception, jump to the last step in the overall steps.
    7. Move on to the next statement, if any, or onto the next overall step otherwise.

    In case of error (or more specifically, if the above substeps say to jump to the "in case of error" steps), run the following substeps:

    1. If the statement had an associated error callback, then invoke that error callback with the

`SQLTransaction` object and a newly constructed `SQLError` object that represents the error that caused these substeps to be run as the two arguments, respectively.

2. If the error callback returns false, then move on to the next statement, if any, or onto the next overall step otherwise.

3. Otherwise, the error callback did not return false, or there was no error callback. Jump to the last step in the overall steps.

7. If a *postflight operation* was defined for this instance of the transaction steps, run that. If it fails, then jump to the last step. (This is basically a hook for the `changeVersion()` method.)

8. Commit the transaction.

9. If an error occurred in the committing of the transaction, jump to the last step.

10. Invoke the *success callback*.

11. End these steps. The next step is only used when something goes wrong.

12. Call the *error callback* with a newly constructed `SQLError` object that represents the last error to have occurred in this transaction. Rollback the transaction. Any still-pending statements in the transaction are discarded.

5.10.3 Disk space

User agents should limit the total amount of space allowed for storage areas and databases.

User agents should guard against sites storing data in the storage areas or databases of subdomains, e.g. storing up to the limit in a1.example.com, a2.example.com, a3.example.com, etc, circumventing the main example.com storage limit.

User agents may prompt the user when 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.

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.

5.10.4 Privacy

5.10.4.1. 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 local storage area or in its client-side database to track a user across multiple sessions, building a profile of the user's interests to allow for highly targeted 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.

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 `localStorage` and database objects to scripts originating at the domain of the top-level document of the browsing context, for instance denying access to the API for pages from other domains running in `iframe`s.

- Expiring stored data: user agents may automatically delete stored data after a period of time.

  For example, a user agent could treat third-party local storage areas 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).

  However, this also puts the user's data at risk.

- Treating persistent storage as cookies: user agents should present the persistent storage and database features to the user in a way that does not distinguish them from HTTP session cookies. [RFC2109] [RFC2965]

  This might encourage users to view persistent storage with healthy suspicion.

- Site-specific white-listing of access to local storage areas and databases: user agents may allow sites to access session storage areas in an unrestricted manner, but require the user to authorize access to local storage areas and databases.

- Origin-tracking of persistent storage data: user agents may record the origins of sites that contained content from third-party origins 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 these APIs for user tracking, they do not block it altogether. Within a single domain, a site can continue to track the user during a session, 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 URLs, 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.

5.10.4.2. Cookie resurrection

If the user interface for persistent storage presents data in the persistent storage features 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.

5.10.5 Security

5.10.5.1. DNS spoofing attacks

Because of the potential for DNS spoofing attacks, one cannot guarantee that a host claiming to be in a certain domain really is from that domain. To mitigate this, pages can use SSL. Pages using SSL can be sure that only pages using SSL that have certificates identifying them as being from the same domain can access their local storage areas and databases.

5.10.5.2. Cross-directory attacks

Different authors sharing one host name, for example users hosting content on `geocities.com`, all share one persistent storage object and one set of databases. There is no feature to restrict the access by pathname. Authors on shared hosts are therefore recommended to avoid using the persistent storage features, as it would be trivial for other authors to read from and write to the same storage area or database.

> ***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.***

5.10.5.3. Implementation risks

The two primary risks when implementing these persistent storage features 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 targeted 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.

Thus, strictly following the origin model described in this specification is important for user security.

5.10.5.4. SQL and user agents

User agent implementors are strongly encouraged to audit all their supported SQL statements for security implications. For example, `LOAD DATA INFILE` is likely to pose security risks and there is little reason to support it.

In general, it is recommended that user agents not support features that control how databases are stored on disk. For example, there is little reason to allow Web authors to control the character encoding used in the disk representation of the data, as all data in ECMAScript is implicitly UTF-16.

5.10.5.5. SQL injection

Authors are strongly recommended to make use of the `?` placeholder feature of the `executeSql()` method, and to never construct SQL statements on the fly.

5.11 Links

5.11.1 Hyperlink elements

The `a`, `area`, and `link` elements can, in certain situations described in the definitions of those elements, represent **hyperlinks**.

The **`href`** attribute on a hyperlink element must have a value that is a valid URL. This URL is the *destination resource* of the hyperlink.

> ***The `href` attribute on `a` and `area` elements is not required; when those elements do not have `href` attributes they do not represent hyperlinks.***
>
> ***The `href` attribute on the `link` element is required, but whether a `link` element represents a hyperlink or not depends on the value of the `rel` attribute of that element.***

The **`target`** attribute, if present, must be a valid browsing context name or keyword. User agents use this name when following hyperlinks.

The **`ping`** attribute, if present, gives the URLs of the resources that are interested in being notified if the user follows the hyperlink. The value must be a space separated list of one or more valid URLs. The value is used by the user agent when following hyperlinks.

For `a` and `area` elements that represent hyperlinks, the relationship between the document containing the hyperlink and the destination resource indicated by the hyperlink is given by the value of the element's **`rel`** attribute, which must be a set of space-separated tokens. The allowed values and their meanings are defined below. The `rel` attribute has no default value. If the attribute is omitted or if none of the values in the attribute are recognized by the 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, is `all`.

The **`hreflang`** attribute on hyperlink elements, if present, gives the language of the linked resource. It is purely advisory. The value must be a valid RFC 3066 language code. [RFC3066] User agents must not consider this attribute authoritative — upon fetching the resource, user agents must use only language information associated with the resource to determine its language, not metadata included in the link to the resource.

The **`type`** attribute, if present, gives the MIME type of the linked resource. It is purely advisory. The value must be a valid MIME type, optionally with parameters. [RFC2046] User agents must not consider the `type` attribute authoritative — upon fetching the resource, user agents must not use metadata included in the link to the resource to determine its type.

5.11.2 Following hyperlinks

When a user *follows a hyperlink*, the user agent must navigate a browsing context to the URL given by the `href` attribute of that hyperlink. In the case of server-side image maps, the URL of the hyperlink must further have its *hyperlink suffix* appended to it.

If the user indicated a specific browsing context when following the hyperlink, or if the user agent is configured to follow hyperlinks by navigating a particular browsing context, then that must be the browsing context that is navigated.

Otherwise, if the hyperlink element is an `a` or `area` element that has a `target` attribute, then the browsing context that is navigated must be chosen by applying the rules for choosing a browsing context given a browsing context name, using the value of the `target` attribute as the browsing context name. If these rules result in the creation of a new browsing context, it must be navigated with replacement enabled.

Otherwise, if the hyperlink element is a sidebar hyperlink and the user agent implements a feature that can be considered a secondary browsing context, such a secondary browsing context may be selected as the browsing context to be navigated.

Otherwise, if the hyperlink element is an `a` or `area` element with no `target` attribute, but one of the child nodes of the `head` element is a `base` element with a `target` attribute, then the browsing context that is navigated must be chosen by applying the rules for choosing a browsing context given a browsing context name, using the value of the `target` attribute of the first such `base` element as the browsing context name. If these rules result in the creation of a new browsing context, it must be navigated with replacement enabled.

Otherwise, the browsing context that must be navigated is the same browsing context as the one which the hyperlink element itself is in.

The navigation must be done with the browsing context that contains the `Document` object with which the hyperlink's element in question is associated as the source browsing context.

5.11.2.1. Hyperlink auditing

If an `a` or `area` hyperlink element has a `ping` attribute and the user follows the hyperlink, the user agent must take the `ping` attribute's value, split that string on spaces, resolve each resulting token, and then should send a request (as described below) to each of the resulting absolute URLs. (Tokens that fail to resolve are ignored.) This may be done in parallel with the primary request, and is independent of the result of that request.

User agents should allow the user to adjust this behavior, for example in conjunction with a setting that disables the sending of HTTP `Referer` headers. Based on the user's preferences, UAs may either ignore the `ping` attribute altogether, or selectively ignore URLs in the list (e.g. ignoring any third-party URLs).

For URLs that are HTTP URLs, the requests must be performed using the POST method (with an empty entity body in the request). All relevant cookie and HTTP authentication headers must be included in the request. Which other headers are required depends on the URIs involved.

↪ **If both the address of the `Document` object containing the hyperlink being audited and the ping URL have the same origin**

The request must include a `Ping-From` HTTP header with, as its value, the address of the document containing the hyperlink, and a `Ping-To` HTTP header with, as its value, the address of

the absolute URL of the target of the hyperlink. The request must not include a `Referer` HTTP header.

↪ **Otherwise, if the origins are different, but the document containing the hyperlink being audited was not retrieved over an encrypted connection**

The request must include a `Referer` HTTP header [sic] with, as its value, the location of the document containing the hyperlink, a `Ping-From` HTTP header with the same value, and a `Ping-To` HTTP header with, as its value, the address of the target of the hyperlink.

↪ **Otherwise, the origins are different and the document containing the hyperlink being audited was retrieved over an encrypted connection**

The request must include a `Ping-To` HTTP header with, as its value, the address of the target of the hyperlink. The request must neither include a `Referer` HTTP header nor include a `Ping-From` HTTP header.

Note: To save bandwidth, implementors might also wish to consider omitting optional headers such as `Accept` from these requests.

User agents must ignore any entity bodies returned in the responses, but must, unless otherwise specified by the user, honor the HTTP headers (including, in particular, redirects and HTTP cookie headers). [RFC2109] [RFC2965]

When the `ping` attribute is present, user agents should clearly indicate to the user that following the hyperlink will also cause secondary requests to be sent in the background, possibly including listing the actual target URLs.

The `ping` attribute is redundant with pre-existing technologies like HTTP redirects and JavaScript in allowing Web pages to track which off-site links are most popular or allowing advertisers to track click-through rates.

However, the `ping` attribute provides these advantages to the user over those alternatives:

- ***It allows the user to see the final target URL unobscured.***
- ***It allows the UA to inform the user about the out-of-band notifications.***
- ***It allows the paranoid user to disable the notifications without losing the underlying link functionality.***
- ***It allows the UA to optimize the use of available network bandwidth so that the target page loads faster.***

Thus, while it is possible to track users without this feature, authors are encouraged to use the `ping` attribute so that the user agent can improve the user experience.

5.11.3 Link types

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

In this section, the term *referenced document* refers to the resource identified by the element representing the link, and the term *current document* refers to the resource within which the element representing the link finds itself.

To determine which link types apply to a `link`, `a`, or `area` element, the element's `rel` attribute must be split on spaces. The resulting tokens are the link types that apply to that element.

Unless otherwise specified, a keyword must not be specified more than once per `rel` attribute.

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.
`external`	*not allowed*	Hyperlink	Indicates that the referenced document is not part of the same site as the current document.
`feed`	Hyperlink	Hyperlink	Gives the address of a syndication feed for the current document.
`first`	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the first document in the series is the referenced document.
`help`	Hyperlink	Hyperlink	Provides a link to context-sensitive help.
`icon`	External Resource	*not allowed*	Imports an icon to represent the current document.
`index`	Hyperlink	Hyperlink	Gives a link to the document that provides a table of contents or index listing the current document.
`last`	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the last document in the series is the referenced document.
`license`	Hyperlink	Hyperlink	Indicates that the current document is covered by the copyright license described by the referenced document.
`next`	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the next document in the series is the referenced document.
`nofollow`	*not allowed*	Hyperlink	Indicates that the current document's original author or publisher does not endorse the referenced document.
`noreferrer`	*not allowed*	Hyperlink	Requires that the user agent not send an HTTP `Referer` header if the user follows the hyperlink.
`pingback`	External Resource	*not allowed*	Gives the address of the pingback server that handles pingbacks to the current document.
`prefetch`	External Resource	*not allowed*	Specifies that the target resource should be preemptively cached.
`prev`	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the previous document in the series is the referenced document.
`search`	Hyperlink	Hyperlink	Gives a link to a resource that can be used to search through the current document and its related pages.
`stylesheet`	External Resource	*not allowed*	Imports a stylesheet.
`sidebar`	Hyperlink	Hyperlink	Specifies that the referenced document, if retrieved, is intended to be shown in the browser's sidebar (if it has one).
`tag`	Hyperlink	Hyperlink	Gives a tag (identified by the given address) that applies to the current document.
`up`	Hyperlink	Hyperlink	Provides a link to a document giving the context for the current document.

Some of the types described below list synonyms for these values. These are to be handled as specified by user agents, but must not be used in documents.

5.11.3.1. Link type "alternate"

The alternate keyword may be used with link, a, and area elements. For link elements, if the rel attribute does not also contain the keyword stylesheet, it creates a hyperlink; but if it *does* also contain the keyword stylesheet, the alternate keyword instead modifies the meaning of the stylesheet keyword in the way described for that keyword, and the rest of this subsection doesn't apply.

The alternate keyword indicates that the referenced document is an alternate representation of the current document.

The nature of the referenced document is given by the media, hreflang, and type attributes.

If the alternate keyword is used with the media attribute, it indicates that the referenced document is intended for use with the media specified.

If the alternate keyword is used with the hreflang attribute, and that attribute's value differs from the root element's language, it indicates that the referenced document is a translation.

If the alternate keyword is used with the type attribute, it indicates that the referenced document is a reformulation of the current document in the specified format.

The media, hreflang, and type attributes can be combined when specified with the alternate keyword.

> For example, the following link is a French translation that uses the PDF format:
>
> ```
> <link rel=alternate type=application/pdf hreflang=fr href=manual-fr>
> ```

If the alternate keyword is used with the type attribute set to the value application/rss+xml or the value application/atom+xml, then the user agent must treat the link as it would if it had the feed keyword specified as well.

The alternate link relationship is transitive — that is, if a document links to two other documents with the link type "alternate", then, in addition to implying that those documents are alternative representations of the first document, it is also implying that those two documents are alternative representations of each other.

5.11.3.2. Link type "archives"

The archives keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The archives keyword indicates that the referenced document describes a collection of records, documents, or other materials of historical interest.

> A blog's index page could link to an index of the blog's past posts with rel="archives".

Synonyms: For historical reasons, user agents must also treat the keyword "archive" like the archives keyword.

5.11.3.3. Link type "author"

The author keyword may be used with link, a, and area elements. For link elements, it creates a

hyperlink.

For `a` and `area` elements, the `author` keyword indicates that the referenced document provides further information about the author of the section that the element defining the hyperlink applies to.

For `link` elements, the `author` keyword indicates that the referenced document provides further information about the author for the page as a whole.

> ***Note: The "referenced document" can be, and often is, a `mailto:` URL giving the e-mail address of the author. [MAILTO]***

Synonyms: For historical reasons, user agents must also treat `link`, `a`, and `area` elements that have a `rev` attribute with the value "`made`" as having the `author` keyword specified as a link relationship.

5.11.3.4. Link type "`bookmark`"

The `bookmark` keyword may be used with `a` and `area` elements.

The `bookmark` keyword gives a permalink for the nearest ancestor `article` element of the linking element in question, or of the section the linking element is most closely associated with, if there are no ancestor `article` elements.

> The following snippet has three permalinks. A user agent could determine which permalink applies to which part of the spec by looking at where the permalinks are given.
>
> ```
> ...
> <body>
>  <h1>Example of permalinks</h1>
>  
>   <h2>First example</h2>
>   <a href="a.html" rel="bookmark">This</a> permalink applies to
>   only the content from the first H2 to the second H2. The DIV isn't
>   exactly that section, but it roughly corresponds to it.
>  
>  <h2>Second example</h2>
>  <article id="b">
>   <a href="b.html" rel="bookmark">This</a> permalink applies to
>   the outer ARTICLE element (which could be, e.g., a blog post).
>   <article id="c">
>    <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>
> ...
> ```

5.11.3.5. Link type "`external`"

The `external` keyword may be used with `a` and `area` elements.

The `external` keyword indicates that the link is leading to a document that is not part of the site that the current document forms a part of.

5.11.3.6. Link type "`feed`"

The `feed` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `feed` keyword indicates that the referenced document is a syndication feed. If the `alternate` link type is also specified, then the feed is specifically the feed for the current document; otherwise, the feed is just a syndication feed, not necessarily associated with a particular Web page.

The first `link`, `a`, or `area` element in the document (in tree order) that creates a hyperlink with the link type `feed` must be treated as the default syndication feed for the purposes of feed autodiscovery.

Note: The `feed` keyword is implied by the `alternate` link type in certain cases (q.v.).

> The following two `link` elements are equivalent: both give the syndication feed for the current page:
>
> ```
> <link rel="alternate" type="application/atom+xml" href="data.xml">
>
> <link rel="feed alternate" href="data.xml">
> ```
>
> The following extract offers various different syndication feeds:
>
> ```
>  You can access the planets database using Atom feeds:
>  <ul>
>   <li><a href="recently-visited-planets.xml" rel="feed">Recently Visited
> Planets</a></li>
>   <li><a href="known-bad-planets.xml" rel="feed">Known Bad Planets</a>
> </li>
>   <li><a href="unexplored-planets.xml" rel="feed">Unexplored Planets</a>
> </li>
>  </ul>
> ```

5.11.3.7. Link type "`help`"

The `help` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

For `a` and `area` elements, the `help` keyword indicates that the referenced document provides further help information for the parent of the element defining the hyperlink, and its children.

> In the following example, the form control has associated context-sensitive help. The user agent could use this information, for example, displaying the referenced document if the user presses the "Help" or "F1" key.
>
> ```
>  <label> Topic: <input name=topic> <a href="help/topic.html"
> rel="help">(Help)</a></label>
> ```

For `link` elements, the `help` keyword indicates that the referenced document provides help for the page as a whole.

5.11.3.8. Link type "icon"

The `icon` keyword may be used with `link` elements, for which it creates an external resource link.

The specified resource is an icon representing the page or site, and should be used by the user agent when representing the page in the user interface.

Icons could be auditory icons, visual icons, or other kinds of icons. If multiple icons are provided, the user agent must select the most appropriate icon according to the `type`, `media`, and `sizes` attributes. If there are multiple equally appropriate icons, user agents must use the last one declared in tree order. If the user agent tries to use an icon but that icon is determined, upon closer examination, to in fact be inappropriate (e.g. because it uses an unsupported format), then the user agent must try the next-most-appropriate icon as determined by the attributes.

There is no default type for resources given by the `icon` keyword. However, for the purposes of determining the type of the resource, user agents must expect the resource to be an image.

The **`sizes`** attribute gives the sizes of icons for visual media.

If specified, the attribute must have a value that is an unordered set of unique space-separated tokens. The values must all be either `any` or a value that consists of two valid non-negative integers that do not have a leading U+0030 DIGIT ZERO (0) character and that are separated by a single U+0078 LATIN SMALL LETTER X character.

The keywords represent icon sizes.

To parse and process the attribute's value, the user agent must first split the attribute's value on spaces, and must then parse each resulting keyword to determine what it represents.

The **`any`** keyword represents that the resource contains a scalable icon, e.g. as provided by an SVG image.

Other keywords must be further parsed as follows to determine what they represent:

- If the keyword doesn't contain exactly one U+0078 LATIN SMALL LETTER X character, then this keyword doesn't represent anything. Abort these steps for that keyword.
- Let *width string* be the string before the "`x`".
- Let *height string* be the string after the "`x`".
- If either *width string* or *height string* start with a U+0030 DIGIT ZERO (0) character or contain any characters other than characters in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9), then this keyword doesn't represent anything. Abort these steps for that keyword.
- Apply the rules for parsing non-negative integers to *width string* to obtain *width*.
- Apply the rules for parsing non-negative integers to *height string* to obtain *height*.
- The keyword represents that the resource contains a bitmap icon with a width of *width* device pixels and a height of *height* device pixels.

The keywords specified on the `sizes` attribute must not represent icon sizes that are not actually available in the linked resource.

If the attribute is not specified, then the user agent must assume that the given icon is appropriate, but less appropriate than an icon of a known and appropriate size.

> The following snippet shows the top part of an application with several icons.
>
> ```
> <!DOCTYPE HTML>
> <html>
>  <head>
>   <title>lsForums — Inbox</title>
>   <link rel=icon href=favicon.png sizes="16x16">
>   <link rel=icon href=windows.ico sizes="32x32 48x48">
>   <link rel=icon href=mac.icns sizes="128x128 512x512 8192x8192
> 32768x32768">
>   <link rel=icon href=iphone.png sizes="59x60">
>   <link rel=icon href=gnome.svg sizes="any">
>   <link rel=stylesheet href=lsforums.css>
>   <script src=lsforums.js></script>
>   <meta name=application-name content="lsForums">
>  </head>
>  <body>
>   ...
> ```

5.11.3.9. Link type "`license`"

The `license` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `license` keyword indicates that the referenced document provides the copyright license terms under which the current document is provided.

Synonyms: For historical reasons, user agents must also treat the keyword "`copyright`" like the `license` keyword.

5.11.3.10. Link type "`nofollow`"

The `nofollow` keyword may be used with `a` and `area` elements.

The `nofollow` keyword indicates that the link is not endorsed by the original author or publisher of the page, or that the link to the referenced document was included primarily because of a commercial relationship between people affiliated with the two pages.

5.11.3.11. Link type "`noreferrer`"

The `noreferrer` keyword may be used with `a` and `area` elements.

If a user agent follows a link defined by an `a` or `area` element that has the `noreferrer` keyword, the user agent must not include a `Referer` HTTP header (or equivalent for other protocols) in the request.

5.11.3.12. Link type "`pingback`"

The `pingback` keyword may be used with `link` elements, for which it creates an external resource link.

For the semantics of the `pingback` keyword, see the Pingback 1.0 specification. [PINGBACK]

5.11.3.13. Link type "`prefetch`"

The `prefetch` keyword may be used with `link` elements, for which it creates an external resource link.

The `prefetch` keyword indicates that preemptively fetching and caching the specified resource is likely to be beneficial, as it is highly likely that the user will require this resource.

There is no default type for resources given by the `prefetch` keyword.

5.11.3.14. Link type "`search`"

The `search` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `search` keyword indicates that the referenced document provides an interface specifically for searching the document and its related resources.

> ***Note: OpenSearch description documents can be used with `link` elements and the `search` link type to enable user agents to autodiscover search interfaces. [OPENSEARCH]***

5.11.3.15. Link type "`stylesheet`"

The `stylesheet` keyword may be used with `link` elements, for which it creates an external resource link that contributes to the styling processing model.

The specified resource is a resource that describes how to present the document. Exactly how the resource is to be processed depends on the actual type of the resource.

If the `alternate` keyword is also specified on the `link` element, then the link is an alternative stylesheet.

The default type for resources given by the `stylesheet` keyword is `text/css`.

Quirk: If the document has been set to quirks mode and the Content-Type metadata of the external resource is not a supported style sheet type, the user agent must instead assume it to be `text/css`.

5.11.3.16. Link type "`sidebar`"

The `sidebar` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `sidebar` keyword indicates that the referenced document, if retrieved, is intended to be shown in a secondary browsing context (if possible), instead of in the current browsing context.

A hyperlink element with with the `sidebar` keyword specified is a **sidebar hyperlink**.

5.11.3.17. Link type "tag"

The tag keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The tag keyword indicates that the *tag* that the referenced document represents applies to the current document.

5.11.3.18. Hierarchical link types

Some documents form part of a hierarchical structure of documents.

A hierarchical structure of documents is one where each document can have various subdocuments. The document of which a document is a subdocument is said to be the document's *parent*. A document with no parent forms the top of the hierarchy.

A document may be part of multiple hierarchies.

5.11.3.18.1. Link type "index"

The index keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The index keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the top of the hierarchy. It conveys more information when used with the up keyword (q.v.).

Synonyms: For historical reasons, user agents must also treat the keywords "top", "contents", and "toc" like the index keyword.

5.11.3.18.2. Link type "up"

The up keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The up keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the parent of the current document.

The up keyword may be repeated within a rel attribute to indicate the hierarchical distance from the current document to the referenced document. Each occurrence of the keyword represents one further level. If the index keyword is also present, then the number of up keywords is the depth of the current page relative to the top of the hierarchy. Only one link is created for the set of one or more up keywords and, if present, the index keyword.

If the page is part of multiple hierarchies, then they should be described in different paragraphs. User agents must scope any interpretation of the up and index keywords together indicating the depth of the hierarchy to the paragraph in which the link finds itself, if any, or to the document otherwise.

When two links have both the up and index keywords specified together in the same scope and contradict each other by having a different number of up keywords, the link with the greater number of up keywords must be taken as giving the depth of the document.

> This can be used to mark up a navigation style sometimes known as bread crumbs. In the following example, the current page can be reached via two paths.
>
> ```
> <nav>
>  
>   <a href="/" rel="index up up up">Main</a> >
>   <a href="/products/" rel="up up">Products</a> >
>   <a href="/products/dishwashers/" rel="up">Dishwashers</a> >
>   <a>Second hand</a>
>  
>  
>   <a href="/" rel="index up up">Main</a> >
>   <a href="/second-hand/" rel="up">Second hand</a> >
>   <a>Dishwashers</a>
>  
> </nav>
> ```

Note: The `relList` DOM attribute (e.g. on the `a` element) does not currently represent multiple `up` keywords (the interface hides duplicates).

5.11.3.19. Sequential link types

Some documents form part of a sequence of documents.

A sequence of documents is one where each document can have a *previous sibling* and a *next sibling*. A document with no previous sibling is the start of its sequence, a document with no next sibling is the end of its sequence.

A document may be part of multiple sequences.

5.11.3.19.1. Link type "`first`"

The `first` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `first` keyword indicates that the document is part of a sequence, and that the link is leading to the document that is the first logical document in the sequence.

Synonyms: For historical reasons, user agents must also treat the keywords "`begin`" and "`start`" like the `first` keyword.

5.11.3.19.2. Link type "`last`"

The `last` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `last` keyword indicates that the document is part of a sequence, and that the link is leading to the document that is the last logical document in the sequence.

Synonyms: For historical reasons, user agents must also treat the keyword "`end`" like the `last` keyword.

5.11.3.19.3. Link type "next"

The `next` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `next` keyword indicates that the document is part of a sequence, and that the link is leading to the document that is the next logical document in the sequence.

5.11.3.19.4. Link type "prev"

The `prev` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `prev` keyword indicates that the document is part of a sequence, and that the link is leading to the document that is the previous logical document in the sequence.

Synonyms: For historical reasons, user agents must also treat the keyword "`previous`" like the `prev` keyword.

5.11.3.20. Other link types

Other than the types defined above, only types defined as extensions in the WHATWG Wiki RelExtensions page may be used with the `rel` attribute on `link`, `a`, and `area` elements. [WHATWGWIKI]

Anyone is free to edit the WHATWG Wiki RelExtensions page at any time to add a type. Extension types must be specified with the following information:

Keyword

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

Effect on... `link`

One of the following:

not allowed

The keyword is not allowed to be specified on `link` elements.

Hyperlink

The keyword may be specified on a `link` element; it creates a hyperlink link.

External Resource

The keyword may be specified on a `link` element; it creates a external resource link.

Effect on... `a` and `area`

One of the following:

not allowed

The keyword is not allowed to be specified on `a` and `area` elements.

Hyperlink

The keyword may be specified on `a` and `area` elements.

Brief description

A short description of what the keyword's meaning is.

Link to more details

A link to a more detailed description of the keyword's semantics and requirements. It could be another page on the Wiki, or a link to an external page.

Synonyms

A list of other keyword values that have exactly the same processing requirements. Authors must not use the values defined to be synonyms, they are only intended to allow user agents to support legacy content.

Status

One of the following:

Proposal

The keyword has not received wide peer review and approval. It is included for completeness because pages use the keyword. Pages should not use the keyword.

Accepted

The keyword has received wide peer review and approval. It has a specification that unambiguously defines how to handle pages that use the keyword, including when they use them in incorrect ways. Pages may use the keyword.

Rejected

The keyword has received wide peer review and it has been found to have significant problems. Pages must not use the keyword. When a keyword has this status, the "Effect on... `link`" and "Effect on... `a` and `area`" information should be set to "not allowed".

If a keyword is added with the "proposal" status and found to be redundant with existing values, it should be removed and listed as a synonym for the existing value. If a keyword is added with the "proposal" status and found to be harmful, then it should be changed to "rejected" status, and its "Effect on..." information should be changed accordingly.

Conformance checkers must use the information given on the WHATWG Wiki RelExtensions page to establish if a value not explicitly defined in this specification is allowed or not. When an author uses a new type not defined by either this specification or the Wiki page, conformance checkers should offer to add the value to the Wiki, with the details described above, with the "proposal" status.

This specification does not define how new values will get approved. It is expected that the Wiki will have a community that addresses this.

6. User Interaction

This section describes various features that allow authors to enable users to edit documents and parts of documents interactively.

6.1 Introduction

This section is non-normative.

Would be nice to explain how these features work together.

6.2 The `irrelevant` attribute

All elements may have the `irrelevant` content attribute set. The `irrelevant` attribute is a boolean attribute. When specified on an element, it indicates that the element is not yet, or is no longer, relevant. User agents should not render elements that have the `irrelevant` attribute specified.

> In the following skeletal example, the attribute is used to hide the Web game's main screen until the user logs in:
>
> ```
> <h1>The Example Game</h1>
> <section id="login">
>  <h2>Login</h2>
>  <form>
>   ...
>   <!-- calls login() once the user's credentials have been checked -->
>  </form>
>  <script>
>   function login() {
>     // switch screens
>     document.getElementById('login').irrelevant = true;
>     document.getElementById('game').irrelevant = false;
>   }
>  </script>
> </section>
> <section id="game" irrelevant>
>  ...
> </section>
> ```

The `irrelevant` attribute must not be used to hide content that could legitimately be shown in another presentation. For example, it is incorrect to use `irrelevant` to hide panels in a tabbed dialog, because the tabbed interface is merely a kind of overflow presentation — showing all the form controls in one big page with a scrollbar would be equivalent, and no less correct.

Elements in a section hidden by the `irrelevant` attribute are still active, e.g. scripts and form controls in such sections still render execute and submit respectively. Only their presentation to the user changes.

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

6.3 Activation

The **click()** method must fire a `click` event at the element, whose default action is the firing of a further `DOMActivate` event at the same element, whose own default action is to go through all the elements the `DOMActivate` event bubbled through (starting at the target node and going towards the `Document` node), looking for an element with an activation behavior; the first element, in reverse tree order, to have one, must have its activation behavior executed.

6.4 Scrolling elements into view

The **`scrollIntoView([`*`top`*`])`** method, when called, must cause the element on which the method was called to have the attention of the user called to it.

> ***Note: In a speech browser, this could happen by having the current playback position move to the start of the given element.***

In visual user agents, if the argument is present and has the value false, the user agent should scroll the element into view such that both the bottom and the top of the element are in the viewport, with the bottom of the element aligned with the bottom of the viewport. If it isn't possible to show the entire element in that way, or if the argument is omitted or is true, then the user agent should instead align the top of the element with the top of the viewport. Visual user agents should further scroll horizontally as necessary to bring the element to the attention of the user.

Non-visual user agents may ignore the argument, or may treat it in some media-specific manner most useful to the user.

6.5 Focus

When an element is *focused*, key events received by the document must be targeted at that element. There may be no element focused; when no element is focused, key events received by the document must be targetted at the `body` element.

User agents may track focus for each browsing context or `Document` individually, or may support only one focused elment per top-level browsing context — user agents should follow platform conventions in this regard.

Which element(s) within a top-level browsing context currently has focus must be independent of whether or not the top-level browsing context itself has the *system focus*.

6.5.1 Focus management

The **focusing steps** are as follows:

1. If focusing the element will remove the focus from another element, then run the unfocusing steps for

that element.

2. Make the element the currently focused element in its top-level browsing context.

   Some elements, most notably `area`, can correspond to more than one distinct focusable area. If a particular area was indicated when the element was focused, then that is the area that must get focus; otherwise, e.g. when using the `focus()` method, the first such region in tree order is the one that must be focused.

3. Fire a simple event that doesn't bubble called `focus` at the element.

User agents must run the focusing steps for an element whenever the user moves the focus to a focusable element.

The **unfocusing steps** are as follows:

1. Unfocus the element.

2. Fire a simple event that doesn't bubble called `blur` at the element.

User agents should run the unfocusing steps for an element whenever the user moves the focus away from any focusable element.

The **`focus()`** method, when invoked, must run the following algorithm:

1. If the element is marked as *locked for focus*, then abort these steps.

2. If the element is not focusable, then abort these steps.

3. Mark the element as **locked for focus**.

4. If the element is not already focused, run the focusing steps for the element.

5. Unmark the element as *locked for focus*.

The **`blur()`** method, when invoked, should run the unfocusing steps for the element. User agents may selectively or uniformly ignore calls to this method for usability reasons.

The **`activeElement`** attribute must return the element in the document that is focused. If no element in the `Document` is focused, this must return the `body` element.

The **`hasFocus()`** method 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.

6.5.2 Sequential focus navigation

The **`tabindex`** content attribute specifies whether the element is focusable, whether it can be reached using sequential focus navigation, and the relative order of the element for the purposes of sequential focus navigation. The name "tab index" comes from the common use of the "tab" key to navigate through the focusable elements. The term "tabbing" refers to moving forward through the focusable elements that can be reached using sequential focus navigation.

The `tabindex` attribute, if specified, must have a value that is a valid integer.

If the attribute is specified, it must be parsed using the rules for parsing integers. The attribute's values have the following meanings:

If the attribute is omitted or parsing the value returns an error

The user agent should follow platform conventions to determine if the element is to be focusable and, if so, whether the element can be reached using sequential focus navigation, and if so, what its relative order should be.

If the value is a negative integer

The user agent must allow the element to be focused, but should not allow the element to be reached using sequential focus navigation.

If the value is a zero

The user agent must allow the element to be focused, should allow the element to be reached using sequential focus navigation, and should follow platform conventions to determine the element's relative order.

If the value is greater than zero

The user agent must allow the element to be focused, should allow the element to be reached using sequential focus navigation, and should place the element in the sequential focus navigation order so that it is:

- before any focusable element whose `tabindex` attribute has been omitted or whose value, when parsed, returns an error,
- before any focusable element whose `tabindex` attribute has a value equal to or less than zero,
- after any element whose `tabindex` attribute has a value greater than zero but less than the value of the `tabindex` attribute on the element,
- after any element whose `tabindex` attribute has a value equal to the value of the `tabindex` attribute on the element but that is earlier in the document in tree order than the element,
- before any element whose `tabindex` attribute has a value equal to the value of the `tabindex` attribute on the element but that is later in the document in tree order than the element, and
- before any element whose `tabindex` attribute has a value greater than the value of the `tabindex` attribute on the element.

An element is **focusable** if the `tabindex` attribute's definition above defines the element to be focusable *and* the element is being rendered.

When an element is 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 must reflect the value of the `tabIndex` content attribute. If the attribute is not present, or parsing its value returns an error, then the DOM attribute must return 0 for elements that are focusable and −1 for elements that are not focusable.

6.6 The text selection APIs

Every browsing context has **a selection**. The selection can be empty, and the selection can have more than one range (a disjointed selection). The user 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**.

User agents may selectively ignore attempts to use the API to adjust the selection made after the user has modified the selection. For example, if the user has just selected part of a word, the user agent could ignore attempts to use the API call to immediately unselect the selection altogether, but could allow attempts to change the selection to select the entire word.

User agents may also allow the user to create selections that are not exposed to the API.

The `datagrid` and `select` elements also have selections, indicating which items have been picked by the user. These are not discussed in this section.

> ***Note: This specification does not specify how selections are presented to the user. The Selectors specification, in conjunction with CSS, can be used to style text selections using the `::selection` pseudo-element. [SELECTORS] [CSS21]***

6.6.1 APIs for the browsing context selection

The **`getSelection()`** method on the `Window` interface must return the `Selection` object representing the selection of that `Window` object's browsing context.

For historical reasons, the **`getSelection()`** method on the `HTMLDocument` interface must return the same `Selection` object.

```
[Stringifies] interface Selection {
  readonly attribute Node anchorNode;
  readonly attribute long anchorOffset;
  readonly attribute Node focusNode;
```

```
  readonly attribute long focusOffset;
  readonly attribute boolean isCollapsed;
  void collapse(in Node parentNode, in long offset);
  void collapseToStart();
  void collapseToEnd();
  void selectAllChildren(in Node parentNode);
  void deleteFromDocument();
  readonly attribute long rangeCount;
  Range getRangeAt(in long index);
  void addRange(in Range range);
  void removeRange(in Range range);
  void removeAllRanges();
};
```

The `Selection` interface is represents a list of `Range` objects. The first item in the list has index 0, and the last item has index *count*-1, where *count* is the number of ranges in the list. [DOM2RANGE]

All of the members of the `Selection` interface are defined in terms of operations on the `Range` objects represented by this object. These operations can raise exceptions, as defined for the `Range` interface; this can therefore result in the members of the `Selection` interface raising exceptions as well, in addition to any explicitly called out below.

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 stringification to return the range's text twice.

The **`removeRange(range)`** method must remove the first occurrence of *range* in the list of ranges, if it appears at all.

The **`removeAllRanges()`** method must remove all the ranges from the list of ranges, such that the `rangeCount` attribute returns 0 after the `removeAllRanges()` method is invoked (and until a new range is added to the list, either through this interface or via user interaction).

Objects implementing this interface must **stringify** to a concatenation of the results of invoking the `toString()` method of the `Range` object on each of the ranges of the selection, in the order they appear in the list (first to last).

> In the following document fragment, the emphasised parts indicate the selection.
>
> ```
> The cute girl likes the <cite>Oxford English Dictionary</cite>.
> ```
>
> If a script invoked `window.getSelection().toString()`, the return value would be "`the Oxford English`".

Note: The `Selection` interface has no relation to the `DataGridSelection` interface.

6.6.2 APIs for the text field selections

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,
> control.selectionEnd);
> ```
>
> ...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.

6.7 The `contenteditable` attribute

The **`contenteditable`** attribute is a common attribute. User agents must support this attribute on all HTML elements.

The `contenteditable` attribute is an enumerated attribute whose keywords are the empty string, `true`, and

`false`. The empty string and the `true` keyword map to the *true* state. The `false` keyword maps to the *false* state. In addition, there is a third state, the *inherit* state, which is the *missing value default* (and the *invalid value default*).

If an HTML element has a `contenteditable` attribute set to the true state, or it has its `contenteditable` attribute set to the inherit state and if its nearest ancestor HTML element with the `contenteditable` attribute set to a state other than the inherit state has its attribute set to the true state, or if it and its ancestors all have their `contenteditable` attribute set to the inherit state but the `Document` has `designMode` enabled, then the UA must treat the element as **editable** (as described below).

Otherwise, either the HTML element has a `contenteditable` attribute set to the false state, or its `contenteditable` attribute is set to the inherit state and its nearest ancestor HTML element with the `contenteditable` attribute set to a state other than the inherit state has its attribute set to the false state, or all its ancestors have their `contenteditable` attribute set to the inherit state and the `Document` itself has `designMode` disabled; either way, the element is not editable.

The **`contentEditable`** DOM attribute, on getting, must return the string "`true`" if the content attribute is set to the true state, `false`" if the content attribute is set to the false state, and "`inherit`" otherwise. On setting, if the new value is case-insensitively equal to the string "`inherit`" then the content attribute must be removed, if the new value is case-insensitively equal to the string "`true`" then the content attribute must be set to the string "`true`", if the new value is case-insensitively equal to the string "`false`" then the content attribute must be set to the string "`false`", and otherwise the attribute setter must raise a `SYNTAX_ERR` exception.

The **`isContentEditable`** DOM attribute, on getting, must return true if the element is editable, and false otherwise.

If an element is editable and its parent element is not, or if an element is editable and it has no parent element, then the element is an **editing host**. Editable elements can be nested. User agents must make editing hosts focusable (which typically means they enter the tab order). An editing host can contain non-editable sections, these are handled as described below. An editing host can contain non-editable sections that contain further editing hosts.

When an editing host has focus, it must have a **caret position** that specifies where the current editing position is. It may also have a selection.

Note: How the caret and selection are represented depends entirely on the UA.

6.7.1 User editing actions

There are several actions that the user agent should allow the user to perform while the user is interacting with an editing host. How exactly each action is triggered is not defined for every action, but when it is not defined, suggested key bindings are provided to guide implementors.

Move the caret

User agents must allow users to move the caret to any position within an editing host, even into nested editable elements. This could be triggered as the default action of `keydown` events with various key identifiers and as the default action of `mousedown` events.

Change the selection

User agents must allow users to change the selection within an editing host, even into nested editable elements. User agents may prevent selections from being made in ways that cross from editable elements into non-editable elements (e.g. by making each non-editable descendant atomically selectable, but not allowing text selection within them). This could be triggered as the default action of `keydown` events with various key identifiers and as the default action of `mousedown` events.

Insert text

This action must be triggered as the default action of a `textInput` event, and may be triggered by other commands as well. It must cause the user agent to insert the specified text (given by the event object's `data` attribute in the case of the `textInput` event) at the caret.

If the caret is positioned somewhere where phrasing content is not allowed (e.g. inside an empty `ol` element), then the user agent must not insert the text directly at the caret position. In such cases the behavior is UA-dependent, but user agents must not, in response to a request to insert text, generate a DOM that is less conformant than the DOM prior to the request.

User agents should allow users to insert new paragraphs into elements that contains only content other than paragraphs.

> For example, given the markup:
>
> ```
> <section>
>  <dl>
>   <dt> Ben </dt>
>   <dd> Goat </dd>
>  </dl>
> </section>
> ```
>
> ...the user agent should allow the user to insert `p` elements before and after the `dl` element, as children of the `section` element.

Break block

UAs should offer a way for the user to request that the current paragraph be broken at the caret, e.g. as the default action of a `keydown` event whose identifier is the "Enter" key and that has no modifiers set.

The exact behavior is UA-dependent, but user agents must not, in response to a request to break a paragraph, generate a DOM that is less conformant than the DOM prior to the request.

Insert a line separator

UAs should offer a way for the user to request an explicit line break at the caret position without breaking the paragraph, e.g. as the default action of a `keydown` event whose identifier is the "Enter" key and that has a shift modifier set. Line separators are typically found within a poem verse or an address. To insert a line break, the user agent must insert a `br` element.

If the caret is positioned somewhere where phrasing content is not allowed (e.g. in an empty `ol` element), then the user agent must not insert the `br` element directly at the caret position. In such cases the behavior is UA-dependent, but user agents must not, in response to a request to insert a line separator, generate a DOM that is less conformant than the DOM prior to the request.

Delete

UAs should offer a way for the user to delete text and elements, including non-editable descendants, e.g. as the default action of `keydown` events whose identifiers are "U+0008" or "U+007F".

Five edge cases in particular need to be considered carefully when implementing this feature: backspacing at the start of an element, backspacing when the caret is immediately after an element, forward-deleting at the end of an element, forward-deleting when the caret is immediately before an element, and deleting a selection whose start and end points do not share a common parent node.

In any case, the exact behavior is UA-dependent, but user agents must not, in response to a request to delete text or an element, generate a DOM that is less conformant than the DOM prior to the request.

Insert, and wrap text in, semantic elements

UAs should offer the user the ability to mark text and paragraphs with semantics that HTML can express.

UAs should similarly offer a way for the user to insert empty semantic elements to subsequently fill by entering text manually.

UAs should also offer a way to remove those semantics from marked up text, and to remove empty semantic element that have been inserted.

In response to a request from a user to mark text up in italics, user agents should use the `i` element to represent the semantic. The `em` element should be used only if the user agent is sure that the user means to indicate stress emphasis.

In response to a request from a user to mark text up in bold, user agents should use the `b` element to represent the semantic. The `strong` element should be used only if the user agent is sure that the user means to indicate importance.

The exact behavior is UA-dependent, but user agents must not, in response to a request to wrap semantics around some text or to insert or remove a semantic element, generate a DOM that is less conformant than the DOM prior to the request.

Select and move non-editable elements nested inside editing hosts

UAs should offer a way for the user to move images and other non-editable parts around the content within an editing host. This may be done using the drag and drop mechanism. User agents must not, in response to a request to move non-editable elements nested inside editing hosts, generate a DOM that is less conformant than the DOM prior to the request.

Edit form controls nested inside editing hosts

When an editable form control is edited, the changes must be reflected in both its current value *and* its default value. For `input` elements this means updating the `defaultValue` DOM attribute as well as the `value` DOM attribute; for `select` elements it means updating the `option` elements' `defaultSelected` DOM attribute as well as the `selected` DOM attribute; for `textarea` elements this means updating the `defaultValue` DOM attribute as well as the `value` DOM attribute. (Updating the `default*` DOM attributes causes content attributes to be updated as well.)

User agents may perform several commands per user request; for example if the user selects a block of text and hits `Enter`, the UA might interpret that as a request to delete the content of the selection followed by a request to break the paragraph at that position.

6.7.2 Making entire documents editable

Documents have a **`designMode`**, which can be either enabled or disabled.

The `designMode` DOM attribute on the `Document` object takes two values, "`on`" and "`off`". When it is set, the new value must be case-insensitively compared to these two values. If it matches the "`on`" value, then `designMode` must be enabled, and if it matches the "`off`" value, then `designMode` must be disabled. Other values must be ignored.

When `designMode` is enabled, the DOM attribute must return the value "`on`", and when it is disabled, it must return the value "`off`".

The last state set must persist until the document is destroyed or the state is changed. Initially, documents must have their `designMode` disabled.

Enabling `designMode` causes scripts in general to be disabled and the document to become editable.

6.8 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).

6.8.1 Introduction

This section is non-normative.

It's also currently non-existent.

6.8.2 The `DragEvent` and `DataTransfer` interfaces

The drag-and-drop processing model involves several events. They all use the `DragEvent` interface.

```
interface DragEvent : UIEvent {
  readonly attribute DataTransfer dataTransfer;
  void initDragEvent(in DOMString typeArg, in boolean canBubbleArg, in
boolean cancelableArg, in AbstractView viewArg, in long detailArg, in
DataTransfer dataTransferArg);
  void initDragEventNS(in DOMString namespaceURIArg, in DOMString typeArg,
in boolean canBubbleArg, in boolean cancelableArg, in AbstractView viewArg,
in long detailArg, in DataTransfer dataTransferArg);
};
```

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.

```
interface DataTransfer {
           attribute DOMString dropEffect;
           attribute DOMString effectAllowed;
  readonly attribute DOMStringList types;
  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 `DataTransfer` object is created, it must be initialized as follows:

- The `DataTransfer` object must initially contain no data, no elements, and have no associated image.
- The `DataTransfer` object's `effectAllowed` attribute must be set to "`uninitialized`".
- The `dropEffect` attribute must be set to "`none`".

The **`dropEffect`** attribute controls the drag-and-drop feedback that the user is given during a drag-and-drop operation.

The attribute must ignore any attempts to set it to a value other than `none`, `copy`, `link`, and `move`. On getting, the attribute must return the last of those four values that it was set to.

The **`effectAllowed`** attribute is used in the drag-and-drop processing model to 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, labeled as being of the type *format*. This must replace any previous data that had been set for that format. If *format* is the value "`Text`", then it must be treated as "`text/plain`". If the *format* is "`URL`", then it must be treated as "`text/uri-list`".

The **`getData(format)`** method must return the data that is associated with the type *format*, if any, and must return the empty string otherwise. If *format* is the value "`Text`", then it must be treated as "`text/plain`". If the *format* is "`URL`", then the data associated with the "`text/uri-list`" format must be parsed as appropriate for `text/uri-list` data, and the first URL from the list must be returned. If there is no data with that format, or if there is but it has no URLs, then the method must return the empty string. [RFC2483]

The **`types`** attribute must return a live `DOMStringList` that contains the list of formats that are stored in the `DataTransfer` object.

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.

6.8.3 Events fired during a drag-and-drop action

The following events are involved in the drag-and-drop model. Whenever the processing model described below causes one of these events to be fired, the event fired must use the `DragEvent` interface defined above, must have the bubbling and cancelable behaviors given in the table below, and must have the context information set up as described after the table, with the `view` attribute set to the view with which the user interacted to trigger the drag-and-drop event, and the `detail` attribute set to zero.

Event Name	Target	Bubbles?	Cancelable?	`dataTransfer`	`effectAllowed`	`dropEffect`	Default Action
`dragstart`	Source node	✓ Bubbles	✓ Cancelable	Contains source node unless a selection is being dragged, in which case it is empty	`uninitialized`	`none`	Initiate the drag-and-drop operation
`drag`	Source node	✓ Bubbles	✓ Cancelable	Empty	Same as last event	`none`	Continue the drag-and-drop operation
`dragenter`	Immediate user selection or the body element	✓ Bubbles	✓ Cancelable	Empty	Same as last event	Based on `effectAllowed` value	Reject immediate user selection as potential target element
`dragleave`	Previous target element	✓ Bubbles	—	Empty	Same as last event	`none`	None

Event Name	Target	Bubbles?	Cancelable?	dataTransfer	effectAllowed	dropEffect	Default Action
dragover	Current target element	✓ Bubbles	✓ Cancelable	Empty	Same as last event	Based on effectAllowed value	Reset the current drag operation to "none"
drop	Current target element	✓ Bubbles	✓ Cancelable	getData() returns data set in dragstart event	Same as last event	Current drag operation	Varies
dragend	Source node	✓ Bubbles	—	Empty	Same as last event	Current drag operation	Varies

The dataTransfer object's contents are empty except for dragstart events and drop events, for which the contents are set as described in the processing model, below.

The effectAllowed attribute must be set to "uninitialized" for dragstart events, and to whatever value the field had after the last drag-and-drop event was fired for all other events (only counting events fired by the user agent for the purposes of the drag-and-drop model described below).

The dropEffect attribute must be set to "none" for dragstart, drag, and dragleave events (except when stated otherwise in the algorithms given in the sections below), to the value corresponding to the current drag operation for drop and dragend events, and to a value based on the effectAllowed attribute's value and to the drag-and-drop source, as given by the following table, for the remaining events (dragenter and dragover):

effectAllowed	dropEffect
none	none
copy, copyLink, copyMove, all	copy
link, linkMove	link
move	move
uninitialized when what is being dragged is a selection from a text field	move
uninitialized when what is being dragged is a selection	copy
uninitialized when what is being dragged is an a element with an href attribute	link
Any other case	copy

6.8.4 Drag-and-drop processing model

When the user attempts to begin a drag operation, the user agent must first determine what is being dragged. If the drag operation was invoked on a selection, then it is the selection that is being dragged. Otherwise, it is the first element, going up the ancestor chain, starting at the node that the user tried to drag, that has the DOM attribute draggable set to true. If there is no such element, then nothing is being dragged, the drag-and-drop operation is never started, and the user agent must not continue with this algorithm.

> ***Note: img elements and a elements with an href attribute have their draggable attribute set to true by default.***

If the user agent determines that something can be dragged, a dragstart event must then be fired.

If it is a selection that is being dragged, then this event must be fired on the node that the user started the drag on (typically the text node that the user originally clicked). If the user did not specify a particular node, for example if the user just told the user agent to begin a drag of "the selection", then the event must be fired on the deepest node that is a common ancestor of all parts of the selection.

We should look into how browsers do other types (e.g. Firefox apparently also adds text/html for internal drag and drop of a selection).

If it is not a selection that is being dragged, then the event must be fired on the element that is being dragged.

The node on which the event is fired is the **source node**. Multiple events are fired on this node during the course of the drag-and-drop operation.

If it is a selection that is being dragged, the `dataTransfer` member of the event must be created with no nodes. Otherwise, it must be created containing just the source node. Script can use the `addElement()` method to add further elements to the list of what is being dragged.

If it is a selection that is being dragged, the `dataTransfer` member of the event must have the text of the selection added to it as the data associated with the `text/plain` format. Otherwise, if it is an `img` element being dragged, then the value of the element's `src` DOM attribute must be added, associated with the `text/uri-list` format. Otherwise, if it is an `a` element being dragged, then the value of the element's `href` DOM attribute must be added, associated with the `text/uri-list` format. Otherwise, no data is added to the object by the user agent.

If the event is canceled, then the drag-and-drop operation must not occur; the user agent must not continue with this algorithm.

If it is not canceled, then the drag-and-drop operation must be initiated.

> ***Note: Since events with no event handlers registered are, almost by definition, never canceled, drag-and-drop is always available to the user if the author does not specifically prevent it.***

The drag-and-drop feedback must be generated from the first of the following sources that is available:

1. The element specified in the last call to the `setDragImage()` method of the `dataTransfer` object of the `dragstart` event, if the method was called. In visual media, if this is used, the *x* and *y* arguments that were passed to that method should be used as hints for where to put the cursor relative to the resulting image. The values are expressed as distances in CSS pixels from the left side and from the top side of the image respectively. [CSS21]

2. The elements that were added to the `dataTransfer` object, both before the event was fired, and during the handling of the event using the `addElement()` method, if any such elements were indeed added.

3. The selection that the user is dragging.

The user agent must take a note of the data that was placed in the `dataTransfer` object. This data will be made available again when the `drop` event is fired.

From this point until the end of the drag-and-drop operation, device input events (e.g. mouse and keyboard events) must be suppressed. In addition, the user agent must track all DOM changes made during the drag-and-drop operation, and add them to its undo history as one atomic operation once the drag-and-drop operation has ended.

During the drag operation, the element directly indicated by the user as the drop target is called the

immediate user selection. (Only elements can be selected by the user; other nodes must not be made available as drop targets.) However, the immediate user selection is not necessarily the **current target element**, which is the element currently selected for the drop part of the drag-and-drop operation. The immediate user selection changes as the user selects different elements (either by pointing at them with a pointing device, or by selecting them in some other way). The current target element changes when the immediate user selection changes, based on the results of event handlers in the document, as described below.

Both the current target element and the immediate user selection can be null, which means no target element is selected. They can also both be elements in other (DOM-based) documents, or other (non-Web) programs altogether. (For example, a user could drag text to a word-processor.) The current target element is initially null.

In addition, there is also a **current drag operation**, which can take on the values "none", "copy", "link", and "move". Initially it has the value "none". It is updated by the user agent as described in the steps below.

User agents must, every 350ms (±200ms), perform the following steps in sequence. (If the user agent is still performing the previous iteration of the sequence when the next iteration becomes due, the user agent must not execute the overdue iteration, effectively "skipping missed frames" of the drag-and-drop operation.)

1. First, the user agent must fire a `drag` event at the source node. If this event is canceled, the user agent must set the current drag operation to none (no drag operation).

2. Next, if the `drag` event was not canceled and the user has not ended the drag-and-drop operation, the user agent must check the state of the drag-and-drop operation, as follows:

    1. First, if the user is indicating a different immediate user selection than during the last iteration (or if this is the first iteration), and if this immediate user selection is not the same as the current target element, then the current target element must be updated, as follows:

        1. If the new immediate user selection is null, or is in a non-DOM document or application, then set the current target element to the same value.

        2. Otherwise, the user agent must fire a `dragenter` event at the immediate user selection.

        3. If the event is canceled, then the current target element must be set to the immediate user selection.

        4. Otherwise, if the current target element is not the body element, the user agent must fire a `dragenter` event at the body element, and the current target element must be set to the body element, regardless of whether that event was canceled or not. (If the body element is null, then the current target element would be set to null too in this case, it wouldn't be set to the `Document` object.)

    2. If the previous step caused the current target element to change, and if the previous target element was not null or a part of a non-DOM document, the user agent must fire a `dragleave` event at the previous target element.

    3. If the current target element is a DOM element, the user agent must fire a `dragover` event at this current target element.

        If the `dragover` event is not canceled, the current drag operation must be reset to "none".

Otherwise, the current drag operation must be set based on the values the `effectAllowed` and `dropEffect` attributes of the `dataTransfer` object had after the event was handled, as per the following table:

`effectAllowed`	`dropEffect`	Drag operation
`uninitialized`, `copy`, `copyLink`, `copyMove`, or `all`	`copy`	"copy"
`uninitialized`, `link`, `copyLink`, `linkMove`, or `all`	`link`	"link"
`uninitialized`, `move`, `copyMove`, `linkMove`, or `all`	`move`	"move"
Any other case		"none"

Then, regardless of whether the `dragover` event was canceled or not, the drag feedback (e.g. the mouse cursor) must be updated to match the current drag operation, as follows:

Drag operation	Feedback
"copy"	Data will be copied if dropped here.
"link"	Data will be linked if dropped here.
"move"	Data will be moved if dropped here.
"none"	No operation allowed, dropping here will cancel the drag-and-drop operation.

4. Otherwise, if the current target element is not a DOM element, the user agent must use platform-specific mechanisms to determine what drag operation is being performed (none, copy, link, or move). This sets the *current drag operation*.

3. Otherwise, if the user ended the drag-and-drop operation (e.g. by releasing the mouse button in a mouse-driven drag-and-drop interface), or if the `drag` event was canceled, then this will be the last iteration. The user agent must execute the following steps, then stop looping.

    1. If the current drag operation is none (no drag operation), or, if the user ended the drag-and-drop operation by canceling it (e.g. by hitting the `Escape` key), or if the current target element is null, then the drag operation failed. If the current target element is a DOM element, the user agent must fire a `dragleave` event at it; otherwise, if it is not null, it must use platform-specific conventions for drag cancellation.

    2. Otherwise, the drag operation was as success. If the current target element is a DOM element, the user agent must fire a `drop` event at it; otherwise, it must use platform-specific conventions for indicating a drop.

        When the target is a DOM element, the `dropEffect` attribute of the event's `dataTransfer` object must be given the value representing the current drag operation (`copy`, `link`, or `move`), and the object must be set up so that the `getData()` method will return the data that was added during the `dragstart` event.

        If the event is canceled, the current drag operation must be set to the value of the `dropEffect` attribute of the event's `dataTransfer` object as it stood after the event was handled.

        Otherwise, the event is not canceled, and the user agent must perform the event's default action, which depends on the exact target as follows:

        ↪ **If the current target element is a text field (e.g. `textarea`, or an `input` element 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.

6.8.4.1. When the drag-and-drop operation starts or ends in another document

The model described above is independent of which `Document` object the nodes involved are from; the events must be fired as described above and the rest of the processing model must be followed as described above, irrespective of how many documents are involved in the operation.

6.8.4.2. When the drag-and-drop operation starts or ends in another application

If the drag is initiated in another application, the source node is not a DOM node, and the user agent must use platform-specific conventions instead when the requirements above involve the source node. User agents in this situation must act as if the dragged data had been added to the `DataTransfer` object when the drag started, even though no `dragstart` event was actually fired; user agents must similarly use platform-specific conventions when deciding on what drag feedback to use.

If a drag is started in a document but ends in another application, then the user agent must instead replace the parts of the processing model relating to handling the *target* according to platform-specific conventions.

In any case, scripts running in the context of the document must not be able to distinguish the case of a drag-

and-drop operation being started or ended in another application from the case of a drag-and-drop operation being started or ended in another document from another domain.

6.8.5 The `draggable` attribute

All elements may have the `draggable` content attribute set. The `draggable` attribute is an enumerated attribute. It has three states. The first state is *true* and it has the keyword `true`. The second state is *false* and it has the keyword `false`. The third state is *auto*; it has no keywords but it is the *missing value default*.

The **`draggable`** DOM attribute, whose value depends on the content attribute's in the way described below, controls whether or not the element is draggable. Generally, only text selections are draggable, but elements whose `draggable` DOM attribute is true become draggable as well.

If an element's `draggable` content attribute has the state *true*, the `draggable` DOM attribute must return true.

Otherwise, if the element's `draggable` content attribute has the state *false*, the `draggable` DOM attribute must return false.

Otherwise, the element's `draggable` content attribute has the state *auto*. If the element is an `img` element, or, if the element is an `a` element with an `href` content attribute, the `draggable` DOM attribute must return true.

Otherwise, the `draggable` DOM must return false.

If the `draggable` DOM attribute is set to the value false, the `draggable` content attribute must be set to the literal value `false`. If the `draggable` DOM attribute is set to the value true, the `draggable` content attribute must be set to the literal value `true`.

6.8.6 Copy and paste

Copy-and-paste is a form of drag-and-drop: the "copy" part is equivalent to dragging content to another application (the "clipboard"), and the "paste" part is equivalent to dragging content *from* another application.

Select-and-paste (a model used by mouse operations in the X Window System) is equivalent to a drag-and-drop operation where the source is the selection.

6.8.6.1. Copy to clipboard

When the user invokes a copy operation, the user agent must act as if the user had invoked a drag on the current selection. If the drag-and-drop operation initiates, then the user agent must act as if the user had indicated (as the immediate user selection) a hypothetical application representing the clipboard. Then, the user agent must act as if the user had ended the drag-and-drop operation without canceling it. If the drag-and-drop operation didn't get canceled, the user agent should then follow the relevant platform-specific conventions for copy operations (e.g. updating the clipboard).

6.8.6.2. Cut to clipboard

When the user invokes a cut operation, the user agent must act as if the user had invoked a copy operation (see the previous section), followed, if the copy was completed successfully, by a selection delete operation.

6.8.6.3. Paste from clipboard

When the user invokes a clipboard paste operation, the user agent must act as if the user had invoked a drag on a hypothetical application representing the clipboard, setting the data associated with the drag as the content on the clipboard (in whatever formats are available).

Then, the user agent must act as if the user had indicated (as the immediate user selection) the element with the keyboard focus, and then ended the drag-and-drop operation without canceling it.

6.8.6.4. Paste from selection

When the user invokes a selection paste operation, the user agent must act as if the user had invoked a drag on the current selection, then indicated (as the immediate user selection) the element with the keyboard focus, and then ended the drag-and-drop operation without canceling it.

6.8.7 Security risks in the drag-and-drop model

User agents must not make the data added to the `DataTransfer` object during the `dragstart` event available to scripts until the `drop` event, because otherwise, if a user were to drag sensitive information from one document to a second document, crossing a hostile third document in the process, the hostile document could intercept the data.

For the same reason, user agents must consider a drop to be successful only if the user specifically ended the drag operation — if any scripts end the drag operation, it must be considered unsuccessful (canceled) and the `drop` event must not be fired.

User agents should take care to not start drag-and-drop operations in response to script actions. For example, in a mouse-and-window environment, if a script moves a window while the user has his mouse button depressed, the UA would not consider that to start a drag. This is important because otherwise UAs could cause data to be dragged from sensitive sources and dropped into hostile documents without the user's consent.

6.9 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).

6.9.1 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 DOMString title);
  [XXX] void remove(in unsigned long index);
  void clearUndo();
  void clearRedo();
  [IndexGetter] DOMObject item(in unsigned long index);
  readonly attribute unsigned long length;
  readonly attribute unsigned long position;
};
```

The **`undoManager`** attribute of the `Window` interface must return the object implementing the `UndoManager` interface for that `Window` object's associated `HTMLDocument` object.

`UndoManager` objects represent their document's undo transaction history. Only undo object entries are visible with this API, but this does not mean that DOM changes entries are absent from the undo transaction history.

The **`length`** attribute must return the number of undo object entries in the undo transaction history.

The **`item(n)`** method must return the *n*th undo object entry in the undo transaction history.

The undo transaction history has a **current position**. This is the position between two entries in the undo transaction history's list where the previous entry represents what needs to happen if the user invokes the "undo" command (the "undo" side, lower numbers), and the next entry represents what needs to happen if the user invokes the "redo" command (the "redo" side, higher numbers).

The **`position`** attribute must return the index of the undo object entry nearest to the undo position, on the "redo" side. If there are no undo object entries on the "redo" side, then the attribute must return the same as the `length` attribute. If there are no undo object entries on the "undo" side of the undo position, the `position` attribute returns zero.

> ***Note: Since the undo transaction history contains both undo object entries and DOM changes entries, but the `position` attribute only returns indices relative to undo object entries, it is possible for several "undo" or "redo" actions to be performed without the value of the `position` attribute changing.***

The **`add(data, title)`** method's behavior depends on the current state. Normally, it must insert the *data* object passed as an argument into the undo transaction history immediately before the undo position, optionally remembering the given *title* to use in the UI. If the method is called during an undo operation, however, the object must instead be added immediately *after* the undo position.

If the method is called and there is neither an undo operation in progress nor a redo operation in progress then any entries in the undo transaction history after the undo position must be removed (as if `clearRedo()` had been called).

We could fire events when someone adds something to the undo history -- one event per undo object entry before the position (or after, during redo addition), allowing the script to decide if that entry should remain or not. Or something. Would make it potentially easier to expire server-held state when the server limitations come into play.

The **`remove(index)`** method must remove the undo object entry with the specified *index*. If the index is less than zero or greater than or equal to `length` then the method must raise an `INDEX_SIZE_ERR` exception. DOM changes entries are unaffected by this method.

The **`clearUndo()`** method must remove all entries in the undo transaction history before the undo position, be they DOM changes entries or undo object entries.

The **`clearRedo()`** method must remove all entries in the undo transaction history after the undo position, be they DOM changes entries or undo object entries.

Another idea is to have a way for scripts to say "startBatchingDOMChangesForUndo()" and after that the changes to the DOM go in as if the user had done them.

6.9.2 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.

6.9.3 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.

6.9.4 The `UndoManagerEvent` interface and the `undo` and `redo` events

```
interface UndoManagerEvent : Event {
  readonly attribute DOMObject data;
  void initUndoManagerEvent(in DOMString typeArg, in boolean canBubbleArg,
in boolean cancelableArg, in DOMObject dataArg);
  void initUndoManagerEventNS(in DOMString namespaceURIArg, in DOMString
typeArg, in boolean canBubbleArg, in boolean cancelableArg, in DOMObject
dataArg);
};
```

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.

6.9.5 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 official (as described above) undo transaction history (such as providing a tree-based approach to document state). Such UI models should be based upon the single undo transaction history described in this section, however, such that to a script there is no detectable difference.

6.10 Command APIs

The **`execCommand(`*`commandId, showUI, value`*`)`** method on the `HTMLDocument` interface allows scripts to perform actions on the current selection or at the current caret position. Generally, these commands would be used to implement editor UI, for example having a "delete" button on a toolbar.

There are three variants to this method, with one, two, and three arguments respectively. The *showUI* and *value* parameters, even if specified, are ignored unless otherwise stated.

When `execCommand()` is invoked, the user agent must follow the following steps:

1. If the given *commandId* maps to an entry in the list below whose "Enabled When" entry has a condition that is currently false, do nothing; abort these steps.

2. Otherwise, execute the "Action" listed below for the given *commandId*.

A document is **ready for editing host commands** if it has a selection that is entirely within an editing host, or if it has no selection but its caret is inside an editing host.

The **`queryCommandEnabled(`*`commandId`*`)`** method, when invoked, must return true if the condition listed below under "Enabled When" for the given *commandId* is true, and false otherwise.

The **`queryCommandIndeterm(`*`commandId`*`)`** method, when invoked, must return true if the condition listed below under "Indeterminate When" for the given *commandId* is true, and false otherwise.

The **`queryCommandState(`*`commandId`*`)`** method, when invoked, must return the value expressed below under "State" for the given *commandId*.

The **`queryCommandSupported(`*`commandId`*`)`** method, when invoked, must return true if the given *commandId* is in the list below, and false otherwise.

The **`queryCommandValue(`*`commandId`*`)`** method, when invoked, must return the value expressed below under "Value" for the given *commandId*.

The possible values for *commandId*, and their corresponding meanings, are as follows. These values are case-insensitive.

`bold`

Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `b` element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: True if the selection, or the caret, if there is no selection, is, or is contained within, a `b` element. False otherwise.
Value: The string "`true`" if the expression given for the "State" above is true, the string "`false`" otherwise.

`createLink`

Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `a` element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA). If the user agent creates an `a` element or modifies an existing `a` element, then if the *showUI* argument is present and has the value false, then the value of the *value* argument must be used as the URL of the link. Otherwise, the user should be prompted for the URL of the link.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`delete`

Action: The user agent must act as if the user had performed a backspace operation.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`formatBlock`

Action: The user agent must run the following steps:

1. If the *value* argument wasn't specified, abort these steps without doing anything.

2. If the *value* argument has a leading U+003C LESS-THAN SIGN character ('<') and a trailing U+003E GREATER-THAN SIGN character ('>'), then remove the first and last characters from *value*.

3. If *value* is (now) a case-insensitive match for the tag name of an element defined by this specification that is defined to be a prose element but not a phrasing element, then, for every position in the selection, take the furthest flow content ancestor element of that position that contains only phrasing content, and, if that element is editable, and has a content model that allows it to contain prose content other than phrasing content, and has a parent element whose content model allows that parent to contain any prose content, rename the element (as if the `Element.renameNode()` method had been used) to *value*, using the HTML namespace.

If there is no selection, then, where in the description above refers to the selection, the user agent must act as if the selection was an empty range (with just one position) at the caret position.

Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

forwardDelete

Action: The user agent must act as if the user had performed a forward delete operation.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

insertImage

Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the *`img`* element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA). If the user agent creates an `img` element or modifies an existing `img` element, then if the *showUI* argument is present and has the value false, then the value of the *value* argument must be used as the URL of the image. Otherwise, the user should be prompted for the URL of the image.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

insertHTML

Action: The user agent must run the following steps:

1. If the *value* argument wasn't specified, abort these steps without doing anything.
2. If there is a selection, act as if the user had requested that the selection be deleted.
3. Invoke the HTML fragment parsing algorithm with an arbitrary orphan `body` element as the *context* element and with the *value* argument as *input*.
4. Insert the nodes returned by the previous step into the document at the location of the caret.

Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

insertLineBreak

Action: The user agent must act as if the user had requested a line separator.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

insertOrderedList

Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `ol` element (or unwrapped, or, if there is no selection, have that semantic inserted or removed — the exact behavior is UA-defined).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

insertUnorderedList

Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `ul` element (or unwrapped, or, if there is no selection, have that semantic inserted or removed — the exact behavior is UA-defined).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

insertParagraph

Action: The user agent must act as if the user had performed a break block editing action.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

insertText

Action: The user agent must act as if the user had inserted text corresponding to the *value* parameter.
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

italic

Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `i` element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: True if the selection, or the caret, if there is no selection, is, or is contained within, a `i` element. False otherwise.
Value: The string "`true`" if the expression given for the "State" above is true, the string "`false`" otherwise.

redo

Action: The user agent must move forward one step in its undo transaction history, restoring the associated state. If the undo position is at the end of the undo transaction history, the user agent must do nothing. See the undo history.
Enabled When: The undo position is not at the end of the undo transaction history.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

selectAll

Action: The user agent must change the selection so that all the content in the currently focused editing host is selected. If no editing host is focused, then the content of the entire document must be selected.
Enabled When: Always.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

subscript

Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the *`sub`* element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: True if the selection, or the caret, if there is no selection, is, or is contained within, a `sub` element. False otherwise.
Value: The string "`true`" if the expression given for the "State" above is true, the string "`false`" otherwise.

superscript

Action: The user agent must act as if the user had requested that the selection be wrapped in the semantics of the `sup` element (or unwrapped, or, if there is no selection, have that semantic inserted or removed — the exact behavior is UA-defined).
Enabled When: The document is ready for editing host commands.
Indeterminate When: Never.
State: True if the selection, or the caret, if there is no selection, is, or is contained within, a `sup` element. False otherwise.
Value: The string "`true`" if the expression given for the "State" above is true, the string "`false`" otherwise.

undo

Action: The user agent must move back one step in its undo transaction history, restoring the associated state. If the undo position is at the start of the undo transaction history, the user agent must do nothing. See the undo history.
Enabled When: The undo position is not at the start of the undo transaction history.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

unlink

Action: The user agent must remove all `a` elements that have `href` attributes and that are partially or completely included in the current selection.
Enabled When: The document has a selection that is entirely within an editing host.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

unselect

Action: The user agent must change the selection so that nothing is selected.

Enabled When: Always.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

`vendorID-customCommandID`

Action: User agents may implement vendor-specific extensions to this API. Vendor-specific extensions to the list of commands should use the syntax *`vendorID-customCommandID`* so as to prevent clashes between extensions from different vendors and future additions to this specification.
Enabled When: UA-defined.
Indeterminate When: UA-defined.
State: UA-defined.
Value: UA-defined.

Anything else

Action: User agents must do nothing.
Enabled When: Never.
Indeterminate When: Never.
State: Always false.
Value: Always the string "`false`".

7. Communication

7.1 Event definitions

Messages in server-sent events, Web sockets, cross-document messaging, and channel messaging use the **`message`** event.

The following interface is defined for this event:

```
interface MessageEvent : Event {
  readonly attribute DOMString data;
  readonly attribute DOMString origin;
  readonly attribute DOMString lastEventId;
  readonly attribute Window source;
  readonly attribute MessagePort messagePort;
  void initMessageEvent(in DOMString typeArg, in boolean canBubbleArg, in
boolean cancelableArg, in DOMString dataArg, in DOMString originArg, in
DOMString lastEventIdArg, in Window sourceArg, in MessagePort
messagePortArg);
  void initMessageEventNS(in DOMString namespaceURI, in DOMString typeArg,
in boolean canBubbleArg, in boolean cancelableArg, in DOMString dataArg, in
DOMString originArg, in DOMString lastEventIdArg, in Window sourceArg, in
MessagePort messagePortArg);
};
```

The **`initMessageEvent()`** and **`initMessageEventNS()`** 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 message being sent.

The **`origin`** attribute represents, in server-sent events and cross-document messaging, the origin of the document that sent the message (typically the scheme, hostname, and port of the document, but not its path or fragment identifier).

The **`lastEventId`** attribute represents, in server-sent events, the last event ID string of the event source.

The **`source`** attribute represents, in cross-document messaging, the `Window` from which the message came.

The **`messagePort`** attribute represents, in cross-document messaging and channel messaging the `MessagePort` being sent, if any.

Unless otherwise specified, when the user agent creates and dispatches a `message` event in the algorithms described in the following sections, the `lastEventId` attribute must be the empty string, the `origin` attribute must be the empty string, the `source` attribute must be null, and the `messagePort` attribute must be null.

7.2 Server-sent events

This section describes a mechanism for allowing servers to dispatch DOM events into documents that expect it. The `eventsource` element provides a simple interface to this mechanism.

7.2.1 The `RemoteEventTarget` interface

Any object that implements the `EventTarget` interface must also implement the `RemoteEventTarget` interface.

```
interface RemoteEventTarget {
  void addEventSource(in DOMString src);
  void removeEventSource(in DOMString src);
};
```

When the **`addEventSource(`*`src`*`)`** method is invoked, the user agent must resolve the URL specified in *src*, and if that succeeds, add the resulting absolute URL to the list of event sources for that object. The same URL can be registered multiple times. If the URL fails to resolve, then the user agent must raise a `SYNTAX_ERR` exception.

When the **`removeEventSource(`*`src`*`)`** method is invoked, the user agent must resolve the URL specified in *src*, and if that succeeds, remove the resulting absolute URL from the list of event sources for that object. If the same URI has been registered multiple times, removing it must remove only one instance of that URI for each invocation of the `removeEventSource()` method. If the URL fails to resolve, the user agent does nothing.

7.2.2 Connecting to an event stream

Each object implementing the `EventTarget` and `RemoteEventTarget` interfaces has a **list of event sources** that are registered for that object.

When a new URI is added to this list, the user agent should, as soon as all currently executing scripts (if any) have finished executing, and if the specified URL isn't removed from the list before they do so, fetch the resource identified by that URL.

When an event source is removed from the list of event sources for an object, if that resource is still being fetched, then the relevant connection must be closed.

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.

Each event source in the list must have associated with it the following:

- The **reconnection time**, in milliseconds. This must initially be a user-agent-defined value, probably in the region of a few seconds.
- The **last event ID string**. This must initially be the empty string.

In general, the semantics of the transport protocol specified by the URLs for the event sources must be followed, including HTTP caching rules.

For HTTP connections, the `Accept` header may be included; if included, it must contain only formats of event framing that are supported by the user agent (one of which must be `text/event-stream`, as described below).

Other formats of event framing may also be supported in addition to `text/event-stream`, but this specification does not define how they are to be parsed or processed.

> ***Note: Such formats could include systems like SMS-push; for example servers could use `Accept` headers and HTTP redirects to an SMS-push mechanism as a kind of protocol negotiation to reduce network load in GSM environments.***

User agents should use the `Cache-Control: no-cache` header in requests to bypass any caches for requests of event sources.

If the event source's last event ID string is not the empty string, then a `Last-Event-ID` HTTP header must be included with the request, whose value is the value of the event source's last event ID string.

For connections to domains other than the document's domain, the semantics of the Access-Control HTTP header must be followed. [ACCESSCONTROL]

HTTP 200 OK responses with a Content-Type header specifying the type `text/event-stream` that are either from the document's domain or explicitly allowed by the Access-Control HTTP headers must be processed line by line as described below.

For the purposes of such successfully opened event streams only, user agents should ignore HTTP cache headers, and instead assume that the resource indicates that it does not wish to be cached.

If such a resource completes loading (i.e. the entire HTTP response body is received or the connection itself closes), the user agent should request the event source resource again after a delay equal to the reconnection time of the event source.

HTTP 200 OK responses that have a Content-Type other than `text/event-stream` (or some other supported type), and HTTP responses whose Access-Control headers indicate that the resource are not to be used, must be ignored and must prevent the user agent from refetching the resource for that event source.

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 resources. 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 refetch the resource after a delay equal to the reconnection time of the event source.

HTTP 300 Multiple Choices responses should be handled automatically if possible (treating the responses as if they were 302 Found 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 reconnect using the new server specified URL instead of the previously specified URL for all subsequent requests for this event source. (It doesn't affect other event sources with the same URL unless they also receive 301 responses, and it doesn't affect future sessions, e.g. if the page is reloaded.)

HTTP 302 Found, 303 See Other, and 307 Temporary Redirect responses must cause the user agent to connect to the new server-specified URL, but if the user agent needs to again request the resource at a later point, it must return to the previously specified URL for this event source.

HTTP 304 Not Modified responses should be handled like HTTP 200 OK responses, with the content coming from the user agent cache. A new request should then be made after a delay equal to the reconnection time of the event source.

HTTP 305 Use Proxy, HTTP 401 Unauthorized, and 407 Proxy Authentication Required should be treated transparently as for any other subresource.

Any other HTTP response code not listed here should cause the user agent to stop trying to process this event source.

DNS errors must be considered fatal, and cause the user agent to not open any connection for that event source.

For non-HTTP protocols, UAs should act in equivalent ways.

7.2.3 Parsing an event stream

This event stream format's MIME type is `text/event-stream`.

The event stream format is (in pseudo-BNF):

```
<stream>          ::= <bom>? <event>*
<event>           ::= [ <comment> | <field> ]* <newline>
<comment>         ::= <colon> <any-char>* <newline>
<field>           ::= <name-char>+ [ <colon> <space>? <any-char>* ]? <newline>

# characters:
<bom>             ::= a single U+FEFF BYTE ORDER MARK character
<space>           ::= a single U+0020 SPACE character (' ')
<newline>         ::= a U+000D CARRIAGE RETURN character
                      followed by a U+000A LINE FEED character
                      | a single U+000D CARRIAGE RETURN character
                      | a single U+000A LINE FEED character
                      | the end of the file
<colon>           ::= a single U+003A COLON character (':')
<name-char>       ::= a single Unicode character other than
                      U+003A COLON, U+000D CARRIAGE RETURN and U+000A LINE
FEED
<any-char>        ::= a single Unicode character other than
                      U+000D CARRIAGE RETURN and U+000A LINE FEED
```

Event streams in this format must always be encoded as UTF-8.

Lines must be separated by either a U+000D CARRIAGE RETURN U+000A LINE FEED (CRLF) character pair, a single U+000A LINE FEED (LF) character, or a single U+000D CARRIAGE RETURN (CR) character.

7.2.4 Interpreting an event stream

Bytes or sequences of bytes that are not valid UTF-8 sequences must be interpreted as the U+FFFD REPLACEMENT CHARACTER.

One leading U+FEFF BYTE ORDER MARK character must be ignored if any are present.

The stream must then be parsed by reading everything line by line, with a U+000D CARRIAGE RETURN U+000A LINE FEED (CRLF) character pair, a single U+000A LINE FEED (LF) character, a single U+000D CARRIAGE RETURN (CR) character, and the end of the file being the four ways in which a line can end.

When a stream is parsed, a *data* buffer and an *event name* buffer must be associated with it. They must be initialized to the empty string

Lines must be processed, in the order they are received, as follows:

↪ **If the line is empty (a blank line)**

Dispatch the event, as defined below.

↪ **If the line starts with a U+003A COLON character (':')**

Ignore the line.

↪ **If the line contains a U+003A COLON character (':') character**

Collect the characters on the line before the first U+003A COLON character (':'), and let *field* be that string.

Collect the characters on the line after the first U+003A COLON character (':'), and let *value* be that string. If *value* starts with a single U+0020 SPACE character, remove it from *value*.

Process the field using the steps described below, using *field* as the field name and *value* as the field value.

↪ **Otherwise, the string is not empty but does not contain a U+003A COLON character (':') character**

Process the field using the steps described below, using the whole line as the field name, and the empty string as the field value.

Once the end of the file is reached, the user agent must dispatch the event one final time, as defined below.

The steps to **process the field** given a field name and a field value depend on the field name, as given in the following list. Field names must be compared literally, with no case folding performed.

↪ **If the field name is "event"**

Set the *event name* buffer the to field value.

↪ **If the field name is "data"**

If the *data* buffer is not the empty string, then append a single U+000A LINE FEED character to the *data* buffer. Append the field value to the *data* buffer.

↪ **If the field name is "id"**

Set the event stream's last event ID to the field value.

↪ **If the field name is "retry"**

If the field value consists of only characters in the range U+0030 DIGIT ZERO ('0') U+0039 DIGIT NINE ('9'), then interpret the field value as an integer in base ten, and set the event stream's reconnection time to that integer. Otherwise, ignore the field.

↪ **Otherwise**

The field is ignored.

When the user agent is required to **dispatch the event**, then the user agent must act as follows:

1. If the *data* buffer is an empty string, set the *data* buffer and the *event name* buffer to the empty string and abort these steps.
2. If the *event name* buffer is not the empty string but is also not a valid NCName, set the *data* buffer and the *event name* buffer to the empty string and abort these steps.
3. Otherwise, create an event that uses the `MessageEvent` interface, with the event name `message`, which does not bubble, is cancelable, and has no default action. The `data` attribute must be set to the value of the *data* buffer, the `origin` attribute must be set to the Unicode serialization of the origin of the event stream's URL, and the `lastEventId` attribute must be set to the last event ID string of the event source.
4. If the *event name* buffer has a value other than the empty string, change the type of the newly created event to equal the value of the *event name* buffer.
5. Set the *data* buffer and the *event name* buffer to the empty string.
6. Dispatch the newly created event at the `RemoteEventTarget` object to which the event stream is registered.

Note: If an event doesn't have an "id" field, but an earlier event did set the event source's last event ID string, then the event's `lastEventId` field will be set to the value of whatever the last seen "id" field was.

The following event stream, once followed by a blank line:

```
data: YHOO
data: -2
data: 10
```

...would cause an event `message` with the interface `MessageEvent` to be dispatched on the `eventsource` element, whose `data` attribute would contain the string `YHOO\n-2\n10` (where `\n` represents a newline).

This could be used as follows:

```
<eventsource src="http://stocks.example.com/ticker.php"
             onmessage="var data = event.data.split('\n');
updateStocks(data[0], data[1], data[2]);">
```

...where `updateStocks()` is a function defined as:

```
function updateStocks(symbol, delta, value) { ... }
```

...or some such.

The following stream contains four blocks. The first block has just a comment, and will fire nothing. The second block has two fields with names "data" and "id" respectively; an event will be fired for this block, with the data "first event", and will then set the last event ID to "1" so that if the connection died between

this block and the next, the server would be sent a `Last-Event-ID` header with the value "1". The third block fires an event with data "second event", and also has an "id" field, this time with no value, which resets the last event ID to the empty string (meaning no `Last-Event-ID` header will now be sent in the event of a reconnection being attempted). Finally the last block just fires an event with the data "third event". Note that the last block doesn't have to end with a blank line, the end of the stream is enough to trigger the dispatch of the last event.

```
: test stream

data: first event
id: 1

data: second event
id

data: third event
```

The following stream fires just one event:

```
data

data
data

data:
```

The first and last blocks do nothing, since they do not contain any actual data (the *data* buffer remains at the empty string, and so nothing gets dispatched). The middle block fires an event with the data set to a single newline character.

The following stream fires two identical events:

```
data:test

data: test
```

This is because the space after the colon is ignored if present.

7.2.5 Notes

Legacy proxy servers are known to, in certain cases, drop HTTP connections after a short timeout. To protect against such proxy servers, authors can include a comment line (one starting with a ':' character) every 15 seconds or so.

Authors wishing to relate event source connections to each other or to specific documents previously served might find that relying on IP addresses doesn't work, as individual clients can have multiple IP addresses (due to having multiple proxy servers) and individual IP addresses can have multiple clients (due to sharing a proxy server). It is better to include a unique identifier in the document when it is served and then pass that identifier as part of the URL in the `src` attribute of the `eventsource` element.

Implementations that support HTTP's per-server connection limitation might run into trouble when opening

multiple pages from a site if each page has an `eventsource` to the same domain. Authors can avoid this using the relatively complex mechanism of using unique domain names per connection, or by allowing the user to enable or disable the `eventsource` functionality on a per-page basis.

7.3 Web sockets

To enable Web applications to maintain bidirectional communications with their originating server, this specification introduces the `WebSocket` interface.

> ***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.***

7.3.1 Introduction

This section is non-normative.

An introduction to the client-side and server-side of using the direct connection APIs.

7.3.2 The `WebSocket` interface

```
[Constructor(in DOMString url)]
interface WebSocket {
  readonly attribute DOMString URL;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSED = 2;
  readonly attribute int readyState;

  // networking
           attribute EventListener onopen;
           attribute EventListener onmessage;
           attribute EventListener onclosed;
  void postMessage(in DOMString data);
  void disconnect();
};
```

`WebSocket` objects must also implement the `EventTarget` interface. [DOM3EVENTS]

The **`WebSocket(`*`url`*`)`** constructor takes one argument, *url*, which specifies the URL to which to connect. When a `WebSocket` object is created, the UA must parse this argument and verify that the URL parses without failure and has a <scheme> component whose value is either "`ws`" or "`wss`", when compared case-insensitively. If it does, it has, and it is, then the user agent must asynchronously establish a Web Socket

connection to *url*. Otherwise, the constructor must raise a `SYNTAX_ERR` exception.

The **`URL`** attribute must return the value that was passed to the constructor.

The **`readyState`** attribute represents the state of the connection. It can have the following values:

`CONNECTING` **(numeric value 0)**

The connection has not yet been established.

`OPEN` **(numeric value 1)**

The Web Socket connection is established and communication is possible.

`CLOSED` **(numeric value 2)**

The connection has been closed or could not be opened.

When the object is created its `readyState` must be set to `CONNECTING` (0).

The **`send(`*`data`*`)`** method transmits data using the connection. If the connection is not established (`readyState` is not `OPEN`), it must raise an `INVALID_STATE_ERR` exception. If the connection *is* established, then the user agent must send *data* using the Web Socket.

The **`disconnect()`** method must close the Web Socket connection or connection attempt, if any. 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 below.

7.3.3 WebSocket Events

The **`open`** event is fired when the Web Socket connection is established.

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).

> ***Note: No information regarding why the connection was closed is passed to the application in this version of this specification.***

The `message` event is fired when when data is received for a connection.

Events that would be fired during script execution (e.g. between the `WebSocket` object being created — and thus the connection being established — and the current script completing; or, during the execution of a `message` event handler) must be buffered, and those events queued up and each one individually fired after the script has completed.

The following are the event handler DOM attributes that must be supported by objects implementing the `WebSocket` interface:

`onopen`

Must be invoked whenever an `open` event is targeted at or bubbles through the `WebSocket` object.

`onmessage`

Must be invoked whenever a `message` event is targeted at or bubbles through the `WebSocket` object.

onclose

Must be invoked whenever an `close` event is targeted at or bubbles through the `WebSocket` object.

7.3.4 The Web Socket protocol

7.3.4.1. Client-side requirements

This section only applies to user agents.

7.3.4.1.1. Handshake

When the user agent is to **establish a Web Socket connection** to *url*, it must run the following steps, in the background (without blocking scripts or anything like that):

1. Resolve the URL *url*.
2. If the <scheme> component of the resulting absolute URL is "`ws`", set *secure* to false; otherwise, the <scheme> component is "`wss`", set *secure* to true.
3. Let *host* be the value of the <host> component in the resulting absolute URL.
4. If the resulting absolute URL has a <port> component, then let *port* be that component's value; otherwise, if *secure* is false, let *port* be 81, otherwise let *port* be 815.
5. Let *resource name* be the value of the <path> component (which might be empty) in the resulting absolute URL.
6. If *resource name* is the empty string, set it to a single character U+002F SOLIDUS (/).
7. If the resulting absolute URL has a <query> component, then append a single 003F QUESTION MARK (?) character to *resource name*, followed by the value of the <query> component.
8. If the user agent is configured to use a proxy to connect to port *port*, then connect to that proxy and ask it to open a TCP/IP connection to the host given by *host* and the port given by *port*.

   > For example, if the user agent uses an HTTP proxy, then if it was to try to connect to port 80 on server example.com, it might send the following lines to the proxy server:
   >
   > ```
   > CONNECT example.com HTTP/1.1
   > ```
   >
   > If there was a password, the connection might look like:
   >
   > ```
   > CONNECT example.com HTTP/1.1
   > Proxy-authorization: Basic ZWRuYW1vZGU6bm9jYXBlcyE=
   > ```

   Otherwise, if the user agent is not configured to use a proxy, then open a TCP/IP connection to the host given by *host* and the port given by *port*.
9. If the connection could not be opened, then fail the Web Socket connection and abort these steps.
10. If *secure* is true, perform a TLS handshake over the connection. If this fails (e.g. the server's certificate could not be verified), then fail the Web Socket connection and abort these steps. Otherwise, all further communication on this channel must run through the encrypted tunnel. [RFC2246]

11. Send the following bytes to the remote side (the server):

    ```
    47 45 54 20
    ```

    Send the *resource name* value, encoded as US-ASCII.

    Send the following bytes:

    ```
    20 48 54 54 50 2f 31 2e  31 0d 0a 55 70 67 72 61
    64 65 3a 20 57 65 62 53  6f 63 6b 65 74 0d 0a 43
    6f 6e 6e 65 63 74 69 6f  6e 3a 20 55 70 67 72 61
    64 65 0d 0a
    ```

    Note: The string "GET ", the path, " HTTP/1.1", CRLF, the string "Upgrade: WebSocket", CRLF, and the string "Connection: Upgrade", CRLF.

12. Send the following bytes:

    ```
    48 6f 73 74 3a 20
    ```

    Send the *host* value, encoded as US-ASCII, if it represents a host name (and not an IP address).

    Send the following bytes:

    ```
    0d 0a
    ```

    Note: The string "Host: ", the host, and CRLF.

13. Send the following bytes:

    ```
    4f 72 69 67 69 6e 3a 20
    ```

    Send the ASCII serialization of the origin of the script that invoked the `WebSocket()` constructor.

    Send the following bytes:

    ```
    0d 0a
    ```

    Note: The string "Origin: ", the origin, and CRLF.

14. If the client has any authentication information or cookies that would be relevant to a resource with a URL that has a scheme of `http` if *secure* is false and `https` if *secure* is true and is otherwise identical to *url*, then HTTP headers that would be appropriate for that information should be sent at this point. [RFC2616] [RFC2109] [RFC2965]

    Each header must be on a line of its own (each ending with a CR LF sequence). For the purposes of this step, each header must not be split into multiple lines (despite HTTP otherwise allowing this with continuation lines).

    > For example, if the server had a username and password that applied to that URL, it could send:
    >
    > ```
    > Authorization: Basic d2FsbGU6ZXZl
    > ```

15. Send the following bytes:

    ```
    0d 0a
    ```

    Note: Just a CRLF (a blank line).

16. Read the first 85 bytes from the server. If the connection closes before 85 bytes are received, or if the first 85 bytes aren't exactly equal to the following bytes, then fail the Web Socket connection and abort these steps.

    ```
    48 54 54 50 2f 31 2e 31  20 31 30 31 20 57 65 62
    20 53 6f 63 6b 65 74 20  50 72 6f 74 6f 63 6f 6c
    20 48 61 6e 64 73 68 61  6b 65 0d 0a 55 70 67 72
    61 64 65 3a 20 57 65 62  53 6f 63 6b 65 74 0d 0a
    43 6f 6e 6e 65 63 74 69  6f 6e 3a 20 55 70 67 72
    61 64 65 0d 0a
    ```

    Note: The string "HTTP/1.1 101 Web Socket Protocol Handshake", CRLF, the string "Upgrade: WebSocket", CRLF, the string "Connection: Upgrade", CRLF.

    What if the response is a 401 asking for credentials?

17. Let *headers* be a list of name-value pairs, initially empty.

18. *Header*: Let *name* and *value* be empty byte arrays.

19. Read a byte from the server.

    If the connection closes before this byte is received, then fail the Web Socket connection and abort these steps.

    Otherwise, handle the byte as described in the appropriate entry below:

    ↪ **If the byte is 0x0d (ASCII CR)**

    If the *name* byte array is empty, then jump to the headers processing step. Otherwise, fail the Web Socket connection and abort these steps.

    ↪ **If the byte is 0x0a (ASCII LF)**

    Fail the Web Socket connection and abort these steps.

    ↪ **If the byte is 0x3a (ASCII ":")**

    Move on to the next step.

    ↪ **If the byte is in the range 0x41 .. 0x5a (ASCII "A" .. "Z")**

    Append a byte whose value is the byte's value plus 0x20 to the *name* byte array and redo this step for the next byte.

    ↪ **Otherwise**

    Append the byte to the *name* byte array and redo this step for the next byte.

    Note: This reads a header name, terminated by a colon, converting upper-case ASCII

letters to lowercase, and aborting if a stray CR or LF is found.

20. Read a byte from the server.

    If the connection closes before this byte is received, then fail the Web Socket connection and abort these steps.

    Otherwise, handle the byte as described in the appropriate entry below:

    ↪ **If the byte is 0x20 (ASCII space)**

    Ignore the byte and move on to the next step.

    ↪ **Otherwise**

    Treat the byte as described by the list in the next step, then move on to that next step for real.

    Note: This skips past a space character after the colon, if necessary.

21. Read a byte from the server.

    If the connection closes before this byte is received, then fail the Web Socket connection and abort these steps.

    Otherwise, handle the byte as described in the appropriate entry below:

    ↪ **If the byte is 0x0d (ASCII CR)**

    Move on to the next step.

    ↪ **If the byte is 0x0a (ASCII LF)**

    Fail the Web Socket connection and abort these steps.

    ↪ **Otherwise**

    Append the byte to the *name* byte array and redo this step for the next byte.

    Note: This reads a header value, terminated by a CRLF.

22. Read a byte from the server.

    If the connection closes before this byte is received, or if the byte is not a 0x0a byte (ASCII LF), then fail the Web Socket connection and abort these steps.

    Note: This skips past the LF byte of the CRLF after the header.

23. Append an entry to the *headers* list that has the name given by the string obtained by interpreting the *name* byte array as a UTF-8 byte stream and the value given by the string obtained by interpreting the *value* byte array as a UTF-8 byte stream.

24. Return to the header step above.

25. *Headers processing*: If there is not exactly one entry in the *headers* list whose name is "`websocket-origin`", or if there is not exactly one entry in the *headers* list whose name is "`websocket-location`", or if there are any entries in the *headers* list whose names are the empty string, then fail

the Web Socket connection and abort these steps.

26. Handle each entry in the *headers* list as follows:

    ↪ **If the entry's name is "`websocket-origin`"**

    If the value is not exactly equal to the ASCII serialization of the origin of the script that invoked the `WebSocket()` constructor, then fail the Web Socket connection and abort these steps.

    ↪ **If the entry's name is "`websocket-location`"**

    If the value is not exactly equal to the absolute URL that resulted from the first step of ths algorithm, then fail the Web Socket connection and abort these steps.

    ↪ **If the entry's name is "`set-cookie`" or "`set-cookie2`" or another cookie-related header name**

    Handle the cookie as defined by the appropriate spec, except pretend that the resource's URL actually has a scheme of `http` if *secure* is false and `https` if *secure* is true and is otherwise identical to *url*. [RFC2109] [RFC2965]

    ↪ **Any other name**

    Ignore it.

27. Change the `readyState` attribute's value to `OPEN` (1).

28. Fire a simple event named `open` at the `WebSocket` object.

29. The **Web Socket connection is established**. Now the user agent must send and receive to and from the connection as described in the next section.

To **fail the Web Socket connection**, the user agent must close the Web Socket connection, and may report the problem to the user (which would be especially useful for developers). However, user agents must not convey the failure information to the script in a way distinguishable from the Web Socket being closed normally.

7.3.4.1.2. Data framing

Once a Web Socket connection is established, the user agent must run through the following state machine for the bytes sent by the server.

1. Try to read a byte from the server. Let *frame type* be that byte.

   If no byte could be read because the Web Socket connection is closed, then abort.

2. Handle the *frame type* byte as follows:

   If the high-order bit of the *frame type* byte is set (i.e. if *frame type and*ed with 0x80 returns 0x80)

   Run these steps. If at any point during these steps a read is attempted but fails because the Web Socket connection is closed, then abort.

   1. Let *length* be zero.

   2. *Length*: Read a byte, let *b* be that byte.

3. Let b_V be integer corresponding to the low 7 bits of *b* (the value you would get by *and*ing *b* with 0x7f).

4. Multiply *length* by 128, add b_V to that result, and store the final result in *length*.

5. If the high-order bit of *b* is set (i.e. if *b and*ed with 0x80 returns 0x80), then return to the step above labeled length.

6. Read *length* bytes.

7. Discard the read bytes.

If the high-order bit of the *frame type* byte is *not* set (i.e. if *frame type and*ed with 0x80 returns 0x00)

Run these steps. If at any point during these steps a read is attempted but fails because the Web Socket connection is closed, then abort.

1. Let *raw data* be an empty byte array.

2. *Data*: Read a byte, let *b* be that byte.

3. If *b* is not 0xff, then append *b* to *raw data* and return to the previous step (labeled data).

4. Interpret *raw data* as a UTF-8 string, and store that string in *data*.

5. If *frame type* is 0x00, create an event that uses the `MessageEvent` interface, with the event name `message`, which does not bubble, is cancelable, has no default action, and whose `data` attribute is set to *data*, and dispatch it at the `WebSocket` object. Otherwise, discard the data.

3. Return to the first step to read the next byte.

If the user agent is faced with content that is too large to be handled appropriately, then it must fail the Web Socket connection.

Once a Web Socket connection is established, the user agent must use the following steps to **send *data* using the Web Socket**:

1. Send a 0x00 byte to the server.

2. Encode *data* using UTF-8 and send the resulting byte stream to the server.

3. Send a 0xff byte to the server.

People often request the ability to send binary blobs over this API; also, once the other postMessage() methods support it, we should look into allowing name/value pairs, arrays, and numbers using postMessage() instead of just strings and binary data.

7.3.4.2. Server-side requirements

This section only applies to servers.

7.3.4.2.1. Minimal handshake

Note: This section describes the minimal requirements for a server-side implementation of Web Sockets.

Listen on a port for TCP/IP. Upon receiving a connection request, open a connection and send the following bytes back to the client:

```
48 54 54 50 2f 31 2e 31  20 31 30 31 20 57 65 62
20 53 6f 63 6b 65 74 20  50 72 6f 74 6f 63 6f 6c
20 48 61 6e 64 73 68 61  6b 65 0d 0a 55 70 67 72
61 64 65 3a 20 57 65 62  53 6f 63 6b 65 74 0d 0a
43 6f 6e 6e 65 63 74 69  6f 6e 3a 20 55 70 67 72
61 64 65 0d 0a
```

Send the string "`WebSocket-Origin`" followed by a U+003A COLON (":") followed by the ASCII serialization of the origin from which the server is willing to accept connections, followed by a CRLF pair (0x0d 0x0a).

> For instance:
>
> ```
> WebSocket-Origin: http://example.com
> ```

Send the string "`WebSocket-Location`" followed by a U+003A COLON (":") followed by the URL of the Web Socket script, followed by a CRLF pair (0x0d 0x0a).

> For instance:
>
> ```
> WebSocket-Location: ws://example.com:80/demo
> ```

Send another CRLF pair (0x0d 0x0a).

Read (and discard) data from the client until four bytes 0x0d 0x0a 0x0d 0x0a are read.

If the connection isn't dropped at this point, go to the data framing section.

7.3.4.2.2. Handshake details

The previous section ignores the data that is transmitted by the client during the handshake.

The data sent by the client consists of a number of fields separated by CR LF pairs (bytes 0x0d 0x0a).

The first field consists of three tokens separated by space characters (byte 0x20). The middle token is the path being opened. If the server supports multiple paths, then the server should echo the value of this field in the initial handshake, as part of the URL given on the `WebSocket-Location` line (after the appropriate scheme and host).

The remaining fields consist of name-value pairs, with the name part separated from the value part by a colon and a space (bytes 0x3a 0x20). Of these, several are interesting:

Host (bytes 48 6f 73 74)

The value gives the hostname that the client intended to use when opening the Web Socket. It would be of interest in particular to virtual hosting environments, where one server might serve multiple hosts, and might therefore want to return different data.

The right host has to be output as part of the URL given on the `WebSocket-Location` line of the handshake described above, to verify that the server knows that it is really representing that host.

Origin (bytes 4f 72 69 67 69 6e)

The value gives the scheme, hostname, and port (if it's not the default port for the given scheme) of the page that asked the client to open the Web Socket. It would be interesting if the server's operator had deals with operators of other sites, since the server could then decide how to respond (or indeed, *whether* to respond) based on which site was requesting a connection.

If the server supports connections from more than one origin, then the server should echo the value of this field in the initial handshake, on the `WebSocket-Origin` line.

Other fields

Other fields can be used, such as "`Cookie`" or "`Authorization`", for authentication purposes.

7.3.4.2.3. Data framing

Note: This section only describes how to handle content that this specification allows user agents to send (text). It doesn't handle any arbitrary content in the same way that the requirements on user agents defined earlier handle any content including possible future extensions to the protocols.

The server should run through the following steps to process the bytes sent by the client:

1. Read a byte from the client. Assuming everything is going according to plan, it will be a 0x00 byte. Behaviour for the server is undefined if the byte is not 0x00.
2. Let *raw data* be an empty byte array.
3. *Data*: Read a byte, let *b* be that byte.
4. If *b* is not 0xff, then append *b* to *raw data* and return to the previous step (labeled data).
5. Interpret *raw data* as a UTF-8 string, and apply whatever server-specific processing should occur for the resulting string.
6. Return to the first step to read the next byte.

The server should run through the followin steps to send strings to the client:

1. Send a 0x00 byte to the client to indicate the start of a string.
2. Encode *data* using UTF-8 and send the resulting byte stream to the client.
3. Send a 0xff byte to the client to indicate the end of the message.

7.3.4.3. Closing the connection

To **close the Web Socket connection**, either the user agent or the server closes the TCP/IP connection. There is no closing handshake. Whether the user agent or the server closes the connection, it is said that the

Web Socket connection is closed.

Servers may close the Web Socket connection whenever desired.

User agents should not close the Web Socket connection arbitrarily.

When the Web Socket connection is closed, the `readyState` attribute's value must be changed to `CLOSED` (2), and the user agent must fire a simple event named `close` at the `WebSocket` object.

7.4 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.

7.4.1 Introduction

This section is non-normative.

> For example, if document A contains an `iframe` element that contains document B, and script in document A calls `postMessage()` on the `Window` object of document B, then a message event will be fired on that object, marked as originating from the `Window` of document A. The script in document A might look like:
>
> ```
> var o = document.getElementsByTagName('iframe')[0];
> o.contentWindow.postMessage('Hello world', 'http://b.example.org/');
> ```
>
> To register an event handler for incoming events, the script would use `addEventListener()` (or similar mechanisms). For example, the script in document B might look like:
>
> ```
> window.addEventListener('message', receiver, false);
> function receiver(e) {
>   if (e.origin == 'http://example.com') {
>     if (e.data == 'Hello world') {
>       e.source.postMessage('Hello', e.origin);
>     } else {
>       alert(e.data);
>     }
>   }
> }
> ```
>
> This script first checks the domain is the expected domain, and then looks at the message, which it either displays to the user, or responds to by sending a message back to the document which sent the message in the first place.

7.4.2 Security

7.4.2.1. Authors

⚠Warning! ***Use of this API requires extra care to protect users from hostile entities abusing a site for their own purposes.***

Authors should check the `origin` attribute to ensure that messages are only accepted from domains that they expect to receive messages from. Otherwise, bugs in the author's message handling code could be exploited by hostile sites.

Authors should not use the wildcard keyword ("*") in the *targetOrigin* argument in messages that contain any confidential information, as otherwise there is no way to guarantee that the message is only delivered to the recipient to which it was intended.

7.4.2.2. User agents

The integrity of this API is based on the inability for scripts of one origin to post arbitrary events (using `dispatchEvent()` or otherwise) to objects in other origins (those that are not the same).

> ***Note: Implementors are urged to take extra care in the implementation of this feature. It allows authors to transmit information from one domain to another domain, which is normally disallowed for security reasons. It also requires that UAs be careful to allow access to certain properties but not others.***

7.4.3 Posting text

When a script invokes the **`postMessage(`*`message, targetOrigin`*`)`** method (with only two arguments) on a `Window` object, the user agent must follow these steps:

1. If the value of the *targetOrigin* argument is not a single U+002A ASTERISK character ("*"), and parsing it as a URL fails, then throw a `SYNTAX_ERR` exception and abort the overall set of steps.
2. Return from the `postMessage()` method, but asynchronously continue running these steps.
3. Wait for all scripts in the unit of related browsing contexts to which the `Window` object on which the method was invoked belongs to have finished executing any pending scripts.
4. If the *targetOrigin* argument has a value other than a single literal U+002A ASTERISK character ("*"), and the active document of the browsing context of the `Window` object on which the method was invoked does not have the same origin as *targetOrigin*, then abort these steps silently.
5. Create an event that uses the `MessageEvent` interface, with the event name `message`, which does not bubble, is cancelable, and has no default action. The `data` attribute must be set to the value passed as the *message* argument to the `postMessage()` method, the `origin` attribute must be set to the Unicode serialization of the origin of the script that invoked the method, and the `source` attribute must be set to the `Window` object of the default view of the browsing context for which the `Document` object with which the script is associated is the active document.
6. Dispatch the event created in the previous step at the `Window` object on which the method was

invoked.

7.4.4 Posting message ports

When a script invokes the **`postMessage(`*`message, messagePort, targetOrigin`*`)`** method (with three arguments) on a `Window` object, the user agent must follow these steps:

1. If the value of the *targetOrigin* argument is not a single U+002A ASTERISK character ("*"), and parsing it as a URL fails, then throw a `SYNTAX_ERR` exception and abort the overall set of steps.
2. If the *messagePort* argument is null, then act as if the method had just been called with two arguments, *message* and *targetOrigin*.
3. Try to obtain a *new port* by cloning the *messagePort* argument with the `Window` object on which the method was invoked as the owner of the clone. If this returns an exception, then throw that exception and abort these steps.
4. Return from the `postMessage()` method, but asynchronously continue running these steps.
5. Wait for all scripts in the unit of related browsing contexts to which the `Window` object on which the method was invoked belongs to have finished executing any pending scripts.
6. If the *targetOrigin* argument has a value other than a single literal U+002A ASTERISK character ("*"), and the active document of the browsing context of the `Window` object on which the method was invoked does not have the same origin as *targetOrigin*, then abort these steps silently.
7. Create an event that uses the `MessageEvent` interface, with the event name `message`, which does not bubble, is cancelable, and has no default action. The `data` attribute must be set to the value passed as the *message* argument to the `postMessage()` method, the `origin` attribute must be set to the Unicode serialization of the origin of the script that invoked the method, and the `source` attribute must be set to the `Window` object of the default view of the browsing context for which the `Document` object with which the script is associated is the active document.
8. Let the `messagePort` attribute of the event be the *new port*.
9. Dispatch the event created in the previous step at the `Window` object on which the method was invoked.

Note: These steps, with the exception of the second and third steps and the penultimate step, are identical to those in the previous section.

7.4.5 Posting structured data

People often request the ability to send name/value pairs, arrays, and numbers using postMessage() instead of just strings.

7.5 Channel messaging

7.5.1 Introduction

This section is non-normative.

An introduction to the channel and port APIs.

7.5.2 Message channels

```
[Constructor]
interface MessageChannel {
  readonly attribute MessagePort port1;
  readonly attribute MessagePort port2;
};
```

When the **`MessageChannel()`** constructor is called, it must run the following algorithm:

1. Create a new `MessagePort` object owned by the script browsing context, and let *port1* be that object.
2. Create a new `MessagePort` object owned by the script browsing context, and let *port2* be that object.
3. Entangle the *port1* and *port2* objects.
4. Set the `active` attribute of the two ports to true.
5. Instantiate a new `MessageChannel` object, and let *channel* be that object.
6. Let the `port1` attribute of the *channel* object be *port1*.
7. Let the `port2` attribute of the *channel* object be *port2*.
8. Return *channel*.

The **`port1`** and **`port2`** attributes must return the values they were assigned when the `MessageChannel` object was created.

7.5.3 Message ports

Each channel has two message ports. Data sent through one port is received by the other port, and vice versa.

```
interface MessagePort {
  readonly attribute Window ownerWindow;
  readonly attribute boolean active;
  boolean postMessage(in DOMString message);
  boolean postMessage(in DOMString message, in MessagePort messagePort);
  void close();

  // event handler attributes
           attribute EventListener onmessage;
```

```
            attribute EventListener onload;
            attribute EventListener onerror;
            attribute EventListener onunload;
};
```

When the user agent is to **create a new `MessagePort` object** owned by a `Window` object *owner*, it must run the following steps:

1. Instantiate a new `MessagePort` object, and let *port* be that object.
2. Set the `ownerWindow` attribute of *port* to *owner*.
3. Set the `active` attribute of *port* be false.

When the user agent is to **entangle** two `MessagePort` objects, it must run the following steps:

1. If either port is already entangled, then let *port1* be that port, and run these substeps:
    1. Let *old port* be the port that *port1* is entangled with.
    2. Unentangle *old port*, so that it is no longer entangled with any ports.
    3. Set the `active` attribute of *old port* to false.
    4. Unentangle *port1*, so that it is no longer entangled with any ports.

    If *port1* and *old port* were the two ports of a `MessageChannel` object, then that `MessageChannel` object no longer represents an actual channel: the two ports in that object are no longer entangled.
2. Associate the two ports to be entangled, so that they form the two parts of a new channel. (There is no `MessageChannel` object that represents this channel.)

When the user agent is to **clone a port** *original port*, with the clone being owned by *owner*, it must run the following steps, which return either a new `MessagePort` object or an exception for the caller to raise:

1. If the *original port* is not entangled without another port, then return an `INVALID_STATE_ERR` exception and abort all these steps.
2. Let the *remote port* be the port with which the *original port* is entangled.
3. Let *original status* be the value of the `active` attribute of the *original port*.
4. Create a new `MessagePort` object owned by *owner*, and let *new port* be that object.
5. Entangle the *remote port* and *new port* objects. The *original port* object will have its `active` attribute permanently set to false by this process.
6. Let the `active` attribute of the *new port* object have the same value as *original status*.
7. Return *new port*. It is the clone.

The **`ownerWindow`** attribute must return the value it was assigned when the `MessagePort` object was

created.

The **`active`** attribute must return the last value that it was set to according to the rules of this specification.

The **`postMessage()`** method, when called on a port *source port*, must cause the user agent to run the following steps:

1. Let *message* be the method's first argument.
2. Let *data port* be the method's second argument, if any.
3. If the *source port*'s `active` attribute is false, then return false and abort these steps.
4. Let *target port* be the port with which *source port* is entangled.
5. Create an event that uses the `MessageEvent` interface, with the name `message`, which does not bubble, is cancelable, and has no default action.
6. Let the `data` attribute of the event have the value of *message*, the method's first argument.
7. If the method was called with a second argument *data port* and that argument isn't null, then run the following substeps:
    1. If the *data port* is the *source port* or the *target port*, then throw an `INVALID_ACCESS_ERR` exception and abort all these steps.
    2. Try to obtain a *new data port* by cloning the *data port* with the owner of the *target port* as the owner of the clone. If this returns an exception, then throw that exception and abort these steps.
    3. Let the `messagePort` attribute of the event be the *new data port*.
8. Return true from the method, but continue with these steps.
9. Wait for all scripts in the unit of related browsing contexts to which the owner of the *target port* belongs to have executed to completion, and then dispatch the event at the *target port* object. If this never happens (e.g. the relevant browsing context is closed by the user before the event can be dispatched), then discard the event.

People often request the ability to send name/value pairs, arrays, and numbers using postMessage() instead of just strings.

The **`close()`** method, when called on a port *local port* that is entangled with another port, must cause the user agents to run the following steps:

1. Unentangle the two ports.
2. Set both ports' `active` attribute to false.
3. At the next available opportunity, after any scripts have finished executing, fire a simple event called `unload` at each of the message ports. If the two message ports are in the same unit of related browsing contexts, then the port on which the method was called must receive the event first.

If the method is called on a port that is not entangled, then the method must do nothing.

The following are the event handler DOM attributes that must be supported by objects implementing the `MessagePort` interface:

onmessage

Must be invoked whenever a `message` event is targeted at or bubbles through the `MessagePort` object.

onload

Must be invoked whenever a `load` event is targeted at or bubbles through the `MessagePort` object.

onerror

Must be invoked whenever an `error` event is targeted at or bubbles through the `MessagePort` object.

onunload

Must be invoked whenever an `unload` event is targeted at or bubbles through the `MessagePort` object.

Note: Nothing in this specification causes any `load` or `error` events to be targetted at a `MessagePort` object. Those features are intended for use with Workers. [WORKERS]

7.5.3.1. Ports and browsing contexts

During session history traversal (e.g. when a user navigates a browsing context to another page, or goes back to a previous page), the user agent will have to deactivate and/or reactivate some ports.

Ports are deactivated and unentangled when the `Document` that was the active document of the browsing context corresponding to the `Window` object that owns them is discarded.

To **deactivate a port** *local port* that is entangled with a second port *remote port*, the user agent must set the `active` attribute of the *remote port* to false.

To **reactivate a port** *local port* that is entangled with a second port *remote port*, the user agent must set the `active` attribute of the *remote port* to true.

Note: It's important to note that activating or deactivating a port actually changes the `active` attribute of the port's entangled port, not its own attribute.

7.5.3.2. Ports and garbage collection

User agents must act as if `MessagePort` objects have a strong reference to their entangled `MessagePort` object so long as that other `MessagePort` object has any `message` or `unload` event listeners registered.

Thus, a message port can be received, given an event listener, and then forgotten, and so long as that event listener could receive a message, the channel will be maintained.

Of course, if this was to occur on both sides of the channel, then both ports would be garbage

collected, since they would not be reachable from live code, despite having a strong reference to each other.

When an entangled message port is about to be garbage collected, it must be deactivated and unentangled, so that the two ports are no longer related and so that the other port's `active` attribute is set to false.

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. Optionally, a single U+FEFF BYTE ORDER MARK (BOM) character.
2. Any number of comments and space characters.
3. A DOCTYPE.
4. Any number of comments and space characters.
5. The root element, in the form of an `html` element.
6. Any number of comments and space characters.

The various types of content mentioned above are described in the next few sections.

In addition, there are some restrictions on how character encoding declarations are to be serialized, as discussed in the section on that topic.

> ***Space characters before the root `html` element, and space characters at the start of the `html` element and before the `head` element, will be dropped when the document is parsed; space characters after the root `html` element will be parsed as if they were at the end of the `body` element. Thus, space characters around the root element do not round-trip.***
>
> ***It is suggested that newlines be inserted after the DOCTYPE, after any comments that are before the root element, after the `html` element's start tag (if it is not omitted), and after any comments that are inside the `html` element but before the `head` element.***

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 five different kinds of **elements**: void elements, CDATA elements, RCDATA elements, foreign elements, and normal elements.

Void elements

`base`, `command`, `eventsource`, `link`, `meta`, `hr`, `br`, `img`, `embed`, `param`, `area`, `col`, `input`, `source`

CDATA elements

`style`, `script`

RCDATA elements

`title`, `textarea`

Foreign elements

Elements from the MathML namespace

Normal elements

All other allowed HTML elements are normal elements.

Tags are used to delimit the start and end of elements in the markup. CDATA, RCDATA, and normal elements have a start tag to indicate where they begin, and an end tag to indicate where they end. The start and end tags of certain normal elements can be omitted, as described later. Those that cannot be omitted must not be omitted. Void elements only have a start tag; end tags must not be specified for void elements. Foreign elements must either have a start tag and an end tag, or a start tag that is marked as self-closing, in which case they must not have an end tag.

The contents of the element must be placed between just after the start tag (which might be implied, in certain cases) and just before the end tag (which again, might be implied in certain cases). The exact allowed contents of each individual element depends on the content model of that element, as described earlier in this specification. Elements must not contain content that their content model disallows. In addition to the restrictions placed on the contents by those content models, however, the five types of elements have additional *syntactic* requirements.

Void elements can't have any contents (since there's no end tag, no content can be put between the start tag and the end tag).

CDATA elements can have text, though it has restrictions described below.

RCDATA elements can have text and character references, but the text must not contain an ambiguous ampersand. There are also further restrictions described below.

Foreign elements whose start tag is marked as self-closing can't have any contents (since, again, as there's no end tag, no content can be put between the start tag and the end tag). Foreign elements whose start tag is *not* marked as self-closing can have text, character references, CDATA sections, other elements, and comments, but the text must not contain the character U+003C LESS-THAN SIGN (<) or an ambiguous ampersand.

Normal elements can have text, character references, other elements, and comments, but the text must not contain the character U+003C LESS-THAN SIGN (<) or an ambiguous ampersand. Some normal elements also have yet more restrictions on what content they are allowed to hold, beyond the restrictions imposed by the content model and those described in this paragraph. Those restrictions are described below.

Tags contain a **tag name**, giving the element's name. HTML elements all have names that only use characters in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A .. U+007A LATIN SMALL LETTER Z, U+0041 LATIN CAPITAL LETTER A .. U+005A LATIN CAPITAL LETTER Z, and U+002D HYPHEN-MINUS (-). In the HTML syntax, tag names may be written with any mix of lower- and uppercase letters that, when converted to all-lowercase, matches the element's tag name; tag names are case-insensitive.

8.1.2.1. Start tags

Start tags must have the following format:

1. The first character of a start tag must be a U+003C LESS-THAN SIGN (<).
2. The next few characters of a start tag must be the element's tag name.
3. If there are to be any attributes in the next step, there must first be one or more space characters.
4. Then, the start tag may have a number of attributes, the syntax for which is described below. Attributes may be separated from each other by one or more space characters.
5. After the attributes, there may be one or more space characters. (Some attributes are required to be followed by a space. See the attributes section below.)
6. Then, if the element is one of the void elements, or if the element is a foreign element, then there may be a single U+002F SOLIDUS (/) character. This character has no effect on void elements, but on foreign elements it marks the start tag as self-closing.
7. Finally, start tags must be closed by a U+003E GREATER-THAN SIGN (>) character.

8.1.2.2. End tags

End tags must have the following format:

1. The first character of an end tag must be a U+003C LESS-THAN SIGN (<).
2. The second character of an end tag must be a U+002F SOLIDUS (/).

3. The next few characters of an end tag must be the element's tag name.

4. After the tag name, there may be one or more space characters.

5. Finally, end tags must be closed by a U+003E GREATER-THAN SIGN (>) character.

8.1.2.3. Attributes

Attributes for an element are expressed inside the element's start tag.

Attributes have a name and a value. **Attribute names** must consist of one or more characters other than the space characters, U+0000 NULL, U+0022 QUOTATION MARK ("), U+0027 APOSTROPHE ('), U+003E GREATER-THAN SIGN (>), U+002F SOLIDUS (/), and U+003D EQUALS SIGN (=) characters, the control characters, and any characters that are not defined by Unicode. In the HTML syntax, attribute names may be written with any mix of lower- and uppercase letters that, when converted to all-lowercase, matches the attribute's name; attribute names are case-insensitive.

Attribute values are a mixture of text and character references, except with the additional restriction that the text cannot contain an ambiguous ampersand.

Attributes can be specified in four different ways:

Empty attribute syntax

Just the attribute name.

> In the following example, the `disabled` attribute is given with the empty attribute syntax:
>
> ```
> <input disabled>
> ```

If an attribute using the empty attribute syntax is to be followed by another attribute, then there must be a space character separating the two.

Unquoted attribute value syntax

The attribute name, followed by zero or more space characters, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters, followed by the attribute value, which, in addition to the requirements given above for attribute values, must not contain any literal space characters, a U+0022 QUOTATION MARK (") characters, U+0027 APOSTROPHE (') characters, U+003D EQUALS SIGN (=) characters, or U+003E GREATER-THAN SIGN (>) characters.

> In the following example, the `value` attribute is given with the unquoted attribute value syntax:
>
> ```
> <input value=yes>
> ```

If an attribute using the unquoted attribute syntax is to be followed by another attribute or by one of the optional U+002F SOLIDUS (/) characters allowed in step 6 of the start tag syntax above, then there must be a space character separating the two.

Single-quoted attribute value syntax

The attribute name, followed by zero or more space characters, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters, followed by a single U+0027 APOSTROPHE (') character, followed by the attribute value, which, in addition to the requirements given above for attribute values, must not contain any literal U+0027 APOSTROPHE (') characters,

and finally followed by a second single U+0027 APOSTROPHE (') character.

> In the following example, the `type` attribute is given with the single-quoted attribute value syntax:
>
> ```
> <input type='checkbox'>
> ```

If an attribute using the single-quoted attribute syntax is to be followed by another attribute, then there must be a space character separating the two.

Double-quoted attribute value syntax

The attribute name, followed by zero or more space characters, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters, followed by a single U+0022 QUOTATION MARK (") character, followed by the attribute value, which, in addition to the requirements given above for attribute values, must not contain any literal U+0022 QUOTATION MARK (") characters, and finally followed by a second single U+0022 QUOTATION MARK (") character.

> In the following example, the `name` attribute is given with the double-quoted attribute value syntax:
>
> ```
> <input name="be evil">
> ```

If an attribute using the double-quoted attribute syntax is to be followed by another attribute, then there must be a space character separating the two.

8.1.2.4. Optional tags

Certain tags can be **omitted**.

An `html` element's start tag may be omitted if the first thing inside the `html` element is not a comment.

An `html` element's end tag may be omitted if the `html` element is not immediately followed by a comment and the element contains a `body` element that is either not empty or whose start tag has not been omitted.

A `head` element's start tag may be omitted if the first thing inside the `head` element is an element.

A `head` element's end tag may be omitted if the `head` element is not immediately followed by a space character or a comment.

A `body` element's start tag may be omitted if the first thing inside the `body` element is not a space character or a comment, except if the first thing inside the `body` element is a `script` or `style` element.

A `body` element's end tag may be omitted if the `body` element is not immediately followed by a comment and the element is either not empty or its start tag has not been omitted.

A `li` element's end tag may be omitted if the `li` element is immediately followed by another `li` element or if there is no more content in the parent element.

A `dt` element's end tag may be omitted if the `dt` element is immediately followed by another `dt` element or a `dd` element.

A `dd` element's end tag may be omitted if the `dd` element is immediately followed by another `dd` element or a `dt` element, or if there is no more content in the parent element.

A `p` element's end tag may be omitted if the `p` element is immediately followed by an `address`, `article`, `aside`, `blockquote`, `datagrid`, `dialog`, `dir`, `div`, `dl`, `fieldset`, `footer`, `form`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `header`, `hr`, `menu`, `nav`, `ol`, `p`, `pre`, `section`, `table`, or `ul`, element, or if there is no more content in the parent element.

An `rt` element's end tag may be omitted if the `rt` element is immediately followed by an `rt` or `rp` element, or if there is no more content in the parent element.

An `rp` element's end tag may be omitted if the `rp` element is immediately followed by an `rt` or `rp` element, or if there is no more content in the parent element.

An `optgroup` element's end tag may be omitted if the `optgroup` element is immediately followed by another `optgroup` element, or if there is no more content in the parent element.

An `option` element's end tag may be omitted if the `option` element is immediately followed by another `option` element, or if there is no more content in the parent element.

A `colgroup` element's start tag may be omitted if the first thing inside the `colgroup` element is a `col` element, and if the element is not immediately preceded by another `colgroup` element whose end tag has been omitted.

A `colgroup` element's end tag may be omitted if the `colgroup` element is not immediately followed by a space character or a comment.

A `thead` element's end tag may be omitted if the `thead` element is immediately followed by a `tbody` or `tfoot` element.

A `tbody` element's start tag may be omitted if the first thing inside the `tbody` element is a `tr` element, and if the element is not immediately preceded by a `tbody`, `thead`, or `tfoot` element whose end tag has been omitted.

A `tbody` element's end tag may be omitted if the `tbody` element is immediately followed by a `tbody` or `tfoot` element, or if there is no more content in the parent element.

A `tfoot` element's end tag may be omitted if the `tfoot` element is immediately followed by a `tbody` element, or if there is no more content in the parent element.

A `tr` element's end tag may be omitted if the `tr` element is immediately followed by another `tr` element, or if there is no more content in the parent element.

A `td` element's end tag may be omitted if the `td` element is immediately followed by a `td` or `th` element, or if there is no more content in the parent element.

A `th` element's end tag may be omitted if the `th` element is immediately followed by a `td` or `th` element, or if there is no more content in the parent element.

However, a start tag must never be omitted if it has any attributes.

8.1.2.5. Restrictions on content models

For historical reasons, certain elements have extra restrictions beyond even the restrictions given by their content model.

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.)

A single U+000A LINE FEED (LF) character may be placed immediately after the start tag of `pre` and `textarea` elements. This does not affect the processing of the element. The otherwise optional U+000A LINE FEED (LF) character *must* be included if the element's contents start with that character (because otherwise the leading newline in the contents would be treated like the optional newline, and ignored).

> The following two `pre` blocks are equivalent:
>
> ```
> <pre>Hello</pre>
>
> <pre>
> Hello</pre>
> ```

8.1.2.6. Restrictions on the contents of CDATA and RCDATA elements

The text in CDATA and RCDATA elements must not contain any occurrences of the string "`</`" (U+003C LESS-THAN SIGN, U+002F SOLIDUS) followed by characters that case-insensitively match the tag name of the element followed by one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), U+0020 SPACE, U+003E GREATER-THAN SIGN (>), or U+002F SOLIDUS (/), unless that string is part of an escaping text span.

An **escaping text span** is a span of text that starts with an escaping text span start that is not itself in an escaping text span, and ends at the next escaping text span end. There cannot be any character references inside an escaping text span.

An **escaping text span start** is a part of text that consists of the four character sequence "`<!--`" (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS).

An **escaping text span end** is a part of text that consists of the three character sequence "`-->`" (U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN) whose U+003E GREATER-THAN SIGN (>).

An escaping text span start may share its U+002D HYPHEN-MINUS characters with its corresponding escaping text span end.

The text in CDATA and RCDATA elements must not have an escaping text span start that is not followed by an escaping text span end.

8.1.3 Text

Text is allowed inside elements, attributes, and comments. Text must consist of Unicode characters. Text must not contain U+0000 characters. Text must not contain permanently undefined Unicode characters. Text must not contain control characters other than space characters. Extra constraints are placed on what is and what is not allowed in text based on where the text is to be put, as described in the other sections.

8.1.3.1. Newlines

Newlines in HTML may be represented either as U+000D CARRIAGE RETURN (CR) characters, U+000A LINE FEED (LF) characters, or pairs of U+000D CARRIAGE RETURN (CR), U+000A LINE FEED (LF) characters in that order.

8.1.4 Character references

In certain cases described in other sections, text may be mixed with **character references**. These can be used to escape characters that couldn't otherwise legally be included in text.

Character references must start with a U+0026 AMPERSAND (`&`). Following this, there are three possible kinds of character references:

Named character references

The ampersand must be followed by one of the names given in the named character references section, using the same case. The name must be one that is terminated by a U+003B SEMICOLON (`;`) character.

Decimal numeric character reference

The ampersand must be followed by a U+0023 NUMBER SIGN (`#`) character, followed by one or more digits in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, representing a base-ten integer that itself is a Unicode code point that is not U+0000, U+000D, in the range U+0080 .. U+009F, or in the range 0xD800 .. 0xDFFF (surrogates). The digits must then be followed by a U+003B SEMICOLON character (`;`).

Hexadecimal numeric character reference

The ampersand must be followed by a U+0023 NUMBER SIGN (`#`) character, which must be followed by either a U+0078 LATIN SMALL LETTER X or a U+0058 LATIN CAPITAL LETTER X character, which must then be followed by one or more digits in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F, and U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F, representing a base-sixteen integer that itself is a Unicode code point that is not U+0000, U+000D, in the range U+0080 .. U+009F, or in the range 0xD800 .. 0xDFFF (surrogates). The digits must then be followed by a U+003B SEMICOLON character (`;`).

An **ambiguous ampersand** is a U+0026 AMPERSAND (`&`) character that is followed by some text other than a space character, a U+003C LESS-THAN SIGN character ('<'), or another U+0026 AMPERSAND (`&`) character.

8.1.5 CDATA sections

CDATA sections must start with the character sequence U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+005B LEFT SQUARE BRACKET, U+0043 LATIN CAPITAL LETTER C, U+0044 LATIN CAPITAL LETTER D, U+0041 LATIN CAPITAL LETTER A, U+0054 LATIN CAPITAL LETTER T, U+0041 LATIN CAPITAL LETTER A, U+005B LEFT SQUARE BRACKET (`<![CDATA[`). Following this sequence, the CDATA section may have text, with the additional restriction that the text must not contain the three character sequence U+005D RIGHT SQUARE BRACKET, U+005D RIGHT SQUARE BRACKET, U+003E GREATER-THAN SIGN (`]]>`). Finally, the CDATA section must be ended by the three character

sequence U+005D RIGHT SQUARE BRACKET, U+005D RIGHT SQUARE BRACKET, U+003E GREATER-THAN SIGN (]]>).

8.1.6 Comments

Comments must start with the four character sequence U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS (<!--). Following this sequence, the comment may have text, with the additional restriction that the text must not start with a single U+003E GREATER-THAN SIGN ('>') character, nor start with a U+002D HYPHEN-MINUS (-) character followed by a U+003E GREATER-THAN SIGN ('>') character, nor contain two consecutive U+002D HYPHEN-MINUS (-) characters, nor end with a U+002D HYPHEN-MINUS (-) character. Finally, the comment must be ended by the three character sequence U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN (-->).

8.2 Parsing HTML documents

This section only applies to user agents, data mining tools, and conformance checkers.

The rules for parsing XML documents (and thus XHTML documents) into DOM trees are covered by the XML and Namespaces in XML specifications, and are out of scope of this specification. [XML] [XMLNS]

For HTML documents, user agents must use the parsing rules described in this section to generate the DOM trees. Together, these rules define what is referred to as the **HTML parser**.

> ***While the HTML form of HTML5 bears a close resemblance to SGML and XML, it is a separate language with its own parsing rules.***
>
> ***Some earlier versions of HTML (in particular from HTML2 to HTML4) were based on SGML and used SGML parsing rules. However, few (if any) web browsers ever implemented true SGML parsing for HTML documents; the only user agents to strictly handle HTML as an SGML application have historically been validators. The resulting confusion — with validators claiming documents to have one representation while widely deployed Web browsers interoperably implemented a different representation — has wasted decades of productivity. This version of HTML thus returns to a non-SGML basis.***
>
> ***Authors interested in using SGML tools in their authoring pipeline are encouraged to use XML tools and the XML serialization of HTML5.***

This specification defines the parsing rules for HTML documents, whether they are syntactically correct or not. Certain points in the parsing algorithm are said to be **parse errors**. The error handling for parse errors is well-defined: user agents must either act as described below when encountering such problems, or must abort processing at the first error that they encounter for which they do not wish to apply the rules described below.

Conformance checkers must report at least one parse error condition to the user if one or more parse error conditions exist in the document and must not report parse error conditions if none exist in the document. Conformance checkers may report more than one parse error condition if more than one parse error conditions exist in the document. Conformance checkers are not required to recover from parse errors.

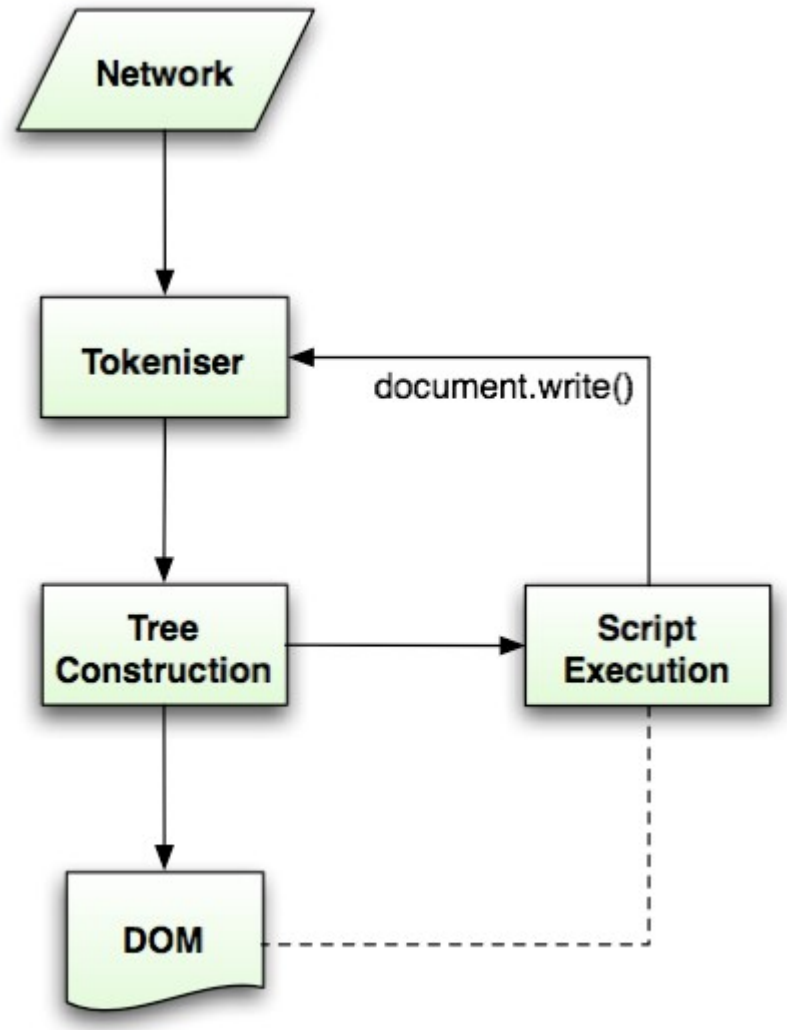
Note: Parse errors are only errors with the syntax of HTML. In addition to checking for parse errors, conformance checkers will also verify that the document obeys all the other conformance requirements described in this specification.

8.2.1 Overview of the parsing model

The input to the HTML parsing process consists of a stream of Unicode characters, which is passed through a tokenization stage (lexical analysis) followed by a tree construction stage (semantic analysis). The output is a `Document` object.

Note: Implementations that do not support scripting do not have to actually create a DOM `Document` object, but the DOM tree in such cases is still used as the model for the rest of the specification.

In the common case, the data handled by the tokenization stage comes from the network, but it can also come from script, e.g. using the `document.write()` API.



There is only one set of state for the tokeniser stage and the tree construction stage, but the tree construction stage is reentrant, meaning that while the tree construction stage is handling one token, the tokeniser might be resumed, causing further tokens to be emitted and processed before the first token's processing is complete.

> In the following example, the tree construction stage will be called upon to handle a "p" start tag token while handling the "script" start tag token:
>
> ```
> ...
> <script>
>  document.write('<p>');
> </script>
> ...
> ```

8.2.2 The input stream

The stream of Unicode characters that consists the input to the tokenization stage will be initially seen by the user agent as a stream of bytes (typically coming over the network or from the local file system). The bytes encode the actual characters according to a particular *character encoding*, which the user agent must use to decode the bytes into characters.

> ***Note: For XML documents, the algorithm user agents must use to determine the character encoding is given by the XML specification. This section does not apply to XML documents. [XML]***

8.2.2.1. Determining the character encoding

In some cases, it might be impractical to unambiguously determine the encoding before parsing the document. Because of this, this specification provides for a two-pass mechanism with an optional pre-scan. Implementations are allowed, as described below, to apply a simplified parsing algorithm to whatever bytes they have available before beginning to parse the document. Then, the real parser is started, using a tentative encoding derived from this pre-parse and other out-of-band metadata. If, while the document is being loaded, the user agent discovers an encoding declaration that conflicts with this information, then the parser can get reinvoked to perform a parse of the document with the real encoding.

User agents must use the following algorithm (the **encoding sniffing algorithm**) to determine the character encoding to use when decoding a document in the first pass. This algorithm takes as input any out-of-band metadata available to the user agent (e.g. the Content-Type metadata of the document) and all the bytes available so far, and returns an encoding and a **confidence**. The confidence is either *tentative* or *certain*. The encoding used, and whether the confidence in that encoding is *tentative* or *confident*, is used during the parsing to determine whether to change the encoding.

1. If the transport layer specifies an encoding, return that encoding with the confidence *certain*, and abort these steps.

2. The user agent may wait for more bytes of the resource to be available, either in this step or at any later step in this algorithm. For instance, a user agent might wait 500ms or 512 bytes, whichever came first. In general preparsing the source to find the encoding improves performance, as it reduces the need to throw away the data structures used when parsing upon finding the encoding information. However, if the user agent delays too long to obtain data to determine the encoding, then the cost of the delay

could outweigh any performance improvements from the preparse.

3. For each of the rows in the following table, starting with the first one and going down, if there are as many or more bytes available than the number of bytes in the first column, and the first bytes of the file match the bytes given in the first column, then return the encoding given in the cell in the second column of that row, with the confidence *certain*, and abort these steps:

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

Note: This step looks for Unicode Byte Order Marks (BOMs).

4. Otherwise, the user agent will have to search for explicit character encoding information in the file itself. This should proceed as follows:

   Let *position* be a pointer to a byte in the input stream, initially pointing at the first byte. If at any point during these substeps the user agent either runs out of bytes or decides that scanning further bytes would not be efficient, then skip to the next step of the overall character encoding detection algorithm. User agents may decide that scanning *any* bytes is not efficient, in which case these substeps are entirely skipped.

   Now, repeat the following "two" steps until the algorithm aborts (either because user agent aborts, as described above, or because a character encoding is found):

   1. If *position* points to:

      ↪ **A sequence of bytes starting with: 0x3C 0x21 0x2D 0x2D (ASCII '<!--')**

      Advance the *position* pointer so that it points at the first 0x3E byte which is preceded by two 0x2D bytes (i.e. at the end of an ASCII '-->' sequence) and comes after the 0x3C byte that was found. (The two 0x2D bytes can be the same as the those in the '<!--' sequence.)

      ↪ **A sequence of bytes starting with: 0x3C, 0x4D or 0x6D, 0x45 or 0x65, 0x54 or 0x74, 0x41 or 0x61, and finally one of 0x09, 0x0A, 0x0C, 0x0D, 0x20, 0x2F (case-insensitive ASCII '<meta' followed by a space or slash)**

      1. Advance the *position* pointer so that it points at the next 0x09, 0x0A, 0x0C, 0x0D, 0x20, or 0x2F byte (the one in sequence of characters matched above).

      2. Get an attribute and its value. If no attribute was sniffed, then skip this inner set of steps, and jump to the second step in the overall "two step" algorithm.

      3. If the attribute's name is neither "`charset`" nor "`content`", then return to step 2 in these inner steps.

      4. If the attribute's name is "`charset`", let *charset* be the attribute's value, interpreted as a character encoding.

      5. Otherwise, the attribute's name is "`content`": apply the algorithm for extracting an encoding from a Content-Type, giving the attribute's value as the

string to parse. If an encoding is returned, let *charset* be that encoding. Otherwise, return to step 2 in these inner steps.

6. If *charset* is a UTF-16 encoding, change it to UTF-8.

7. If *charset* is a supported character encoding, then return the given encoding, with confidence *tentative*, and abort all these steps.

8. Otherwise, return to step 2 in these inner steps.

↪ **A sequence of bytes starting with a 0x3C byte (ASCII '<'), optionally a 0x2F byte (ASCII '/'), and finally a byte in the range 0x41-0x5A or 0x61-0x7A (an ASCII letter)**

1. Advance the *position* pointer so that it points at the next 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), or 0x3E (ASCII '>') byte.

2. Repeatedly get an attribute until no further attributes can be found, then jump to the second step in the overall "two step" algorithm.

↪ **A sequence of bytes starting with: 0x3C 0x21 (ASCII '<!')**

↪ **A sequence of bytes starting with: 0x3C 0x2F (ASCII '</')**

↪ **A sequence of bytes starting with: 0x3C 0x3F (ASCII '<?')**

Advance the *position* pointer so that it points at the first 0x3E byte (ASCII '>') that comes after the 0x3C byte that was found.

↪ **Any other byte**

Do nothing with that byte.

2. Move *position* so it points at the next byte in the input stream, and return to the first step of this "two step" algorithm.

When the above "two step" algorithm says to **get an attribute**, it means doing this:

1. If the byte at *position* is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), or 0x2F (ASCII '/') then advance *position* to the next byte and redo this substep.

2. If the byte at *position* is 0x3E (ASCII '>'), then abort the "get an attribute" algorithm. There isn't one.

3. Otherwise, the byte at *position* is the start of the attribute name. Let *attribute name* and *attribute value* be the empty string.

4. *Attribute name*: Process the byte at *position* as follows:

↪ **If it is 0x3D (ASCII '='), and the *attribute name* is longer than the empty string**

Advance *position* to the next byte and jump to the step below labeled *value*.

↪ **If it is 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space)**

Jump to the step below labeled *spaces*.

- **If it is 0x2F (ASCII '/') or 0x3E (ASCII '>')**

  Abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name*, its value is the empty string.

- **If it is in the range 0x41 (ASCII 'A') to 0x5A (ASCII 'Z')**

  Append the Unicode character with codepoint *b*+0x20 to *attribute name* (where *b* is the value of the byte at *position*).

- **Anything else**

  Append the Unicode character with the same codepoint as the value of the byte at *position*) to *attribute name*. (It doesn't actually matter how bytes outside the ASCII range are handled here, since only ASCII characters can contribute to the detection of a character encoding.)

5. Advance *position* to the next byte and return to the previous step.

6. *Spaces.* If the byte at *position* is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space) then advance *position* to the next byte, then, repeat this step.

7. If the byte at *position* is *not* 0x3D (ASCII '='), abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name*, its value is the empty string.

8. Advance *position* past the 0x3D (ASCII '=') byte.

9. *Value.* If the byte at *position* is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space) then advance *position* to the next byte, then, repeat this step.

10. Process the byte at *position* as follows:

    - **If it is 0x22 (ASCII '"') or 0x27 ("'")**

      1. Let *b* be the value of the byte at *position*.

      2. Advance *position* to the next byte.

      3. If the value of the byte at *position* is the value of *b*, then advance *position* to the next byte and abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name*, and its value is the value of *attribute value*.

      4. Otherwise, if the value of the byte at *position* is in the range 0x41 (ASCII 'A') to 0x5A (ASCII 'Z'), then append a Unicode character to *attribute value* whose codepoint is 0x20 more than the value of the byte at *position*.

      5. Otherwise, append a Unicode character to *attribute value* whose codepoint is the same as the value of the byte at *position*.

      6. Return to the second step in these substeps.

    - **If it is 0x3E (ASCII '>')**

      Abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name*, its value is the empty string.

- **If it is in the range 0x41 (ASCII 'A') to 0x5A (ASCII 'Z')**

  Append the Unicode character with codepoint *b*+0x20 to *attribute value* (where *b* is the value of the byte at *position*). Advance *position* to the next byte.

- **Anything else**

  Append the Unicode character with the same codepoint as the value of the byte at *position*) to *attribute value*. Advance *position* to the next byte.

11. Process the byte at *position* as follows:

    - **If it is 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), or 0x3E (ASCII '>')**

      Abort the "get an attribute" algorithm. The attribute's name is the value of *attribute name* and its value is the value of *attribute value*.

    - **If it is in the range 0x41 (ASCII 'A') to 0x5A (ASCII 'Z')**

      Append the Unicode character with codepoint *b*+0x20 to *attribute value* (where *b* is the value of the byte at *position*).

    - **Anything else**

      Append the Unicode character with the same codepoint as the value of the byte at *position*) to *attribute value*.

12. Advance *position* to the next byte and return to the previous step.

For the sake of interoperability, user agents should not use a pre-scan algorithm that returns different results than the one described above. (But, if you do, please at least let us know, so that we can improve this algorithm and benefit everyone...)

5. If the user agent has information on the likely encoding for this page, e.g. based on the encoding of the page when it was last visited, then return that encoding, with the confidence *tentative*, and abort these steps.

6. The user agent may attempt to autodetect the character encoding from applying frequency analysis or other algorithms to the data stream. If autodetection succeeds in determining a character encoding, then return that encoding, with the confidence *tentative*, and abort these steps. [UNIVCHARDET]

7. Otherwise, return an implementation-defined or user-specified default character encoding, with the confidence *tentative*. In non-legacy environments, the more comprehensive `UTF-8` encoding is recommended. Due to its use in legacy content, `windows-1252` is recommended as a default in predominantly Western demographics instead. Since these encodings can in many cases be distinguished by inspection, a user agent may heuristically decide which to use as a default.

The document's character encoding must immediately be set to the value returned from this algorithm, at the same time as the user agent uses the returned value to select the decoder to use for the input stream.

8.2.2.2. Character encoding requirements

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

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

distinct character encodings.

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

When comparing a string specifying a character encoding with the name or alias of a character encoding to determine if they are equal, user agents must ignore the all characters in the ranges U+0009 to U+000D, U+0020 to U+002F, U+003A to U+0040, U+005B to U+0060, and U+007B to U+007E (all whitespace and punctuation characters in ASCII) in both names, and then perform the comparison case-insensitively.

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

When a user agent would otherwise use an encoding given in the first column of the following table, it must instead use the encoding given in the cell in the second column of the same row. Any bytes that are treated differently due to this encoding aliasing must be considered parse errors.

Character encoding overrides

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

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

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

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

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

8.2.2.3. Preprocessing the input stream

Given an encoding, the bytes in the input stream must be converted to Unicode characters for the tokeniser, as described by the rules for that encoding, except that the leading U+FEFF BYTE ORDER MARK character, if any, must not be stripped by the encoding layer (it is stripped by the rule below).

Bytes or sequences of bytes in the original byte stream that could not be converted to Unicode characters must be converted to U+FFFD REPLACEMENT CHARACTER code points.

Note: Bytes or sequences of bytes in the original byte stream that did not conform to the

encoding specification (e.g. invalid UTF-8 byte sequences in a UTF-8 input stream) are errors that conformance checkers are expected to report.

One leading U+FEFF BYTE ORDER MARK character must be ignored if any are present.

All U+0000 NULL characters in the input must be replaced by U+FFFD REPLACEMENT CHARACTERs. Any occurrences of such characters is a parse error.

Any occurrences of any characters in the ranges U+0001 to U+0008, U+000E to U+001F, U+007F to U+009F, U+D800 to U+DFFF , U+FDD0 to U+FDDF, and characters U+FFFE, U+FFFF, U+1FFFE, U+1FFFF, U+2FFFE, U+2FFFF, U+3FFFE, U+3FFFF, U+4FFFE, U+4FFFF, U+5FFFE, U+5FFFF, U+6FFFE, U+6FFFF, U+7FFFE, U+7FFFF, U+8FFFE, U+8FFFF, U+9FFFE, U+9FFFF, U+AFFFE, U+AFFFF, U+BFFFE, U+BFFFF, U+CFFFE, U+CFFFF, U+DFFFE, U+DFFFF, U+EFFFE, U+EFFFF, U+FFFFE, U+FFFFF, U+10FFFE, and U+10FFFF are parse errors. (These are all control characters or permanently undefined Unicode characters.)

U+000D CARRIAGE RETURN (CR) characters, and U+000A LINE FEED (LF) characters, are treated specially. Any CR characters that are followed by LF characters must be removed, and any CR characters not followed by LF characters must be converted to LF characters. Thus, newlines in HTML DOMs are represented by LF characters, and there are never any CR characters in the input to the tokenization stage.

The **next input character** is the first character in the input stream that has not yet been **consumed**. Initially, the *next input character* is the first character in the input.

The **insertion point** is the position (just before a character or just before the end of the input stream) where content inserted using `document.write()` is actually inserted. The insertion point is relative to the position of the character immediately after it, it is not an absolute offset into the input stream. Initially, the insertion point is uninitialized.

The "EOF" character in the tables below is a conceptual character representing the end of the input stream. If the parser is a script-created parser, then the end of the input stream is reached when an **explicit "EOF" character** (inserted by the `document.close()` method) is consumed. Otherwise, the "EOF" character is not a real character in the stream, but rather the lack of any further characters.

8.2.2.4. Changing the encoding while parsing

When the parser requires the user agent to **change the encoding**, it must run the following steps. This might happen if the encoding sniffing algorithm described above failed to find an encoding, or if it found an encoding that was not the actual encoding of the file.

1. If the new encoding is a UTF-16 encoding, change it to UTF-8.

2. If the new encoding is identical or equivalent to the encoding that is already being used to interpret the input stream, then set the confidence to *confident* and abort these steps. This happens when the encoding information found in the file matches what the encoding sniffing algorithm determined to be the encoding, and in the second pass through the parser if the first pass found that the encoding sniffing algorithm described in the earlier section failed to find the right encoding.

3. If all the bytes up to the last byte converted by the current decoder have the same Unicode interpretations in both the current encoding and the new encoding, and if the user agent supports changing the converter on the fly, then the user agent may change to the new converter for the encoding on the fly. Set the document's character encoding and the encoding used to convert the input

stream to the new encoding, set the confidence to *confident*, and abort these steps.

4. Otherwise, navigate to the document again, with replacement enabled, and using the same source browsing context, but this time skip the encoding sniffing algorithm and instead just set the encoding to the new encoding and the confidence to *confident*. Whenever possible, this should be done without actually contacting the network layer (the bytes should be re-parsed from memory), even if, e.g., the document is marked as not being cacheable.

8.2.3 Parse state

8.2.3.1. The insertion mode

Initially the insertion mode is "initial". It can change to "before html", "before head", "in head", "in head noscript", "after head", "in body", "in table", "in caption", "in column group", "in table body", "in row", "in cell", "in select", "in select in table", "in foreign content", "after body", "in frameset", "after frameset", "after after body", and "after after frameset" during the course of the parsing, as described in the tree construction stage. The insertion mode affects how tokens are processed and whether CDATA sections are supported.

Seven of these modes, namely "in head", "in body", "in table", "in table body", "in row", "in cell", and "in select", are special, in that the other modes defer to them at various times. When the algorithm below says that the user agent is to do something "**using the rules for** the *m* insertion mode", where *m* is one of these modes, the user agent must use the rules described under that insertion mode's section, but must leave the insertion mode unchanged (unless the rules in that section themselves switch the insertion mode).

When the insertion mode is switched to "in foreign content", the **secondary insertion mode** is also set. This secondary mode is used within the rules for the "in foreign content" mode to handle HTML (i.e. not foreign) content.

When the steps below require the UA to **reset the insertion mode appropriately**, it means the UA must follow these steps:

1. Let *last* be false.
2. Let *node* be the last node in the stack of open elements.
3. If *node* is the first node in the stack of open elements, then set *last* to true and set *node* to the *context* element. (fragment case)
4. If *node* is a `select` element, then switch the insertion mode to "in select" and abort these steps. (fragment case)
5. If *node* is a `td` or `th` element and *last* is false, then switch the insertion mode to "in cell" and abort these steps.
6. If *node* is a `tr` element, then switch the insertion mode to "in row" and abort these steps.
7. If *node* is a `tbody`, `thead`, or `tfoot` element, then switch the insertion mode to "in table body" and abort these steps.
8. If *node* is a `caption` element, then switch the insertion mode to "in caption" and abort these steps.
9. If *node* is a `colgroup` element, then switch the insertion mode to "in column group" and abort these steps. (fragment case)

10. If *node* is a `table` element, then switch the insertion mode to "in table" and abort these steps.
11. If *node* is an element from the MathML namespace, then switch the insertion mode to "in foreign content", let the secondary insertion mode be "in body", and abort these steps.
12. If *node* is a `head` element, then switch the insertion mode to "in body" ("in body"! *not "in head"*!) and abort these steps. (fragment case)
13. If *node* is a `body` element, then switch the insertion mode to "in body" and abort these steps.
14. If *node* is a `frameset` element, then switch the insertion mode to "in frameset" and abort these steps. (fragment case)
15. If *node* is an `html` element, then: if the `head` element pointer is null, switch the insertion mode to "before head", otherwise, switch the insertion mode to "after head". In either case, abort these steps. (fragment case)
16. If *last* is true, then switch the insertion mode to "in body" and abort these steps. (fragment case)
17. Let *node* now be the node before *node* in the stack of open elements.
18. Return to step 3.

8.2.3.2. The stack of open elements

Initially the **stack of open elements** is empty. The stack grows downwards; the topmost node on the stack is the first one added to the stack, and the bottommost node of the stack is the most recently added node in the stack (notwithstanding when the stack is manipulated in a random access fashion as part of the handling for misnested tags).

The "before html" insertion mode creates the `html` root element node, which is then added to the stack.

In the fragment case, the stack of open elements is initialized to contain an `html` element that is created as part of that algorithm. (The fragment case skips the "before html" insertion mode.)

The `html` node, however it is created, is the topmost node of the stack. It never gets popped off the stack.

The **current node** is the bottommost node in this stack.

The **current table** is the last `table` element in the stack of open elements, if there is one. If there is no `table` element in the stack of open elements (fragment case), then the current table is the first element in the stack of open elements (the `html` element).

Elements in the stack fall into the following categories:

Special

The following HTML elements have varying levels of special parsing rules: `address`, `area`, `article`, `aside`, `base`, `basefont`, `bgsound`, `blockquote`, `body`, `br`, `center`, `col`, `colgroup`, `command`, `datagrid`, `dd`, `details`, `dialog`, `dir`, `div`, `dl`, `dt`, `embed`, `eventsource` `fieldset`, `figure`, `footer`, `form`, `frame`, `frameset`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `head`, `header`, `hr`, `iframe`, `img`, `input`, `isindex`, `li`, `link`, `listing`, `menu`, `meta`, `nav`, `noembed`, `noframes`, `noscript`, `ol`, `optgroup`, `option`, `p`, `param`, `plaintext`, `pre`, `script`, `section`, `select`, `spacer`, `style`, `tbody`,

`textarea`, `tfoot`, `thead`, `title`, `tr`, `ul`, and `wbr`.

Scoping

The following HTML elements introduce new scopes for various parts of the parsing: `applet`, `button`, `caption`, `html`, `marquee`, `object`, `table`, `td`, `th`.

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: `eventsource`, `section`, `nav`, `article`, `aside`, `header`, `footer`, `datagrid`, `command`

The stack of open elements is said to **have an element in 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 one of the following elements, terminate in a failure state:
    - `applet` in the HTML namespace
    - `caption` in the HTML namespace
    - `html` in the HTML namespace
    - `table` in the HTML namespace
    - `td` in the HTML namespace
    - `th` in the HTML namespace
    - `button` in the HTML namespace
    - `marquee` in the HTML namespace
    - `object` in the HTML namespace
4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2. (This will never fail, since the loop will always terminate in the previous step if the top of the stack — an `html` element — is reached.)

The stack of open elements is said to **have an element in *table scope*** when the following algorithm terminates in a match state:

1. 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 one of the following elements, terminate in a failure state:
    - `html` in the HTML namespace
    - `table` in the HTML namespace
4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2. (This will never fail, since the loop will always terminate in the previous step if the top of the stack — an `html`

element — is reached.)

Nothing happens if at any time any of the elements in the stack of open elements are moved to a new location in, or removed from, the `Document` tree. In particular, the stack is not changed in this situation. This can cause, amongst other strange effects, content to be appended to nodes that are no longer in the DOM.

Note: In some cases (namely, when closing misnested formatting elements), the stack is manipulated in a random-access fashion.

8.2.3.3. The list of active formatting elements

Initially the **list of active formatting elements** is empty. It is used to handle mis-nested formatting element tags.

The list contains elements in the formatting category, and scope markers. The scope markers are inserted when entering `applet` elements, buttons, `object` elements, marquees, table cells, and table captions, and are used to prevent formatting from "leaking" into `applet` elements, buttons, `object` elements, marquees, and tables.

When the steps below require the UA to **reconstruct the active formatting elements**, the UA must perform the following steps:

1. If there are no entries in the list of active formatting elements, then there is nothing to reconstruct; stop this algorithm.
2. If the last (most recently added) entry in the list of active formatting elements is a marker, or if it is an element that is in the stack of open elements, then there is nothing to reconstruct; stop this algorithm.
3. Let *entry* be the last (most recently added) element in the list of active formatting elements.
4. If there are no entries before *entry* in the list of active formatting elements, then jump to step 8.
5. Let *entry* be the entry one earlier than *entry* in the list of active formatting elements.
6. If *entry* is neither a marker nor an element that is also in the stack of open elements, go to step 4.
7. Let *entry* be the element one later than *entry* in the list of active formatting elements.
8. Perform a shallow clone of the element *entry* to obtain *clone*. [DOM3CORE]
9. Append *clone* to the current node and push it onto the stack of open elements so that it is the new current node.
10. Replace the entry for *entry* in the list with an entry for *clone*.
11. If the entry for *clone* in the list of active formatting elements is not the last entry in the list, return to step 7.

This has the effect of reopening all the formatting elements that were opened in the current body, cell, or caption (whichever is youngest) that haven't been explicitly closed.

Note: The way this specification is written, the list of active formatting elements always

consists of elements in chronological order with the least recently added element first and the most recently added element last (except for while steps 8 to 11 of the above algorithm are being executed, of course).

When the steps below require the UA to **clear the list of active formatting elements up to the last marker**, the UA must perform the following steps:

1. Let *entry* be the last (most recently added) entry in the list of active formatting elements.
2. Remove *entry* from the list of active formatting elements.
3. If *entry* was a marker, then stop the algorithm at this point. The list has been cleared up to the last marker.
4. Go to step 1.

8.2.3.4. The element pointers

Initially the **`head` element pointer** and the **`form` element pointer** are both null.

Once a `head` element has been parsed (whether implicitly or explicitly) the `head` element pointer gets set to point to this node.

The `form` element pointer points to the last `form` element that was opened and whose end tag has not yet been seen. It is used to make form controls associate with forms in the face of dramatically bad markup, for historical reasons.

8.2.3.5. The scripting state

The **scripting flag** is set to "enabled" if the `Document` with which the parser is associated was with script when the parser was created, and "disabled" otherwise.

8.2.4 Tokenization

Implementations must act as if they used the following state machine to tokenise HTML. The state machine must start in the data state. Most states consume a single character, which may have various side-effects, and either switches the state machine to a new state to *reconsume* the same character, or switches it to a new state (to consume the next character), or repeats the same state (to consume the next character). Some states have more complicated behavior and can consume several characters before switching to another state.

The exact behavior of certain states depends on a **content model flag** that is set after certain tokens are emitted. The flag has several states: *PCDATA*, *RCDATA*, *CDATA*, and *PLAINTEXT*. Initially it must be in the PCDATA state. In the RCDATA and CDATA states, a further **escape flag** is used to control the behavior of the tokeniser. It is either true or false, and initially must be set to the false state. The insertion mode and the stack of open elements also affects tokenization.

The output of the tokenization step is a series of zero or more of the following tokens: DOCTYPE, start tag, end tag, comment, character, end-of-file. DOCTYPE tokens have a name, a public identifier, a system identifier, and a *force-quirks flag*. When a DOCTYPE token is created, its name, public identifier, and system

identifier must be marked as missing (which is a distinct state from the empty string), and the *force-quirks flag* must be set to *off* (its other state is *on*). Start and end tag tokens have a tag name, a *self-closing flag*, and a list of attributes, each of which has a name and a value. When a start or end tag token is created, its *self-closing flag* must be unset (its other state is that it be set), and its attributes list must be empty. Comment and character tokens have data.

When a token is emitted, it must immediately be handled by the tree construction stage. The tree construction stage can affect the state of the content model flag, and can insert additional characters into the stream. (For example, the `script` element can result in scripts executing and using the dynamic markup insertion APIs to insert characters into the stream being tokenised.)

When a start tag token is emitted with its *self-closing flag* set, if the flag is not **acknowledged** when it is processed by the tree construction stage, that is a parse error.

When an end tag token is emitted, the content model flag must be switched to the PCDATA state.

When an end tag token is emitted with attributes, that is a parse error.

When an end tag token is emitted with its *self-closing flag* set, that is a parse error.

Before each step of the tokeniser, the user agent may check to see if either one of the scripts in the list of scripts that will execute as soon as possible or the first script in the list of scripts that will execute asynchronously, has completed loading. If one has, then it must be executed and removed from its list.

The tokeniser state machine consists of the states defined in the following subsections.

8.2.4.1. Data state

Consume the next input character:

↪ **U+0026 AMPERSAND (&)**

When the content model flag is set to one of the PCDATA or RCDATA states and the escape flag is false: switch to the character reference data state.
Otherwise: treat it as per the "anything else" entry below.

↪ **U+002D HYPHEN-MINUS (-)**

If the content model flag is set to either the RCDATA state or the CDATA state, and the escape flag is false, and there are at least three characters before this one in the input stream, and the last four characters in the input stream, including this one, are U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, and U+002D HYPHEN-MINUS ("<!--"), then set the escape flag to true.

In any case, emit the input character as a character token. Stay in the data state.

↪ **U+003C LESS-THAN SIGN (<)**

When the content model flag is set to the PCDATA state: switch to the tag open state.
When the content model flag is set to either the RCDATA state or the CDATA state and the escape flag is false: switch to the tag open state.
Otherwise: treat it as per the "anything else" entry below.

↪ **U+003E GREATER-THAN SIGN (>)**

If the content model flag is set to either the RCDATA state or the CDATA state, and the escape flag

is true, and the last three characters in the input stream including this one are U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN ("-->"), set the escape flag to false.

In any case, emit the input character as a character token. Stay in the data state.

↪ **EOF**

Emit an end-of-file token.

↪ **Anything else**

Emit the input character as a character token. Stay in the data state.

8.2.4.2. Character reference data state

(This cannot happen if the content model flag is set to the CDATA state.)

Attempt to consume a character reference, with no additional allowed character.

If nothing is returned, emit a U+0026 AMPERSAND character token.

Otherwise, emit the character token that was returned.

Finally, switch to the data state.

8.2.4.3. Tag open state

The behavior of this state depends on the content model flag.

If the content model flag is set to the RCDATA or CDATA states

Consume the next input character. If it is a U+002F SOLIDUS (/) character, switch to the close tag open state. Otherwise, emit a U+003C LESS-THAN SIGN character token and reconsume the current input character in the data state.

If the content model flag is set to the PCDATA state

Consume the next input character:

↪ **U+0021 EXCLAMATION MARK (!)**

Switch to the markup declaration open state.

↪ **U+002F SOLIDUS (/)**

Switch to the close tag open state.

↪ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

Create a new start tag token, set its tag name to the lowercase version of the input character (add 0x0020 to the character's code point), then switch to the tag name state. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+0061 LATIN SMALL LETTER A through to U+007A LATIN SMALL LETTER Z**

Create a new start tag token, set its tag name to the input character, then switch to the tag name state. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Emit a U+003C LESS-THAN SIGN character token and a U+003E GREATER-THAN SIGN character token. Switch to the data state.

↪ **U+003F QUESTION MARK (?)**

Parse error. Switch to the bogus comment state.

↪ **Anything else**

Parse error. Emit a U+003C LESS-THAN SIGN character token and reconsume the current input character in the data state.

8.2.4.4. Close tag open state

If the content model flag is set to the RCDATA or CDATA states but no start tag token has ever been emitted by this instance of the tokeniser (fragment case), or, if the content model flag is set to the RCDATA or CDATA states and the next few characters do not match the tag name of the last start tag token emitted (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+000C FORM FEED (FF)
- U+0020 SPACE
- U+003E GREATER-THAN SIGN (>)
- U+002F SOLIDUS (/)
- EOF

...then emit a U+003C LESS-THAN SIGN character token, a U+002F SOLIDUS character token, and switch to the data state to process the next input character.

Otherwise, if the content model flag is set to the PCDATA state, or if the next few characters *do* match that tag name, consume the next input character:

↪ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

Create a new end tag token, set its tag name to the lowercase version of the input character (add 0x0020 to the character's code point), then switch to the tag name state. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+0061 LATIN SMALL LETTER A through to U+007A LATIN SMALL LETTER Z**

Create a new end tag token, set its tag name to the input character, then switch to the tag name state. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Switch to the data state.

↪ **EOF**

Parse error. Emit a U+003C LESS-THAN SIGN character token and a U+002F SOLIDUS character token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Switch to the bogus comment state.

8.2.4.5. Tag name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Switch to the before attribute name state.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state.

↪ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the current tag token's tag name. Stay in the tag name state.

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

↪ **U+002F SOLIDUS (/)**

Switch to the self-closing start tag state.

↪ **Anything else**

Append the current input character to the current tag token's tag name. Stay in the tag name state.

8.2.4.6. Before attribute name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the before attribute name state.

↪ **U+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 (/)**

Switch to the self-closing start tag state.

↪ **U+0022 QUOTATION MARK (")**

↪ **U+0027 APOSTROPHE (')**

↪ **U+003D EQUALS SIGN (=)**

Parse error. Treat it as per the "anything else" entry below.

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

↪ **Anything else**

Start a new attribute in the current tag token. Set that attribute's name to the current input

character, and its value to the empty string. Switch to the attribute name state.

8.2.4.7. Attribute name state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Switch to the after attribute name state.

- **U+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 (/)**

  Switch to the self-closing start tag state.

- **U+0022 QUOTATION MARK (")**
- **U+0027 APOSTROPHE (')**

  Parse error. Treat it as per the "anything else" entry below.

- **EOF**

  Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

- **Anything else**

  Append the current input character to the current attribute's name. Stay in the attribute name state.

When the user agent leaves the attribute name state (and before emitting the tag token, if appropriate), the complete attribute's name must be compared to the other attributes on the same token; if there is already an attribute on the token with the exact same name, then this is a parse error and the new attribute must be dropped, along with the value that gets associated with it (if any).

8.2.4.8. After attribute name state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Stay in the after attribute name state.

↪ **U+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 (/)**

Switch to the self-closing start tag state.

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

↪ **Anything else**

Start a new attribute in the current tag token. Set that attribute's name to the current input character, and its value to the empty string. Switch to the attribute name state.

8.2.4.9. Before attribute value state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000C FORM FEED (FF)**
↪ **U+0020 SPACE**

Stay in the before attribute value state.

↪ **U+0022 QUOTATION MARK (")**

Switch to the attribute value (double-quoted) state.

↪ **U+0026 AMPERSAND (&)**

Switch to the attribute value (unquoted) state and reconsume this input character.

↪ **U+0027 APOSTROPHE (')**

Switch to the attribute value (single-quoted) state.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state.

↪ **U+003D EQUALS SIGN (=)**

Parse error. Treat it as per the "anything else" entry below.

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the character in the data state.

↪ **Anything else**

Append the current input character to the current attribute's value. Switch to the attribute value (unquoted) state.

8.2.4.10. Attribute value (double-quoted) state

Consume the next input character:

↪ **U+0022 QUOTATION MARK (")**

Switch to the after attribute value (quoted) state.

↪ **U+0026 AMPERSAND (&)**

Switch to the character reference in attribute value state, with the additional allowed character being U+0022 QUOTATION MARK (").

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the character in the data state.

↪ **Anything else**

Append the current input character to the current attribute's value. Stay in the attribute value (double-quoted) state.

8.2.4.11. Attribute value (single-quoted) state

Consume the next input character:

↪ **U+0027 APOSTROPHE (')**

Switch to the after attribute value (quoted) state.

↪ **U+0026 AMPERSAND (&)**

Switch to the character reference in attribute value state, with the additional allowed character being U+0027 APOSTROPHE (').

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the character in the data state.

↪ **Anything else**

Append the current input character to the current attribute's value. Stay in the attribute value (single-quoted) state.

8.2.4.12. Attribute value (unquoted) state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Switch to the before attribute name state.

↪ **U+0026 AMPERSAND (&)**

Switch to the character reference in attribute value state, with no additional allowed character.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state.

- **U+0022 QUOTATION MARK (")**
- **U+0027 APOSTROPHE (')**
- **U+003D EQUALS SIGN (=)**

  Parse error. Treat it as per the "anything else" entry below.

- **EOF**

  Parse error. Emit the current tag token. Reconsume the character in the data state.

- **Anything else**

  Append the current input character to the current attribute's value. Stay in the attribute value (unquoted) state.

8.2.4.13. Character reference in attribute value state

Attempt to consume a character reference.

If nothing is returned, append a U+0026 AMPERSAND character to the current attribute's value.

Otherwise, append the returned character token to the current attribute's value.

Finally, switch back to the attribute value state that you were in when were switched into this state.

8.2.4.14. After attribute value (quoted) state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Switch to the before attribute name state.

- **U+003E GREATER-THAN SIGN (>)**

  Emit the current tag token. Switch to the data state.

- **U+002F SOLIDUS (/)**

  Switch to the self-closing start tag state.

- **EOF**

  Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

- **Anything else**

  Parse error. Reconsume the character in the before attribute name state.

8.2.4.15. Self-closing start tag state

Consume the next input character:

- **U+003E GREATER-THAN SIGN (>)**

  Set the *self-closing flag* of the current tag token. Emit the current tag token. Switch to the data

state.

↪ **EOF**

Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Reconsume the character in the before attribute name state.

8.2.4.16. Bogus comment state

(This can only happen if the content model flag is set to the PCDATA state.)

Consume every character up to and including the first U+003E GREATER-THAN SIGN character (>) or the end of the file (EOF), whichever comes first. Emit a comment token whose data is the concatenation of all the characters starting from and including the character that caused the state machine to switch into the bogus comment state, up to and including the character immediately before the last consumed character (i.e. up to the character just before the U+003E or EOF character). (If the comment was started by the end of the file (EOF), the token is empty.)

Switch to the data state.

If the end of the file was reached, reconsume the EOF character.

8.2.4.17. Markup declaration open state

(This can only happen if the content model flag is set to the PCDATA state.)

If the next two characters are both U+002D HYPHEN-MINUS (-) characters, consume those two characters, create a comment token whose data is the empty string, and switch to the comment start state.

Otherwise, if the next seven characters are a case-insensitive match for the word "DOCTYPE", then consume those characters and switch to the DOCTYPE state.

Otherwise, if the insertion mode is "in foreign content" and the current node is not an element in the HTML namespace and the next seven characters are a case-sensitive match for the string "[CDATA[" (the five uppercase letters "CDATA" with a U+005B LEFT SQUARE BRACKET character before and after), then consume those characters and switch to the CDATA section state (which is unrelated to the content model flag's CDATA state).

Otherwise, this is a parse error. Switch to the bogus comment state. The next character that is consumed, if any, is the first character that will be in the comment.

8.2.4.18. Comment start state

Consume the next input character:

↪ **U+002D HYPHEN-MINUS (-)**

Switch to the comment start dash state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Emit the comment token. Switch to the data state.

↪ **EOF**

Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**

Append the input character to the comment token's data. Switch to the comment state.

8.2.4.19. Comment start dash state

Consume the next input character:

↪ **U+002D HYPHEN-MINUS (-)**

Switch to the comment end state

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Emit the comment token. Switch to the data state.

↪ **EOF**

Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**

Append a U+002D HYPHEN-MINUS (-) character and the input character to the comment token's data. Switch to the comment state.

8.2.4.20. Comment state

Consume the next input character:

↪ **U+002D HYPHEN-MINUS (-)**

Switch to the comment end dash state

↪ **EOF**

Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**

Append the input character to the comment token's data. Stay in the comment state.

8.2.4.21. Comment end dash state

Consume the next input character:

↪ **U+002D HYPHEN-MINUS (-)**

Switch to the comment end state

↪ **EOF**

Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**

Append a U+002D HYPHEN-MINUS (-) character and the input character to the comment token's data. Switch to the comment state.

8.2.4.22. Comment end state

Consume the next input character:

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the comment token. Switch to the data state.

↪ **U+002D HYPHEN-MINUS (-)**

Parse error. Append a U+002D HYPHEN-MINUS (-) character to the comment token's data. Stay in the comment end state.

↪ **EOF**

Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Append two U+002D HYPHEN-MINUS (-) characters and the input character to the comment token's data. Switch to the comment state.

8.2.4.23. DOCTYPE state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000C FORM FEED (FF)**
↪ **U+0020 SPACE**

Switch to the before DOCTYPE name state.

↪ **Anything else**

Parse error. Reconsume the current character in the before DOCTYPE name state.

8.2.4.24. Before DOCTYPE name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000C FORM FEED (FF)**
↪ **U+0020 SPACE**

Stay in the before DOCTYPE name state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Create a new DOCTYPE token. Set its *force-quirks flag* to *on*. Emit the token. Switch to the data state.

↪ **EOF**

Parse error. Create a new DOCTYPE token. Set its *force-quirks flag* to *on*. Emit the token. Reconsume the EOF character in the data state.

↪ **Anything else**

Create a new DOCTYPE token. Set the token's name to the current input character. Switch to the DOCTYPE name state.

8.2.4.25. DOCTYPE name state

First, consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Switch to the after DOCTYPE name state.

- **U+003E GREATER-THAN SIGN (>)**

  Emit the current DOCTYPE token. Switch to the data state.

- **EOF**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

- **Anything else**

  Append the current input character to the current DOCTYPE token's name. Stay in the DOCTYPE name state.

8.2.4.26. After DOCTYPE name state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Stay in the after DOCTYPE name state.

- **U+003E GREATER-THAN SIGN (>)**

  Emit the current DOCTYPE token. Switch to the data state.

- **EOF**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

- **Anything else**

  If the next six characters are a case-insensitive match for the word "PUBLIC", then consume those characters and switch to the before DOCTYPE public identifier state.

  Otherwise, if the next six characters are a case-insensitive match for the word "SYSTEM", then consume those characters and switch to the before DOCTYPE system identifier state.

  Otherwise, this is the parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Switch to the bogus DOCTYPE state.

8.2.4.27. Before DOCTYPE public identifier state

Consume the next input character:

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**

  Stay in the before DOCTYPE public identifier state.

- **U+0022 QUOTATION MARK (")**

  Set the DOCTYPE token's public identifier to the empty string (not missing), then switch to the DOCTYPE public identifier (double-quoted) state.

- **U+0027 APOSTROPHE (')**

  Set the DOCTYPE token's public identifier to the empty string (not missing), then switch to the DOCTYPE public identifier (single-quoted) state.

- **U+003E GREATER-THAN SIGN (>)**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

- **EOF**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

- **Anything else**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Switch to the bogus DOCTYPE state.

8.2.4.28. DOCTYPE public identifier (double-quoted) state

Consume the next input character:

- **U+0022 QUOTATION MARK (")**

  Switch to the after DOCTYPE public identifier state.

- **U+003E GREATER-THAN SIGN (>)**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

- **EOF**

  Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

- **Anything else**

  Append the current input character to the current DOCTYPE token's public identifier. Stay in the DOCTYPE public identifier (double-quoted) state.

8.2.4.29. DOCTYPE public identifier (single-quoted) state

Consume the next input character:

↪ **U+0027 APOSTROPHE (')**

Switch to the after DOCTYPE public identifier state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Append the current input character to the current DOCTYPE token's public identifier. Stay in the DOCTYPE public identifier (single-quoted) state.

8.2.4.30. After DOCTYPE public identifier state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000C FORM FEED (FF)**
↪ **U+0020 SPACE**

Stay in the after DOCTYPE public identifier state.

↪ **U+0022 QUOTATION MARK (")**

Set the DOCTYPE token's system identifier to the empty string (not missing), then switch to the DOCTYPE system identifier (double-quoted) state.

↪ **U+0027 APOSTROPHE (')**

Set the DOCTYPE token's system identifier to the empty string (not missing), then switch to the DOCTYPE system identifier (single-quoted) state.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current DOCTYPE token. Switch to the data state.

↪ **EOF**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Switch to the bogus DOCTYPE state.

8.2.4.31. Before DOCTYPE system identifier state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the before DOCTYPE system identifier state.

↪ **U+0022 QUOTATION MARK (")**

Set the DOCTYPE token's system identifier to the empty string (not missing), then switch to the DOCTYPE system identifier (double-quoted) state.

↪ **U+0027 APOSTROPHE (')**

Set the DOCTYPE token's system identifier to the empty string (not missing), then switch to the DOCTYPE system identifier (single-quoted) state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Switch to the bogus DOCTYPE state.

8.2.4.32. DOCTYPE system identifier (double-quoted) state

Consume the next input character:

↪ **U+0022 QUOTATION MARK (")**

Switch to the after DOCTYPE system identifier state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Append the current input character to the current DOCTYPE token's system identifier. Stay in the DOCTYPE system identifier (double-quoted) state.

8.2.4.33. DOCTYPE system identifier (single-quoted) state

Consume the next input character:

↪ **U+0027 APOSTROPHE (')**

Switch to the after DOCTYPE system identifier state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Append the current input character to the current DOCTYPE token's system identifier. Stay in the DOCTYPE system identifier (single-quoted) state.

8.2.4.34. After DOCTYPE system identifier state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**
↪ **U+000A LINE FEED (LF)**
↪ **U+000C FORM FEED (FF)**
↪ **U+0020 SPACE**

Stay in the after DOCTYPE system identifier state.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current DOCTYPE token. Switch to the data state.

↪ **EOF**

Parse error. Set the DOCTYPE token's *force-quirks flag* to *on*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Switch to the bogus DOCTYPE state. (This does *not* set the DOCTYPE token's *force-quirks flag* to *on*.)

8.2.4.35. Bogus DOCTYPE state

Consume the next input character:

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the DOCTYPE token. Switch to the data state.

↪ **EOF**

Emit the DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Stay in the bogus DOCTYPE state.

8.2.4.36. CDATA section state

(This can only happen if the content model flag is set to the PCDATA state, and is unrelated to the content model flag's CDATA state.)

Consume every character up to the next occurrence of the three character sequence U+005D RIGHT SQUARE BRACKET U+005D RIGHT SQUARE BRACKET U+003E GREATER-THAN SIGN (]]>), or the end of the file (EOF), whichever comes first. Emit a series of text tokens consisting of all the characters consumed except the matching three character sequence at the end (if one was found before the end of the file).

Switch to the data state.

If the end of the file was reached, reconsume the EOF character.

8.2.4.37. Tokenizing character references

This section defines how to **consume a character reference**. This definition is used when parsing character references in text and in attributes.

The behavior depends on the identity of the next character (the one immediately after the U+0026 AMPERSAND character):

- **U+0009 CHARACTER TABULATION**
- **U+000A LINE FEED (LF)**
- **U+000C FORM FEED (FF)**
- **U+0020 SPACE**
- **U+003C LESS-THAN SIGN**
- **U+0026 AMPERSAND**
- **EOF**
- **The *additional allowed character*, if there is one**

  Not a character reference. No characters are consumed, and nothing is returned. (This is not an error, either.)

- **U+0023 NUMBER SIGN (#)**

  Consume the U+0023 NUMBER SIGN.

  The behavior further depends on the character after the U+0023 NUMBER SIGN:

  - **U+0078 LATIN SMALL LETTER X**
  - **U+0058 LATIN CAPITAL LETTER X**

    Consume the X.

    Follow the steps below, but using the range of characters U+0030 DIGIT ZERO through to U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A through to U+0066 LATIN SMALL LETTER F, and U+0041 LATIN CAPITAL LETTER A, through to U+0046 LATIN CAPITAL LETTER F (in other words, 0-9, A-F, a-f).

    When it comes to interpreting the number, interpret it as a hexadecimal number.

  - **Anything else**

    Follow the steps below, but using the range of characters U+0030 DIGIT ZERO through to U+0039 DIGIT NINE (i.e. just 0-9).

    When it comes to interpreting the number, interpret it as a decimal number.

  Consume as many characters as match the range of characters given above.

If no characters match the range, then don't consume any characters (and unconsume the U+0023 NUMBER SIGN character and, if appropriate, the X character). This is a parse error; nothing is returned.

Otherwise, if the next character is a U+003B SEMICOLON, consume that too. If it isn't, there is a parse error.

If one or more characters match the range, then take them all and interpret the string of characters as a number (either hexadecimal or decimal as appropriate).

If that number is one of the numbers in the first column of the following table, then this is a parse error. Find the row with that number in the first column, and return a character token for the Unicode character given in the second column of that row.

Number	Unicode character	
0x0D	U+000A	LINE FEED (LF)
0x80	U+20AC	EURO SIGN ('€')
0x81	U+FFFD	REPLACEMENT CHARACTER
0x82	U+201A	SINGLE LOW-9 QUOTATION MARK (',')
0x83	U+0192	LATIN SMALL LETTER F WITH HOOK ('ƒ')
0x84	U+201E	DOUBLE LOW-9 QUOTATION MARK (',,')
0x85	U+2026	HORIZONTAL ELLIPSIS ('…')
0x86	U+2020	DAGGER ('†')
0x87	U+2021	DOUBLE DAGGER ('‡')
0x88	U+02C6	MODIFIER LETTER CIRCUMFLEX ACCENT ('ˆ')
0x89	U+2030	PER MILLE SIGN ('‰')
0x8A	U+0160	LATIN CAPITAL LETTER S WITH CARON ('Š')
0x8B	U+2039	SINGLE LEFT-POINTING ANGLE QUOTATION MARK ('‹')
0x8C	U+0152	LATIN CAPITAL LIGATURE OE ('Œ')
0x8D	U+FFFD	REPLACEMENT CHARACTER
0x8E	U+017D	LATIN CAPITAL LETTER Z WITH CARON ('Ž')
0x8F	U+FFFD	REPLACEMENT CHARACTER
0x90	U+FFFD	REPLACEMENT CHARACTER
0x91	U+2018	LEFT SINGLE QUOTATION MARK ('‘')
0x92	U+2019	RIGHT SINGLE QUOTATION MARK ('’')
0x93	U+201C	LEFT DOUBLE QUOTATION MARK ('“')
0x94	U+201D	RIGHT DOUBLE QUOTATION MARK ('”')
0x95	U+2022	BULLET ('•')
0x96	U+2013	EN DASH ('–')
0x97	U+2014	EM DASH ('—')
0x98	U+02DC	SMALL TILDE ('˜')
0x99	U+2122	TRADE MARK SIGN ('™')
0x9A	U+0161	LATIN SMALL LETTER S WITH CARON ('š')
0x9B	U+203A	SINGLE RIGHT-POINTING ANGLE QUOTATION MARK ('›')
0x9C	U+0153	LATIN SMALL LIGATURE OE ('œ')
0x9D	U+FFFD	REPLACEMENT CHARACTER
0x9E	U+017E	LATIN SMALL LETTER Z WITH CARON ('ž')
0x9F	U+0178	LATIN CAPITAL LETTER Y WITH DIAERESIS ('Ÿ')

Otherwise, if the number is in the range 0x0000 to 0x0008, 0x000E to 0x001F, 0x007F to 0x009F,

0xD800 to 0xDFFF , 0xFDD0 to 0xFDDF, or is one of 0xFFFE, 0xFFFF, 0x1FFFE, 0x1FFFF, 0x2FFFE, 0x2FFFF, 0x3FFFE, 0x3FFFF, 0x4FFFE, 0x4FFFF, 0x5FFFE, 0x5FFFF, 0x6FFFE, 0x6FFFF, 0x7FFFE, 0x7FFFF, 0x8FFFE, 0x8FFFF, 0x9FFFE, 0x9FFFF, 0xAFFFE, 0xAFFFF, 0xBFFFE, 0xBFFFF, 0xCFFFE, 0xCFFFF, 0xDFFFE, 0xDFFFF, 0xEFFFE, 0xEFFFF, 0xFFFFE, 0xFFFFF, 0x10FFFE, or 0x10FFFF, or is higher than 0x10FFFF, then this is a parse error; return a character token for the U+FFFD REPLACEMENT CHARACTER character instead.

Otherwise, return a character token for the Unicode character whose code point is that number.

↪ **Anything else**

Consume the maximum number of characters possible, with the consumed characters case-sensitively matching one of the identifiers in the first column of the named character references table.

If no match can be made, then this is a parse error. No characters are consumed, and nothing is returned.

If the last character matched is not a U+003B SEMICOLON (`;`), there is a parse error.

If the character reference is being consumed as part of an attribute, and the last character matched is not a U+003B SEMICOLON (`;`), and the next character is in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE, U+0041 LATIN CAPITAL LETTER A to U+005A LATIN CAPITAL LETTER Z, or U+0061 LATIN SMALL LETTER A to U+007A LATIN SMALL LETTER Z, then, for historical reasons, all the characters that were matched after the U+0026 AMPERSAND (&) must be unconsumed, and nothing is returned.

Otherwise, return a character token for the character corresponding to the character reference name (as given by the second column of the named character references table).

> If the markup contains `I'm &notit; I tell you`, the character reference is parsed as "not", as in, `I'm ¬it; I tell you`. But if the markup was `I'm &notin; I tell you`, the character reference would be parsed as "notin;", resulting in `I'm ∉ I tell you`.

8.2.5 Tree construction

The input to the tree construction stage is a sequence of tokens from the tokenization stage. The tree construction stage is associated with a DOM `Document` object when a parser is created. The "output" of this stage consists of dynamically modifying or extending that document's DOM tree.

This specification does not define when an interactive user agent has to render the `Document` so that it is available to the user, or when it has to begin accepting user input.

As each token is emitted from the tokeniser, the user agent must process the token according to the rules given in the section corresponding to the current insertion mode.

When the steps below require the UA to **insert a character** into a node, if that node has a child immediately before where the character is to be inserted, and that child is a `Text` node, then the character must be appended to that `Text` node; otherwise, a new `Text` node whose data is just that character must be inserted in the appropriate place.

DOM mutation events must not fire for changes caused by the UA parsing the document. (Conceptually, the

parser is not mutating the DOM, it is constructing it.) This includes the parsing of any content inserted using `document.write()` and `document.writeln()` calls. [DOM3EVENTS]

Note: Not all of the tag names mentioned below are conformant tag names in this specification; many are included to handle legacy content. They still form part of the algorithm that implementations are required to implement to claim conformance.

Note: The algorithm described below places no limit on the depth of the DOM tree generated, or on the length of tag names, attribute names, attribute values, text nodes, etc. While implementors are encouraged to avoid arbitrary limits, it is recognized that practical concerns will likely force user agents to impose nesting depths.

8.2.5.1. Creating and inserting elements

When the steps below require the UA to **create an element for a token** in a particular namespace, the UA must create a node implementing the interface appropriate for the element type corresponding to the tag name of the token in the given namespace (as given in the specification that defines that element, e.g. for an `a` element in the HTML namespace, this specification defines it to be the `HTMLAnchorElement` interface), with the tag name being the name of that element, with the node being in the given namespace, and with the attributes on the node being those given in the given token.

The interface appropriate for an element in the HTML namespace that is not defined in this specification is `HTMLElement`. The interface appropriate for an element in another namespace that is not defined by that namespace's specification is `Element`.

When the steps below require the UA to **insert an HTML element** for a token, the UA must first create an element for the token in the HTML namespace, and then append this node to the current node, and push it onto the stack of open elements so that it is the new current node.

The steps below may also require that the UA insert an HTML element in a particular place, in which case the UA must follow the same steps except that it must insert or append the new node in the location specified instead of appending it to the current node. (This happens in particular during the parsing of tables with invalid content.)

When the steps below require the UA to **insert a foreign element** for a token, the UA must first create an element for the token in the given namespace, and then append this node to the current node, and push it onto the stack of open elements so that it is the new current node. If the newly created element has an `xmlns` attribute in the XMLNS namespace whose value is not exactly the same as the element's namespace, that is a parse error.

When the steps below require the user agent to **adjust foreign attributes** for a token, then, if any of the attributes on the token match the strings given in the first column of the following table, let the attribute be a namespaced attribute, with the prefix being the string given in the corresponding cell in the second column, the local name being the string given in the corresponding cell in the third column, and the namespace being the namespace given in the corresponding cell in the fourth column. (This fixes the use of namespaced attributes, in particular `xml:lang`.)

Attribute name	Prefix	Local name	Namespace
`xlink:actuate`	`xlink`	`actuate`	XLink namespace
`xlink:arcrole`	`xlink`	`arcrole`	XLink namespace

Attribute name	Prefix	Local name	Namespace
`xlink:href`	`xlink`	`href`	XLink namespace
`xlink:role`	`xlink`	`role`	XLink namespace
`xlink:show`	`xlink`	`show`	XLink namespace
`xlink:title`	`xlink`	`title`	XLink namespace
`xlink:type`	`xlink`	`type`	XLink namespace
`xml:base`	`xml`	`base`	XML namespace
`xml:lang`	`xml`	`lang`	XML namespace
`xml:space`	`xml`	`space`	XML namespace
`xmlns`	(none)	`xmlns`	XMLNS namespace
`xmlns:xlink`	`xmlns`	`xlink`	XMLNS namespace

The **generic CDATA element parsing algorithm** and the **generic RCDATA element parsing algorithm** consist of the following steps. These algorithms are always invoked in response to a start tag token.

1. Create an element for the token in the HTML namespace.
2. Append the new element to the current node.
3. If the algorithm that was invoked is the generic CDATA element parsing algorithm, switch the tokeniser's content model flag to the CDATA state; otherwise the algorithm invoked was the generic RCDATA element parsing algorithm, switch the tokeniser's content model flag to the RCDATA state.
4. 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 tokenizing.
5. 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.
6. The tokeniser's content model flag will have switched back to the PCDATA state.
7. If the next token is an end tag token with the same tag name as the start tag token, ignore it. Otherwise, it's an end-of-file token, and this is a parse error.

8.2.5.2. Closing elements that have implied end tags

When the steps below require the UA to **generate implied end tags**, then, while the current node is a `dd` element, a `dt` element, an `li` element, an `option` element, an `optgroup` element, a `p` element, an `rp` element, or an `rt` element, the UA must pop the current node off the stack of open elements.

If a step requires the UA to generate implied end tags but lists an element to exclude from the process, then the UA must perform the above steps as if that element was not in the above list.

8.2.5.3. Foster parenting

Foster parenting happens when content is misnested in tables.

When a node *node* is to be **foster parented**, the node *node* must be inserted into the *foster parent element*, and the current table must be marked as **tainted**. (Once the current table has been tainted, whitespace characters are inserted into the *foster parent element* instead of the current node.)

The **foster parent element** is the parent element of the last `table` element in the stack of open elements, if there is a `table` element and it has such a parent element. If there is no `table` element in the stack of open elements (fragment case), then the *foster parent element* is the first element in the stack of open elements (the `html` element). Otherwise, if there is a `table` element in the stack of open elements, but the last `table` element in the stack of open elements has no parent, or its parent node is not an element, then the *foster parent element* is the element before the last `table` element in the stack of open elements.

If the *foster parent element* is the parent element of the last `table` element in the stack of open elements, then *node* must be inserted immediately *before* the last `table` element in the stack of open elements in the *foster parent element*; otherwise, *node* must be *appended* to the *foster parent element*.

8.2.5.4. The "initial" insertion mode

When the insertion mode is "initial", tokens must be handled as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Ignore the token.

↪ **A comment token**

Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

If the DOCTYPE token's `name` does not case-insensitively match the string "`HTML`", or if the token's public identifier is not missing, or if the token's system identifier is not missing, then there is a parse error. Conformance checkers may, instead of reporting this error, switch to a conformance checking mode for another language (e.g. based on the DOCTYPE token a conformance checker could recognize that the document is an HTML4-era document, and defer to an HTML4 conformance checker.)

Append a `DocumentType` node to the `Document` node, with the `name` attribute set to the name given in the DOCTYPE token; the `publicId` attribute set to the public identifier given in the DOCTYPE token, or the empty string if the public identifier was missing; the `systemId` attribute set to the system identifier given in the DOCTYPE token, or the empty string if the system identifier was missing; and the other attributes specific to `DocumentType` objects set to null and empty lists as appropriate. Associate the `DocumentType` node with the `Document` object so that it is returned as the value of the `doctype` attribute of the `Document` object.

Then, if the DOCTYPE token matches one of the conditions in the following list, then set the document to quirks mode:

- The *force-quirks flag* is set to *on*.
- The name is set to anything other than "`HTML`".
- The public identifier starts with: "`+//Silmaril//dtd html Pro v0r11 19970101//`"
- The public identifier starts with: "`-//AdvaSoft Ltd//DTD HTML 3.0 asWedit + extensions//`"
- The public identifier starts with: "`-//AS//DTD HTML 3.0 asWedit + extensions//`"
- The public identifier starts with: "`-//IETF//DTD HTML 2.0 Level 1//`"
- The public identifier starts with: "`-//IETF//DTD HTML 2.0 Level 2//`"
- The public identifier starts with: "`-//IETF//DTD HTML 2.0 Strict Level 1//`"

- The public identifier starts with: "-//IETF//DTD HTML 2.0 Strict Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Strict//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0//"
- The public identifier starts with: "-//IETF//DTD HTML 2.1E//"
- The public identifier starts with: "-//IETF//DTD HTML 3.0//"
- The public identifier starts with: "-//IETF//DTD HTML 3.2 Final//"
- The public identifier starts with: "-//IETF//DTD HTML 3.2//"
- The public identifier starts with: "-//IETF//DTD HTML 3//"
- The public identifier starts with: "-//IETF//DTD HTML Level 0//"
- The public identifier starts with: "-//IETF//DTD HTML Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML Level 3//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 0//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 3//"
- The public identifier starts with: "-//IETF//DTD HTML Strict//"
- The public identifier starts with: "-//IETF//DTD HTML//"
- The public identifier starts with: "-//Metrius//DTD Metrius Presentational//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 HTML Strict//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 HTML//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 Tables//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 HTML Strict//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 HTML//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 Tables//"
- The public identifier starts with: "-//Netscape Comm. Corp.//DTD HTML//"
- The public identifier starts with: "-//Netscape Comm. Corp.//DTD Strict HTML//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML 2.0//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML Extended 1.0//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML Extended Relaxed 1.0//"
- The public identifier starts with: "-//SoftQuad Software//DTD HoTMetaL PRO 6.0::19990601::extensions to HTML 4.0//"
- The public identifier starts with: "-//SoftQuad//DTD HoTMetaL PRO 4.0::19971010::extensions to HTML 4.0//"
- The public identifier starts with: "-//Spyglass//DTD HTML 2.0 Extended//"
- The public identifier starts with: "-//SQ//DTD HTML 2.0 HoTMetaL + extensions//"
- The public identifier starts with: "-//Sun Microsystems Corp.//DTD HotJava HTML//"
- The public identifier starts with: "-//Sun Microsystems Corp.//DTD HotJava Strict HTML//"
- The public identifier starts with: "-//W3C//DTD HTML 3 1995-03-24//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2 Draft//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2 Final//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2S Draft//"
- The public identifier starts with: "-//W3C//DTD HTML 4.0 Frameset//"
- The public identifier starts with: "-//W3C//DTD HTML 4.0 Transitional//"
- The public identifier starts with: "-//W3C//DTD HTML Experimental 19960712//"
- The public identifier starts with: "-//W3C//DTD HTML Experimental 970421//"

- The public identifier starts with: "-//W3C//DTD W3 HTML//"
- The public identifier starts with: "-//W3O//DTD W3 HTML 3.0//"
- The public identifier is set to: "-//W3O//DTD W3 HTML Strict 3.0//EN//"
- The public identifier starts with: "-//WebTechs//DTD Mozilla HTML 2.0//"
- The public identifier starts with: "-//WebTechs//DTD Mozilla HTML//"
- The public identifier is set to: "-/W3C/DTD HTML 4.0 Transitional/EN"
- The public identifier is set to: "HTML"
- The system identifier is set to: "http://www.ibm.com/data/dtd/v11/ibmxhtml1-transitional.dtd"
- The system identifier is missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Frameset//"
- The system identifier is missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Transitional//"

Otherwise, if the DOCTYPE token matches one of the conditions in the following list, then set the document to limited quirks mode:

- The public identifier starts with: "-//W3C//DTD XHTML 1.0 Frameset//"
- The public identifier starts with: "-//W3C//DTD XHTML 1.0 Transitional//"
- The system identifier is not missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Frameset//"
- The system identifier is not missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Transitional//"

The name, system identifier, and public identifier strings must be compared to the values given in the lists above in a case-insensitive manner. A system identifier whose value is the empty string is not considered missing for the purposes of the conditions above.

Then, switch the insertion mode to "before html".

↪ **Anything else**

Parse error.

Set the document to quirks mode.

Switch the insertion mode to "before html", then reprocess the current token.

8.2.5.5. The "before html" insertion mode

When the insertion mode is "before html", tokens must be handled as follows:

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A comment token**

Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Ignore the token.

↪ **A start tag whose tag name is "html"**

Create an element for the token in the HTML namespace. Append it to the `Document` object. Put

this element in the stack of open elements.

If the token has an attribute "manifest", then resolve the value of that attribute to an absolute URL, and if that is successful, run the application cache selection algorithm with the resulting absolute URL. Otherwise, if there is no such attribute or resolving it fails, run the application cache selection algorithm with no manifest.

Switch the insertion mode to "before head".

↪ **Anything else**

Create an `HTMLElement` node with the tag name `html`, in the HTML namespace. Append it to the `Document` object. Put this element in the stack of open elements.

Run the application cache selection algorithm with no manifest.

Switch the insertion mode to "before head", then reprocess the current token.

Should probably make end tags be ignored, so that "</head><!-- --><html>" puts the comment before the root node (or should we?)

The root element can end up being removed from the `Document` object, e.g. by scripts; nothing in particular happens in such cases, content continues being appended to the nodes as described in the next section.

8.2.5.6. The "before head" insertion mode

When the insertion mode is "before head", tokens must be handled as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Ignore the token.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "head"**

Insert an HTML element for the token.

Set the `head` element pointer to the newly created `head` element.

Switch the insertion mode to "in head".

↪ **An end tag whose tag name is one of: "head", "br"**

Act as if a start tag token with the tag name "head" and no attributes had been seen, then reprocess the current token.

↪ **Any other end tag**

Parse error. Ignore the token.

↪ **Anything else**

Act as if a start tag token with the tag name "head" and no attributes had been seen, then reprocess the current token.

Note: This will result in an empty `head` element being generated, with the current token being reprocessed in the "after head" insertion mode.

8.2.5.7. The "in head" insertion mode

When the insertion mode is "in head", tokens must be handled as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Insert the character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is one of: "base", "command", "eventsource", "link"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "meta"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

If the element has a `charset` attribute, and its value is a supported encoding, and the confidence is currently *tentative*, then change the encoding to the encoding given by the value of the `charset` attribute.

Otherwise, if the element has a `content` attribute, and applying the algorithm for extracting an encoding from a Content-Type to its value returns a supported encoding *encoding*, and the confidence is currently *tentative*, then change the encoding to the encoding *encoding*.

↪ **A start tag whose tag name is "title"**

Follow the generic RCDATA element parsing algorithm.

↪ **A start tag whose tag name is "noscript", if the scripting flag is enabled**

↪ **A start tag whose tag name is one of: "noframes", "style"**

Follow the generic CDATA element parsing algorithm.

↪ **A start tag whose tag name is "noscript", if the scripting flag is disabled**

Insert an HTML element for the token.

Switch the insertion mode to "in head noscript".

↪ **A start tag whose tag name is "script"**

Create an element for the token in the HTML namespace.

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 tokenizing.

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 for the HTML fragment parsing algorithm, then mark the `script` element as "already executed", and skip the rest of the processing described for this token (including the part below where "pending external scripts" are executed). (fragment case)

> ***Note: Marking the `script` element as "already executed" prevents it from executing when it is inserted into the document a few paragraphs below. Thus, 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. 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 pending external script, then:

↪ **If the tree construction stage is being called reentrantly, say from a call to**

`document.write()`:

Abort the processing of any nested invocations of the tokeniser, yielding control back to the caller. (Tokenization will resume when the caller returns to the "outer" tree construction stage.)

↪ **Otherwise:**

Follow these steps:

1. Let *the script* be the pending external script. There is no longer a pending external script.
2. Pause until the script has completed loading.
3. Let the insertion point be just before the next input character.
4. Execute the script.
5. Let the insertion point be undefined again.
6. If there is once again a pending external script, then repeat these steps from step 1.

↪ **An end tag whose tag name is "head"**

Pop the current node (which will be the `head` element) off the stack of open elements.

Switch the insertion mode to "after head".

↪ **An end tag whose tag name is "br"**

Act as described in the "anything else" entry below.

↪ **A start tag whose tag name is "head"**

↪ **Any other end tag**

Parse error. Ignore the token.

↪ **Anything else**

Act as if an end tag token with the tag name "head" had been seen, and reprocess the current token.

In certain UAs, some elements don't trigger the "in body" mode straight away, but instead get put into the head. Do we want to copy that?

8.2.5.8. The "in head noscript" insertion mode

When the insertion mode is "in head noscript", tokens must be handled as follows:

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **An end tag whose tag name is "noscript"**

Pop the current node (which will be a `noscript` element) from the stack of open elements; the new current node will be a `head` element.

Switch the insertion mode to "in head".

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

↪ **A comment token**

↪ **A start tag whose tag name is one of: "link", "meta", "noframes", "style"**

Process the token using the rules for the "in head" insertion mode.

↪ **An end tag whose tag name is one of: "br"**

Act as described in the "anything else" entry below.

↪ **A start tag whose tag name is one of: "head", "noscript"**

↪ **Any other end tag**

Parse error. Ignore the token.

↪ **Anything else**

Parse error. Act as if an end tag with the tag name "noscript" had been seen and reprocess the current token.

8.2.5.9. The "after head" insertion mode

When the insertion mode is "after head", tokens must be handled as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Insert the character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "body"**

Insert an HTML element for the token.

Switch the insertion mode to "in body".

↪ **A start tag whose tag name is "frameset"**

Insert an HTML element for the token.

Switch the insertion mode to "in frameset".

↪ **A start tag token whose tag name is one of: "base", "link", "meta", "noframes", "script", "style", "title"**

Parse error.

Push the node pointed to by the `head` element pointer onto the stack of open elements.

Process the token using the rules for the "in head" insertion mode.

Pop the current node (which will be the node pointed to by the `head` element pointer) off the stack of open elements.

↪ **A start tag whose tag name is "head"**

↪ **Any other end tag**

Parse error. Ignore the token.

↪ **Anything else**

Act as if a start tag token with the tag name "body" and no attributes had been seen, and then reprocess the current token.

8.2.5.10. The "in body" insertion mode

When the insertion mode is "in body", tokens must be handled as follows:

↪ **A character token**

Reconstruct the active formatting elements, if any.

Insert the token's character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Parse error. For each attribute on the token, check to see if the attribute is already present on the top element of the stack of open elements. If it is not, add the attribute and its corresponding value to that element.

↪ **A start tag token whose tag name is one of: "base", "command", "eventsource", "link", "meta", "noframes", "script", "style", "title"**

Process the token using the rules for the "in head" insertion mode.

↪ **A start tag whose tag name is "body"**

Parse error.

If the second element on the stack of open elements is not a `body` element, or, if the stack of open elements has only one node on it, then ignore the token. (fragment case)

Otherwise, for each attribute on the token, check to see if the attribute is already present on the

`body` element (the second element) on the stack of open elements. If it is not, add the attribute and its corresponding value to that element.

↪ **An end-of-file token**

If there is a node in the stack of open elements that is not either a `dd` element, a `dt` element, an `li` element, a `p` element, a `tbody` element, a `td` element, a `tfoot` element, a `th` element, a `thead` element, a `tr` element, the `body` element, or the `html` element, then this is a parse error.

Stop parsing.

↪ **An end tag whose tag name is "body"**

If the stack of open elements does not have a `body` element in scope, this is a parse error; ignore the token.

Otherwise, if there is a node in the stack of open elements that is not either a `dd` element, a `dt` element, an `li` element, a `p` element, a `tbody` element, a `td` element, a `tfoot` element, a `th` element, a `thead` element, a `tr` element, the `body` element, or the `html` element, then this is a parse error.

Switch the insertion mode to "after body". Otherwise, ignore the token.

↪ **An end tag whose tag name is "html"**

Act as if an end tag with tag name "body" had been seen, then, if that token wasn't ignored, reprocess the current token.

Note: The fake end tag token here can only be ignored in the fragment case.

↪ **A start tag whose tag name is one of: "address", "article", "aside", "blockquote", "center", "datagrid", "details", "dialog", "dir", "div", "dl", "fieldset", "figure", "footer", "h1", "h2", "h3", "h4", "h5", "h6", "header", "menu", "nav", "ol", "p", "section", "ul"**

If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token.

↪ **A start tag whose tag name is one of: "pre", "listing"**

If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token.

If the next token is a U+000A LINE FEED (LF) character token, then ignore that token and move on to the next one. (Newlines at the start of `pre` blocks are ignored as an authoring convenience.)

↪ **A start tag whose tag name is "form"**

If the `form` element pointer is not null, then this is a parse error; ignore the token.

Otherwise:

If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token, and set the `form` element pointer to point to the element created.

↪ **A start tag whose tag name is "li"**

Run the following algorithm:

1. Initialise *node* to be the current node (the bottommost node of the stack).
2. If *node* is an `li` element, then act as if an end tag with the tag name "li" had been seen, then jump to the last step.
3. If *node* is not in the formatting category, and is not in the phrasing category, and is not an `address`, `div`, or `p` element, then jump to the last step.
4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2.
5. If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

   Finally, insert an HTML element for the token.

↪ **A start tag whose tag name is one of: "dd", "dt"**

Run the following algorithm:

1. Initialise *node* to be the current node (the bottommost node of the stack).
2. If *node* is a `dd` or `dt` element, then act as if an end tag with the same tag name as *node* had been seen, then jump to the last step.
3. If *node* is not in the formatting category, and is not in the phrasing category, and is not an `address`, `div`, or `p` element, then jump to the last step.
4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2.
5. If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

   Finally, insert an HTML element for the token.

↪ **A start tag whose tag name is "plaintext"**

If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token.

Switch the content model flag to the PLAINTEXT state.

Note: Once a start tag with the tag name "plaintext" has been seen, that will be the last token ever seen other than character tokens (and the end-of-file token), because there is no way to switch the content model flag out of the PLAINTEXT state.

↪ **An end tag whose tag name is one of: "address", "article", "aside", "blockquote", "center",**

"datagrid", "details", "dialog", "dir", "div", "dl", "fieldset", "figure", "footer", "header", "listing", "menu", "nav", "ol", "pre", "section", "ul"

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.
2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.
3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.

↪ **An end tag whose tag name is "form"**

Set the `form` element pointer to null.

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.
2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.
3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.

↪ **An end tag whose tag name is "p"**

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; act as if a start tag with the tag name `p` had been seen, then reprocess the current token.

Otherwise, run these steps:

1. Generate implied end tags, except for elements with the same tag name as the token.
2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.
3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.

↪ **An end tag whose tag name is one of: "dd", "dt", "li"**

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags, except for elements with the same tag name as the token.

2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.

3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.

↪ **An end tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

If the stack of open elements does not have an element in scope whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.

2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.

3. Pop elements from the stack of open elements until an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6" has been popped from the stack.

↪ **An end tag whose tag name is "sarcasm"**

Take a deep breath, then act as described in the "any other end tag" entry below.

↪ **A start tag whose tag name is "a"**

If the list of active formatting elements contains an element whose tag name is "a" between the end of the list and the last marker on the list (or the start of the list if there is no marker on the list), then this is a parse error; act as if an end tag with the tag name "a" had been seen, then remove that element from the list of active formatting elements and the stack of open elements if the end tag didn't already remove it (it might not have if the element is not in table scope).

> In the non-conforming stream `<a href="a">a<table><a href="b">b</table>x`, the first `a` element would be closed upon seeing the second one, and the "x" character would be inside a link to "b", not to "a". This is despite the fact that the outer `a` element is not in table scope (meaning that a regular `</a>` end tag at the start of the table wouldn't close the outer `a` element).

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Add that element to the list of active formatting elements.

↪ **A start tag whose tag name is one of: "b", "big", "em", "font", "i", "s", "small", "strike", "strong", "tt", "u"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Add that element to the list of active formatting elements.

↪ **A start tag whose tag name is "nobr"**

Reconstruct the active formatting elements, if any.

If the stack of open elements has a `nobr` element in scope, then this is a parse error; act as if an end tag with the tag name "nobr" had been seen, then once again reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Add that element to the list of active formatting elements.

↪ **An end tag whose tag name is one of: "a", "b", "big", "em", "font", "i", "nobr", "s", "small", "strike", "strong", "tt", "u"**

Follow these steps:

1. Let the *formatting element* be the last element in the list of active formatting elements that:
   - is between the end of the list and the last scope marker in the list, if any, or the start of the list otherwise, and
   - has the same tag name as the token.

   If there is no such node, or, if that node is also in the stack of open elements but the element is not in scope, then this is a parse error; ignore the token, and abort these steps.

   Otherwise, if there is such a node, but that node is not in the stack of open elements, then this is a parse error; remove the element from the list, and abort these steps.

   Otherwise, there is a *formatting element* and that element is in the stack and is in scope. If the element is not the current node, this is a parse error. In any case, proceed with the algorithm as written in the following steps.

2. Let the *furthest block* be the topmost node in the stack of open elements that is lower in the stack than the *formatting element*, and is not an element in the phrasing or formatting categories. There might not be one.

3. If there is no *furthest block*, then the UA must skip the subsequent steps and instead just pop all the nodes from the bottom of the stack of open elements, from the current node up to and including the *formatting element*, and remove the *formatting element* from the list of active formatting elements.

4. Let the *common ancestor* be the element immediately above the *formatting element* in the stack of open elements.

5. 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 above *node* in the stack of open elements.

   2. If *node* is not in the list of active formatting elements, then remove *node* from the stack of open elements and then go back to step 1.

   3. Otherwise, if *node* is the *formatting element*, then go to the next step in the overall algorithm.

   4. Otherwise, if *last node* is the *furthest block*, then move the aforementioned bookmark to be immediately after the *node* in the list of active formatting elements.

   5. 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. If the *common ancestor* node is a `table`, `tbody`, `tfoot`, `thead`, or `tr` element, then, foster parent whatever *last node* ended up being in the previous step.

   Otherwise, append whatever *last node* ended up being in the previous step to the *common ancestor* node, first removing it from its previous parent node if any.

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 below 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 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.

If the `form` element pointer is not null, then associate the `button` element with the `form` element pointed to by the `form` element pointer.

Insert a marker at the end of the list of active formatting elements.

↪ **A start tag token whose tag name is one of: "applet", "marquee", "object"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Insert a marker at the end of the list of active formatting elements.

↪ **An end tag token whose tag name is one of: "applet", "button", "marquee", "object"**

If the stack of open elements does not have an element in scope with the same tag name as that of the token, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.
2. If the current node is not an element with the same tag name as that of the token, then this is a parse error.
3. Pop elements from the stack of open elements until an element with the same tag name as the token has been popped from the stack.
4. Clear the list of active formatting elements up to the last marker.

↪ **A start tag whose tag name is "xmp"**

Reconstruct the active formatting elements, if any.

Follow the generic CDATA element parsing algorithm.

↪ **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.

Switch the insertion mode to "in table".

↪ **A start tag whose tag name is one of: "area", "basefont", "bgsound", "br", "embed", "img", "spacer", "wbr"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is one of: "param", "source"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "hr"**

If the stack of open elements has a `p` element in scope, then act as if an end tag with the tag name "p" had been seen.

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **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 HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

If the `form` element pointer is not null, then associate the newly created `input` element with the `form` element pointed to by the `form` element pointer.

↪ **A start tag whose tag name is "isindex"**

Parse error.

If the `form` element pointer is not null, then ignore the token.

Otherwise:

Acknowledge the token's *self-closing flag*, if it is set.

Act as if a start tag token with the tag name "form" had been seen.

If the token has an attribute called "action", set the `action` attribute on the resulting `form` element to the value of the "action" attribute of the token.

Act as if a start tag token with the tag name "hr" had been seen.

Act as if a start tag token with the tag name "p" had been seen.

Act as if a start tag token with the tag name "label" had been seen.

Act as if a stream of character tokens had been seen (see below for what they should say).

Act as if a start tag token with the tag name "input" had been seen, with all the attributes from the "isindex" token except "name", "action", and "prompt". Set the `name` attribute of the resulting `input` element to the value "`isindex`".

Act as if a stream of character tokens had been seen (see below for what they should say).

Act as if an end tag token with the tag name "label" had been seen.

Act as if an end tag token with the tag name "p" had been seen.

Act as if a start tag token with the tag name "hr" had been seen.

Act as if an end tag token with the tag name "form" had been seen.

If the token has an attribute with the name "prompt", then the first stream of characters must be the same string as given in that attribute, and the second stream of characters must be empty. Otherwise, the two streams of character tokens together should, together with the `input` element, express the equivalent of "This is a searchable index. Insert your search keywords here: (input field)" in the user's preferred language.

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 in the HTML namespace. Append the new element to the current node.

If the `form` element pointer is not null, then associate the newly created `textarea` element with the `form` element pointed to by the `form` element pointer.

Switch the tokeniser's content model flag to the RCDATA state.

If the next token is a U+000A LINE FEED (LF) character token, then ignore that token and move on to the next one. (Newlines at the start of `textarea` elements are ignored as an authoring convenience.)

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 tokenizing.

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. Otherwise, this is a parse error.

↪ **A start tag whose tag name is one of: "iframe", "noembed"**

↪ **A start tag whose tag name is "noscript", if the scripting flag is enabled**

Follow the generic CDATA element parsing algorithm.

↪ **A start tag whose tag name is "select"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

If the `form` element pointer is not null, then associate the `select` element with the `form` element pointed to by the `form` element pointer.

If the insertion mode is one of in table", "in caption", "in column group", "in table body", "in row", or "in cell", then switch the insertion mode to "in select in table". Otherwise, switch the insertion mode to "in select".

↪ **A start tag whose tag name is one of: "rp", "rt"**

If the stack of open elements has a `ruby` element in scope, then generate implied end tags. If the current node is not then a `ruby` element, this is a parse error; pop all the nodes from the current node up to the node immediately before the bottommost `ruby` element on the stack of open elements.

Insert an HTML element for the token.

↪ **An end tag whose tag name is "br"**

Parse error. Act as if a start tag token with the tag name "br" had been seen. Ignore the end tag token.

↪ **A start tag whose tag name is "math"**

Reconstruct the active formatting elements, if any.

Adjust foreign attributes for the token. (This fixes the use of namespaced attributes, in particular XLink.)

Insert a foreign element for the token, in the MathML namespace.

If the token has its *self-closing flag* set, pop the current node off the stack of open elements and acknowledge the token's *self-closing flag*.

Otherwise, let the secondary insertion mode be the current insertion mode, and then switch the insertion mode to "in foreign content".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "frame", "frameset", "head", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↪ **Any other start tag**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Note: This element will be a phrasing element.

↪ **Any other end tag**

Run the following steps:

1. 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 these steps.
3. Otherwise, if *node* is in neither the formatting category nor the phrasing category, then this is a parse error; ignore the token, and abort these steps.

4. Set *node* to the previous entry in the stack of open elements.

5. Return to step 2.

8.2.5.11. The "in table" insertion mode

When the insertion mode is "in table", tokens must be handled as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

If the current table is tainted, then act as described in the "anything else" entry below.

Otherwise, insert the character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "caption"**

Clear the stack back to a table context. (See below.)

Insert a marker at the end of the list of active formatting elements.

Insert an HTML element for the token, then switch the insertion mode to "in caption".

↪ **A start tag whose tag name is "colgroup"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in column group".

↪ **A start tag whose tag name is "col"**

Act as if a start tag token with the tag name "colgroup" had been seen, then reprocess the current token.

↪ **A start tag whose tag name is one of: "tbody", "tfoot", "thead"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in table body".

↪ **A start tag whose tag name is one of: "td", "th", "tr"**

Act as if a start tag token with the tag name "tbody" had been seen, then reprocess the current token.

↪ **A start tag whose tag name is "table"**

Parse error. Act as if an end tag token with the tag name "table" had been seen, then, if that token wasn't ignored, reprocess the current token.

Note: The fake end tag token here can only be ignored in the fragment case.

↪ **An end tag whose tag name is "table"**

If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Pop elements from this stack until a `table` element has been popped from the stack.

Reset the insertion mode appropriately.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↪ **A start tag whose tag name is one of: "style", "script"**

If the current table is tainted then act as described in the "anything else" entry below.

Otherwise, process the token using the rules for the "in head" insertion mode.

↪ **A start tag whose tag name is "input"**

If the token does not have an attribute with the name "type", or if it does, but that attribute's value is not a case-insensitive match for the string "hidden", or, if the current table is tainted, then: act as described in the "anything else" entry below.

Otherwise:

Parse error.

Insert an HTML element for the token.

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.

↪ **An end-of-file token**

If the current node is not the root `html` element, then this is a parse error.

Note: It can only be the current node in the fragment case.

Stop parsing.

↪ **Anything else**

Parse error. Process the token using the rules for the "in body" insertion mode, except that if the current node is a `table`, `tbody`, `tfoot`, `thead`, or `tr` element, then, whenever a node would be inserted into the current node, it must instead be foster parented.

When the steps above require the UA to **clear the stack back to a table context**, it means that the UA must, while the current node is not a `table` element or an `html` element, pop elements from the stack of open elements.

Note: The current node being an `html` element after this process is a fragment case.

8.2.5.12. The "in caption" insertion mode

When the insertion mode is "in caption", tokens must be handled as follows:

- **An end tag whose tag name is "caption"**

  If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (fragment case)

  Otherwise:

  Generate implied end tags.

  Now, if the current node is not a `caption` element, then this is a parse error.

  Pop elements from this stack until a `caption` element has been popped from the stack.

  Clear the list of active formatting elements up to the last marker.

  Switch the insertion mode to "in table".

- **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**
- **An end tag whose tag name is "table"**

  Parse error. Act as if an end tag with the tag name "caption" had been seen, then, if that token wasn't ignored, reprocess the current token.

  Note: The fake end tag token here can only be ignored in the fragment case.

- **An end tag whose tag name is one of: "body", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

  Parse error. Ignore the token.

- **Anything else**

  Process the token using the rules for the "in body" insertion mode.

8.2.5.13. The "in column group" insertion mode

When the insertion mode is "in column group", tokens must be handled as follows:

- **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

  Insert the character into the current node.

- **A comment token**

  Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

- **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "col"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **An end tag whose tag name is "colgroup"**

If the current node is the root `html` element, then this is a parse error; ignore the token. (fragment case)

Otherwise, pop the current node (which will be a `colgroup` element) from the stack of open elements. Switch the insertion mode to "in table".

↪ **An end tag whose tag name is "col"**

Parse error. Ignore the token.

↪ **An end-of-file token**

If the current node is the root `html` element, then stop parsing. (fragment case)

Otherwise, act as described in the "anything else" entry below.

↪ **Anything else**

Act as if an end tag with the tag name "colgroup" had been seen, and then, if that token wasn't ignored, reprocess the current token.

Note: The fake end tag token here can only be ignored in the fragment case.

8.2.5.14. The "in table body" insertion mode

When the insertion mode is "in table body", tokens must be handled as follows:

↪ **A start tag whose tag name is "tr"**

Clear the stack back to a table body context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in row".

↪ **A start tag whose tag name is one of: "th", "td"**

Parse error. Act as if a start tag with the tag name "tr" had been seen, then reprocess the current token.

↪ **An end tag whose tag name is one of: "tbody", "tfoot", "thead"**

If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token.

Otherwise:

Clear the stack back to a table body context. (See below.)

Pop the current node from the stack of open elements. Switch the insertion mode to "in table".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "tfoot", "thead"**

↪ **An end tag whose tag name is "table"**

If the stack of open elements does not have a `tbody`, `thead`, or `tfoot` element in table scope, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Clear the stack back to a table body context. (See below.)

Act as if an end tag with the same tag name as the current node ("tbody", "tfoot", or "thead") had been seen, then reprocess the current token.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "td", "th", "tr"**

Parse error. Ignore the token.

↪ **Anything else**

Process the token using the rules for the "in table" insertion mode.

When the steps above require the UA to **clear the stack back to a table body context**, it means that the UA must, while the current node is not a `tbody`, `tfoot`, `thead`, or `html` element, pop elements from the stack of open elements.

Note: The current node being an `html` element after this process is a fragment case.

8.2.5.15. The "in row" insertion mode

When the insertion mode is "in row", tokens must be handled as follows:

↪ **A start tag whose tag name is one of: "th", "td"**

Clear the stack back to a table row context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in cell".

Insert a marker at the end of the list of active formatting elements.

↪ **An end tag whose tag name is "tr"**

If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Clear the stack back to a table row context. (See below.)

Pop the current node (which will be a `tr` element) from the stack of open elements. Switch the insertion mode to "in table body".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "tfoot", "thead", "tr"**

↪ **An end tag whose tag name is "table"**

Act as if an end tag with the tag name "tr" had been seen, then, if that token wasn't ignored, reprocess the current token.

Note: The fake end tag token here can only be ignored in the fragment case.

↪ **An end tag whose tag name is one of: "tbody", "tfoot", "thead"**

If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token.

Otherwise, act as if an end tag with the tag name "tr" had been seen, then reprocess the current token.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "td", "th"**

Parse error. Ignore the token.

↪ **Anything else**

Process the token using the rules for the "in table" insertion mode.

When the steps above require the UA to **clear the stack back to a table row context**, it means that the UA must, while the current node is not a `tr` element or an `html` element, pop elements from the stack of open elements.

Note: The current node being an `html` element after this process is a fragment case.

8.2.5.16. The "in cell" insertion mode

When the insertion mode is "in cell", tokens must be handled as follows:

↪ **An end tag whose tag name is one of: "td", "th"**

If the stack of open elements does not have an element in table scope with the same tag name as that of the token, then this is a parse error and the token must be ignored.

Otherwise:

Generate implied end tags.

Now, if the current node is not an element with the same tag name as the token, then this is a parse error.

Pop elements from this stack until an element with the same tag name as the token has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in row". (The current node will be a `tr` element at this point.)

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

If the stack of open elements does *not* have a `td` or `th` element in table scope, then this is a parse

error; ignore the token. (fragment case)

Otherwise, close the cell (see below) and reprocess the current token.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html"**

Parse error. Ignore the token.

↪ **An end tag whose tag name is one of: "table", "tbody", "tfoot", "thead", "tr"**

If the stack of open elements does not have an element in table scope with the same tag name as that of the token (which can only happen for "tbody", "tfoot" and "thead", or, in the fragment case), then this is a parse error and the token must be ignored.

Otherwise, close the cell (see below) and reprocess the current token.

↪ **Anything else**

Process the token using the rules for the "in body" insertion mode.

Where the steps above say to **close the cell**, they mean to run the following algorithm:

1. If the stack of open elements has a `td` element in table scope, then act as if an end tag token with the tag name "td" had been seen.
2. Otherwise, the stack of open elements will have a `th` element in table scope; act as if an end tag token with the tag name "th" had been seen.

Note: The stack of open elements cannot have both a `td` and a `th` element in table scope at the same time, nor can it have neither when the insertion mode is "in cell".

8.2.5.17. The "in select" insertion mode

When the insertion mode is "in select", tokens must be handled as follows:

↪ **A character token**

Insert the token's character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "option"**

If the current node is an `option` element, act as if an end tag with the tag name "option" had been seen.

Insert an HTML element for the token.

↪ **A start tag whose tag name is "optgroup"**

If the current node is an `option` element, act as if an end tag with the tag name "option" had been seen.

If the current node is an `optgroup` element, act as if an end tag with the tag name "optgroup" had been seen.

Insert an HTML element for the token.

↪ **An end tag whose tag name is "optgroup"**

First, if the current node is an `option` element, and the node immediately before it in the stack of open elements is an `optgroup` element, then act as if an end tag with the tag name "option" had been seen.

If the current node is an `optgroup` element, then pop that node from the stack of open elements. Otherwise, this is a parse error; ignore the token.

↪ **An end tag whose tag name is "option"**

If the current node is an `option` element, then pop that node from the stack of open elements. Otherwise, this is a parse error; ignore the token.

↪ **An end tag whose tag name is "select"**

If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (fragment case)

Otherwise:

Pop elements from the stack of open elements until a `select` element has been popped from the stack.

Reset the insertion mode appropriately.

↪ **A start tag whose tag name is "select"**

Parse error. Act as if the token had been an end tag with the tag name "select" instead.

↪ **A start tag whose tag name is one of: "input", "textarea"**

Parse error. Act as if an end tag with the tag name "select" had been seen, and reprocess the token.

↪ **An end-of-file token**

If the current node is not the root `html` element, then this is a parse error.

Note: It can only be the current node in the fragment case.

Stop parsing.

↪ **Anything else**

Parse error. Ignore the token.

8.2.5.18. The "in select in table" insertion mode

When the insertion mode is "in select in table", tokens must be handled as follows:

↪ **A start tag whose tag name is one of: "caption", "table", "tbody", "tfoot", "thead", "tr", "td", "th"**

Parse error. Act as if an end tag with the tag name "select" had been seen, and reprocess the token.

↪ **An end tag whose tag name is one of: "caption", "table", "tbody", "tfoot", "thead", "tr", "td", "th"**

Parse error.

If the stack of open elements has an element in table scope with the same tag name as that of the token, then act as if an end tag with the tag name "select" had been seen, and reprocess the token. Otherwise, ignore the token.

↪ **Anything else**

Process the token using the rules for the "in select" insertion mode.

8.2.5.19. The "in foreign content" insertion mode

When the insertion mode is "in foreign content", tokens must be handled as follows:

↪ **A character token**

Insert the token's character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `mi` element in the MathML namespace.**

↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `mo` element in the MathML namespace.**

↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `mn` element in the MathML namespace.**

↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `ms` element in the MathML namespace.**

↪ **A start tag whose tag name is neither "mglyph" nor "malignmark", if the current node is an `mtext` element in the MathML namespace.**

↪ **A start tag, if the current node is an element in the HTML namespace.**

↪ **An end tag**

Process the token using the rules for the secondary insertion mode.

If, after doing so, the insertion mode is still "in foreign content", but there is no element in scope that has a namespace other than the HTML namespace, switch the insertion mode to the secondary insertion mode.

↪ **A start tag whose tag name is one of: "b", "big", "blockquote", "body", "br", "center", "code",**

"dd", "div", "dl", "dt", "em", "embed", "font", "h1", "h2", "h3", "h4", "h5", "h6", "head", "hr", "i", "img", "li", "listing", "menu", "meta", "nobr", "ol", "p", "pre", "ruby", "s", "small", "span", "strong", "strike", "sub", "sup", "table", "tt", "u", "ul", "var"

↪ **An end-of-file token**

Parse error.

Pop elements from the stack of open elements until the current node is in the HTML namespace.

Switch the insertion mode to the secondary insertion mode, and reprocess the token.

↪ **Any other start tag**

Adjust foreign attributes for the token. (This fixes the use of namespaced attributes, in particular XLink in SVG.)

Insert a foreign element for the token, in the same namespace as the current node.

If the token has its *self-closing flag* set, pop the current node off the stack of open elements and acknowledge the token's *self-closing flag*.

8.2.5.20. The "after body" insertion mode

When the insertion mode is "after body", tokens must be handled as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Process the token using the rules for the "in body" insertion mode.

↪ **A comment token**

Append a `Comment` node to the first element in the stack of open elements (the `html` element), with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **An end tag whose tag name is "html"**

If the parser was originally created as part of the HTML fragment parsing algorithm, this is a parse error; ignore the token. (fragment case)

Otherwise, switch the insertion mode to "after after body".

↪ **An end-of-file token**

Stop parsing.

↪ **Anything else**

Parse error. Switch the insertion mode to "in body" and reprocess the token.

8.2.5.21. The "in frameset" insertion mode

When the insertion mode is "in frameset", tokens must be handled as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Insert the character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **A start tag whose tag name is "frameset"**

Insert an HTML element for the token.

↪ **An end tag whose tag name is "frameset"**

If the current node is the root `html` element, then this is a parse error; ignore the token. (fragment case)

Otherwise, pop the current node from the stack of open elements.

If the parser was *not* originally created as part of the HTML fragment parsing algorithm (fragment case), and the current node is no longer a `frameset` element, then switch the insertion mode to "after frameset".

↪ **A start tag whose tag name is "frame"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "noframes"**

Process the token using the rules for the "in head" insertion mode.

↪ **An end-of-file token**

If the current node is not the root `html` element, then this is a parse error.

Note: It can only be the current node in the fragment case.

Stop parsing.

↪ **Anything else**

Parse error. Ignore the token.

8.2.5.22. The "after frameset" insertion mode

When the insertion mode is "after frameset", tokens must be handled as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

Insert the character into the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **An end tag whose tag name is "html"**

Switch the insertion mode to "after after frameset".

↪ **A start tag whose tag name is "noframes"**

Process the token using the rules for the "in head" insertion mode.

↪ **An end-of-file token**

Stop parsing.

↪ **Anything else**

Parse error. Ignore the token.

This doesn't handle UAs that don't support frames, or that do support frames but want to show the NOFRAMES content. Supporting the former is easy; supporting the latter is harder.

8.2.5.23. The "after after body" insertion mode

When the insertion mode is "after after body", tokens must be handled as follows:

↪ **A comment token**

Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **An end-of-file token**

Stop parsing.

↪ **Anything else**

Parse error. Switch the insertion mode to "in body" and reprocess the token.

8.2.5.24. The "after after frameset" insertion mode

When the insertion mode is "after after frameset", tokens must be handled as follows:

↪ **A comment token**

Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A DOCTYPE token**

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), or U+0020 SPACE**

↪ **A start tag whose tag name is "html"**

Process the token using the rules for the "in body" insertion mode.

↪ **An end-of-file token**

Stop parsing.

↪ **A start tag whose tag name is "noframes"**

Process the token using the rules for the "in head" insertion mode.

↪ **Anything else**

Parse error. Ignore the token.

8.2.6 The end

Once the user agent **stops parsing** the document, the user agent must follow the steps in this section.

First, the current document readiness must be set to "interactive".

Then, the rules for when a script completes loading start applying (script execution is no longer managed by the parser).

If any of the scripts in the list of scripts that will execute as soon as possible have completed loading, or if the list of scripts that will execute asynchronously is not empty and the first script in that list has completed loading, then the user agent must act as if those scripts just completed loading, following the rules given for that in the `script` element definition.

Then, if the list of scripts that will execute when the document has finished parsing is not empty, and the first item in this list has already completed loading, then the user agent must act as if that script just finished loading.

By this point, there will be no scripts that have loaded but have not yet been executed.

The user agent must then fire a simple event called `DOMContentLoaded` at the `Document`.

Once everything that **delays the load event** has completed, the user agent must set the current document readiness to "complete", and then fire a `load` event at the `body` element.

delaying the load event for things like image loads allows for intranet port scans (even without javascript!). Should we really encode that into the spec?

8.2.7 Coercing an HTML DOM into an infoset

When an application uses an HTML parser in conjunction with an XML pipeline, it is possible that the constructed DOM is not compatible with the XML tool chain in certain subtle ways. For example, an XML toolchain might not be able to represent attributes with the name `xmlns`, since they conflict with the Namespaces in XML syntax. There is also some data that the HTML parser generates that isn't included in the DOM itself. This section specifies some rules for handling these issues.

If the XML API being used doesn't support DOCTYPEs, tools may drop DOCTYPEs altogether.

If the XML API doesn't support attributes in no namespace that are named "`xmlns`", attributes whose names start with "`xmlns:`", or attributes in the XMLNS namespace, then the tool may drop such attributes.

The tool may annotate the output with any namespace declarations required for proper operation.

If the XML API being used restricts the allowable characters in the local names of elements and attributes, then the tool may map all element and attribute local names that the API wouldn't support to a set of names that *are* allowed, by replacing any character that isn't supported with the upper case letter U and the five digits of the character's Unicode codepoint when expressed in hexadecimal, using digits 0-9 and capital letters A-F as the symbols, in increasing numeric order.

> For example, the element name `foo<bar`, which can be output by the HTML parser, though it is neither a legal HTML element name nor a well-formed XML element name, would be converted into `fooU0003Cbar`, which *is* a well-formed XML element name (though it's still not legal in HTML by any means).

> As another example, consider the attribute `xlink:href`. Used on a MathML element, it becomes, after being adjusted, an attribute with a prefix "`xlink`" and a local name "`href`". However, used on an HTML element, it becomes an attribute with no prefix and the local name "`xlink:href`", which is not a valid NCName, and thus might not be accepted by an XML API. It could thus get converted, becoming "`xlinkU0003Ahref`".

Note: The resulting names from this conversion conveniently can't clash with any attribute generated by the HTML parser, since those are all either lowercase or those listed in the adjust foreign attributes algorithm's table.

If the XML API restricts comments from having two consecutive U+002D HYPHEN-MINUS characters (--), the tool may insert a single U+0020 SPACE character between any such offending characters.

If the XML API restricts comments from ending in a U+002D HYPHEN-MINUS character (-), the tool may insert a single U+0020 SPACE character at the end of such comments.

If the XML API restricts allowed characters in character data, the tool may replace any U+000C FORM FEED (FF) character with a U+0020 SPACE character, and any other literal non-XML character with a U+FFFD REPLACEMENT CHARACTER.

If the tool has no way to convey out-of-band information, then the tool may drop the following information:

- Whether the document is set to *no quirks mode*, *limited quirks mode*, or *quirks mode*
- The association between form controls and forms that aren't their nearest `form` element ancestor (use of the `form` element pointer in the parser)

Note: The mutatiosn allowed by this section apply after the HTML parser's rules have been applied. For example, a `<a::>` start tag will be closed by a `</a::>` end tag, and never by a `</aU0003AU0003A>` end tag, even if the user agent is using the rules above to then generate an actual element in the DOM with the name `aU0003AU0003A` for that start tag.

8.3 Namespaces

The **HTML namespace** is: `http://www.w3.org/1999/xhtml`

The **MathML namespace** is: `http://www.w3.org/1998/Math/MathML`

The **SVG namespace** is: `http://www.w3.org/2000/svg`

The **XLink namespace** is: `http://www.w3.org/1999/xlink`

The **XML namespace** is: `http://www.w3.org/XML/1998/namespace`

The **XMLNS namespace** is: `http://www.w3.org/2000/xmlns/`

8.4 Serializing HTML fragments

The following steps form the **HTML fragment serialization algorithm**. The algorithm takes as input a DOM `Element` or `Document`, referred to as *the node*, and either returns a string or raises an exception.

Note: This algorithm serializes the children of the node being serialized, not the node itself.

1. Let *s* be a string, and initialise it to the empty string.
2. For each child node of *the node*, in tree order, run the following steps:
    1. Let *current node* be the child node being processed.
    2. Append the appropriate string from the following list to *s*:

        ↪ **If *current node* is an `Element`**

        Append a U+003C LESS-THAN SIGN (<) character, followed by the element's tag name. (For nodes created by the HTML parser, `Document.createElement()`, or `Document.renameNode()`, the tag name will be lowercase.)

        For each attribute that the element has, append a U+0020 SPACE character, the attribute's name (which, for attributes set by the HTML parser or by `Element.setAttributeNode()` or `Element.setAttribute()`, will be lowercase), a U+003D EQUALS SIGN (=) character, a U+0022 QUOTATION MARK

(") character, the attribute's value, escaped as described below in *attribute mode*, and a second U+0022 QUOTATION MARK (") character.

While the exact order of attributes is UA-defined, and may depend on factors such as the order that the attributes were given in the original markup, the sort order must be stable, such that consecutive invocations of this algorithm serialize an element's attributes in the same order.

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

If *current node* is an `area`, `base`, `basefont`, `bgsound`, `br`, `col`, `embed`, `frame`, `hr`, `img`, `input`, `link`, `meta`, `param`, `spacer`, or `wbr` element, then continue on to the next child node at this point.

If *current node* is a `pre` `textarea`, or `listing` element, append a U+000A LINE FEED (LF) character.

Append the value of running the HTML fragment serialization algorithm on the *current node* element (thus recursing into this algorithm for that element), followed by a U+003C LESS-THAN SIGN (<) character, a U+002F SOLIDUS (/) character, the element's tag name again, and finally a U+003E GREATER-THAN SIGN (>) character.

↪ **If *current node* is a `Text` or `CDATASection` node**

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

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

↪ **If *current node* is a `Comment`**

Append the literal string `<!--` (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS), followed by the value of *current node*'s `data` DOM attribute, followed by the literal string `-->` (U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN).

↪ **If *current node* is a `ProcessingInstruction`**

Append the literal string `<?` (U+003C LESS-THAN SIGN, U+003F QUESTION MARK), followed by the value of *current node*'s `target` DOM attribute, followed by a single U+0020 SPACE character, followed by the value of *current node*'s `data` DOM attribute, followed by a single U+003E GREATER-THAN SIGN character ('>').

↪ **If *current node* is a `DocumentType`**

Append the literal string `<!DOCTYPE` (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+0044 LATIN CAPITAL LETTER D, U+004F LATIN CAPITAL LETTER O, U+0043 LATIN CAPITAL LETTER C, U+0054 LATIN CAPITAL LETTER T, U+0059 LATIN CAPITAL LETTER Y, U+0050 LATIN CAPITAL LETTER P, U+0045 LATIN CAPITAL LETTER E), followed by a space (U+0020 SPACE), followed by the value of *current node*'s `name` DOM attribute, followed by the literal

string > (U+003E GREATER-THAN SIGN).

Other node types (e.g. `Attr`) cannot occur as children of elements. If, despite this, they somehow do occur, this algorithm must raise an `INVALID_STATE_ERR` exception.

3. The result of the algorithm is the string *s*.

Escaping a string (for the purposes of the algorithm above) consists of replacing any occurrences of the "`&`" character by the string "`&`", any occurrences of the "`<`" character by the string "`<`", any occurrences of the "`>`" character by the string "`>`", any occurrences of the U+00A0 NO-BREAK SPACE character by the string "` `", and, if the algorithm was invoked in the *attribute mode*, any occurrences of the "`"`" character by the string "`&quot;`".

Note: Entity reference nodes are assumed to be expanded by the user agent, and are therefore not covered in the algorithm above.

Note: It is possible that the output of this algorithm, if parsed with an HTML parser, will not return the original tree structure. For instance, if a `textarea` element to which a `Comment` node has been appended is serialized and the output is then reparsed, the comment will end up being displayed in the text field. Similarly, if, as a result of DOM manipulation, an element contains a comment that contains the literal string "-->", then when the result of serializing the element is parsed, the comment will be truncated at that point and the rest of the comment will be interpreted as markup. More examples would be making a `script` element contain a text node with the text string "</script>", or having a `p` element that contains a `ul` element (as the `ul` element's start tag would imply the end tag for the `p`).

8.5 Parsing HTML fragments

The following steps form the **HTML fragment parsing algorithm**. The algorithm takes as input a DOM `Element`, referred to as the *context* element, which gives the context for the parser, as well as *input*, a string to parse, and returns a list of zero or more nodes.

Note: Parts marked* fragment case *in algorithms in the parser section are parts that only occur if the parser was created for the purposes of this algorithm. The algorithms have been annotated with such markings for informational purposes only; such markings have no normative weight. If it is possible for a condition described as a fragment case to occur even when the parser wasn't created for the purposes of handling this algorithm, then that is an error in the specification.

1. Create a new `Document` node, and mark it as being an HTML document.

2. Create a new HTML parser, and associate it with the just created `Document` node.

3. Set the HTML parser's tokenization stage's content model flag according to the *context* element, as follows:

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

   Set the content model flag to the RCDATA state.

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

Set the content model flag to the CDATA state.

↪ **If it is a `noscript` element**

If the scripting flag is enabled, set the content model flag to the CDATA state. Otherwise, set the content model flag to the PCDATA state.

↪ **If it is a `plaintext` element**

Set the content model flag to PLAINTEXT.

↪ **Otherwise**

Set the content model flag to the PCDATA state.

4. Let *root* be a new `html` element with no attributes.
5. Append the element *root* to the `Document` node created above.
6. Set up the parser's stack of open elements so that it contains just the single element *root*.
7. Reset the parser's insertion mode appropriately.

Note: The parser will reference the context element as part of that algorithm.

8. Set the parser's `form` element pointer to the nearest node to the *context* element that is a `form` element (going straight up the ancestor chain, and including the element itself, if it is a `form` element), or, if there is no such `form` element, to null.
9. Place into the input stream for the HTML parser just created the *input*.
10. Start the parser and let it run until it has consumed all the characters just inserted into the input stream.
11. Return all the child nodes of *root*, preserving the document order.

8.6 Named character references

This table lists the character reference names that are supported by HTML, and the code points to which they refer. It is referenced by the previous sections.

Name	Character
AElig;	U+000C6
AElig	U+000C6
AMP;	U+00026
AMP	U+00026
Aacute;	U+000C1
Aacute	U+000C1
Abreve;	U+00102
Acirc;	U+000C2
Acirc	U+000C2
Acy;	U+00410
Afr;	U+1D504
Agrave;	U+000C0
Agrave	U+000C0
Alpha;	U+00391
Amacr;	U+00100
And;	U+02A53
Aogon;	U+00104
Aopf;	U+1D538
ApplyFunction;	U+02061
Aring;	U+000C5
Aring	U+000C5
Ascr;	U+1D49C
Assign;	U+02254
Atilde;	U+000C3
Atilde	U+000C3
Auml;	U+000C4
Auml	U+000C4
Backslash;	U+02216
Barv;	U+02AE7
Barwed;	U+02306
Bcy;	U+00411
Because;	U+02235
Bernoullis;	U+0212C
Beta;	U+00392
Bfr;	U+1D505
Bopf;	U+1D539
Breve;	U+002D8
Bscr;	U+0212C
Bumpeq;	U+0224E
CHcy;	U+00427
COPY;	U+000A9
COPY	U+000A9
Cacute;	U+00106
Cap;	U+022D2
CapitalDifferentialD;	U+02145
Cayleys;	U+0212D
Ccaron;	U+0010C
Ccedil;	U+000C7
Ccedil	U+000C7
Ccirc;	U+00108
Cconint;	U+02230
Cdot;	U+0010A
Cedilla;	U+000B8
CenterDot;	U+000B7
Cfr;	U+0212D
Chi;	U+003A7
CircleDot;	U+02299
CircleMinus;	U+02296
CirclePlus;	U+02295
CircleTimes;	U+02297
ClockwiseContourIntegral;	U+02232
CloseCurlyDoubleQuote;	U+0201D
CloseCurlyQuote;	U+02019
Colon;	U+02237
Colone;	U+02A74
Congruent;	U+02261
Conint;	U+0222F
ContourIntegral;	U+0222E
Copf;	U+02102
Coproduct;	U+02210
CounterClockwiseContourIntegral;	U+02233
Cross;	U+02A2F
Cscr;	U+1D49E
Cup;	U+022D3
CupCap;	U+0224D
DD;	U+02145
DDotrahd;	U+02911
DJcy;	U+00402
DScy;	U+00405

9. Rendering and user-agent behavior

This section will probably include details on how to render DATAGRID (including its pseudo-elements), drag-and-drop, etc, in a visual medium, in concert with CSS. 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.

must define letting the user "**obtain a physical form** (or a representation of a physical form)" of a document (printing) and what this means for the UA, in particular creating a new view for the print media.

Must define that in CSS, tag names in HTML documents, and class names in quirks mode documents, are case-insensitive.

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` interface, during event propagation. There is no way currently to enumerate all the views.

9.2 Rendering and menus/toolbars

9.2.1 The 'icon' property

UAs should use the command's Icon as the default generic icon provided by the user agent when the 'icon' property computes to 'auto' on an element that either defines a command or refers to one using the `command` attribute, but when the property computes to an actual image, it should use that image instead.

9.3 Obsolete elements, attributes, and APIs

9.3.1 The `body` element

Need to define the content attributes in terms of CSS or something.

```
[XXX] interface HTMLDocument {
           attribute DOMString fgColor;
           attribute DOMString bgColor;
           attribute DOMString linkColor;
           attribute DOMString vlinkColor;
           attribute DOMString alinkColor;
};
```

The **`fgColor`** attribute on the `Document` object must reflect the `text` attribute on the `body` element.

The **`bgColor`** attribute on the `Document` object must reflect the `bgcolor` attribute on the `body` element.

The **`linkColor`** attribute on the `Document` object must reflect the `link` attribute on the `body` element.

The **`vLinkColor`** attribute on the `Document` object must reflect the `vlink` attribute on the `body` element.

The **`aLinkColor`** attribute on the `Document` object must reflect the `alink` attribute on the `body` element.

```
[XXX] interface HTMLBodyElement {
           attribute DOMString text;
           attribute DOMString bgColor;
           attribute DOMString background;
           attribute DOMString link;
           attribute DOMString vLink;
           attribute DOMString aLink;
};
```

The **`text`** DOM attribute of the `body` element must reflect the element's `text` content attribute.

The **`bgColor`** DOM attribute of the `body` element must reflect the element's `bgcolor` content attribute.

The **`background`** DOM attribute of the `body` element must reflect the element's `background` content attribute.

The **`link`** DOM attribute of the `body` element must reflect the element's `link` content attribute.

The **`aLink`** DOM attribute of the `body` element must reflect the element's `alink` content attribute.

The **`vLink`** DOM attribute of the `body` element must reflect the element's `vlink` content attribute.

9.3.2 The `applet` element

The `applet` element is a Java-specific variant of the `embed` element. In HTML5 the `applet` element is

obsoleted so that all extension frameworks (Java, .NET, Flash, etc) are handled in a consistent manner.

If the sandboxed plugins browsing context flag is set on the browsing context for which the `applet` element's document is the active document, then the element must be ignored (it represents nothing).

Otherwise, define how the element works, if supported.

```
[XXX] interface HTMLDocument {
  readonly attribute HTMLCollection applets;
};
```

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

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 Localization

If you wish to create localized versions of an HTML application, the best solution is to preprocess the files on the server, and then use HTTP content negotiation to serve the appropriate language.

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 Timers

This section is expected to be moved to the Window Object specification in due course.

Objects that implement the `Window` interface must also implement the `WindowTimers` interface:

```
[NoInterfaceObject] interface WindowTimers {
  // timers
  long setTimeout(in TimeoutHandler handler, in long timeout);
  long setTimeout(in TimeoutHandler handler, in long timeout, arguments...);
  long setTimeout(in DOMString code, in long timeout);
  long setTimeout(in DOMString code, in long timeout, in DOMString
language);
  void clearTimeout(in long handle);
  long setInterval(in TimeoutHandler handler, in long timeout);
  long setInterval(in TimeoutHandler handler, in long timeout,
arguments...);
  long setInterval(in DOMString code, in long timeout);
  long setInterval(in DOMString code, in long timeout, in DOMString
```

```
language);
  void clearInterval(in long handle);
};

interface TimeoutHandler {
  void handleEvent([Variadic] in any args);
};
```

The `setTimeout` and `setInterval` methods allow authors to schedule timer-based events.

The **`setTimeout(`*`handler`*`, `*`timeout`*`[, `*`arguments...`*`])`** method takes a reference to a `TimeoutHandler` object and a length of time in milliseconds. It must return a handle to the timeout created, and then asynchronously wait *timeout* milliseconds and then invoke `handleEvent()` on the *handler* object. If any *arguments...* were provided, they must be passed to the *handler* as arguments to the `handleEvent()` function.

Alternatively, **`setTimeout(`*`code`*`, `*`timeout`*`[, `*`language`*`])`** may be used. This variant takes a string instead of a `TimeoutHandler` object. That string must be parsed using the specified *language* (defaulting to ECMAScript if the third argument is omitted) and executed in the scope of the browsing context associated with the `Window` object on which the `setTimeout()` method was invoked.

Need to define *language* values.

The **`setInterval(...)`** variants must work in the same way as the `setTimeout` variants except that if *timeout* is a value greater than zero, 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.)

Index

This section is non-normative.

References

This section will be written in a future draft.

Acknowledgements

Thanks to Aankhen, Aaron Boodman, Aaron Leventhal, Adam Barth, Adam Roben, Addison Phillips, Adele Peterson, Adrian Sutton, Agustín Fernández, Alastair Campbell, Alexey Feldgendler, Anders Carlsson, Andrew Gove, Andrew Sidwell, Anne van Kesteren, Anthony Hickson, Anthony Ricaud, Antti Koivisto, Arphen Lin, Asbjørn Ulsberg, Ashley Sheridan, Aurelien Levy, Ben Godfrey, Ben Meadowcroft, Ben Millard, Benjamin Hawkes-Lewis, Bert Bos, Billy Wong, Bjoern Hoehrmann, Boris Zbarsky, Brad Fults, Brad Neuberg, Brady Eidson, Brendan Eich, Brett Wilson, Brian Campbell, Brian Smith, Bruce Miller, Cameron McCormack, Carlos Perelló Marín, Chao Cai, 윤석찬 (Channy Yun), Charl van Niekerk, Charles Iliya Krempeaux, Charles McCathieNevile, Christian Biesinger, Christian Johansen, Chriswa, Cole Robison, Collin Jackson, Daniel Barclay, Daniel Brumbaugh Keeney, Daniel Glazman, Daniel Peng, Daniel Spång, Daniel Steinberg, Danny Sullivan, Darin Adler, Darin Fisher, Dave Camp, Dave Singer, Dave Townsend, David Baron, David Bloom, David Carlisle, David Flanagan, David Håsäther, David Hyatt, Dean Edridge, Debi Orton, Derek Featherstone, DeWitt Clinton, Dimitri Glazkov, dolphinling, Doron Rosenberg, Doug Kramer, Eira Monstad, Elliotte Harold, Eric Law, Erik Arvidsson, Evan Martin, Evan Prodromou, fantasai, Felix Sasaki, Franck 'Shift' Quélain, Garrett Smith, Geoffrey Garen, Geoffrey Sneddon, Håkon Wium Lie, Henri Sivonen, Henrik Lied, Henry Mason, Hugh Winkler, Ignacio Javier, Ivo Emanuel Gonçalves, J. King, Jacques Distler, James Graham, James Justin Harrell, James M Snell, James Perrett, Jan-Klaas Kollhof, Jason White, Jasper Bryant-Greene, Jeff Cutsinger, Jeff Walden, Jens Bannmann, Jens Fendler, Jeroen van der Meer, Jim Jewett, Jim Meehan, Joe Clark, Jjgod Jiang, Joel Spolsky, Johan Herland, John Boyer, John Bussjaeger, John Harding, Johnny Stenback, Jon Perlow, Jonathan Worent, Jorgen Horstink, Josh Levenberg, Joshua Randall, Jukka K. Korpela, Julian Reschke, Kai Hendry, Kornel Lesinski, 黒澤剛志 (KUROSAWA Takeshi), Kristof Zelechovski, Lachlan Hunt, Larry Page, Lars Gunther, Laura L. Carlson, Laura Wisewell, Laurens Holst, Lee Kowalkowski, Leif Halvard Silli, Lenny Domnitser, Léonard Bouchet, Leons Petrazickis, Logan, Loune, Maciej Stachowiak, Magnus Kristiansen, Malcolm Rowe, Mark Nottingham, Mark Rowe, Mark Schenk, Martijn Wargers, Martin Atkins, Martin Dürst, Martin Honnen, Masataka Yakura, Mathieu Henri, Matthew Mastracci, Matthew Raymond, Matthew Thomas, Mattias Waldau, Max Romantschuk, Michael 'Ratt' Iannarelli, Michael A. Nachbaur, Michael A. Puls II, Michael Carter, Michael Gratton, Michael Powers, Michael(tm) Smith, Michel Fortin, Michiel van der Blonk, Mihai Şucan, Mike Brown, Mike Dierken, Mike Dixon, Mike Schinkel, Mike Shaver, Mikko Rantalainen, Neil Deakin, Neil Soiffer, Olaf Hoffmann, Olav Junker Kjær, Oliver Hunt, Peter Karlsson, Peter Kasting, Philip Jägenstedt, Philip Taylor, Philip TAYLOR, Rachid Finge, Rajas Moonka, Ralf Stoltze, Ralph Giles, Raphael Champeimont, Rene Saarsoo, Richard Ishida, Rimantas Liubertas, Robert Blaut, Robert O'Callahan, Robert Sayre, Roman Ivanov, Ryan King, S. Mike Dierken, Sam Ruby, Sam Weinig, Scott Hess, Sean Knapp, Shaun Inman, Silvia Pfeiffer, Simon Pieters, Stefan Haustein, Stephen Ma, Steve Faulkner, Steve Runyon, Steven Garrity, Stewart Brodie, Stuart Parmenter, Sunava Dutta, Tantek Çelik, Terrence Wood, Thomas Broyer, Thomas O'Connor, Tim Altman, Tim Johansson, Tyler Close, Vladimir Vukićević, Wakaba, Wayne Pollock, William Swanson, Yi-An Huang, and Øistein E. Andersen, for their useful and substantial comments.

Thanks also to everyone who has ever posted about HTML5 to their blogs, public mailing lists, or forums, including the W3C public-html list and the various WHATWG lists.

Special thanks to Richard Williamson for creating the first implementation of `canvas` in Safari, from which the canvas feature was designed.

Special thanks also to the Microsoft employees who first implemented the event-based drag-and-drop mechanism, `contenteditable`, and other features first widely deployed by the Windows Internet Explorer browser.

Special thanks and $10,000 to David Hyatt who came up with a broken implementation of the adoption agency algorithm that the editor had to reverse engineer and fix before using it in the parsing section.

Thanks to the many sources that provided inspiration for the examples used in the specification.

Thanks also to the Microsoft blogging community for some ideas, to the attendees of the W3C Workshop on Web Applications and Compound Documents for inspiration, and to the #mrt crew, the #mrt.no crew, the #whatwg crew, and the cabal for their ideas and support.

Specification annotation system: Login...