HTML 5
Working Draft — 12 January 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!

One-page version:

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

Multiple-page version:

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

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

Editor:

> Ian Hickson, Google, ian@hixie.ch

Abstract

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

Status of this document

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

Implementors should be aware that this specification is not stable. **Implementors who are not taking part in the discussions are likely to find the specification changing out from under**

them in incompatible ways. Vendors interested in implementing this specification before it eventually reaches the call for implementations should join the WHATWG mailing list and take part in the discussions.

This 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.

<div style="border:2px solid red;color:red;">

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).

</div>

Table of contents

1. Introduction

This section is non-normative.

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

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

1.1. Scope

This section is non-normative.

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

The scope of this specification does not include addressing presentation concerns (although default

rendering rules for Web browsers are included at the end of this specification).

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

The scope of this specification is not to describe an entire operating system. In particular, hardware configuration software, image manipulation tools, and applications that users would be expected to use with high-end workstations on a daily basis are out of scope. In terms of applications, this specification is 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 and Macromedia's Flash). These solutions are evolving faster than any standards process could follow, and the requirements are evolving even faster. These systems are also significantly more complicated to specify, and are orders of magnitude more difficult to achieve interoperability with, than the solutions described in this document. Platform-specific solutions for such sophisticated applications (for example the MacOS X Core APIs) are even further ahead.

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

This section is non-normative.

This specification represents a new version of HTML4 and XHTML1, along with a new version of the associated DOM2 HTML API. Migration from HTML4 or XHTML1 to the format and APIs described in this specification should in most cases be straightforward, as care has been taken to ensure that backwards-compatibility is retained.

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

1.1.2. 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.

1.1.3. 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, vender-neutral language, HTML provides for a solution to the same problems without the risk of vendor lock-in.

1.2. Structure of this specification

This section is non-normative.

This specification is divided into the following important sections:

The DOM

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

The Semantics

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

Browsing Contexts

> 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.

APIs

> The Editing APIs: HTML documents can provide a number of mechanisms for users to 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 serialised form and sent to other people, and so this section defines the syntax of HTML, along with rules for how to parse HTML.

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

1.2.1. How to read this specification

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

1.3. Conformance requirements

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

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

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 labelled as `text/html` as described in this specification, so that users can interact with them.

Non-interactive presentation user agents

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

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

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

User agents with no scripting support

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

> *Note: Scripting can form an integral part of an application. Web browsers that do not support scripting, or that have scripting disabled, might be unable to*

fully convey the author's intent.

Conformance checkers

Conformance checkers must verify that a document conforms to the applicable conformance criteria described in this specification. Conformance checkers are exempt from detecting errors that require interpretation of the author's intent (for example, while a document is non-conforming if the content of a `blockquote` element is not a quote, conformance checkers do not have to check that `blockquote` elements only contain quoted material).

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

The term "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 tools 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 errorneous 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, and this specification makes certain concessions to WYSIWYG editors.

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 behaviour. The former category of requirements are requirements on documents and authoring tools. The second category are requirements on user agents.

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

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

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

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

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

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 entities for characters in XHTML documents is unsafe (except

for <, >, &, " and '). For interoperability, authors are advised to avoid optional features of XML.

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

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

1.3.1. Common conformance requirements for APIs exposed to JavaScript

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 other 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 other specified, if a DOM attribute that is a signed numeric type is assigned a negative value, a `NOT_SUPPORTED_ERR` exception must be raised.

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

Unless other specified, if a method is expecting, as one of its arguments, as defined by its IDL definition, an object implementing a particular interface *X*, and the argument passed is an object whose [[Class]] property is neither that interface *X*, nor the name of an interface *Y* where this specification requires that all objects implementing interface *Y* also implement interface *X*, nor the name of an interface that inherits from the expected interface *X*, then a `TYPE_MISMATCH_ERR` exception must be raised.

Anything else? Passing the wrong type of object, maybe? Implied conversions to int/float?

1.3.2. Dependencies

This specification relies on several other underlying specifications.

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

XML Base
> User agents must follow the rules given by XML Base to resolve relative URIs in HTML and

XHTML fragments. That is the mechanism used in this specification for resolving relative URIs in DOM trees. [XMLBASE]

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

DOM

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

ECMAScript

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

This specification does not require support of any particular network transport protocols, 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.*

1.3.3. Features defined in other specifications

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

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

The 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?rev=1.35&content-type=text/html;%20charset=utf-8

Certain features are defined in terms of CSS <color> values. When the CSS value `currentColor` is specified in this context, 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. [CSS3COLOR]

> If a canvas gradient's `addColorStop()` method is called with the `currentColor` keyword as the color, then the computed value of the 'color' property on the `canvas` element is the one that is used.

1.4. Terminology

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

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

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

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

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

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

For readability, the term URI is used to refer to both ASCII URIs and Unicode IRIs, as those terms are defined by RFC 3986 and RFC 3987 respectively. On the rare occasions where IRIs are not allowed but ASCII URIs are, this is called out explicitly. [RFC3986] [RFC3987]

The term **root element**, when not qualified to explicitly refer to the document's root element, means the furthest ancestor element node of whatever node is being discussed, or the node itself is there is none. When the node is a part of the document, then that is indeed the document's root element. However, if the node is not currently part of the document tree, the root element will be an orphaned node.

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

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

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

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

xml

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

html

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

svg

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

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

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

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

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

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

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

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

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

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

1.4.1. HTML vs XHTML

This section is non-normative.

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

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

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

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

The second concrete syntax uses XML, and is known as "XHTML5". When a document is transmitted with an XML MIME type, such as `application/xhtml+xml`, then it is processed by an XML processor by Web browsers, and treated as an "XHTML5" document. Generally speaking, authors are discouraged from trying to use XML on the Web, because XML has much stricter syntax rules than the "HTML5" variant described above, and is relatively newer and therefore less mature.

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

2. The Document Object Model

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

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

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

‖ For example, an <u>ol</u> element represents an ordered list.

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

2.1. Documents

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

`Document` objects are assumed to be **XML documents** unless they are flagged as being **HTML**

documents when they are created. Whether a document is an HTML document or an XML document affects the behaviour of certain APIs, as well as a few CSS rendering rules. [CSS21]

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

All `Document` objects (in user agents implementing this specification) must also implement the `HTMLDocument` interface, available using binding-specific methods. (This is the case whether or not the document in question is an HTML document or indeed whether it contains any HTML elements at all.) `Document` objects must also implement the document-level interface of any other namespaces found in the document that the UA supports. For example, if an HTML implementation also supports SVG, then the `Document` object must implement `HTMLDocument` and `SVGDocument`.

> *Note: Because the `HTMLDocument` interface is now obtained using binding-specific casting methods instead of simply being the primary interface of the document object, it is no longer defined as inheriting from `Document`.*

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

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

  // Dynamic markup insertion
           attribute DOMString innerHTML;
  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);
```

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

   // Commands
   readonly attribute HTMLCollection commands;

   // Editing
            attribute boolean designMode;
   boolean execCommand(in DOMString commandId);
   boolean execCommand(in DOMString commandId, in boolean doShowUI);
   boolean execCommand(in DOMString commandId, in boolean doShowUI, in
DOMString value);
   Selection getSelection();



};
```

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

2.1.1. Security

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

2.1.2. Resource metadata management

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

The **domain** attribute must be initialised to the document's domain, if it has one, and null otherwise. On getting, the attribute must return its current value. On setting, if the new value is an allowed value (as defined below), the attribute's value must be changed to the new value. If the new value is not an allowed value, then a security exception must be raised instead.

A new value is an allowed value for the `document.domain` attribute if it is equal to the attribute's current value, or if the new value, prefixed by a U+002E FULL STOP ("."), exactly matches the end of the current value. If the current value is null, new values other than null will never be allowed.

If the `Document` object's address is hierarchical and uses a server-based naming authority, then its **domain** is the <host>/<ihost> part of that address. Otherwise, it has no domain.

Note: The domain attribute is used to enable pages on different hosts of a domain to access each others' DOMs , though this is not yet defined by this specification .

we should handle IP addresses here

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

The **cookie** attribute must, on getting, return the same string as the value of the `Cookie` HTTP header it would include if fetching the resource indicated by the document's address over HTTP, as per RFC 2109 section 4.3.4. [RFC2109]

On setting, the `cookie` attribute must cause the user agent to act as it would when processing cookies if it had just attempted to fetch the document's address over HTTP, and had received a response with a `Set-Cookie` header whose value was the specified value, as per RFC 2109 sections 4.3.1, 4.3.2, and 4.3.3. [RFC2109]

> *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 filesystem 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`.

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`". The document can also be set to **limited quirks mode** (also known as "almost standards" mode). By default, the document is set to **no quirks mode** (also known as

"standards mode").

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).

2.2. Elements

The nodes representing HTML elements in the DOM must implement, and expose to scripts, the interfaces listed for them in the relevant sections of this specification. This includes XHTML elements in XML documents, even when those documents are in another context (e.g. inside an XSLT transform).

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

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

  // dynamic markup insertion
           attribute DOMString innerHTML;

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

  // interaction
           attribute boolean irrelevant;
           attribute long tabIndex;
  void click();
  void focus();
  void blur();
  void scrollIntoView();
  void scrollIntoView(in boolean top);

  // commands
           attribute HTMLMenuElement contextMenu;

  // editing
           attribute boolean draggable;
           attribute DOMString contentEditable;
```

```
// 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 onsubmit;
        attribute EventListener onunload;

};
```

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

2.2.1. Reflecting content attributes in DOM attributes

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

attribute changes the value of the content attribute to the given value.

If a reflecting DOM attribute is a `DOMString` attribute whose content attribute is defined to contain a URI, then on getting, the DOM attribute must return the value of the content attribute, resolved to an absolute URI, and on setting, must set the content attribute to the specified literal value. If the content attribute is absent, the DOM attribute must return the default value, if the content attribute has one, or else the empty string.

If a reflecting DOM attribute is a `DOMString` attribute whose content attribute is defined to contain one or more URIs, then on getting, the DOM attribute must split the content attribute on spaces and return the concatenation of each token URI, resolved to an absolute URI, with a single U+0020 SPACE character between each URI; and on setting, must set the content attribute to the specified literal value. If the content attribute is absent, the DOM attribute must return the default value, if the content attribute has one, or else the empty string.

If a reflecting DOM attribute is a `DOMString` 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 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 that 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 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 the content attribute must be parsed according to [the rules for parsing signed integers]() first. If that fails, or if the attribute is absent, the default value must be returned instead, or 0 if there is no default value. On setting, the given value must be converted to a string representing the number as a [valid integer]() in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is an *unsigned* integer type (`unsigned long`) then the content attribute must be parsed according to [the rules for parsing unsigned integers]() first. If that fails, or if the attribute is absent, the default value must be returned instead, or 0 if there is no default value. On setting, the given value must be converted to a string representing the number as a [valid non-negative integer]() in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is 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 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 a string representing the number as a [valid non-negative integer]() in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is a floating point number type (`float`) and the content attribute is defined to contain a time offset, then the content attribute must be parsed according to [the rules for parsing time ofsets]() first. If that fails, or if the attribute is absent, the default value must be returned

instead, or the not-a-number value (NaN) if there is no default value. On setting, the given value must be converted to a string using the [time offset serialisation rules](), and 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 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.

2.3. Common DOM interfaces

2.3.1. Collections

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

When a [collection]() is created, a filter and a root are associated with the collection.

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

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

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

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

2.3.1.1. HTMLCollection

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

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

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

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

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

- It is an `a`, applet, `area`, form, `img`, or `object` element with a name attribute equal to *key*, or,

- It is an [HTML element](#) of any kind with an `id` attribute equal to *key*. (Non-HTML elements, even if they have IDs, are not searched for the purposes of `namedItem()`.)

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

In ECMAScript implementations, objects that implement the `HTMLCollection` interface must also have a [[Get]] method that, when invoked with a property name that is a number, acts like the `item()` method would when invoked with that argument, and when invoked with a property name that is a string, acts like the `namedItem()` method would when invoked with that argument.

2.3.1.2. HTMLFormControlsCollection

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

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

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

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

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

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

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

3. Otherwise, create a `NodeList` object representing a live view of the `HTMLFormControlsCollection` object, further filtered so that the only nodes in the

NodeList object are those that have either an id attribute or a name attribute equal to *key*. The nodes in the NodeList object must be sorted in <u>tree order</u>.

4. Return that NodeList object.

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

2.3.1.3. HTMLOptionsCollection

The <u>HTMLOptionsCollection</u> interface represents a list of option elements.

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

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

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

> *Note: Setting <u>length</u> 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 <u>id</u> 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 <u>id</u> 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 <u>HTMLOptionsCollection</u> object, further filtered so that the only nodes in the NodeList object are those that have either an <u>id</u> attribute or a name attribute equal to *key*. The nodes in the NodeList object must be sorted in <u>tree order</u>.

4. Return that NodeList object.

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

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

2.3.2. 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).

```
interface DOMTokenList {
  readonly attribute unsigned long length;
  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.

In ECMAScript implementations, objects that implement the `DOMTokenList` interface must also have a [[Get]] method that, when invoked with a property name that is a number, acts like the `item()` method would when invoked with that argument.

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

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

2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.

3. If the token indicated by *token* is one of the tokens in the object's underlying string then return true and stop this algorithm.

4. Otherwise, return false.

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

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

2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.

3. If the given *token* is already one of the tokens in the DOMTokenList object's underlying string then stop the algorithm.

4. Otherwise, if the last character of the DOMTokenList object's underlying string is not a space character, then append a U+0020 SPACE character to the end of that string.

5. Append the value of *token* to the end of the DOMTokenList object's underlying string.

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

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

2. Otherwise, remove the given *token* from the underlying string.

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

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

2. Otherwise, split the underlying string on spaces to get the list of tokens in the object's underlying string.

3. If the given *token* is already one of the tokens in the DOMTokenList object's underlying string then remove the given *token* from the underlying string, and stop the algorithm, returning false.

4. Otherwise, if the last character of the DOMTokenList object's underlying string is not a space character, then append a U+0020 SPACE character to the end of that string.

5. Append the value of *token* to the end of the DOMTokenList object's underlying string.

6. Return true.

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

2.3.3. DOM feature strings

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

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

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

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

2.4. DOM tree accessors

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

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

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

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

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

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

The **getElementsByName(*name*)** method a string *name*, and must return a live `NodeList`
containing all the `a`, `applet`, `button`, `form`, `iframe`, `img`, `input`, `map`, `meta`, `object`, `select`,
and `textarea` elements in that document that have a `name` attribute whose value is equal to the
name argument.

The **getElementsByClassName(*classNames*)** method takes 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 labelled
as being in specific classes. UAs must not assume that all attributes of the name `class` for
elements in any namespace work in this way, however, and must not assume that such attributes,
when used as global attributes, label other elements as being in specific classes.

> Given the following XHTML fragment:
>
> ```
> <div id="example">
>  <p id="p1" class="aaa bbb"/>
>  <p id="p2" class="aaa ccc"/>
>  <p id="p3" class="bbb ccc"/>
> </div>
> ```

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.

2.5. Dynamic markup insertion

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

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

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

Regardless of the parsing mode, the **`document.writeln(...)`** 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).

2.5.1. Controlling the input stream

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

When called with two or fewer arguments, the method must act as follows:

1. Let *type* be the value of the first argument, if there is one, or "`text/html`" otherwise.

2. Let *replace* be true if there is a second argument and it has the value "replace", and false otherwise.

3. If the document has an active parser that isn't a script-created parser, and the insertion point associated with that parser's input stream is not undefined (that is, it *does* point to somewhere in the input stream), then the method does nothing. Abort these steps 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

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. 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).

8. Mark the document as being an HTML document (it might already be so-marked).

9. If *type* does not have the value "`text/html`", then act as if the tokeniser had emitted a `pre` element start tag, then set the HTML parser's tokenisation stage's content model flag to *PLAINTEXT*.

10. 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.)

11. Finally, set the insertion point to point at just before the end of the input stream (which at this point will be empty).

12. 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.

2.5.2. 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 script that will execute as soon as the parser resumes, then the method must now return without further processing of the input stream.

4. Otherwise, the tokeniser must process the characters that were inserted, one at a time, processing resulting tokens as they are emitted, and stopping when the tokeniser reaches the insertion point or when the processing of the tokeniser is aborted by the tree construction stage (this can happen if a `script` start tag token is emitted by the tokeniser).

   Note: If the `document.write()` method was called from script executing inline (i.e. executing because the parser parsed a set of `script` tags), then this is a reentrant invocation of the parser.

5. Finally, the method must return.

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

On getting, the `innerHTML` DOM attribute must return the result of running the HTML fragment serialisation 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* 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.

2.5.3. 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 and `HTMLDocument`s, on getting, must return a string in the form of an [internal general parsed entity](#) that is XML namespace-well-formed, the string being an isomorphic serialisation of all of that node's child nodes, in document order. User agents may adjust prefixes and namespace declarations in the serialisation (and indeed might be forced to do so in some cases to obtain namespace-well-formed XML). [XML] [XMLNS]

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

- 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 (":").

- A `Text` 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 unserialisable. 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 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.

2.6. APIs in HTML documents

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

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

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

These attributes return tag names in all uppercase 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. Semantics and structure of HTML elements

3.1. Introduction

This section is non-normative.

An introduction to marking up a document.

3.2. Common microsyntaxes

There are various places in HTML that accept particular data types, such as dates or numbers. This

section describes what the conformance criteria for content in those formats is, and how to parse them.

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

3.2.1. Common parser idioms

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

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

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

1. Let *input* and *position* be the same variables as those of the same name in the algorithm that invoked these steps.

2. Let *result* be the empty string.

3. While *position* doesn't point past the end of *input* and the character at *position* is one of the *characters*, append that character to the end of *result* and advance *position* to the next character in *input*.

4. Return *result*.

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

3.2.2. 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 the attribute's canonical name, exactly, with no leading or trailing whitespace, and in lowercase.

3.2.3. Numbers

3.2.3.1. 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*.

3.2.3.2. 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*.

3.2.3.3. 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*.

3.2.3.4. 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 guarenteed to be a valid floating point number).

6. Return *number*.

3.2.3.5. Percentages and dimensions

<div style="border: 2px solid red;">

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)

</div>

3.2.3.6. Lists of integers

A **valid list of integers** is a number of <u>valid integers</u> separated by U+002C COMMA characters, with no other characters (e.g. no <u>space characters</u>). 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 U+002C COMMA character or a U+0020 SPACE 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+002C COMMA character or a U+0020 SPACE character, return to step 4.

7. Let *negated* be false.

8. Let *value* be 0.

9. Let *multiple* be 1.

10. Let *started* be false.

11. Let *finished* be false.

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 *finished* is true, skip to the next step in the overall set of steps.
    >
    >   2. If *started* is true or if *bogus* is true, let *negated* be false.
    >
    >   3. Otherwise, if *started* is false and if *bogus* is false, let *negated* be true.
    >
    >   4. 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. Let *n* be the value of the digit, interpreted in base ten, multiplied by *multiple*.

3. Add *n* to *value*.

4. If *value* is greater than zero, multiply *multiple* by ten.

5. Let *started* be true.

↪ **A U+002C COMMA character**
↪ **A U+0020 SPACE character**

Follow these substeps:

1. If *started* is false, return the *numbers* list and abort.

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. Let *finished* be true.

↪ **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 *started* is true, then append *value* to the *numbers* list, return that list, and abort.

18. Return the *numbers* list and abort.

3.2.4. 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]

3.2.4.1. 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 $0 \le h \le 23$, the minute values (m) in the range $0 \le m \le 59$, and the second value (s) being in the range $0 \le h < 60$. [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 a 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*.

3.2.4.2. 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 and a timezone. Even if the algorithm returns one or more values, the string can still be invalid.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9. Let *maxday* be the [number of days in month *month* of year *year*](#).

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

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

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

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

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

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

16. For the "in content" variant: [skip Zs characters](#); for the "in attributes" variant: [skip whitespace](#).

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

18. For the "in content" variant: [skip Zs characters](#); for the "in attributes" variant: [skip whitespace](#).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    1. For the "in content" variant: [skip Zs characters](#); for the "in attributes" variant: [skip whitespace](#).

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

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

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

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

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

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

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

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

        3. [Collect a sequence of characters](#) in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9). If the collected sequence is not exactly two characters long, then the string is invalid.

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

        5. If $timezone_{hours}$ is not a number in the range 0 ≤ $timezone_{hours}$ ≤ 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).

3.2.5. Time offsets

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

3.2.6. 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.

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. <u>Collect a sequence of characters</u> that are not <u>space characters</u>.

2. Add the string collected in the previous step to *tokens*.

3. <u>Skip whitespace</u>

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 <u>space characters</u>.

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 <u>space character</u>:

    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. <u>Collect a sequence of characters</u> that are not <u>space characters</u>, and let that be *s*.

8. If *s* is exactly equal to *token*, then:

    1. <u>Skip whitespace</u> (in *input*).

    2. Remove any <u>space characters</u> currently at the end of *output*.

    3. If *position* is not past the end of *input*, and *output* is not the empty string, append a single U+0020 SPACE character at the end of *output*.

9. Otherwise, append *s* to the end of *output*.

10. Return to step 6 in the overall set of steps.

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

3.2.7. 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 simply 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.*

3.2.8. References

A **valid hashed ID 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 id attribute of an element in the document with type *type*.

The **rules for parsing a hashed ID 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*.

3.3. Documents and document fragments

3.3.1. Semantics

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

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

> For example, the following document is non-conforming, despite being syntactically correct:
>
> ```
> <!DOCTYPE html>
> ```

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

...because the data placed in the cells is clearly not tabular data. A corrected version of this document might be:

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

All the elements in this specification have a defined content model, which describes what nodes are allowed inside the elements, and thus what the structure of an HTML document or fragment must look like. Authors must only put elements inside an element if that element allows them to be there according to its content model.

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

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

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

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

> The SVG specification defines the SVG `foreignObject` element as allowing foreign namespaces to be included, thus allowing compound documents to be created by inserting subdocument content under that element. *This* specification defines the XHTML `html` element as being allowed where subdocument fragments are allowed in a compound document. Together, these two definitions mean that placing an XHTML `html` element as a child of an SVG `foreignObject` element is conforming.

3.3.3. Kinds of 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
- Prose 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.

3.3.3.1. Metadata content

Metadata content is content that sets up the presentation or behaviour 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.

3.3.3.2. Prose content

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

As a general rule, elements whose content model allows any prose 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.

3.3.3.3. 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.

3.3.3.4. 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).

3.3.3.5. 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 prose content. Any content model that expects prose 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 categorised as phrasing content can only contain elements that are themselves categorised as phrasing content, not any prose content.*

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

3.3.3.6. 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 prose content). Any content model that expects phrasing content (or prose 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.

3.3.3.7. 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.*

3.3.4. 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 only contain content that would still 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 prose content.

3.3.5. Paragraphs

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

Paragraphs in prose 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 <em>first</em> paragraph in this example.
  <p>This is the second.</p>
  <!-- This is not a paragraph. -->
</section>
```

The following example takes that markup and puts `ins` and `del` elements around some of the markup to show that the text was changed (though in this case, the changes don't really make much sense, admittedly). Notice how this example has exactly the same paragraphs as the previous one, despite the `ins` and `del` elements.

```
<section>
  <ins><h1>Example of paragraphs</h1>
  This is the <em>first</em> paragraph in</ins> this example<del>.
  <p>This is the second.</p></del>
  <!-- This is not a paragraph. -->
</section>
```

3.4. Global attributes

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

Global attributes:

> class
> contenteditable
> contextmenu
> dir
> draggable
> id
> irrelevant
> lang
> ref
> registrationmark
> tabindex
> template
> title

In addition, the following <u>event handler content attributes</u> 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
onsubmit
onunload
```

3.4.1. The `id` attribute

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

3.4.2. The `title` attribute

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

If this attribute is omitted from an element, then it implies that the `title` attribute of the nearest ancestor 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 `dfn`, define additional semantics for the `title` attribute beyond the semantics described above.

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

3.4.3. The `lang` (HTML only) and `xml:lang` (XML only) attributes

The `lang` attribute specifies the primary **language** for the element's contents and for any of the element's attributes that contain text. Its value must be a valid RFC 3066 language code, or the empty string. [RFC3066]

The `xml:lang` attribute is defined in XML. [XML]

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

that the primary language is unknown.

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

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

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 <u>HTML element</u> 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 <u>ignored</u> for the purposes of determining the element's language.

If no explicit language is given for the <u>root element</u>, then language information from a higher-level protocol (such as HTTP), if any, must be used as the final fallback language. In the absence of any language information, the default value is unknown (the empty string).

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

The **lang** DOM attribute must <u>reflect</u> the `lang` content attribute.

3.4.4. The `dir` attribute

The `dir` attribute specifies the element's text directionality. The attribute is an <u>enumerated attribute</u> 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.

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 <u>reflect</u> the `dir` content attribute of that element, <u>limited to only known values</u>.

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

3.4.5. The `class` attribute

Every <u>HTML element</u> may have a `class` attribute specified.

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

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

> 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.

3.4.6. 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.

3.5. Interaction

3.5.1. Activation

The **click()** method must <u>fire a `click` event</u> 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 <u>activation behavior</u>; the first element, in reverse tree order, to have one, must have its activation behavior executed.

3.5.2. Focus

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

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

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

3.5.2.1. Focus management

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

Some elements, most notably `area`, can correspond to more than one distinct focusable area. When such an element is focused using the `focus()` method, the first such region in tree order is the one that must be focused.

> Well that clearly needs more.

The **activeElement** attribute must return the element in the document that has focus. If no element specifically has focus, this must return <u>the `body` element</u>.

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

3.5.2.2. Sequential focus navigation

The **tabindex** attribute specifies the relative order of elements for the purposes of sequential focus navigation. The name "tab index" comes from the common use of the "tab" key to navigate through the focusable elements. The term "tabbing" refers to moving forward through the focusable elements.

The `tabindex` attribute, if specified, must have a value that is a <u>valid integer</u>.

If the attribute is specified, it must be parsed using the <u>rules for parsing integers</u>. If parsing the value returns an error, the attribute is ignored for the purposes of focus management (as if it wasn't specified).

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

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

If the attribute is absent (or invalid), then the user agent must treat the element as if it had the value

0 or the value -1, based on platform conventions.

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

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

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

3.5.3. 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 must instead simply align the top of the element with the top of the viewport.

Non-visual user agents may ignore the argument, or may treat it in some media-specific manner most useful to the user.

3.6. The root element

3.6.1. 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:
> No difference from `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 URI (or IRI).

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). Furthermore, as it is processed before any base elements are seen, its value is not subject to being made relative to any base URI.

Though it has absolutely no effect and no meaning, the html element, in [HTML documents](#), may have an xmlns attribute specified, if, and only if, it has the exact value "http://www.w3.org /1999/xhtml". This does not apply to [XML documents](#).

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

> *Note: In XML, an xmlns attribute is part of the namespace declaration mechanism, and an element cannot actually have an xmlns attribute in the null namespace specified.*

3.7. Document metadata

3.7.1. 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:

> No difference from HTMLElement.

The head element collects the document's metadata.

3.7.2. 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:

    No difference from HTMLElement.

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

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.

3.7.3. 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's base URI for the purposes of resolving relative URIs, 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 URI (or IRI).

A `base` element, if it has an `href` attribute, must come before any other elements in the tree that have attributes with URIs (except the `html` element and its `manifest` attribute).

User agents must use the value of the `href` attribute of the first `base` element that is both a child of [the `head` element](#) and has an `href` attribute, if there is such an element, as the document entity's base URI for the purposes of section 5.1.1 of RFC 3986 ("Establishing a Base URI": "Base URI Embedded in Content"). This base URI from RFC 3986 is referred to by the algorithm given in XML Base, which [is a normative part of this specification](#). [RFC3986]

If the base URI given by this attribute is a relative URI, it must be resolved relative to the higher-level base URIs (i.e. the base URI from the encapsulating entity or the URI used to retrieve the entity) to obtain an absolute base URI. All `xml:base` attributes must be ignored when resolving relative URIs in this `href` attribute.

> *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](#). 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](#).

The **`href`** and **`target`** DOM attributes must [reflect](#) the content attributes of the same name.

3.7.4. 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
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;
};
```

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 URI (or IRI). If the `href` attribute is absent, then the element does not define a link.

The type of link indicated (the relationship) is given by the value of the **rel** attribute, which must be present, 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). User agents should process the links on a per-link basis, not a per-element basis.

The exact behaviour for links to external resources depends on the exact relationship, as defined for the relevant link type. Some of the attributes control whether or not the external resource is to be applied (as defined below). For external resources that are represented in the DOM (for example, style sheets), the DOM representation must be made available even if the resource is not applied. (However, user agents may opt to only fetch such resources when they are needed, instead of pro-actively downloading all the external resources that are not applied.)

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

- The relationship between this document and the resource (given by the `rel` attribute)

- The title of the resource (given by the `title` attribute).

- The URI of the resource (given by the `href` attribute).

- The language of the resource (given by the `hreflang` attribute).

- The optimum media for the resource (given by the `media` attribute).

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

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

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

However, if the link is an external resource link, then the `media` attribute is prescriptive. The user agent must only apply the external resource to views while their state match the listed media.

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

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

The **type** attribute gives the MIME type of the linked resource. It is purely advisory. The value must be a valid MIME type, optionally with parameters. [RFC2046]

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

User agents must not consider the `type` attribute authoritative — upon fetching the resource, user agents must not use metadata included in the link to the resource to determine its type.

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

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

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

Some versions of HTTP defined a `Link:` header, to be processed like a series of `link` elements. When processing links, those must be taken into consideration as well. For the purposes of ordering,

links defined by HTTP headers must be assumed to come before any links in the document, in the order that they were given in the HTTP entity header. Relative URIs in these headers must be resolved according to the rules given in HTTP, not relative to base URIs set by the document (e.g. using a base element or xml:base attributes). [RFC2616] [RFC2068]

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

The DOM attribute **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.

3.7.5. The meta element

Categories

> Metadata content.

Contexts in which this element may be used:

> If the charset attribute is present: as the first element in a head element.
> If the http-equiv attribute is present: in a head element.
> If the http-equiv attribute is present: 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 serialised 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 may only be specified in HTML documents, 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.

3.7.5.1. Standard metadata names

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

generator

> The value must be a free-form string that identifies the software used to generate the document. This value must not be used on hand-authored pages. WYSIWYG editors have additional constraints on the value used with this metadata name.

dns

> The value must be an ordered set of unique space-separated tokens, each word of which is a host name. The list allows authors to provide a list of host names that the user is expected to subsequently need. User agents may, according to user preferences and prevailing network conditions, pre-emptively resolve the given DNS names (extracting the names from the value using the rules for splitting a string on spaces), thus precaching the DNS information for those hosts and potentially reducing the time between page loads for subsequent user interactions. Higher priority should be given to host names given earlier in the list.

3.7.5.2. Other metadata names

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

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

Keyword

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

Brief description

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

Link to more details

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

Synonyms

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

Status

> One of the following:

> **Proposal**

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

> **Accepted**

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

> **Unendorsed**

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

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

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

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

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

3.7.5.3. 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 the rows with keywords give the states to which those keywords map.

State	Keywords
Refresh	refresh

State	Keywords
Default style	`default-style`

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:

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 URI using the base URI of the `meta` element.

22. Set a timer so that in *time* seconds, if the user has not canceled the redirect, the user agent navigates to *url*, with replacement enabled.

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 URI (or IRI).

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 URI.

Default style state

1. ...

3.7.5.4. Specifying the document's character encoding

The `meta` element may also be used to provide UAs with character encoding information for HTML files, by setting the **charset** attribute to the name of a character encoding. This is called a character encoding declaration.

The following restrictions apply to character encoding declarations:

- The character encoding name given must be the name of the character encoding used to serialise the file.

- The value must be a valid character encoding name, and must be the preferred name for that encoding. [IANACHARSET]

- The attribute value must be serialised without the use of character entity references 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 a superset of US-ASCII (specifically, ANSI_X3.4-1968) for bytes in the range 0x09 - 0x0D, 0x20, 0x21, 0x22, 0x26, 0x27, 0x2C - 0x3F, 0x41 - 0x5A, and 0x61 - 0x7A , 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.

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.

3.7.6. The `style` element

Categories

> Metadata content.
> If the `scoped` attribute is present: prose 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 prose 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 only apply the styles to views while their state match the listed media. [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 only apply the specified style information 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.

If the scoped attribute is not specified, the style element must be the child of a head element or of a noscript element that is a child of a head element.

If the scoped attribute *is* specified, then the style element must be the child of a [prose content]() element, before any text nodes other than [inter-element whitespace](), and before any elements other than other style elements.

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]().

3.7.7. 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 <u>external resource links that contribute to the styling processing model</u> (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: <u>[CSSOM]</u>

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 <u>Content-Type metadata of the specified resource</u>.

The location (`href` DOM attribute)

> For `link` elements, the location must be the URI given by the element's `href` content attribute. 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.

3.8. 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 <u>sectioning content</u> element *e* are all the *x* elements whose nearest <u>sectioning content</u> ancestor is *e*.

3.8.1. The `body` element

Categories

> <u>Sectioning content</u>.

Contexts in which this element may be used:

> As the second element in an `html` element.

Content model:

> <u>Prose content</u>.

Element-specific attributes:

> None.

DOM interface:

> No difference from `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.*

3.8.2. The `section` element

Categories
> [Prose content](#).
> [Sectioning content](#).

Contexts in which this element may be used:
> Where [prose content](#) is expected.

Content model:
> [Prose content](#).

Element-specific attributes:
> None.

DOM interface:
> No difference from `HTMLElement`.

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

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

3.8.3. The `nav` element

Categories
> [Prose content](#).
> [Sectioning content](#).

Contexts in which this element may be used:
> Where [prose content](#) is expected.

Content model:
> [Prose content](#).

Element-specific attributes:
> None.

DOM interface:
> No difference from `HTMLElement`.

The `nav` element represents a section of a page that links to other pages or to parts within the page: a section with navigation links.

3.8.4. The `article` element

Categories

> [Prose content](#).
> [Sectioning content](#).

Contexts in which this element may be used:

> Where [prose content](#) is expected.

Content model:

> [Prose content](#).

Element-specific attributes:

> None.

DOM interface:

> No difference from [HTMLElement](#).

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

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

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

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

3.8.5. The `blockquote` element

Categories

> [Prose content](#).
> [Sectioning content](#).

Contexts in which this element may be used:

> Where [prose content](#) is expected.

Content model:

> [Prose 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 URI, if it has one, should be cited in the **cite** attribute.

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

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

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

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

3.8.6. The `aside` element

Categories

Prose content.
Sectioning content.

Contexts in which this element may be used:

Where prose content is expected.

Content model:

Prose content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

Categories

Prose content.
Heading content.

Contexts in which this element may be used:

Where prose content is expected.

Content model:

Phrasing content.

Element-specific attributes:

None.

DOM interface:

No difference from HTMLElement.

These elements define headers for their sections.

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

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

3.8.8. The `header` element

Categories

Prose content.
Heading content.

Contexts in which this element may be used:

Where prose content is expected.

Content model:

Prose 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:

No difference from HTMLElement.

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

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.

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>
     <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>
```

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

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

3.8.9. The footer element

Categories

Prose content.

Contexts in which this element may be used:

Where prose content is expected.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from HTMLElement.

The `footer` element represents the footer for the section it applies to. A footer typically contains information about its section such as who wrote it, links to related documents, copyright data, and the like.

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

3.8.10. The `address` element

Categories

> Prose content.

Contexts in which this element may be used:

> Where prose content is expected.

Content model:

> Prose 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:

> No difference from `HTMLElement`.

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

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

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

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

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

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

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

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

3.8.11. Headings and sections

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

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

> For the following fragment:
>
> ```
> <body>
>  <h1>Foo</h1>
>  <h2>Bar</h2>
>  <blockquote>
>   <h3>Bla</h3>
>  </blockquote>
>  <p>Baz</p>
>  <h2>Quux</h2>
>  <section>
>   <h3>Thud</h3>
>  </section>
>  <p>Grunt</p>
> </body>
> ```
>
> ...the structure would be:
>
> 1. Foo (heading of explicit `body` section)
>   1. Bar (heading starting implied section)
>   1. Bla (heading of explicit `blockquote` section)
>   Baz (paragraph)
>   2. Quux (heading starting implied section)
>   3. Thud (heading of explicit `section` section)
>   Grunt (paragraph)
>
> Notice how the `blockquote` nests inside an implicit section while the `section` does not (and in fact, ends the earlier implicit section so that a later paragraph is back at the top level).

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

Authors are also encouraged to explictly wrap sections in 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.

3.8.11.1. Creating an outline

This section will be rewritten at some point. The algorithm likely won't change, but its description will be dramatically simplified.

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

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

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

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

UAs must take this hypothetical tree (which will become the outline) and mutate it by walking it depth first in tree order and, for each element of heading content that is not the first element of its parent

sectioning content element, inserting a new element of sectioning content, as follows:

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

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

↳ **Otherwise:**

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

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

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

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

> Selecting the first section of the document therefore always takes the user to the top of the document, regardless of where the first header in the `body` is to be found.

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

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

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

  // Skip any descendants of headings.
  if ((n.parentNode && n.parentNode.namespaceURI ==
'http://www.w3.org/1999/xhtml') &&
      (n.parentNode.localName == 'h1' || n.parentNode.localName
== 'h2' ||
       n.parentNode.localName == 'h3' || n.parentNode.localName
== 'h4' ||
       n.parentNode.localName == 'h5' || n.parentNode.localName
== 'h6' ||
```

```
           n.parentNode.localName == 'header'))
        return NodeFilter.FILTER_REJECT;

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

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

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

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

> This section will be rewritten at some point. The algorithm likely won't change, but its description will be dramatically simplified.

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

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

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

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

4. If the node is an element of <u>sectioning content</u>, then the associated section is itself. The UA must then generate the <u>hypothetical section tree</u> described in the previous section, rooted at the section itself. If the section element, in that hypothetical tree, has a child element that is

an `h1`-`h6` element or a `header` element, then that element is the associated heading. Otherwise, there is no associated heading element.

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

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

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

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

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

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

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

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

Note: Not all nodes have an associated header or section. For example, if a section is implied, as when multiple headers are found in one element of sectioning content, then a node in that section has an anonymous associated section (its section is not

represented by a real element), and the algorithm above does not associate that node with any particular element of <u>sectioning content</u>.

For the following fragment:

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

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

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

3.8.11.3. Distinguishing site-wide headers from page headers

Given the <u>hypothetical section tree</u>, but ignoring any sections created for `nav` and `aside` elements, and any of their descendants, if the root of the tree is <u>the `body` element</u>'s section, and it has only a single subsection which is created by an `article` element, then the header of <u>the `body` element</u> should be assumed to be a site-wide header, and the header of the `article` element should be assumed to be the page's header.

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 <u>hypothetical section tree</u>, ignoring any sections created for `nav` and `aside` elements and any of their descendants, the root of the tree is <u>the `body` element</u>'s section, its heading is the site-wide heading, <u>the `body` element</u> has just one subsection, that subsection is created by an `article` element, and that `article`'s header 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 <u>hypothetical section tree</u>, ignoring any sections created for `nav` and `aside` elements and any of their descendants, either <u>the `body` element</u> 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.

> *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.*

3.9. Prose

3.9.1. The `p` element

Categories
> Prose content.

Contexts in which this element may be used:
> Where prose content is expected.

Content model:
> Phrasing content.

Element-specific attributes:
> None.

DOM interface:
> No difference from `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>
```

3.9.2. The `hr` element

Categories

[Prose content](#).

Contexts in which this element may be used:

Where [prose content](#) is expected.

Content model:

Empty.

Element-specific attributes:

None.

DOM interface:

No difference from HTMLElement.

The `hr` element represents a [paragraph](#)-level thematic break, e.g. a scene change in a story, or a transition to another topic within a section of a reference book.

3.9.3. The `br` element

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:

> No difference from `HTMLElement`.

The `br` element represents a line break.

`br` elements must be empty. Any content inside `br` elements must not be considered part of the surrounding text.

`br` elements must only be used for line breaks that are actually part of the content, as in poems or addresses.

> The following example is correct usage of the `br` element:
>
> ```
> <p>P. Sherman<br>
> 42 Wallaby Way<br>
> Sydney</p>
> ```

`br` elements must not be used for separating thematic groups in a paragraph.

> The following examples are non-conforming, as they abuse the `br` element:
>
> ```
> <p><a ...>34 comments.</a><br>
> <a ...>Add a comment.<a></p>
>
> <p>Name: <input name="name"><br>
> Address: <input name="address"></p>
> ```
>
> Here are alternatives to the above, which are correct:
>
> ```
> <p><a ...>34 comments.</a></p>
> <p><a ...>Add a comment.<a></p>
>
> <p>Name: <input name="name"></p>
> <p>Address: <input name="address"></p>
> ```

If a [paragraph](#) consists of nothing but a single `br` element, it represents a placeholder blank line (e.g. as in a template). Such blank lines must not be used for presentation purposes.

3.9.4. The `dialog` element

Categories

> [Prose content](#).

Contexts in which this element may be used:

> Where [prose content](#) is expected.

Content model:

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

Element-specific attributes:

> None.

DOM interface:

> No difference from `HTMLElement`.

The `dialog` element represents a conversation.

Each part of the conversation must have an explicit talker (or speaker) given by a <u>dt</u> element, and a discourse (or quote) given by a <u>dd</u> 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 <u>dt</u> element in a <u>dialog</u> element is implicitly the source of the text given in the following <u>dd</u> element, and the contents of the <u>dd</u> element are implicitly a quote from that speaker. There is thus no need to include <u>cite</u>, <u>q</u>, or <u>blockquote</u> elements in this markup. Indeed, a <u>q</u> element inside a <u>dd</u> element in a conversation would actually imply the people talking were themselves quoting someone else. See the <u>cite</u>, <u>q</u>, and <u>blockquote</u> elements for other ways to cite or quote.

3.10. Preformatted text

3.10.1. The `pre` element

Categories
> [Prose content](#).

Contexts in which this element may be used:
> Where [prose content](#) is expected.

Content model:
> [Phrasing content](#).

Element-specific attributes:
> None.

DOM interface:
> No difference from `HTMLElement`.

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

Some examples of cases where the <u>pre</u> 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>
```

3.11. Lists

3.11.1. The `ol` element

Categories

> Prose content.

Contexts in which this element may be used:

> Where prose content is expected.

Content model:

> Zero or more `li` elements.

Element-specific attributes:

> `start`

DOM interface:

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

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

The **start** attribute, if present, must 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.

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

The first item in the list has the ordinal value given by the `ol` element's `start` attribute, unless 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.

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

3.11.2. The `ul` element

Categories

> Prose content.

Contexts in which this element may be used:

> Where prose content is expected.

Content model:

> Zero or more `li` elements.

Element-specific attributes:

> None.

DOM interface:

> No difference from HTMLElement.

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

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

3.11.3. The li element

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: prose 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 first list item.

If the value attribute is present, user agents must parse it as an integer, in order to determine the attribute's value. If the attribute's value cannot be converted to a number, 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.

3.11.4. The dl element

Categories

> Prose content.

Contexts in which this element may be used:

> Where prose content is expected.

Content model:

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

Element-specific attributes:

> None.

DOM interface:

> No difference from HTMLElement.

The dl element introduces an unordered association list consisting of zero or more name-value groups (a description list). Each group must consist of one or more names (dt elements) followed by one or more values (dd elements).

Name-value groups may be terms and definitions, metadata topics and values, or any other groups of name-value data.

> The following are all conforming HTML fragments.
>
> In the following example, one entry ("Authors") is linked to two values ("John" and "Luke").
>
> ```
> <dl>
>  <dt> Authors
>  <dd> John
>  <dd> Luke
>  <dt> Editor
>  <dd> Frank
> </dl>
> ```
>
> In the following example, one definition is linked to two terms.
>
> ```
> <dl>
>  <dt lang="en-US"> <dfn>color</dfn> </dt>
>  <dt lang="en-GB"> <dfn>colour</dfn> </dt>
>  <dd> A sensation which (in humans) derives from the ability of
>  the fine structure of the eye to distinguish three differently
>  filtered analyses of a view. </dd>
> </dl>
> ```
>
> The following example illustrates the use of the dl element to mark up metadata of sorts. At the end of the example, one group has two metadata labels ("Authors" and "Editors") and two values ("Robert Rothman" and "Daniel Jackson").
>
> ```
> <dl>
>  <dt> Last modified time </dt>
>  <dd> 2004-12-23T23:33Z </dd>
>  <dt> Recommended update interval </dt>
>  <dd> 60s </dd>
>  <dt> Authors </dt>
>  <dt> Editors </dt>
>  <dd> Robert Rothman </dd>
>  <dd> Daniel Jackson </dd>
> </dl>
> ```

If a dl element is empty, it contains no groups.

If a `dl` element contains non-whitespace text nodes, or elements other than `dt` and `dd`, then those elements or text nodes do not form part of any groups in that `dl`, and the document is non-conforming.

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

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

> *Note: The `dl` element is inappropriate for marking up dialogue, since dialogue is ordered (each speaker/line pair comes after the next). For an example of how to mark up dialogue, see the `dialog` element.*

3.11.5. The `dt` element

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:
> No difference from `HTMLElement`.

The `dt` element represents the term, or name, part of a term-description group in a description list (`dl` element), and the talker, or speaker, part of a talker-discourse pair in a conversation (`dialog` element).

> *Note: The `dt` element itself, when used in a `dl` element, does not indicate that its contents are a term being defined, but this can be indicated using the `dfn` element.*

3.11.6. The `dd` element

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:
> Prose content.

Element-specific attributes:
> None.

DOM interface:

No difference from `HTMLElement`.

The `dd` element represents the description, definition, or value, part of a term-description group in a description list (`dl` element), and the discourse, or quote, part in a conversation (`dialog` element).

3.12. Phrase elements

3.12.1. 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:

```
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.

3.12.2. 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 a part of a paragraph quoted from another source.

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

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

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

3.12.3. The cite element

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:

> No difference from HTMLElement.

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

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

This is incorrect usage:

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

This is the correct way to do it:

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

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

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

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

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

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

3.12.4. The `em` element

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:

No difference from `HTMLElement`.

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

The level of emphasis that a particlar piece of content has is given by its number of ancestor `em` elements.

The placement of emphasis changes the meaning of the sentence. The element thus forms an integral part of the content. The precise way in which emphasis is used in this way depends on the language.

These examples show how changing the emphasis changes the meaning. First, a general statement of fact, with no emphasis:

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

By emphasising the first word, the statement implies that the kind of animal under discussion is in question (maybe someone is asserting that dogs are cute):

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

Moving the emphasis to the verb, one highlights that the truth of the entire sentence is in question (maybe someone is saying cats are not cute):

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

By moving it to the adjective, the exact nature of the cats is reasserted (maybe someone suggested cats were *mean* animals):

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

Similarly, if someone asserted that cats were vegetables, someone correcting this might emphasise the last word:

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

By emphasising the entire sentence, it becomes clear that the speaker is fighting hard to get the point across. This kind of emphasis also typically affects the punctuation, hence the exclamation mark here.

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

Anger mixed with emphasising the cuteness could lead to markup such as:

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

3.12.5. 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:

> No difference from [HTMLElement](#).

The `strong` element represents strong importance for its contents.

The relative level of importance of a piece of content is given by its number of ancestor `strong` elements; each `strong` element increases the importance of its contents.

Changing the importance of a piece of text with the `strong` element does not change the meaning of the sentence.

> Here is an example of a warning notice in a game, with the various parts marked up according to how important they are:
>
> ```
> <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>
> ```

3.12.6. 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:

No difference from HTMLElement.

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

Note: The small element does not "de-emphasise" or lower the importance of text emphasised by the em element or marked as important with the strong element.

In this example the footer contains contact information and a copyright.

```
<footer>
 <address>
  For more details, contact
  <a href="mailto:js@example.com">John Smith</a>.
 </address>
 <p><small>© copyright 2038 Example Corp.</small></p>
</footer>
```

In this second example, the small element is used for a side comment.

```
<p>Example Corp today announced record profits for the
second quarter <small>(Full Disclosure: Foo News is a subsidiary of
Example Corp)</small>, leading to speculation about a third quarter
merger with Demo Group.</p>
```

In this last example, the small element is marked as being *important* small print.

```
<p><strong><small>Continued use of this service will result in a
kiss.</small></strong></p>
```

3.12.7. The m 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:

No difference from HTMLElement.

> This section has a large number of outstanding comments and will likely be rewritten or removed from the spec.

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

> In the following snippet, a paragraph of text refers to a specific part of a code fragment.
>
> ```
> <p>The highlighted part below is where the error lies:</p>
> <pre><code>var i: Integer;
> begin
>    i := <m>1.1</m>;
> end.</code></pre>
> ```
>
> Another example of the m element is highlighting parts of a document that are matching some search string. If someone looked at a document, and the server knew that the user was searching for the word "kitten", then the server might return the document with one paragraph modified as follows:
>
> ```
> <p>I also have some <m>kitten</m>s who are visiting me
> these days. They're really cute. I think they like my garden!</p>
> ```

3.12.8. The `dfn` element

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:

> No difference from HTMLElement.

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

Defining term: If the dfn element has a **title** attribute, then the exact value of that attribute is the term being defined. Otherwise, if it contains exactly one element child node and no child text nodes, and that child element is an abbr element with a title attribute, then the exact value of *that* attribute is the term being defined. Otherwise, it is the exact textContent of the dfn element that gives the term being defined.

If the title attribute of the dfn element is present, then it must only contain the term being defined.

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

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

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

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

3.12.9. 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:

> No difference from `HTMLElement`.

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

> The paragraph below contains an abbreviation marked up with the `abbr` element.
>
> ```
> <p>The <abbr title="Web Hypertext Application Technology
> Working Group">WHATWG</abbr> is a loose unofficial collaboration of
> Web browser manufacturers and interested parties who wish to
> develop
> new technologies designed to allow authors to write and deploy
> Applications over the World Wide Web.</p>
> ```

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

> In the example below, the word "Zat" is used as an abbreviation in the second paragraph. The

abbreviation is defined in the first, so the explanatory `title` attribute has been omitted. Because of the way `dfn` elements are defined, the second `abbr` element in this example would be connected (in some UA-specific way) to the first.

```
<p>The <dfn><abbr>Zat</abbr></dfn>, short for Zat'ni'catel, is a
weapon.</p>
<p>Jack used a <abbr>Zat</abbr> to make the boxes of evidence
disappear.</p>
```

3.12.10. 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 <u>date</u>; if *result* contains a time, then that is the <u>time</u>; and if *result* contains a timezone, then the timezone is the element's <u>timezone</u>. (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 <u>date</u> is unknown, and otherwise must return the time corresponding to midnight UTC (i.e. the first second) of the given <u>date</u>.

The **time** DOM attribute must return null if the <u>time</u> is unknown, and otherwise must return the time corresponding to the given <u>time</u> of 1970-01-01, with the timezone UTC.

The **timezone** DOM attribute must return null if the <u>timezone</u> is unknown, and otherwise must return the time corresponding to 1970-01-01 00:00 UTC in the given <u>timezone</u>, 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:
>
> ```
> <p>Our first date was <time datetime="2006-09-23">a
> saturday</time>.</p>
> ```
>
> ...the <u>time</u> element's <u>date</u> attribute would have the value 1,158,969,600,000ms, and the <u>time</u> and <u>timezone</u> attributes would return null.
>
> In the following snippet:
>
> ```
> <p>We stopped talking at <time datetime="2006-09-24 05:00 -7">5am
> the next morning</time>.</p>
> ```
>
> ...the <u>time</u> element's <u>date</u> attribute would have the value 1,159,056,000,000ms, the <u>time</u> attribute would have the value 18,000,000ms, and the <u>timezone</u> attribute would return -25,200,000ms. To obtain the actual time, the three attributes can be added together, obtaining 1,159,048,800,000, which is the specified date and time in UTC.
>
> Finally, in the following snippet:
>
> ```
> <p>Many people get up at <time>08:00</time>.</p>
> ```
>
> ...the <u>time</u> element's <u>date</u> attribute would have the value null, the <u>time</u> attribute would have the value 28,800,000ms, and the <u>timezone</u> attribute would return null.

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

3.12.11. The **progress** element

Categories
    <u>Phrasing content</u>.

Contexts in which this element may be used:
    Where <u>phrasing content</u> is expected.

Content model:
    <u>Phrasing content</u>.

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 simply include the current value and the maximum value inline as text inside the element.

> Here is a snippet of a Web application that shows the progress of some automated task:
>
> ```
> <section>
>  <h2>Task Progress</h2>
>  <p><label>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.

User agent requirements: User agents must parse the max and value attributes' values according to the rules for parsing floating point number values.

If the value attribute is omitted, then user agents must also parse the textContent of the progress element in question using the steps for finding one or two numbers of a ratio in a string. These steps will return nothing, one number, one number with a denominator punctuation character,

or two numbers.

Using the results of this processing, user agents must determine whether the progress bar is an indeterminate progress bar, or whether it is a determinate progress bar, and in the latter case, what its current and maximum values are, all as follows:

1. If the `max` attribute is omitted, and the `value` is omitted, and the results of parsing the `textContent` was nothing, then the progress bar is an indeterminate progress bar. Abort these steps.

2. Otherwise, it is a determinate progress bar.

3. If the `max` attribute is included, then, if a value could be parsed out of it, then the maximum value is that value.

4. Otherwise, if the `max` attribute is absent but the `value` attribute is present, or, if the `max` attribute is present but no value could be parsed from it, then the maximum is 1.

5. Otherwise, if neither attribute is included, then, if the `textContent` contained one number with an associated denominator punctuation character, then the maximum value is the value associated with that denominator punctuation character; otherwise, if the `textContent` contained two numbers, the maximum value is the higher of the two values; otherwise, the maximum value is 1.

6. If the `value` attribute is present on the element and a value could be parsed out of it, that value is the current value of the progress bar. Otherwise, if the attribute is present but no value could be parsed from it, the current value is zero.

7. Otherwise if the `value` attribute is absent and the `max` attribute is present, then, if the `textContent` was parsed and found to contain just one number, with no associated denominator punctuation character, then the current value is that number. Otherwise, if the `value` attribute is absent and the `max` attribute is present then the current value is zero.

8. Otherwise, if neither attribute is present, then the current value is the lower of the one or two numbers that were found in the `textContent` of the element.

9. If the maximum value is less than or equal to zero, then it is reset to 1.

10. If the current value is less than zero, then it is reset to zero.

11. Finally, if the current value is greater than the maximum value, then the current value is reset to the maximum value.

UA requirements for showing the progress bar: When representing a `progress` element to the user, the UA should indicate whether it is a determinate or indeterminate progress bar, and in the former case, should indicate the relative position of the current value relative to the maximum value.

The **max** and **value** DOM attributes must reflect the elements' content attributes of the same name. When the relevant content attributes are absent, the DOM attributes must return zero. The value parsed from the `textContent` never affects the DOM values.

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

If the progress bar is an indeterminate progress bar, then the **position** DOM attribute must return -1. Otherwise, it must return the result of dividing the current value by the maximum value.

3.12.12. 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 long value;
          attribute long min;
          attribute long max;
          attribute long low;
          attribute long high;
          attribute long optimum;
};
```

The meter element represents a scalar measurement within a known range, or a fractional value; for example disk usage, the relevance of a query result, or the fraction of a voting population to have selected a particular candidate.

This is also known as a gauge.

> *Note: The meter element should not be used to indicate progress (as in a progress bar). For that role, HTML provides a separate progress element.*

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

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

> The following examples all represent a measurement of three quarters (of the maximum of whatever is being measured):
>
> ```
> <meter>75%</meter>
> <meter>750‰</meter>
> <meter>3/4</meter>
> <meter>6 blocks used (out of 8 total)</meter>
> <meter>max: 100; current: 75</meter>
> <meter><object data="graph75.png">0.75</object></meter>
> <meter min="0" max="100" value="75"></meter>
> ```
>
> The following example is incorrect use of the element, because it doesn't give a range (and since the default maximum is 1, both of the gauges would end up looking maxed out):
>
> ```
> <p>The grapefruit pie had a radius of <meter>12cm</meter>
> and a height of <meter>2cm</meter>.</p> <!-- BAD! -->
> ```
>
> Instead, one would either not include the meter element, or use the meter element with a defined range to give the dimensions in context compared to other pies:
>
> ```
> <p>The grapefruit pie had a radius of 12cm and a height of
> 2cm.</p>
> <dl>
>  <dt>Radius: <dd> <meter min=0 max=20 value=12>12cm</meter>
>  <dt>Height: <dd> <meter min=0 max=10 value=2>2cm</meter>
> </dl>
> ```

There is no explicit way to specify units in the `meter` element, but the units may be specified in the `title` attribute in freeform 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>
>   <p><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></p>
> ```

```
    <p>Group description: <strong>Layout/presentation on the
  WWW.</strong></p>
    <p><meter value="0.5">Moderate activity,</meter> Usenet, 618
  subscribers</p>
   </li>
   <li>
    <p><a href="/group/netscape.public.mozilla.xpinstall
  /view">netscape.public.mozilla.xpinstall</a> -
        <a href="/group/netscape.public.mozilla.xpinstall
  /subscribe">join</a></p>
    <p>Group description: <strong>Mozilla XPInstall discussion.
  </strong></p>
    <p><meter value="0.25">Low activity,</meter> Usenet, 22
  subscribers</p>
   </li>
   <li>
    <p><a href="/group/mozilla.dev.general
  /view">mozilla.dev.general</a> -
        <a href="/group/mozilla.dev.general/subscribe">join</a></p>
    <p><meter value="0.25">Low activity,</meter> Usenet, 66
  subscribers</p>
   </li>
  </ul>
```

Might be rendered as follows:

With the <meter> elements rendered as inline green bars of varying lengths.

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...

3.12.13. 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, but the `title` attribute has special semantics on this element when used with the `dfn` element.

DOM interface:

    No difference from `HTMLElement`.

The `code` element represents a fragment of computer code. This could be an XML element name, a filename, a computer program, or any other string that a computer would recognise.

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 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 detais.

3.12.14. 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, but the `title` attribute has special semantics on this element when used with the `dfn` element.

DOM interface:

    No difference from `HTMLElement`.

The `var` element represents a variable. This could be an actual variable in a mathematical expression or programming context, or it could just be a term used as a placeholder in prose.

> In the paragraph below, the letter "n" is being used as a variable in prose:
>
> ```
> <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>
> ```

3.12.15. 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, but the `title` attribute has special semantics on this element when used with the `dfn` element.

DOM interface:

No difference from `HTMLElement`.

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

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

This example shows the `samp` element being used inline:

```
<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>
```

3.12.16. 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:

No difference from `HTMLElement`.

The `kbd` element represents user input (typically keyboard input, although it may also be used to represent other input, such as voice commands).

When the `kbd` element is nested inside a `samp` element, it represents the input as it was echoed by the system.

When the `kbd` element *contains* a `samp` element, it represents input based on system output, for example invoking a menu item.

When the `kbd` element is nested inside another `kbd` element, it represents an actual key or other single unit of input as appropriate for the input mechanism.

Here the `kbd` element is used to indicate keys to press:

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

In this second example, the user is told to pick a particular menu item. The outer `kbd` element marks up a block of input, with the inner `kbd` elements representing each individual step of the input, and the `samp` elements inside them indicating that the steps are input based on something being displayed by the system, in this case menu labels:

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

3.12.17. 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:

No difference from `HTMLElement`.

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

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

```
<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>
```

3.12.18. 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, but the `title` attribute has special semantics on this element when used with the `dfn` element.

DOM interface:

No difference from `HTMLElement`.

The `span` element doesn't mean anything on its own, but can be useful when used together with other attributes, e.g. `class`, `lang`, or `dir`, or when used in conjunction with the `dfn` element.

3.12.19. The `i` element

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 when used with the `dfn` element.

DOM interface:

No difference from `HTMLElement`.

The `i` element represents a span of text in an alternate voice or mood, or otherwise offset from the normal prose, such as a taxonomic designation, a technical term, an idiomatic phrase from another language, a thought, a ship name, or some other prose whose typical typographic presentation is italicized.

Terms in languages different from the main text should be annotated with `lang` attributes (`xml:lang` in XML).

The examples below show uses of the `i` element:

```
<p>The <i>felis silvestris catus</i> is cute.</p>
<p>The 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.

Note: Style sheets can be used to format `i` elements, just like any other element can be restyled. Thus, it is not the case that content in `i` elements will necessarily be italicised.

3.12.20. The `b` element

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:

No difference from `HTMLElement`.

The b element represents a span of text to be stylistically offset from the normal prose without conveying any extra importance, such as key words in a document abstract, product names in a review, or other spans of text whose typical typographic presentation is boldened.

> The following example shows a use of the b element to highlight key words without marking them up as important:
>
> ```
> <p>The <b>frobonitor</b> and <b>barbinator</b> components are
> fried.</p>
> ```
>
> The following would be *incorrect* usage:
>
> ```
> <p><b>WARNING!</b> Do not frob the barbinator!</p>
> ```
>
> In the previous example, the correct element to use would have been strong, not b.
>
> In the following example, objects in a text adventure are highlighted as being special by use of the b element.
>
> ```
> <p>You enter a small room. Your <b>sword</b> glows
> brighter. A <b>rat</b> scurries past the corner wall.</p>
> ```
>
> 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>
>  <p><b>Six abandoned kittens have found an unexpected new
>  mother figure — a pet rabbit.</b></p>
>  <p>Veterinary nurse Melanie Humble took the three-week-old
>  kittens to her Aberdeen home.</p>
>  [...]
> ```

The b element should be used as a last resort when no other element is more appropriate. In particular, headers should use the h1 to h6 elements, stress emphasis should use the em element, importance should be denoted with the strong element, and text marked or highlighted should use the m element.

> *Note: Style sheets can be used to format b elements, just like any other element can be restyled. Thus, it is not the case that content in b elements will necessarily be boldened.*

3.12.21. The bdo element

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:

No difference from `HTMLElement`.

The `bdo` element allows authors to override the Unicode bidi algorithm by explicitly specifying a direction override. [BIDI]

Authors must specify the `dir` attribute on this element, with the value `ltr` to specify a left-to-right override and with the value `rtl` to specify a right-to-left override.

If the element has the `dir` attribute set to the exact value `ltr`, then for the purposes of the bidi algorithm, the user agent must act as if there was a U+202D LEFT-TO-RIGHT OVERRIDE character at the start of the element, and a U+202C POP DIRECTIONAL FORMATTING at the end of the element.

If the element has the `dir` attribute set to the exact value `rtl`, then for the purposes of the bidi algorithm, the user agent must act as if there was a U+202E RIGHT-TO-LEFT OVERRIDE character at the start of the element, and a U+202C POP DIRECTIONAL FORMATTING at the end of the element.

The requirements on handling the `bdo` element for the bidi algorithm may be implemented indirectly through the style layer. For example, an HTML+CSS user agent should implement these requirements by implementing the CSS `unicode-bidi` property. [CSS21]

3.13. Edits

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

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.</p>
</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.</p>
 <p><del>This sentence was deleted too.</del> That
 sentence needed a separate &lt;del> 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__.

3.13.1. The `ins` element

Categories

When the element only contains phrasing content: phrasing content.
Otherwise: prose content.

Contexts in which this element may be used:

When the element only contains phrasing content: where phrasing content is expected.
Otherwise: where prose 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>
```

3.13.2. The `del` element

Categories

When the element only contains phrasing content: phrasing content.
Otherwise: prose content.

Contexts in which this element may be used:

When the element only contains phrasing content: where phrasing content is expected.
Otherwise: where prose 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.

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

The **cite** attribute may be used to specify a URI that explains the change. When that document is long, for instance the minutes of a meeting, authors are encouraged to include a fragment identifier pointing to the specific part of that document that discusses the change.

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

The **datetime** attribute may be used to specify the time and date of the change.

If present, the `datetime` attribute must be a valid datetime value.

User agents must parse the `datetime` attribute according to the parse a string as a datetime value algorithm. If that doesn't return a time, then the modification has no associated timestamp (the value is non-conforming; it is not a valid datetime). Otherwise, the modification is marked as having been made at the given datetime. User agents should use the associated timezone information to determine which timezone to present the given datetime in.

The `ins` and `del` elements must implement the `HTMLModElement` interface:

```
interface HTMLModElement : HTMLElement {
          attribute DOMString cite;
          attribute DOMString dateTime;
};
```

The **cite** DOM attribute must reflect the element's >`cite` content attribute. The **dateTime** DOM attribute must reflect the element's `datetime` content attribute.

3.14. Embedded content

3.14.1. The `figure` element

Categories

> Prose content.

Contexts in which this element may be used:

> Where prose content is expected.

Content model:

> Either one `legend` element followed by prose content.
> Or: Prose content followed by one `legend` element.

Element-specific attributes:

> None.

DOM interface:

> No difference from HTMLElement.

The figure element represents some prose content with a caption.

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 captioned content.

3.14.2. 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:

```
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;
};
```

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

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 URI (or IRI).

Should we restrict the URI 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 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 tokenisation
> > 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 realise 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 recognise 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 labelled with an icon indicating their topic.

```
<body>
 <article>
  <header>
   <h1>Ratatouille wins <i>Best Movie of the Year</i>
award</h1>
   <p><img src="movies.png" alt="Movies"></p>
  </header>
  <p>Pixar has won yet another <i>Best Movie of the Year</i>
award,
  making this its 8th win in the last 12 years.</p>
 </article>
 <article>
  <header>
   <h1>Latest TWiT episode is online</h1>
   <p><img src="podcasts.png" alt="Podcasts"></p>
  </header>
  <p>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.</p>
    </article>
   </body>
```

Many pages include logos, insignia, flags, or emblems, which stand for a particular entity such as a company, organisation, 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
 specialising 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>
```

```
    <p><img src="shalott.jpeg" alt=""></p>
    <p>On either side the river lie<br>
    Long fields of barley and of rye,<br>
    That clothe the wold and meet the sky;<br>
    And through the field the road run by<br>
    To many-tower'd Camelot;<br>
    And up and down the people go,<br>
    Gazing where the lilies blow<br>
    Round an island there below,<br>
    The island of Shalott.</p>
```

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 that doesn't have an obvious textual alternative

In certain rare cases, the image is simply a critical part of the content, and there might even be no alternative text available. This could be the case, for instance, in a photo gallery, where a user has uploaded 3000 photos from a vacation trip, without providing any descriptions of the images. The images are the whole *point* of the pages containing 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. If an image is a key part of the content, the alt attribute must not be specified with an empty value.

A photo on a photo-sharing site:

```
<figure>
 <img src="1100670787_6a7c664aef.jpg">
 <legend>Bubbles traveled everywhere with us.</legend>
</figure>
```

A screenshot in a gallery of screenshots for a new OS:

```
<figure>
 <img src="KDE%20Light%20desktop.png">
 <legend>Screenshot of a KDE desktop.</legend>
</figure>
```

In both cases, though, it would be better if a detailed description of the important parts of the image were included.

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.

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

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

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

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

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

The `usemap` attribute, if present, can indicate that the image has an associated image map.

The **ismap** attribute, when used on an element that is a descendant of an `a` element with an `href` attribute, indicates by its presence that the element 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 **height** and **width** must return the rendered height and width of the image, in CSS pixels, if the image is being rendered, and is being rendered to a visual medium, or 0 otherwise. [CSS21]

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

3.14.3. 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

DOM interface:

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

```
};
```

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

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

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

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

If the `src` attribute is not set when the element is created, the browsing context will remain at the initial `about:blank` page.

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**.*

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 attribute **src** must reflect the content attribute of the same name.

3.14.4. 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 URI (or IRI).

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

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

The download of the resource must delay the load event.

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

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

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

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

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

1. If the element has a type attribute, then the value of the type attribute is the content's type.

2. Otherwise, if the specified resource has explicit Content-Type metadata, then that is the

content's type.

3. Otherwise, the content has no type and there can be no appropriate handler for it.

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

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

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

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

The `embed` element supports dimension attributes.

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

3.14.5. 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
> usemap
> width
> height

DOM interface:

```
interface HTMLObjectElement : HTMLElement {
         attribute DOMString data;
         attribute DOMString type;
         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 third-party software package.

The **data** attribute, if present, specifies the address of the resource. If present, the attribute must be a URI (or IRI).

The **type** attribute, if present, specifies the type of the resource. If present, the attribute must be a valid MIME type, optionally with parameters. [RFC2046]

One or both of the `data` and `type` attributes must be present.

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

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

   1. Begin a load for the resource.

      The download of the resource must delay the `load` event.

   2. If the resource is not yet available (e.g. because the resource was not available in the cache, so that loading the resource required making a request over the network), then jump to step 3 in the overall set of steps (fallback). When the resource becomes available, or if the load fails, restart this algorithm from this step. Resources can load incrementally; user agents may opt to consider a resource "available" whenever enough data has been obtained to begin processing the resource.

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

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

      > This says to trust the type. Should we instead use the same mechanism as for browsing contexts?

      ↳ **If the resource has associated Content-Type metadata**
      > The type is the type specified in the resource's Content-Type metadata.

      ↳ **Otherwise, if the `type` attribute is present**
      > The type is the type specified in the `type` attribute.

      ↳ **Otherwise, there is no explicit type information**
      > The type is the sniffed type of the resource.

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

      ↳ **If the resource requires a special handler (e.g. a plugin)**
      > The user agent should find an appropriate handler for the specified resource, based on the *resource type* found in the previous step, and pass

the content of the resource to that handler. If the handler supports a scriptable interface, the `HTMLObjectElement` object representing the element should expose that interface. The handler is not a nested browsing context. If no appropriate handler can be found, then jump to step 3 in the overall set of steps (fallback).

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

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

> this doesn't completely duplicate the navigation section, since it handles <param>, etc, but surely some work should be done to work with it

↪ **If the type of the resource is an XML MIME type**
↪ **If the type of the resource is HTML**
↪ **If the type of the resource does not start with "`image/`"**

The `object` element must be associated with a nested browsing context, if it does not already have one. The element's nested browsing context must then be navigated to the given resource, with replacement enabled. (The `data` attribute of the `object` element doesn't get updated if the browsing context gets further navigated to other locations.)

> navigation might end up treating it as something else, because it can do sniffing. how should we handle that?

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

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

> shouldn't we use the image-sniffing stuff here?

↪ **Otherwise**

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

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

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

2. If the `data` attribute is absent but the `type` attribute is present, and if the user agent can find a handler suitable according to the value of the `type` attribute, then that handler should be used. If the handler supports a scriptable interface, the `HTMLObjectElement` object

representing the element should expose that interface. The handler is not a nested browsing context. If no suitable handler can be found, jump to the next step (fallback).

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

In the absence of other factors (such as style sheets), user agents must show the user what the `object` element represents. Thus, the contents of `object` elements act as fallback content, to be used only when referenced resources can't be shown (e.g. because it returned a 404 error). This allows multiple `object` elements to be nested inside each other, targeting multiple user agents with different capabilities, with the user agent picking the best one it supports.

The `usemap` attribute, if present while the `object` element represents an image, can indicate that the object has an associated image map. The attribute must be ignored if the `object` element doesn't represent an image.

The `object` element supports dimension attributes.

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

3.14.6. The `param` element

Categories
> None.

Contexts in which this element may be used:
> As a child of an `object` element, before any prose 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 handlers invoked by `object` elements.

The **name** attribute gives the name of the parameter.

The **value** attribute gives the value of the parameter.

Both attributes must be present. They may have any value.

If both attributes are present, and if the parent element of the `param` is an `object` element, then the element defines a **parameter** with the given name/value pair.

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

3.14.7. 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 `video` element supports dimension attributes.

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 URI (or IRI).

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

The **videoWidth** DOM attribute must return the native width of the video in CSS pixels. The **videoHeight** DOM attribute must return the native height of the video in CSS pixels. In the absence of resolution information defining the mapping of pixels in the video to physical dimensions, user agents may assume that one pixel in the video corresponds to one CSS pixel. If no video data is available, then the attributes must return 0.

When no video data is available (the element's `networkState` attribute is either `EMPTY`, `LOADING`, or `LOADED_METADATA`), `video` elements represent either the image given by the `poster` attribute, or nothing.

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 synchronised with the current playback position, at the specified volume with the specified mute state.

When a `video` element is paused, the element represents the frame of video corresponding to the current playback position, or, if that is not available yet (e.g. because the video is seeking or buffering), the last rendered frame of video.

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.*

Video content should be rendered inside the element's playback area such that the video content is shown centered in the playback area at the largest possible size that fits completely within it, with the video content's aspect ratio being preserved. Thus, if the aspect ratio of the playback area does not match the aspect ratio of the video, the video will be shown letterboxed. Areas of the element's playback area that do not contain the video represent nothing.

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.

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.*

3.14.7.1. 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.*

3.14.8. 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:

```
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 synchronised 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.

3.14.8.1. 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.

3.14.9. 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 TimeRanges buffered;
    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 float start, in float
end, in boolean pauseOnExit, in VoidCallback enterCallback, in
VoidCallback 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.

3.14.9.1. 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.

3.14.9.2. 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 URI (or IRI).

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. If the media element has a `src`, then the address given in that attribute is the address of the media resource; jump to the last step.

2. Otherwise, let *candidate* be the first `source` element child in the media element, or null if there is no such child.

3. If either:

   - *candidate* is null, or

   - the *candidate* element has no `src` attribute, 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 address given in that *candidate* element's `src` attribute is the address of the media resource; jump to the last step.

4. Let *candidate* be the next `source` element child in the media element, or null if there are no more such children.

5. If *candidate* is not null, return to step 3.

6. There is no media resource. Abort these steps.

7. Let the address of the **chosen media resource** be the one that was found before jumping to this step.

   Note: A `source` element with no `src` attribute is assumed to be the last `source` element — any `source` elements after it are ignored (and are invalid).

The **currentSrc** DOM attribute must return the empty string if the media element's `networkState` has the value EMPTY, and the absolute URL of the chosen media resource otherwise.

3.14.9.3. 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 initialised. 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.

3.14.9.4. 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.

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 fire a progress event called `begin` 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 only consider a resource erroneous 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 **buffered** attribute must return a static normalised `TimeRanges` object that represents the ranges of the media resource, if any, that the user agent has downloaded, 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.

3.14.9.5. Offsets into the media resource

The **`duration`** attribute must return the length of the <u>media resource</u>, in seconds. If no <u>media data</u> is available, then the attributes must return 0. If <u>media data</u> is available but the length is not known, the attribute must return the Not-a-Number (NaN) value. If the <u>media resource</u> is known to be unbounded (e.g. a streaming radio), then the attribute must return the positive Infinity value.

When the length of the <u>media resource</u> 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, <u>fire a simple event</u> called `durationchange` at the <u>media element</u>.

<u>Media elements</u> have a **current playback position**, which must initially be zero. The current position is a time.

The **`currentTime`** attribute must, on getting, return the <u>current playback position</u>, expressed in seconds. On setting, the user agent must <u>seek</u> to the new value (which might raise an exception).

The **`start`** content attribute gives the offset into the <u>media resource</u> at which playback is to begin. The default value is the default start position of the <u>media resource</u>, or 0 if not enough <u>media data</u> 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 <u>start</u> DOM attribute and the end of the <u>media resource</u>.

The **`loopstart`** content attribute gives the offset into the <u>media resource</u> at which playback is to begin when looping a clip. The default value of the <u>loopstart</u> content attribute is the value of the <u>start</u> DOM attribute.

The ***effective loop start*** is the smaller of the <u>loopStart</u> DOM attribute and the end of the <u>media resource</u>.

The **`loopend`** content attribute gives an offset into the <u>media resource</u> at which playback is to jump back to the <u>loopstart</u>, when looping the clip. The default value of the <u>loopend</u> content attribute is the value of the <u>end</u> DOM attribute.

The ***effective loop end*** is the greater of the <u>start</u>, <u>loopStart</u>, and <u>loopEnd</u> DOM attributes, except if that is greater than the end of the <u>media resource</u>, in which case that's its value.

The **`end`** content attribute gives an offset into the <u>media resource</u> at which playback is to end. The default value is infinity.

The ***effective end*** is the greater of the <u>start</u>, <u>loopStart</u>, and <u>end</u> DOM attributes, except if that is greater than the end of the <u>media resource</u>, in which case that's its value.

The <u>start</u>, <u>loopstart</u>, <u>loopend</u>, and <u>end</u> attributes must, if specified, contain value time offsets. To get the time values they represent, user agents must use the <u>rules for parsing time offsets</u>.

The **`start`**, **`loopStart`**, **`loopEnd`**, and **`end`** DOM attributes must <u>reflect</u> the <u>start</u>, <u>loopstart</u>, <u>loopend</u>, and <u>end</u> content attributes on the <u>media element</u> respectively.

The **`playcount`** content attribute gives the number of times to play the clip. The default value is 1.

The **`playCount`** DOM attribute must <u>reflect</u> the <u>playcount</u> content attribute on the <u>media element</u>.

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`, and `playCount` DOM attributes change value (either through content attribute mutations reflecting into the DOM attribute, or 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.

3.14.9.6. 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 ot 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 leaded-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 guarenteed 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.

3.14.9.7. 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.

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, 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*.

3. The playbackRate attribute must be set to the value of the defaultPlaybackRate attribute.

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.

   Note: If the second step above involved a seek, the user agent will fire a simple event called timeupdate at the media element.

   Note: If the third step above caused the playbackRate attribute to change value, the user agent will fire a simple event called ratechange at the media element.

   Note: If the fourth step above changed the value of paused, the user agent must fire a simple event called play at the media 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`, 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, 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 normalised `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.

3.14.9.8. 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 the value of `playCount`, 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. The user agent must fire a simple event called `timeupdate` at the element.

10. As soon as the user agent has 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 `seeking` DOM attribute must be set to false.

The **`seekable`** attribute must return a static normalised `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 `seeking` 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).*

3.14.9.9. 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.

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, 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 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, initialised 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, initialised 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.

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.

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 the `VoidCallback` interface) means calling its `handleEvent()` method.

```
interface VoidCallback {
  void handleEvent();
};
```

The **handleEvent** method of objects implementing the `VoidCallback` interface is the entrypoint for the callback represented by the object.

In the ECMAScript DOM binding, the ECMAScript native `Function` type must implement the `VoidCallback` interface such that invoking the `handleEvent()` method of that interface on the object from another language binding invokes the function itself. In the ECMAScript binding itself, however, the `handleEvent()` method of the interface is not directly accessible on `Function` objects. Such functions, when invoked, must be called at the scope of the browsing context.

3.14.9.10. User interface

The **controls** attribute is a boolean attribute. If the attribute is present, or if scripting is disabled,

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 0.5, 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](#).

3.14.9.11. Time range

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 **normalised 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, 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.

3.14.9.12. Event summary

The following events fire on media elements as part of the processing model described above:

Event name	Interface	Dispatched when...	Preconditions
begin	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.

Event name	Interface	Dispatched when...	Preconditions
error	ProgressEvent [PROGRESS]	An error occurs while fetching the media data.	error is an object with the code MEDIA_ERR_NETWORK_ERRC 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 the 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 curre playback position is not contained in any of the range in buffered. It is possible fo playback to stop for two other reasons without paused bein false, but those two reasons do not fire this event: maybe playback ended, or playback stopped due to errors.
timeupdate	Event	The current playback position changed in an interesting way, for	

Event name	Interface	Dispatched when...	Preconditions
		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	

Event name	Interface	Dispatched when...	Preconditions
		returned.	

3.14.9.13. 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)

3.14.10. The `source` element

Categories

> None.

Contexts in which this element may be used:

> As a child of a media element, before any prose content.

Content model:

> Empty.

Element-specific attributes:

> src
> type
> media

DOM interface:

```
interface HTMLSourceElement : HTMLElement {
          attribute DOMString src;
          attribute DOMString type;
          attribute DOMString media;
};
```

The `source` element allows authors to specify multiple media resources for media elements.

The **src** attribute gives the address of the media resource. The value must be a URI (or IRI). 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 downloading 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=&quot;
avc1.58A01E, mp4a.40.2&quot;">
```

H.264 Main profile video level 3 and Low-Complexity AAC audio in MP4 container

```
<source src="video.mp4" type="video/mp4; codecs=&quot;
avc1.4D401E, mp4a.40.2&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=&quot;
avc1.64001E, mp4a.40.2&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=&quot;
mp4v.20.8, mp4a.40.2&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=&quot;
mp4v.20.240, mp4a.40.2&quot;">
```

MPEG-4 Visual Simple Profile Level 0 video and AMR audio in 3GPP container

```
<source src="video.3gp" type="video/3gpp; codecs=&quot;
mp4v.20.8, samr&quot;">
```

Theora video and Vorbis audio in Ogg container

```
<source src="video.ogv" type="video/ogg; codecs=&quot;theora,
vorbis&quot;">
```

Theora video and Speex audio in Ogg container

```
<source src="video.ogv" type="video/ogg; codecs=&quot;theora,
speex&quot;">
```

Vorbis audio alone in Ogg container

```
<source src="audio.oga" type="audio/ogg; codecs=vorbis">
```

Speex audio alone in Ogg container

```
<source src="audio.oga" type="audio/ogg; codecs=speex">
```

Flac audio alone in Ogg container

```
<source src="audio.oga" type="audio/ogg; codecs=flac">
```

Dirac video and Vorbis audio in Ogg container

```
<source src="video.ogv" type="video/ogg; codecs=&quot;dirac,
vorbis&quot;">
```

Theora video and Vorbis audio in Matroska container

```
<source src="video.mkv" type="video/x-matroska; codecs=&quot;
theora, 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.

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.

3.14.11. 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 with scripting enabled, the canvas element is an embedded element with a dynamically created image.

In non-interactive, static, visual media, if the `canvas` element has been previously painted on (e.g. if the page was viewed in an interactive visual medium and is now being printed, or if some script that ran during the page layout process painted on the element), then the `canvas` element must be treated as embedded content with the current image and size. Otherwise, the element's fallback content must be used instead.

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

The `canvas` element has two attributes to control the size of the coordinate space: **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.

The canvas must initially be fully transparent black.

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 reinitialised 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
> ```

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 tis *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.

> **Note: A future version of this specification will probably define a *3d* context (probably based on the OpenGL ES API).**

The **`toDataURL()`** method must, when called with no arguments, return a `data:` URI containing a representation of the image as a PNG file. [PNG].

The **`toDataURL(type)`** method (when called with one *or more* arguments) must return a `data:` URI containing a representation of the image in the format given by *type*. The possible values are MIME types with no parameters, for example `image/png`, `image/jpeg`, or even maybe `image/svg+xml` if the implementation actually keeps enough information to reliably render an SVG image from the canvas.

Only support for `image/png` is required. User agents may support other types. If the user agent does not support the requested type, it must return the image using the PNG format.

User agents must convert the provided type to lower case before establishing if they support that type and before creating the `data:` URI.

> **Note: When trying to use types other than *image/png*, authors can check if the image was really returned in the requested format by checking to see if the returned string starts with one the exact strings "*data:image/png,*" or "*data:image/png;*". If it does, the image is PNG, and thus the requested type was not supported.**

Arguments other than the *type* must be ignored, and must not cause the user agent to raise an exception (as would normally occur if a method was called with the wrong number of arguments). A future version of this specification will probably allow extra parameters to be passed to `toDataURL()` to allow authors to more carefully control compression settings, image metadata, etc.

Security: To prevent *information leakage*, the `toDataURL()` and `getImageData()` methods should raise a security exception if the canvas has ever had an image painted on it whose origin is different from that of the script calling the method.

3.14.11.1. 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, DOMString
repetition);
  CanvasPattern createPattern(in HTMLCanvasElement image, 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);

  // 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 getImageData(in float sx, in float sy, in float sw, in
float sh);
  void putImageData(in ImageData image, in float dx, in float dy);

  // drawing text is not supported in this version of the API
  // (there is no way to predict what metrics the fonts will have,
  // which makes fonts very hard to use for painting)

};

interface CanvasGradient {
  // opaque object
  void addColorStop(in float offset, in DOMString color);
};
```

```
interface CanvasPattern {
  // opaque object
};


interface ImageData {
  readonly attribute long int width;
  readonly attribute long int height;
  readonly attribute int[] data;
};
```

The **canvas** attribute must return the canvas element that the context paints on.

3.14.11.1.1. THE CANVAS STATE

Each context maintains a stack of drawing states. **Drawing states** consist of:

- The current transformation matrix.
- The current clip region.
- The current values of the following attributes: strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, globalCompositeOperation.

Note: The current path and the current bitmap are not part of the drawing state. The current path is persistent, and can only be reset using the beginPath() method. The current bitmap is a property of the canvas, not the context.

The **save()** method 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.

3.14.11.1.2. TRANSFORMATIONS

The transformation matrix is applied to all drawing operations prior to their being rendered. It is also applied when creating the clip region.

When the context is created, the transformation matrix must initially be the identity transform. It may then be adjusted using the transformation methods.

The transformation matrix can become infinite, at which point nothing is drawn anymore.

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. If either argument is Infinity the transformation matrix must be marked as infinite instead of the method throwing an exception.

The **rotate(angle)** method must add the rotation transformation described by the argument to the transformation matrix. The *angle* argument represents a clockwise rotation angle expressed in

radians.

The **translate(*x*, *y*)** method must add the translation transformation described by the arguments to the transformation matrix. The *x* argument represents the translation distance in the horizontal direction and the *y* argument represents the translation distance in the vertical direction. The arguments are in coordinate space units. If either argument is Infinity the transformation matrix must be marked as infinite instead of the method throwing an exception.

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

If any of the arguments are Infinity the transformation matrix must be marked as infinite instead of the method throwing an exception.

The **setTransform(*m11*, *m12*, *m21*, *m22*, *dx*, *dy*)** method must reset the current transform to the identity matrix, and then invoke the <u>transform</u>(*m11*, *m12*, *m21*, *m22*, *dx*, *dy*) method with the same arguments. If any of the arguments are Infinity the transformation matrix must be marked as infinite instead of the method throwing an exception.

3.14.11.1.3. COMPOSITING

All drawing operations are affected by the global compositing attributes, <u>globalAlpha</u> and <u>globalCompositeOperation</u>.

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 <u>globalAlpha</u> 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 <u>globalAlpha</u> 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 only recognise values that 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 recognise the specified value, it must be ignored, leaving the value of globalCompositeOperation unaffected.

When the context is created, the globalCompositeOperation attribute must initially have the value source-over.

3.14.11.1.4. 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, CanvasGradients, or CanvasPatterns. 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: if it has alpha equal to 1.0, then the color must be returned as 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). If the value has alpha less than 1.0, then the value must instead be returned in the CSS `rgba()` functional-notation format: the literal string `rgba` (U+0072 U+0067 U+0062 U+0061) followed by a U+0028 LEFT PARENTHESIS, a base-ten integer in the range 0-255 representing the red component (using digits 0-9, U+0030 to U+0039, in the shortest form possible), a literal U+002C COMMA and U+0020 SPACE, an integer for the green component, a comma and a space, an integer for the blue component, another comma and space, a U+0030 DIGIT ZERO, a U+002E FULL STOP (representing the decimal point), one or more digits in the range 0-9 (U+0030 to U+0039) representing the fractional part of the alpha value, and finally a U+0029 RIGHT PARENTHESIS.

Otherwise, if it is not a color but a `CanvasGradient` or `CanvasPattern`, then 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.)

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 or greater than 1 then an `INDEX_SIZE_ERR` exception must be raised. If the *color* cannot be parsed as a CSS color, then a `SYNTAX_ERR` exception must be raised. Otherwise, the gradient must 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, representing the start point ($x0$, $y0$) and end point ($x1$, $y1$) of the gradient, in coordinate space units, and must return a linear `CanvasGradient` initialised with that line.

Linear gradients must be rendered such that at and before the starting point on the canvas the color at offset 0 is used, that at and after the ending point the color at offset 1 is used, and 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 described above).

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. The method must return a radial `CanvasGradient` initialised with those two circles. If either of $r0$ or $r1$ are negative, an `INDEX_SIZE_ERR` exception must be raised.

Radial gradients must be rendered by following these steps:

1. 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 of the gradient at offset 0.0 for all values of ω less than 0.0, the color at offset 1.0 for all values of ω greater than 1.0, and the color at the given offset for values of ω in the range $0.0 \leq \omega \leq 1.0$

2. 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.

If $x_0 = x_1$ and $y_0 = y_1$ and $r_0 = r_1$, then the radial gradient must paint nothing.

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 only be painted where the relevant stroking or filling effects requires that they be drawn.

Support for actually painting gradients is optional. Instead of painting the gradients, user agents may instead just paint the first stop's color. However, `createLinearGradient()` and `createRadialGradient()` must always return objects when passed valid arguments.

Patterns are represented by objects implementing the opaque **CanvasPattern** interface.

To create objects of this type, the **createPattern(image, repetition)** method is used. The first argument gives the image to use as the pattern (either an `HTMLImageElement` or an `HTMLCanvasElement`). Modifying this image after calling the `createPattern()` method must not affect the pattern. The second argument must be a string with one of the following values: `repeat`, `repeat-x`, `repeat-y`, `no-repeat`. If the empty string or null is specified, `repeat` must be assumed. If an unrecognised value is given, then the user agent must raise a `SYNTAX_ERR` exception. User agents must recognise the four values described above exactly (e.g. they must not do case folding). The method must return a `CanvasPattern` object suitably initialised.

The *image* argument must be an instance of an `HTMLImageElement` or `HTMLCanvasElement`. If the *image* is of the wrong type, the implementation must raise a `TYPE_MISMATCH_ERR` exception. If the *image* argument is an `HTMLImageElement` object whose `complete` attribute is false, 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 be scaled by this process; one CSS pixel of the image must be painted on one coordinate space unit. Of course, patterns must only actually painted where the stroking or filling effect requires that they be drawn,

and are affected by the current transformation matrix.

Support for patterns is optional. If the user agent doesn't support patterns, then `createPattern()` must return null.

3.14.11.1.5. LINE STYLES

The **lineWidth** attribute gives the default width of lines, in coordinate space units. On setting, zero and negative values must be ignored, leaving the value unchanged.

When the context is created, the lineWidth attribute must initially have the value `1.0`.

The **lineCap** attribute defines the type of endings that UAs shall place on the end of lines. The three valid values are butt, round, and square. The butt value means that the end of each line is a flat edge perpendicular to the direction of the line. The round value means that a semi-circle with the diameter equal to the width of the line is then added on to the end of the line. The square value means that at the end of each line is a rectangle with the length of the line width and the width of half the line width, placed flat against the edge perpendicular to the direction of the line. On setting, any other value than the literal strings butt, round, and square must be ignored, leaving the value unchanged.

When the context is created, the lineCap attribute must initially have the value butt.

The **lineJoin** attribute defines the type of corners that that UAs will place where two lines meet. The three valid values are round, bevel, and miter.

On setting, any other value than the literal strings round, bevel and miter must be ignored, leaving the value unchanged.

When the context is created, the lineJoin attribute must initially have the value miter.

The round value means that a filled arc connecting the corners on the outside of the join, with the diameter equal to the line width, and the origin at the point where the inside edges of the lines touch, must be rendered at joins. The bevel value means that a filled triangle connecting those two corners with a straight line, the third point of the triangle being the point where the lines touch on the inside of the join, must be rendered at joins. The miter value means that a filled four- or five-sided polygon must be placed at the join, with two of the lines being the perpendicular edges of the joining lines, and the other two being continuations of the outside edges of the two joining lines, as long as required to intersect without going over the miter limit.

The miter length is the distance from the point where the lines touch on the inside of the join to the intersection of the line edges on the outside of the join. The miter limit ratio is the maximum allowed ratio of the miter length to the line width. If the miter limit would be exceeded, then a fifth line must be added to the polygon, connecting the two outside lines, such that the distance from the inside point of the join to the point in the middle of this fifth line is the maximum allowed value for the miter length.

The miter limit ratio can be explicitly set using the **miterLimit** attribute. On setting, zero and negative values must be ignored, leaving the value unchanged.

When the context is created, the miterLimit attribute must initially have the value `10.0`.

3.14.11.1.6. SHADOWS

All drawing operations are affected by the four global shadow attributes. Shadows form part of the

source image during composition.

The **shadowColor** attribute sets the color of the shadow.

When the context is created, the shadowColor attribute initially must be fully-transparent black.

The **shadowOffsetX** and **shadowOffsetY** attributes specify the distance that the shadow will be offset in the positive horizontal and positive vertical distance respectively. Their values are in coordinate space units.

When the context is created, the shadow offset attributes initially have the value 0.

The **shadowBlur** attribute specifies the number of coordinate space units that the blurring is to cover. On setting, negative numbers must be ignored, leaving the attribute unmodified.

When the context is created, the shadowBlur attribute must initially have the value 0.

Support for shadows is optional. When they are supported, then, when shadows are drawn, they must be rendered using the specified color, offset, and blur radius.

3.14.11.1.7. 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 and height of the rectangle, respectively.

Shapes are painted without affecting the current path, and are subject to transformations, shadow effects, global alpha, clipping paths, and global composition operators.

Negative values for width and height must cause the implementation to raise an INDEX_SIZE_ERR exception.

The **clearRect()** method must clear the pixels in the specified rectangle to a fully transparent black, erasing any previous image. If either height or width are zero, this method has no effect.

The **fillRect()** method must paint the specified rectangular area using the fillStyle. If either height or width are zero, this method has no effect.

The **strokeRect()** method must draw stroke the path that would be created for the outline of a rectangle of the specified size 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).

3.14.11.1.8. 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 **beginPath()** method must empty the list of subpaths so that the context once again has zero

subpaths.

The `moveTo(x, y)` method must create a new subpath with the specified point as its first (and only) point.

The `closePath()` method must do nothing if the context has no subpaths. Otherwise, it must mark the last subpath as closed, create a new subpath whose first point is the same as the previous subpath's first point, and finally add this new subpath to the path. (If the last subpath had more than one point in its list of points, then this is equivalent to adding a straight line connecting the last point back to the first point, thus "closing" the shape, and then repeating the last moveTo() call.)

New points and the lines connecting them are added to subpaths using the methods described below. In all cases, the methods only modify the last subpath in the context's paths.

The `lineTo(x, y)` method must do nothing if the context has no subpaths. Otherwise, it must connect the last point in the subpath to the given point (x, y) using a straight line, and must then add the given point (x, y) to the subpath.

The `quadraticCurveTo(cpx, cpy, x, y)` method must do nothing if the context has no subpaths. Otherwise it must connect the last point in the subpath to the given point (x, y) by a quadratic curve with control point (cpx, cpy), and must then add the given point (x, y) to the subpath.

The `bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)` method must do nothing if the context has no subpaths. Otherwise, it must connect the last point in the subpath to the given point (x, y) using a bezier curve with control points (cp1x, cp1y) and (cp2x, cp2y). Then, it must add the point (x, y) to the subpath.

The `arcTo(x1, y1, x2, y2, radius)` method must do nothing if the context has no subpaths. If the context *does* have a subpath, then the behaviour depends on the arguments and the last point in the subpath.

Let the point (x0, y0) be the last point in the subpath. Let *The Arc* be the shortest arc given by circumference of the circle that has one point tangent to the line defined by the points (x0, y0) and (x1, y1), another point tangent to the line defined by the points (x1, y1) and (x2, y2), and that has radius *radius*. The points at which this circle touches these two lines are called the start and end tangent points respectively.

If the point (x2, y2) is on the line defined by the points (x0, y0) and (x1, y1) then the method must do nothing, as no arc would satisfy the above constraints.

Otherwise, the method must connect the point (x0, y0) to the start tangent point by a straight line, then connect the start tangent point to the end tangent point by *The Arc*, and finally add the start and end tangent points to the subpath.

Negative or zero values for *radius* must cause the implementation to raise an INDEX_SIZE_ERR exception.

The `arc(x, y, radius, startAngle, endAngle, anticlockwise)` method draws an arc. If the context has any subpaths, then the method must add a straight line from the last point in the subpath to the start point of the arc. In any case, it must draw the arc between the start point of the arc and the end point of the arc, and add the start and end points of the arc to the subpath. The arc and its start and end points are defined as follows:

Consider a circle that has its origin at (x, y) and that has radius *radius*. The points at *startAngle* and *endAngle* along the circle's circumference, measured in radians clockwise from the positive x-axis, are the start and end points respectively. The arc is the path along the circumference of this circle

from the start point to the end point, going anti-clockwise if the *anticlockwise* argument is true, and clockwise otherwise.

Negative or zero values for *radius* must cause the implementation to raise an INDEX_SIZE_ERR exception.

The **rect(*x*, *y*, *w*, *h*)** method must create a new subpath containing just the four points (*x*, *y*), (*x+w*, *y*), (*x+w*, *y+h*), (*x*, *y+h*), with those four points connected by straight lines, and must then mark the subpath as closed. It must then create a new subpath with the point (*x*, *y*) as the only point in the subpath.

Negative values for *w* and *h* must cause the implementation to raise an INDEX_SIZE_ERR exception.

The **fill()** method must fill each subpath of the current path in turn, using <u>fillStyle</u>, and using the non-zero winding number rule. Open subpaths must be implicitly closed when being filled (without affecting the actual subpaths).

The **stroke()** method must stroke each subpath of the current path in turn, using the <u>strokeStyle</u>, <u>lineWidth</u>, <u>lineJoin</u>, and (if appropriate) <u>miterLimit</u> attributes.

Paths, when filled or stroked, must be painted without affecting the current path, and must be subject to [transformations](#), [shadow effects](#), [global alpha](#), [clipping paths](#), and [global composition operators](#).

> *Note: The transformation is applied to the path when it is drawn, not when the path is constructed. Thus, a single path can be constructed and then drawn according to different transformations without recreating the path.*

The **clip()** method must create a new **clipping path** by calculating the intersection of the current clipping path and the area described by the current path (after applying the current transformation), using the non-zero winding number rule. Open subpaths must be implicitly closed when computing the clipping path, without affecting the actual subpaths.

When the context is created, the initial clipping path is the rectangle with the top left corner at (0,0) and the width and height of the coordinate space.

The **isPointInPath(*x*, *y*)** method must return true if the point given by the *x* and *y* coordinates passed to the method, when treated as coordinates in the canvas' coordinate space unaffected by the current transformation, is within the area of the canvas that would be filled if the current path was to be filled; and must return false otherwise.

3.14.11.1.9. IMAGES

To draw images onto the canvas, the **drawImage** method can be used.

This method is overloaded with three variants: drawImage(*image*, *dx*, *dy*), drawImage(*image*, *dx*, *dy*, *dw*, *dh*), and drawImage(*image*, *sx*, *sy*, *sw*, *sh*, *dx*, *dy*, *dw*, *dh*). (Actually it is overloaded with six; each of those three can take either an HTMLImageElement or an HTMLCanvasElement for the *image* argument.) If not specified, the *dw* and *dh* arguments default to the values of *sw* and *sh*, interpreted such that one CSS pixel in the image is treated as one unit in the canvas coordinate space. If the *sx*, *sy*, *sw*, and *sh* arguments are omitted, they default to 0, 0, the image's intrinsic width in image pixels, and the image's intrinsic height in image pixels, respectively.

The *image* argument must be an instance of an `HTMLImageElement` or `HTMLCanvasElement`. If the *image* is of the wrong type, the implementation must raise a `TYPE_MISMATCH_ERR` exception. If one of the *sy*, *sw*, *sw*, and *sh* arguments is outside the size of the image, or if one of the *dw* and *dh* arguments is negative, the implementation must raise an `INDEX_SIZE_ERR` exception. If the *image* argument is an `HTMLImageElement` object whose `complete` attribute is false, then the implementation must raise an `INVALID_STATE_ERR` exception.

When `drawImage()` is invoked, the specified region of the image specified by the source rectangle (*sx*, *sy*, *sw*, *sh*) must be painted on the region of the canvas specified by the destination rectangle (*dx*, *dy*, *dw*, *dh*).

Images are painted without affecting the current path, and are subject to transformations, shadow effects, global alpha, clipping paths, and global composition operators.

3.14.11.1.10. PIXEL MANIPULATION

The **getImageData(*sx*, *sy*, *sw*, *sh*)** method must return an `ImageData` object representing the underlying pixel data for the area of the canvas denoted by the rectangle which has one corner at the (*sx*, *sy*) coordinate, and that has width *sw* and height *sh*. Pixels outside the canvas must be returned as transparent black. Pixels must be returned as non-premultiplied alpha values.

`ImageData` objects must be initialised so that their **height** attribute is set to *h*, the number of rows in the image data, their **width** attribute is set to *w*, the number of physical device pixels per row in the image data, and the **data** attribute is initialised to an array of *h*×*w*×4 integers. The pixels must be represented in this array in left-to-right order, row by row, starting at the top left, with each pixel's red, green, blue, and alpha components being given in that order. Each component of each device pixel represented in this array must be in the range 0..255, representing the 8 bit value for that component. At least one pixel must be returned.

> *Note: The width and height (w and h) might be different than the sw and sh arguments to the function, e.g. if the canvas is backed by a high-resolution bitmap.*

If the `getImageData(`*sx*, *sy*, *sw*, *sh*`)` method is called with either the *sw* or *sh* arguments set to zero or negative values, the method must raise an `INDEX_SIZE_ERR` exception.

The **putImageData(*image*, *dx*, *dy*)** method must take the given `ImageData` structure, and draw it at the specified location *dx*,*dy* in the canvas coordinate space, mapping each pixel represented by the `ImageData` structure into one device pixel.

If the first argument to the method is not an object whose [[Class]] property is `ImageData`, but all of the following conditions are true, then the method must treat the first argument as if it was an `ImageData` object (and thus not raise the `TYPE_MISMATCH_ERR` exception):

- The method's first argument is an object with `width` and `height` attributes with integer values and a `data` attribute whose value is an integer array.

- The `ImageData` object's `width` is greater than zero.

- The `ImageData` object's `height` is greater than zero.

- The `ImageData` object's `width` multiplied by its `height` multiplied by 4 is equal to the number of entries in the `ImageData` object's `data` array.

- The `ImageData` object's `data` array only contains entries that are in the range 0 to 255 inclusive.

The handling of pixel rounding when the specified coordinates do not exactly map to the device coordinate space is not defined by this specification, except that the following must result in no visible changes to the rendering:

```
context.putImageData(context.getImageData(x, y, w, h), x, y);
```

...for any value of *x* and *y*. In other words, while user agents may round the arguments of the two methods so that they map to device pixel boundaries, any rounding performed must be performed consistently for both the `getImageData()` and `putImageData()` operations.

The current transformation matrix must not affect the `getImageData()` and `putImageData()` methods.

> 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. Thus, while one could create an `ImageData` object, one would net necessarily know what resolution the canvas expected (how many pixels the canvas wants to paint over one coordinate space unit pixel).
>
> In the following example, the script first obtains the size of the canvas backing store, and then generates a few new `ImageData` objects which can be used.
>
> ```
> // canvas is a reference to a <canvas> element
> // (note: this example uses JavaScript 1.7 features)
> var context = canvas.getContext('2d');
> var backingStore = context.getImageData(0, 0, canvas.width,
> canvas.height);
> var actualWidth = backingStore.width;
> var actualHeight = backingStore.height;
>
> function CreateImageData(w, h) {
>   return {
>     height: h,
>     width: w,
>     data: [i for (i in function (n) { for (let i = 0; i < n; i +=
> 1) yield 0 }(w*h*4)) ]
>   };
> }
>
> // create some plasma
> var plasma = CreateImageData(actualWidth, actualHeight);
> FillPlasma(plasma, 'green'); // green plasma
>
> // create a cloud
> var could = CreateImageData(actualWidth, actualHeight);
> FillCloud(cloud, actualWidth/2, actualHeight/2); // put a cloud
> in the middle
>
> // paint them on top of each other
> context.putImageData(plasma, 0, 0);
> context.putImageData(cloud, 0, 0);
> ```

```
            function FillPlasma(data) { ... }
            function FillCload(data, x, y) { ... }
```

3.14.11.1.11. DRAWING MODEL

When a shape or image is painted, user agents must follow these steps, in the order given (or act as if they do):

1. If the current transformation matrix is infinite, then do nothing. Abort these steps.

2. The coordinates are transformed by the current transformation matrix.

3. The shape or image is rendered, creating image *A*, as described in the previous sections. For shapes, the current fill, stroke, and line styles must be honoured.

4. The shadow is rendered from image *A*, using the current shadow styles, creating image *B*.

5. Image *A* is composited over image *B* creating the source image.

6. The source image has its alpha adjusted by globalAlpha.

7. Within the clip region (as affected by the current transformation matrix), the source image is composited over the current canvas bitmap using the current composition operator.

3.14.12. The map element

Categories

[Prose content](#).

Contexts in which this element may be used:

Where [prose content](#) is expected.

Content model:

[Prose content](#).

Element-specific attributes:

None, but the id global attribute has special requirements on this element.

DOM interface:

```
interface HTMLMapElement : HTMLElement {
  readonly attribute HTMLCollection areas;
  readonly attribute HTMLCollection images;
};
```

The map element, in conjuction with any area element descendants, defines an [image map](#).

There must always be an id attribute present on map elements.

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.

3.14.13. 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 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 ommited. 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`, `shape`, `href`, `target`, `ping`, `rel`, `media`, `hreflang`, and `type`, each must [reflect](#) the respective content attributes of the same name.

The DOM attribute `relList` must [reflect](#) the `rel` content attribute.

3.14.14. 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 hashed ID 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 hashed ID 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` or `input` element.

   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.

3.14.15. 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 logical width to the logical height in the resource. The two attributes must be omitted if the resource in question does not have both a logical width and a logical

height.

To parse the attributes, user agents must use the <u>rules for parsing dimension values</u>. 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 <u>in the rendering section</u>. 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 <u>embed</u>, <u>object</u>, and <u>video</u> elements must reflect the content attributes of the same name.

3.15. Tabular data

3.15.1. The `table` element

Categories

> <u>Prose content</u>.

Contexts in which this element may be used:

> Where <u>prose content</u> is expected.

Content model:

> In this order: optionally a <u>caption</u> element, followed by either zero or more <u>colgroup</u> elements, followed optionally by a <u>thead</u> element, followed optionally by a <u>tfoot</u> element, followed by either zero or more <u>tbody</u> elements *or* one or more <u>tr</u> elements, followed optionally by a <u>tfoot</u> element (but there can only be one <u>tfoot</u> 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;
  readonly attribute HTMLCollection rows;
  HTMLElement insertRow(in long index);
  void deleteRow(in long index);
};
```

The <u>table</u> element represents data with more than one dimension (a <u>table</u>).

> 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.

6. Zero or one `tfoot` element, if there are no other `tfoot` elements in the table.

The `table` element takes part in the [table model](#).

The **caption** DOM attribute must return, on getting, the first `caption` element child of the `table` element. On setting, 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. 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. 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 **rows** attribute must return an `HTMLCollection` rooted at the `table` node, whose filter matches only `tr` elements that are either children of the `table` element, or children of `thead`, `tbody`, or `tfoot` elements that are themselves children of the `table` element. The elements in the collection must be ordered such that those elements whose parent is a `thead` are included first, in tree order, followed by those elements whose parent is either a `table` or `tbody` element, again in tree order, followed finally by those elements whose parent is a `tfoot` element, still in tree order.

The behaviour of the **insertRow(*index*)** method depends on the state of the table. When it is called, the method must act as required by the first item in the following list of conditions that describes the state of the table and the *index* argument:

⤷ **If *index* is less than -1 or greater than the number of elements in `rows` collection:**
> The method must raise an `INDEX_SIZE_ERR` exception.

⤷ **If the `rows` collection has zero elements in it, and the `table` has no `tbody` elements in it:**
> The method must create a `tbody` element, then create a `tr` element, then append the `tr` element to the `tbody` element, then append the `tbody` element to the `table` element, and finally return the `tr` element.

⤷ **If the `rows` collection has zero elements in it:**
> The method must create a `tr` element, append it to the last `tbody` element in the table, and return the `tr` element.

⤷ **If *index* is equal to -1 or equal to the number of items in `rows` collection:**
> The method must create a `tr` element, and append it to the parent of the last `tr` element in the `rows` collection. Then, the newly created `tr` element must be returned.

⤷ **Otherwise:**
> The method must create a `tr` element, insert it immediately before the *index*th `tr` element in the `rows` collection, in the same parent, and finally must return the newly created `tr` element.

The **deleteRow(*index*)** method must remove the *index*th element in the `rows` collection from its parent. If *index* is less than zero or greater than or equal to the number of elements in the `rows` collection, the method must instead raise an `INDEX_SIZE_ERR` exception.

3.15.2. The `caption` element

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:

> No difference from HTMLElement.

The caption element represents the title of the table that is its parent, if it has a parent and that is a table element.

The caption element takes part in the table model.

3.15.3. The colgroup element

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. Its default value, which must be used if parsing the attribute as a non-negative integer returns either an error or zero, is 1.

The colgroup element and its span attribute take part in the table model.

The **span** DOM attribute must reflect the content attribute of the same name, with the exception that on setting, if the new value is 0, then an INDEX_SIZE_ERR exception must be raised.

3.15.4. The col element

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. Its default value, which must be used if parsing the attribute as a non-negative integer returns either an error or zero, is 1.

The col element and its span attribute take part in the table model.

The **span** DOM attribute must reflect the content attribute of the same name, with the exception that on setting, if the new value is 0, then an INDEX_SIZE_ERR exception must be raised.

3.15.5. The tbody element

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:

One 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.

3.15.6. 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:

> One 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.

3.15.7. 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:

One 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.

3.15.8. 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:

One 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** element must, if the element has a parent `table` element, or a parent `tbody`, `thead`, or `tfoot` element and a *grandparent* `table` element, return the index of the `tr` element in that `table` element's rows collection. If there is no such `table` element, then the attribute must return 0.

The **sectionRowIndex** DOM attribute must, if the element has a parent <u>table</u>, <u>tbody</u>, <u>thead</u>, or <u>tfoot</u> element, return the index of the <u>tr</u> element in the parent element's <u>rows</u> collection (for tables, that's the <u>rows</u> collection; for table sections, that's the <u>rows</u> collection). If there is no such parent element, then the attribute must return 0.

The **cells** attribute must return an <u>HTMLCollection</u> rooted at the <u>tr</u> element, whose filter matches only <u>td</u> and <u>th</u> elements that are children of the <u>tr</u> element.

The **insertCell(*index*)** method must act as follows:

If *index* is less than -1 or greater than the number of elements in the <u>cells</u> collection, the method must raise an INDEX_SIZE_ERR exception.

If *index* is equal to -1 or equal to the number of items in <u>cells</u> collection, the method must create a <u>td</u> element, append it to the <u>tr</u> element, and return the newly created <u>td</u> element.

Otherwise, the method must create a <u>td</u> element, insert it as a child of the <u>tr</u> element, immediately before the *index*th <u>td</u> or <u>th</u> element in the <u>cells</u> collection, and finally must return the newly created <u>td</u> element.

The **deleteCell(*index*)** method must remove the *index*th element in the <u>cells</u> collection from its parent. If *index* is less than zero or greater than or equal to the number of elements in the <u>cells</u> collection, the method must instead raise an INDEX_SIZE_ERR exception.

3.15.9. The <u>td</u> element

Categories

None.

Contexts in which this element may be used:

As a child of a <u>tr</u> element.

Content model:

[Prose content].

Element-specific attributes:

<u>colspan</u>
<u>rowspan</u>

DOM interface:

```
interface HTMLTableCellElement : HTMLElement {
            attribute long colSpan;
            attribute long rowSpan;
  readonly attribute long cellIndex;
};
```

The <u>td</u> element represents a data [cell] in a table.

The <u>td</u> element may have a **colspan** content attribute specified, whose value must be a [valid non-negative integer] greater than zero. Its default value, which must be used if [parsing the attribute as a non-negative integer] returns either an error or zero, is 1.

The <u>td</u> element may also have a **rowspan** content attribute specified, whose value must be a [valid]

non-negative integer. Its default value, which must be used if parsing the attribute as a non-negative integer returns an error, is also 1.

The td element and its colspan and rowspan attributes take part in the table model.

The **colspan** DOM attribute must reflect the content attribute of the same name, with the exception that on setting, if the new value is 0, then an INDEX_SIZE_ERR exception must be raised.

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

The **cellIndex** DOM attribute must, if the element has a parent tr element, return the index of the cell's element in the parent element's cells collection. If there is no such parent element, then the attribute must return 0.

> There has been some suggestion that the headers attribute from HTML4, or some other mechanism that is more powerful than scope="", should be included. This has not yet been considered.

3.15.10. 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 **colspan** content attribute specified, whose value must be a valid non-negative integer greater than zero. Its default value, which must be used if parsing the attribute as a non-negative integer returns either an error or zero, is 1.

The th element may also have a **rowspan** content attribute specified, whose value must be a valid non-negative integer. Its default value, which must be used if parsing the attribute as a non-negative integer returns an error, is also 1.

The th element may have a **scope** content attribute specified. 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`, `rowspan`, and `scope` attributes take part in the [table model](#).

The `scope` DOM attribute must [reflect](#) the content attribute of the same name.

The `HTMLTableHeaderCellElement` interface inherits from the `HTMLTableCellElement` interface and therefore also has the DOM attributes defined above in the `td` section.

3.15.11. Processing model

The various table elements and their content attributes together define the **table model**.

A **table** consists of cells aligned on a two-dimensional grid of **slots** with coordinates (*x*, *y*). The grid is finite, and is either empty or has one or more slots. If the grid has one or more slots, then the *x* coordinates are always in the range $1 \le x \le x_{max}$, and the *y* coordinates are always in the range $1 \le y \le y_{max}$. If one or both of x_{max} and y_{max} are zero, then the table is empty (has no slots). Tables correspond to `table` elements.

A **cell** is a set of slots anchored at a slot ($cell_x$, $cell_y$), and with a particular *width* and *height* such that the cell covers all the slots with coordinates (*x*, *y*) where $cell_x \le x < cell_x+width$ and $cell_y \le y < cell_y+height$. Cell can either be *data cells* or *header cells*. Data cells correspond to `td` elements, and have zero or more associated header cells. Header cells correspond to `th` elements.

A **row** is a complete set of slots from *x*=1 to *x*=x_{max}, for a particular value of *y*. Rows correspond to `tr` elements.

A **column** is a complete set of slots from *y*=1 to *y*=y_{max}, for a particular value of *x*. Columns can correspond to `col` elements, but in the absense of `col` elements are implied.

A **row group** is a set of [rows](#) anchored at a slot (1, $group_y$) with a particular *height* such that the row group covers all the slots with coordinates (*x*, *y*) where $1 \le x < x_{max}$ and $group_y \le y < group_y+height$. Row groups correspond to `tbody`, `thead`, and `tfoot` elements. Not every row is necessarily in a row group.

A **column group** is a set of [columns](#) anchored at a slot ($group_x$, 1) with a particular *width* such that

the column group covers all the slots with coordinates (x, y) where $group_x ≤ x < group_x+width$ and $1 ≤ y < y_{max}$. Column groups correspond to `colgroup` elements. Not every column is necessarily in a column group.

Row groups cannot overlap each other. Similarly, column groups cannot overlap each other.

A cell cannot 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.

3.15.11.1. Forming a table

To determine which elements correspond to which slots in a table associated with a `table` element, to determine the dimensions of the table (x_{max} and y_{max}), and to determine if there are any table model errors, user agents must use the following algorithm:

1. Let x_{max} be zero.

2. Let y_{max} be zero.

3. Let *the table* be the table represented by the `table` element. The x_{max} and y_{max} variables give *the table*'s extent. *The table* is initially empty.

4. If the `table` element has no table children, then return *the table* (which will be empty), and abort these steps.

5. Let the *current element* be the first element child of the `table` element.

   If a step in this algorithm ever requires the *current element* to be advanced to the next child of the `table` when there is no such next child, then the algorithm must be aborted at that point and the algorithm must return *the table*.

6. While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:

   - `caption`
   - `colgroup`
   - `thead`
   - `tbody`
   - `tfoot`
   - `tr`

7. If the *current element* is a `caption`, then that is the `caption` element associated with *the table*. Otherwise, it has no associated `caption` element.

8. If the *current element* is a `caption`, then while the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:

   - `colgroup`
   - `thead`
   - `tbody`
   - `tfoot`

- $\circ$ <u>tr</u>

(Otherwise, the *current element* will already be one of those elements.)

9. If the *current element* is a <u>colgroup</u>, follow these substeps:

1. *Column groups.* Process the *current element* according to the appropriate one of the following two cases:

   $\hookrightarrow$ **If the *current element* has any <u>col</u> element children**

   Follow these steps:

   1. Let x_{start} have the value x_{max}+1.

   2. Let the *current column* be the first <u>col</u> element child of the <u>colgroup</u> element.

   3. *Columns.* If the *current column* <u>col</u> element has a <u>span</u> attribute, then parse its value using the <u>rules for parsing non-negative integers</u>.

      If the result of parsing the value is not an error or zero, then let *span* be that value.

      Otherwise, if the <u>col</u> element has no <u>span</u> attribute, or if trying to parse the attribute's value resulted in an error, then let *span* be 1.

   4. Increase x_{max} by *span*.

   5. Let the last *span* <u>columns</u> in *the table* correspond to the *current column* <u>col</u> element.

   6. If *current column* is not the last <u>col</u> element child of the <u>colgroup</u> element, then let the *current column* be the next <u>col</u> element child of the <u>colgroup</u> element, and return to the third step of this innermost group of steps (columns).

   7. Let all the last <u>columns</u> in *the table* from x=x_{start} to x=x_{max} form a new <u>column group</u>, anchored at the slot (x_{start}, 1), with width x_{max}-x_{start}-1, corresponding to the <u>colgroup</u> element.

   $\hookrightarrow$ **If the *current element* has no <u>col</u> element children**

   1. If the <u>colgroup</u> element has a <u>span</u> attribute, then parse its value using the <u>rules for parsing non-negative integers</u>.

      If the result of parsing the value is not an error or zero, then let *span* be that value.

      Otherwise, if the <u>colgroup</u> element has no <u>span</u> attribute, or if trying to parse the attribute's value resulted in an error, then let *span* be 1.

   2. Increase x_{max} by *span*.

   3. Let the last *span* <u>columns</u> in *the table* form a new <u>column group</u>, anchored at the slot (x_{max}-*span*+1, 1), 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 step 1 in these substeps (column groups).

10. Let $y_{current}$ be zero. When the algorithm is aborted, if $y_{current}$ does not equal y_{max}, then that is a [table model error](#).

11. Let the *list of downward-growing cells* be an empty list.

12. *Rows.* While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table`:

- `thead`
- `tbody`
- `tfoot`
- `tr`

13. If the *current element* is a `tr`, then run the [algorithm for processing rows](#) (defined below), then return to the previous step (rows).

14. Otherwise, run the [algorithm for ending a row group](#).

15. Let y_{start} have the value $y_{max}+1$.

16. For each `tr` element that is a child of the *current element*, in tree order, run the [algorithm for processing rows](#) (defined below).

17. If $y_{max} \geq y_{start}$, then let all the last [rows](#) in *the table* from y=y_{start} to y=y_{max} form a new [row group](#), anchored at the slot with coordinate (1, y_{start}), with height y_{max}-y_{start}+1, corresponding to the *current element*.

18. Run the [algorithm for ending a row group](#) again.

19. Return to step 12 (rows).

The **algorithm for ending a row group**, which is invoked by the set of steps above when starting and ending a block of rows, is:

1. If $y_{current}$ is less than y_{max}, then this is a [table model error](#).

2. While $y_{current}$ is less than y_{max}, follow these steps:

1. Increase $y_{current}$ by 1.

2. Run the [algorithm for growing downward-growing cells](#).

3. Empty the *list of downward-growing cells*.

The **algorithm for processing rows**, which is invoked by the set of steps above for processing tr elements, is:

1. Increase $y_{current}$ by 1.

2. Run the <u>algorithm for growing downward-growing cells</u>.

3. Let $x_{current}$ be 1.

4. If the tr element being processed contains no td or th elements, then abort this set of steps and return to the algorithm above.

5. Let *current cell* be the first td or th element in the tr element being processed.

6. *Cells.* While $x_{current}$ is less than or equal to x_{max} and the slot with coordinate ($x_{current}$, $y_{current}$) already has a cell assigned to it, increase $x_{current}$ by 1.

7. If $x_{current}$ is greater than x_{max}, increase x_{max} by 1 (which will make them equal).

8. If the *current cell* has a $colspan$ attribute, then parse that attribute's value, and let *colspan* be the result.

   If parsing that value failed, or returned zero, or if the attribute is absent, then let *colspan* be 1, instead.

9. If the *current cell* has a $rowspan$ attribute, then parse that attribute's value, and let *rowspan* be the result.

   If parsing that value failed or if the attribute is absent, then let *rowspan* be 1, instead.

10. If *rowspan* is zero, then let *cell grows downward* be true, and set *rowspan* to 1. Otherwise, let *cell grows downward* be false.

11. If $x_{max} < x_{current}+colspan$-1, then let x_{max} be $x_{current}+colspan$-1.

12. If $y_{max} < y_{current}+rowspan$-1, then let y_{max} be $y_{current}+rowspan$-1.

13. Let the slots with coordinates (x, y) such that $x_{current} \le x < x_{current}+colspan$ and $y_{current} \le y < y_{current}+rowspan$ be covered by a new <u>cell</u> *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 <u>algorithm for assigning header cells to data cells</u> described in the next section.

    If any of the slots involved already had a <u>cell</u> covering them, then this is a <u>table model error</u>. Those slots now have two cells overlapping.

14. If *cell grows downward* is true, then add the tuple {*c*, $x_{current}$, *colspan*} to the *list of downward-growing cells*.

15. Increase $x_{current}$ by *colspan*.

16. If *current cell* is the last td or th element in the tr element being processed, then abort this set of steps and return to the algorithm above.

17. Let *current cell* be the next td or th element in the tr element being processed.

18. Return to step 5 (cells).

The **algorithm for growing downward-growing cells**, used when adding a new row, is as follows:

1. If the *list of downward-growing cells* is empty, do nothing. Abort these steps; return to the step that invoked this algorithm.

2. Otherwise, if y_{max} is less than $y_{current}$, then increase y_{max} by 1 (this will make it equal to $y_{current}$).

3. For each {*cell*, *cell$_x$*, *width*} tuple in the *list of downward-growing cells*, extend the [cell](#) *cell* so that it also covers the slots with coordinates (x, $y_{current}$), where $cell_x \leq x < cell_x + width$-1.

If, after establishing which elements correspond to which slots, there exists a [column](#) in the [table](#) containing only slots that do not have a [cell](#) anchored to them, then this is a [table model error](#).

3.15.11.2. Forming relationships between data cells and header cells

Each data cell can be assigned zero or more header cells. The **algorithm for assigning header cells to data cells** is as follows.

For each header cell in the table, in [tree order](#):

1. Let (*header$_x$*, *header$_y$*) be the coordinate of the slot to which the header cell is anchored.

2. 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**

   > Assign the header cell to any data cells anchored at slots with coordinates (*data$_x$*, *data$_y$*) where $header_x < data_x \leq x_{max}$ and $data_y = header_y$.

   ↳ **If it is in the [column](#) state**

   > Assign the header cell to any data cells anchored at slots with coordinates (*data$_x$*, *data$_y$*) where $data_x = header_x$ and $header_y < data_y \leq y_{max}$.

   ↳ **If it is in the [row group](#) state**

   > If the header cell is not in a [row group](#), then don't assign the header cell to any data cells.
   >
   > Otherwise, let (1, *group$_y$*) be the slot at which the row group is anchored, let *height* be the number of rows in the row group, and assign the header cell to any data cells anchored at slots with coordinates (*data$_x$*, *data$_y$*) where $header_x \leq data_x \leq x_{max}$ and $header_y \leq data_y < group_y + height$.

   ↳ **If it is in the [column group](#) state**

   > If the header cell is not in a [column group](#), then don't assign the header cell to any data cells.
   >
   > Otherwise, let (*group$_x$*, 1) be the slot at which the column group is anchored, let *width* be the number of columns in the column group, and assign the header cell to any data cells anchored at slots with coordinates (*data$_x$*, *data$_y$*) where $header_x \leq data_x < group_x + width$ and $header_y \leq data_y \leq y_{max}$.

   ↳ **Otherwise, it is in the *auto* state**

   > If the header cell is not in the first row of the table, or not in the first cell of a row,

then don't assign the header cell to any data cells.

Otherwise, if the header cell is in the first row of the table, assign the header cell to any data cells anchored at slots with coordinates ($data_x$, $data_y$) where $data_x = header_x$ and $header_y < data_y \leq y_{max}$.

Otherwise, the header cell is in the first column of the table; assign the header cell to any data cells anchored at slots with coordinates ($data_x$, $data_y$) where $header_x < data_x \leq x_{max}$ and $data_y = header_y$.

3.16. Forms

This section will contain definitions of the `form` element and so forth.

This section will be a rewrite of the HTML4 Forms and Web Forms 2.0 specifications, with hopefully no normative changes.

3.16.1. The `form` element

3.16.2. The `fieldset` element

3.16.3. The `input` element

3.16.4. The `button` element

3.16.5. The `label` element

3.16.6. The `select` element

3.16.7. The `datalist` element

3.16.8. The `optgroup` element

3.16.9. The `option` element

3.16.10. The `textarea` element

3.16.11. The `output` element

3.16.12. Processing model

See WF2 for now

3.16.12.1. Form submission

See WF2 for now

3.17. Scripting

3.17.1. The `script` element

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`

DOM interface:

```
interface HTMLScriptElement : HTMLElement {
          attribute DOMString src;
          attribute boolean async;
          attribute boolean defer;
          attribute DOMString type;
          attribute DOMString text;
};
```

The `script` element allows authors to include dynamic script in their documents.

When the **src** attribute is set, the `script` element refers to an external file. The value of the attribute must be a URI (or IRI).

If the `src` attribute is not set, then the script is given by the contents of the element.

The language of the script may be given by the **type** attribute. If the attribute is present, its value must be a valid MIME type, optionally with parameters. [RFC2046]

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`, `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 three 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 piece of metadata is *the script's type*. It is 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 but 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, 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, if 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 scripting is disabled, or if the `Document` has `designMode` enabled, 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.

3. The user agent must set the element's "already executed" flag.

4. 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.

5. 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

↳ **If the element has an `async` attribute and a `src` attribute**

> The element must be added to the end of the list of scripts that will execute asynchronously. The user agent must jump to the next set of steps once the script is ready.

↳ **If the element has an `async` attribute but no `src` attribute, and the list of scripts that will execute asynchronously is not empty**

> The element must be added to the end of the list of scripts that will execute asynchronously.

↳ **If the element has a `src` attribute and has been flagged as "parser-inserted"**

> The element is the script that will execute as soon as the parser resumes. (There can only be one such script at a time.)

↳ **If the element has a `src` attribute**

> The element must be added to the end of the list of scripts that will execute as soon as possible. The user agent must jump to the next set of steps when the script is ready.

↳ **Otherwise**

> The user agent must immediately execute the script, even if other scripts are already executing.

When a script completes loading: If a script whose element was added to one of the lists mentioned above completes loading while the document is still being parsed, then the parser handles it. Otherwise, when a script completes loading, the UA must 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 script that will execute as soon as the parser resumes:**

The script will be handled when the parser resumes (amazingly enough).

The download of an external script must delay the `load` event.

Executing a script block: If the load resulted in an error (for example a DNS error, or an HTTP 404 error), then executing the script must just consist of firing an `error` event at the element.

If the load was successful, then first the user agent must fire a `load` event at the element, and then, if scripting is enabled, and the `Document` does not have `designMode` enabled, and the `Document` is the active document in its browsing context, the user agent must execute the script:

If the script is from an external file, then that file must be used as the file to execute.

If the script is inline, then, for scripting languages that consist of pure text, user agents must use the value of the DOM `text` attribute (defined below) as the script to execute, and for XML-based scripting languages, user agents must use all the child nodes of the `script` element as the script to execute.

In any case, the user agent must execute the script according to the semantics defined by the language associated with *the script's type* (see the scripting languages section below).

Scripts must be executed in the scope of the browsing context of the element's `Document`.

> *Note: The element's attributes' values might have changed between when the element was inserted into the document and when the script has finished loading, as may its other attributes; similarly, the element itself might have been taken back out of the DOM, or had other changes made. These changes do not in any way affect the above steps; only the values of the attributes at the time the `script` element is first inserted into the document matter.*

The DOM attributes **src**, **type**, **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.

3.17.1.1. 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.

3.17.2. 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:

> When scripting is disabled, in a `head` element: in any order, zero or more `link` elements, zero or more `style` elements, and zero or more `meta` elements.
> When scripting is disabled, not in a `head` element: transparent, but there must be no `noscript` element descendants.
> When scripting is enabled: text that conforms to the requirements given in the prose.

Element-specific attributes:

> None.

DOM interface:

> No difference from `HTMLElement`.

The `noscript` element does not represent anything. It is used to present different markup to user agents that support scripting and those that don't support scripting, by affecting how the document is parsed.

The `noscript` element must not be used in XML documents.

When used in HTML documents, the allowed content model depends on whether scripting is enabled or not, and whether the element is in a `head` element or not.

In a `head` element, if scripting is disabled, then the content model of a `noscript` element must contain only `link`, `style`, and `meta` elements. If scripting is enabled, 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* 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 scripting is disabled, 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 scripting is enabled, then the content model of a `noscript` element is

text, except that the text must be such that running the following algorithm results in a conforming document with no `noscript` elements and no `script` elements, and such that no step in the algorithm causes an HTML parser to flag a parse error:

1. Remove every `script` element from the document.

2. Make a list of every `noscript` element in the document. For every `noscript` element in that list, perform the following steps:
    1. Let the *parent element* be the parent element of the `noscript` element.
    2. Take all the children of the *parent element* that come before the `noscript` element, and call these elements *the before children*.
    3. Take all the children of the *parent element* that come *after* the `noscript` element, and call these elements *the after children*.
    4. Let *s* be the concatenation of all the text node children of the `noscript` element.
    5. Set the `innerHTML` attribute of the *parent element* to the value of *s*. (This, as a side-effect, causes the `noscript` element to be removed from the document.)
    6. Insert *the before children* at the start of the *parent element*, preserving their original relative order.
    7. Insert *the after children* at the end of the *parent element*, preserving their original relative order.

The `noscript` element has no other requirements. In particular, children of the `noscript` element are not exempt from form submission, scripting, and so forth, even when scripting is enabled.

Note: All these contortions are required because, for historical reasons, the `noscript` element causes the HTML parser to act differently based on whether scripting is enabled or not. The element is not allowed in XML, because in XML the parser is not affected by such state, and thus the element would not have the desired effect.

3.17.3. The `event-source` 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 `event-source` element represents a target for events generated by a remote server.

The **src** attribute, if specified, must give a URI (or IRI) pointing to a resource that uses the `application/x-dom-event-stream` format.

When the element is inserted into the document, if it has the `src` attribute specified, the user agent must act as if the `addEventSource()` method on the `event-source` element had been invoked with the URI resulting from resolving the `src` attribute's value to an absolute URI.

While the element is in a document, if its `src` attribute is mutated, the user agent must act as if first the `removeEventSource()` method on the `event-source` element had been invoked with the URI resulting from resolving the old value of the attribute to an absolute URI, and then as if the `addEventSource()` method on the element had been invoked with the URI resulting from resolving the *new* value of the `src` attribute to an absolute URI.

When the element is removed from the document, if it has the `src` attribute specified, or, when the `src` attribute is about to be removed, the user agent must act as if the `removeEventSource()` method on the `event-source` element had been invoked with the URI resulting from resolving the `src` attribute's value to an absolute URI.

There can be more than one `event-source` 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.

3.18. Interactive elements

3.18.1. The `details` element

Categories

Prose element.

Contexts in which this element may be used:

Where [prose content](#) is expected.

Content model:

One `legend` element followed by [prose 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.

<div style="border:2px solid red">

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.

</div>

<div style="border:2px solid red">

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?

</div>

3.18.2. The **datagrid** element

Categories

> Prose element.
> Interactive element.

Contexts in which this element may be used:

> Where prose content is expected.

Content model:

> Either: Nothing.
> Or: Prose content, but where the first element child node, if any, is not a table 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.

3.18.2.1. The `datagrid` data model

This section is non-normative.

In the `datagrid` data model, data is structured as a set of rows representing a tree, each row being split into a number of columns. The columns are always present in the data model, although individual columns may be hidden in the presentation.

Each row can have child rows. Child rows may be hidden or shown, by closing or opening (respectively) the parent row.

Rows are referred to by the path along the tree that one would take to reach the row, using zero-based indices. Thus, the first row of a list is row "0", the second row is row "1"; the first child row of the first row is row "0,0", the second child row of the first row is row "0,1"; the fourth child of the seventh child of the third child of the tenth row is "9,2,6,3", etc.

The columns can have captions. Those captions are not considered a row in their own right, they are obtained separately.

Selection of data in a `datagrid` operates at the row level. If the `multiple` attribute is present, multiple rows can be selected at once, otherwise the user can only select one row at a time.

The datagrid element can be disabled entirely by setting the disabled attribute.

Columns, rows, and cells can each have specific flags, known as classes, applied to them by the data provider. These classes affect the functionality of the datagrid element, and are also passed to the style system. They are similar in concept to the class attribute, except that they are not specified on elements but are given by scripted data providers.

3.18.2.2. How rows are identified

The chains of numbers that give a row's path, or identifier, are represented by objects that implement the RowSpecification interface.

```
interface RowSpecification {
  // binding-specific interface
};
```

In ECMAScript, two classes of objects are said to implement this interface: Numbers representing non-negative integers, and homogeneous arrays of Numbers representing non-negative integers. Thus, [1,0,9] is a RowSpecification, as is 1 on its own. However, [1,0.2,9] is not a RowSpecification object, since its second value is not an integer.

User agents must always represent RowSpecifications in ECMAScript by using arrays, even if the path only has one number.

The root of the tree is represented by the empty path; in ECMAScript, this is the empty array ([]). Only the getRowCount() and GetChildAtPosition() methods ever get called with the empty path.

3.18.2.3. The data provider interface

The conformance criteria in this section apply to any implementation of the DataGridDataProvider, including (and most commonly) the content author's implementation(s).

```
// To be implemented by Web authors as a JS object
interface DataGridDataProvider {
  void initialize(in HTMLDataGridElement datagrid);
  unsigned long getRowCount(in RowSpecification row);
  unsigned long getChildAtPosition(in RowSpecification parentRow, in
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 $0 \le x < n$, where n is the value returned by getRowCount(*parentRow*).

getColumnCount()

> Must return the number of columns currently in the data model (including columns that might be hidden). May be omitted if there is only one column. If the value that this method would return changes, the datagrid's updateEverything() method must be called.

getCaptionText(*column*)

> Must return the caption, or label, for column *column*. May be omitted if the columns have no captions. If the value that this method would return changes, the datagrid's updateColumnChanged() method must be called with the appropriate column index.

getCaptionClasses(*column*, *classes*)

> Must add the classes that apply to column *column* to the *classes* object. May be omitted if the columns have no special classes. If the classes that this method would add changes, the datagrid's updateColumnChanged() method must be called with the appropriate column index. Some classes have predefined meanings.

getRowImage(*row*)

> Must return a URI to an image that represents row *row*, or the empty string if there is no applicable image. May be omitted if no rows have associated images. If the value that this method would return changes, the `datagrid`'s update methods must be called to update the row in question.

getRowMenu(*row*)

> Must return an `HTMLMenuElement` object that is to be used as a context menu for row *row*, or null if there is no particular context menu. May be omitted if none of the rows have a special context menu. As this method is called immediately before showing the menu in question, no precautions need to be taken if the return value of this method changes.

getRowClasses(*row*, *classes*)

> Must add the classes that apply to row *row* to the *classes* object. May be omitted if the rows have no special classes. If the classes that this method would add changes, the `datagrid`'s update methods must be called to update the row in question. Some classes have [predefined meanings](#).

getCellData(*row*, *column*)

> Must return the value of the cell on row *row* in column *column*. For text cells, this must be the text to show for that cell. For [progress bar cells](#), this must be either a floating point number in the range 0.0 to 1.0 (converted to a string representation), indicating the fraction of the progress bar to show as full (1.0 meaning complete), or the empty string, indicating an indeterminate progress bar. If the value that this method would return changes, the `datagrid`'s update methods must be called to update the rows that changed. If only one cell changed, the `updateCellChanged()` method may be used.

getCellClasses(*row*, *column*, *classes*)

> Must add the classes that apply to the cell on row *row* in column *column* to the *classes* object. May be omitted if the cells have no special classes. If the classes that this method would add changes, the `datagrid`'s update methods must be called to update the rows or cells in question. Some classes have [predefined meanings](#).

toggleColumnSortState(*column*)

> Called by the `datagrid` when the user tries to sort the data using a particular column *column*. The data provider must update its state so that the `GetChildAtPosition()` method returns the new order, and the classes of the columns returned by `getCaptionClasses()` represent the new sort status. There is no need to tell the `datagrid` that it the data has changed, as the `datagrid` automatically assumes that the entire data model will need updating.

setCellCheckedState(*row*, *column*, *state*)

> Called by the `datagrid` when the user changes the state of a checkbox cell on row *row*, column *column*. The checkbox should be toggled to the state given by *state*, which is a positive integer (1) if the checkbox is to be checked, zero (0) if it is to be unchecked, and a negative number (-1) if it is to be set to the indeterminate state. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

cycleCell(*row*, *column*)

> Called by the `datagrid` when the user changes the state of a cyclable cell on row *row*, column *column*. The data provider should change the state of the cell to the new state, as appropriate. There is no need to tell the `datagrid` that the cell has changed, as the

`datagrid` automatically assumes that the given cell will need updating.

editCell(*row*, *column*, *data*)

Called by the `datagrid` when the user edits the cell on row *row*, column *column*. The new value of the cell is given by *data*. The data provider should update the cell accordingly. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

The following classes (for rows, columns, and cells) may be usefully used in conjunction with this interface:

Class name	Applies to	Description
checked	Cells	The cell has a checkbox and it is checked. (The `cyclable` and `progress` classes override this, though.)
cyclable	Cells	The cell can be cycled through multiple values. (The `progress` class overrides this, though.)
editable	Cells	The cell can be edited. (The `cyclable`, `progress`, `checked`, `unchecked` and `indeterminate` classes override this, though.)
header	Rows	The row is a heading, not a data row.
indeterminate	Cells	The cell has a checkbox, and it can be set to an indeterminate state. If neither the `checked` nor `unchecked` classes are present, then the checkbox is in that state, too. (The `cyclable` and `progress` classes override this, though.)
initially-hidden	Columns	The column will not be shown when the `datagrid` is initially rendered. If this class is not present on the column when the `datagrid` is initially rendered, the column will be visible if space allows.
initially-closed	Rows	The row will be closed when the `datagrid` is initially rendered. If neither this class nor the `initially-open` class is present on the row when the `datagrid` is initially rendered, the initial state will depend on platform conventions.
initially-open	Rows	The row will be opened when the `datagrid` is initially rendered. If neither this class nor the `initially-closed` class is present on the row when the `datagrid` is initially rendered, the initial state will depend on platform conventions.
progress	Cells	The cell is a progress bar.
reversed	Columns	If the cell is sorted, the sort direction is descending, instead of ascending.
selectable-separator	Rows	The row is a normal, selectable, data row, except that instead of having data, it only has a separator. (The `header` and `separator` classes override this, though.)
separator	Rows	The row is a separator row, not a data row. (The `header` class overrides this, though.)
sortable	Columns	The data can be sorted by this column.
sorted	Columns	The data is sorted by this column. Unless the `reversed` class is also present, the sort direction is ascending.

unchecked	Cells	The cell has a checkbox and, unless the `checked` class is present as well, it is unchecked. (The `cyclable` and `progress` classes override this, though.)

3.18.2.4. The default data provider

The user agent must supply a default data provider for the case where the `datagrid`'s `data` attribute is null. It must act as described in this section.

The behaviour of the default data provider depends on the nature of the first element child of the `datagrid`.

↳ **While the first element child is a `table` 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 <u>ignored</u> completely by this algorithm.*
>
> **getChildAtPosition(*row, i*)**: The default data provider must return the mapping appropriate to the <u>current sort order</u>.
>
> **getColumnCount()**: The number of columns returned must be the number of `td` element children in the first `tr` element child of the first `tbody` element child of the `table`, if there are any such `tbody` elements. If there are no such `tbody` elements, then it must be the number of `td` element children in the first `tr` element child of the `table`, if any, or otherwise 1. If the number that would be returned by these rules is 0, then 1 must be returned instead.
>
> **getCaptionText(*i*)**: If the `table` has no `thead` element child, or if its first `thead` element child has no `tr` element child, the default data provider must return the empty string for all captions. Otherwise, the value of the `textContent` attribute of the *i*th `th` element child of the first `tr` element child of the first `thead` element child of the `table` element must be returned. If there is no such `th` element, the empty string must be returned.
>
> **getCaptionClasses(*i, classes*)**: If the `table` has no `thead` element child, or if its first `thead` element child has no `tr` element child, the default data provider must not add any classes for any of the captions. Otherwise, each class in the `class` attribute of the *i*th `th` element child of the first `tr` element child of the first `thead` element child of the `table` element must be added to the *classes*. If there is no such `th` element, no classes must be added. The user agent must then:
>
> > 1. Remove the `sorted` and `reversed` classes.

2. If the `table` element has a `class` attribute that includes the `sortable` class, add the `sortable` class.

3. If the column is the one currently being used to sort the data, add the `sorted` class.

4. If the column is the one currently being used to sort the data, and it is sorted in descending order, add the `reversed` class as well.

The various row- and cell- related methods operate relative to a particular element, the element of the row or cell specified by their arguments.

For rows: Since the default data provider for a `table` always returns 0 as the number of children for any row other than the root, the path to the row passed to these methods will always consist of a single number. In the prose below, this number is referred to as *i*.

If the `table` has `tbody` element children, the element for the *i*th row is the *i*th `tr` element that is a child of a `tbody` element that is a child of the `table` element. If the `table` does not have `tbody` element children, then the element for the *i*th real row is the *i*th `tr` element that is a child of the `table` element.

For cells: Given a row and its element, the row's *i*th cell's element is the *i*th `td` element child of the row element.

Note: The `colspan` and `rowspan` attributes are ignored by this algorithm.

`getRowImage(i)` : If the row's first cell's element has an `img` element child, then the URI of the row's image is the URI of the first `img` element child of the row's first cell's element. Otherwise, the URI of the row's image is the empty string.

`getRowMenu(i)` : If the row's first cell's element has a `menu` element child, then the row's menu is the first `menu` element child of the row's first cell's element. Otherwise, the row has no menu.

`getRowClasses(i, classes)` : The default data provider must never add a class to the row's classes.

`toggleColumnSortState(i)` : If the data is already being sorted on the given column, then the user agent must change the current sort mapping to be the inverse of the current sort mapping; if the sort order was ascending before, it is now descending, otherwise it is now ascending. Otherwise, if the current sort column is another column, or the data model is currently not sorted, the user agent must create a new mapping, which maps rows in the data model to rows in the DOM so that the rows in the data model are sorted by the specified column, in ascending order. (Which sort comparison operator to use is left up to the UA to decide.)

When the sort mapping is changed, the values returned by the `getChildAtPosition()` method for the default data provider will change appropriately.

`getCellData(i, j)`, **`getCellClasses(i, j, classes)`**, **`getCellCheckedState(i, j, state)`**, **`cycleCell(i, j)`**, and **`editCell(i, j, data)`** : See the common definitions below.

The data provider must call the `datagrid`'s update methods appropriately whenever the

descendants of the <u>datagrid</u> mutate. For example, if a <u>tr</u> is removed, then the <u>updateRowsRemoved()</u> 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 <u>table</u> element stops being the first child of the <u>datagrid</u>, then the data provider must call the <u>updateEverything()</u> method on the <u>datagrid</u>. 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 <u>updateEverything()</u> 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 <u>datagrid</u> element (the <u>select</u> or <u>datalist</u> element), that skips all nodes other than <u>optgroup</u> and <u>option</u> elements, as well as any descendents of any <u>option</u> 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/>
>     <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.

<u>getRowCount(*row*)</u> 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 <u>select</u> 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 initialised, reflect the selection in the select, reflect the multiple attribute somehow.

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate.

↳ **While the first element child is another element**

The default data provider must return 1 for the column count, the empty string for the column's caption, and must not add any classes to the column's classes.

For the rows, assume the existence of a node filter view of the descendants of the `datagrid` that skips all nodes other than `li`, `h1`-`h6`, and `hr` elements, and skips any descendants of `menu` elements.

Given this view, each element in the view represents a row in the data model. The element corresponding to a path *row* is the one obtained by drilling into the view, taking the child given by the path each time. The element of the row of a particular method call is the element given by drilling into the view along the path given by the method's arguments.

`getRowCount(row)` must return the number of child elements in this view for the given row, or the number of elements at the root of the view if the *row* is empty.

> In the following example, the elements are identified by the paths given by their child text nodes:
>
> ```
>     <datagrid>
>      <ol>
>       <li> row 0 </li>
> ```

```
     <li> row 1
      <ol>
       <li> row 1,0 </li>
      </ol>
     </li>
     <li> row 2 </li>
    </ol>
  </datagrid>
```

In this example, only the `li` elements actually appear in the data grid; the `ol` element does not affect the data grid's processing model.

`getChildAtPosition(row, position)` must return *position*. (The generic default data provider does not support sorting the data grid.)

`getRowImage(i)` must return the URI of the image given by the first `img` element descendant (in the real DOM) of the row's element, that is not also a descendant of another element in the filtered view that is a descendant of the row's element.

In the following example, the row with path "1,0" returns "http://example.com/a" as its image URI, and the other rows (including the row with path "1") return the empty string:

```
  <datagrid>
   <ol>
    <li> row 0 </li>
    <li> row 1
     <ol>
       <li> row 1,0 <img src="http://example.com/a" alt="">
  </li>
     </ol>
    </li>
    <li> row 2 </li>
   </ol>
  </datagrid>
```

`getRowMenu(i)` must return the first `menu` element descendant (in the real DOM) of the row's element, that is not also a descendant of another element in the filtered view that is a decsendant of the row's element. (This is analogous to the image case above.)

`getRowClasses(i, classes)` must add the classes from the following list to *classes* when their condition is met:

- If the row's element contains other elements that are also in the view, and the element's `class` attribute contains the `closed` class: `initially-closed`

- If the row's element contains other elements that are also in the view, and the element's `class` attribute contains the `open` class: `initially-open`

- If the row's element is an `h1`-`h6` element: `header`

- If the row's element is an `hr` element: `separator`

The `getCellData(i, j)`, `getCellClasses(i, j, classes)`, `getCellCheckedState(i, j, state)`, `cycleCell(i, j)`, and `editCell(i,`

j, *data*) methods must act as described in the common definitions below, treating the row's element as being the cell's element.

> selection handling?

The data provider must call the datagrid's update methods appropriately whenever the descendants of the datagrid mutate.

↳ **Otherwise, while there is no element child**

The data provider must return 0 for the number of rows, 1 for the number of columns, the empty string for the first column's caption, and must add no classes when asked for that column's classes. If the datagrid's child list changes such that there is a first element child, then the data provider must call the updateEverything() method on the datagrid.

3.18.2.4.1. COMMON DEFAULT DATA PROVIDER METHOD DEFINITIONS FOR CELLS

These definitions are used for the cell-specific methods of the default data providers (other than in the select/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 optgroups are not displayed).

The getCellClasses() method must add the cyclable class and then all the classes of the current option element.

The cycleCell() method must change the selection of the select element such that the next option element after the current option element is the only one that is selected (in tree order). If the current option element is the last option element descendent of the select, then the first option element descendent must be selected instead.

The setCellCheckedState() and editCell() methods must do nothing.

Progress bar cells

If the first element child of a cell's element is a progress element, then the cell acts as a progress bar cell.

The getCellData() method must return the value returned by the progress element's position DOM attribute.

The getCellClasses() method must add the progress class.

The setCellCheckedState(), cycleCell(), and editCell() methods must do nothing.

Checkbox cells

> If the first element child of a cell's element is an `input` element that has a `type` attribute with the value `checkbox`, then the cell acts as a check box cell.
>
> The `getCellData()` method must return the `textContent` of the cell element.
>
> The `getCellClasses()` method must add the `checked` class if the `input` element is checked, and the `unchecked` class otherwise.
>
> The `setCellCheckedState()` method must set the `input` element's checkbox state to checked if the method's third argument is 1, and to unchecked otherwise.
>
> The `cycleCell()` and `editCell()` methods must do nothing.

Editable cells

> If the first element child of a cell's element is an `input` element that has a `type` attribute with the value `text` or that has no `type` attribute at all, then the cell acts as an editable cell.
>
> The `getCellData()` method must return the `value` of the `input` element.
>
> The `getCellClasses()` method must add the `editable` class.
>
> The `editCell()` method must set the `input` element's `value` DOM attribute to the value of the third argument to the method.
>
> The `setCellCheckedState()` and `cycleCell()` methods must do nothing.

3.18.2.5. Populating the `datagrid` element

A `datagrid` must be disabled until its end tag has been parsed (in the case of a `datagrid` element in the original document markup) or until it has been inserted into the document (in the case of a dynamically created element). After that point, the element must fire a single `load` event at itself, which doesn't bubble and cannot be canceled.

> The end-tag parsing thing should be moved to the parsing section.

The `datagrid` must then populate itself using the data provided by the data provider assigned to the `data` DOM attribute. After the view is populated (using the methods described below), the `datagrid` must invoke the `initialize()` method on the data provider specified by the `data` attribute, passing itself (the `HTMLDataGridElement` object) as the only argument.

When the `data` attribute is null, the `datagrid` must use the default data provider described in the previous section.

To obtain data from the data provider, the element must invoke methods on the data provider object in the following ways:

To determine the total number of columns

> Invoke the `getColumnCount()` method with no arguments. The return value is the number of columns. If the return value is zero or negative, not an integer, or simply not a numeric type, or if the method is not defined, then 1 must be used instead.

To get the captions to use for the columns

> Invoke the `getCaptionText()` method with the index of the column in question. The index *i*

must be in the range 0 ≤ 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 ≤ 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 indentity transform; the data is rendered in its natural order).

To establish what classes apply to a row

Invoke the `getRowClasses()` method with a `RowSpecification` object representing the row in question, 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 is entirely up to the UA.

To obtain a URI to an image representing a row

Invoke the `getRowImage()` method with a `RowSpecification` object representing the row in question. The return value is a string representing a URI (or IRI) to an image. Relative URIs must be interpreted relative to the `datagrid`'s base URI. If the method returns the empty string, null, or if the method is not defined, then the row has no associated image.

To obtain a context menu appropriate for a particular row

Invoke the `getRowMenu()` method with a `RowSpecification` object representing the row in question. The return value is a reference to an object implementing the `HTMLMenuElement` interface, i.e. a `menu` element DOM node. (This element must then be interpreted as described in the section on context menus to obtain the actual context menu to use.) If the method returns something that is not an `HTMLMenuElement`, or if the method is not defined, then the row has no associated context menu. User agents may provide their own default context menu, and may add items to the author-provided context menu. For example, such a menu could allow the user to change the presentation of the `datagrid` element.

To establish the value of a particular cell

Invoke the `getCellData()` method with the first argument being a `RowSpecification` object representing the row of the cell in question and the second argument being the index of the cell's column. The second argument must be a non-negative integer less than the total number of columns. The return value is the value of the cell. If the return value is null or the empty string, or if the method is not defined, then the cell has no data. (For progress bar cells, the cell's value must be further interpreted, as described below.)

To establish what classes apply to a cell

Invoke the `getCellClasses()` method with the first argument being a `RowSpecification` object representing the row of the cell in question, the second argument being the index of the cell's column, and the third being an object implementing the `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 appplies to the cell and the cell's checkbox is in an indeterminate state. When the `indeterminate` class appplies to the cell, the checkbox is a tristate checkbox, and the user can set it to the indeterminate state. Otherwise, only the `checked` and/or `unchecked` classes apply to the cell, and the cell can only be toggled betwen those two states.

If the data provider ever raises an exception while the `datagrid` is invoking one of its methods, the `datagrid` must act, for the purposes of that particular method call, as if the relevant method had not been defined.

A `RowSpecification` object p with n path components passed to a method of the data provider must fulfill the constraint $0 \leq p_i < m$-1 for all integer values of i in the range $0 \leq i < n$-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 honour the most recently returned values.

User agents may cache returned values so that the data provider is never asked for data that could contradict earlier data. User agents must not cache the return value of the `getRowMenu` method.

The exact algorithm used to populate the data grid is not defined here, since it will differ based on the presentation used. However, the behaviour of user agents must be consistent with the descriptions above. For example, it would be non-conformant for a user agent to make cells have

both a checkbox and be editable, as the descriptions above state that cells that have a checkbox cannot be edited.

3.18.2.6. Updating the `datagrid`

Whenever the `data` attribute is set to a new value, the `datagrid` must clear the current selection, remove all the displayed rows, and plan to repopulate itself using the information from the new data provider at the earliest opportunity.

There are a number of update methods that can be invoked on the `datagrid` element to cause it to refresh itself in slightly less drastic ways:

When the **`updateEverything()`** method is called, the user agent must repopulate the entire `datagrid`. If the number of rows decreased, the selection must be updated appropriately. If the number of rows increased, the new rows should be left unselected.

When the **`updateRowsChanged(`***`row, count`***`)`** method is called, the user agent must refresh the rendering of the rows starting from the row specified by *row*, and including the *count* next siblings of the row (or as many next siblings as it has, if that is less than *count*), including all descendant rows.

When the **`updateRowsInserted(`***`row, count`***`)`** method is called, the user agent must assume that *count* new rows have been inserted, such that the first new row is indentified by *row*. The user agent must update its rendering and the selection accordingly. The new rows should not be selected.

When the **`updateRowsRemoved(`***`row, count`***`)`** method is called, the user agent must assume that *count* rows have been removed starting from the row that used to be identifier by *row*. The user agent must update its rendering and the selection accordingly.

The **`updateRowChanged(`***`row`***`)`** method must be exactly equivalent to calling `updateRowsChanged(`*`row`*`, 1)`.

When the **`updateColumnChanged(`***`column`***`)`** method is called, the user agent must refresh the rendering of the specified column *column*, for all rows.

When the **`updateCellChanged(`***`row, column`***`)`** method is called, the user agent must refresh the rendering of the cell on row *row*, in column *column*.

Any effects the update methods have on the `datagrid`'s selection is not considered a change to the selection, and must therefore not fire the `select` event.

These update methods should only be called by the data provider, or code acting on behalf of the data provider. In particular, calling the `updateRowsInserted()` and `updateRowsRemoved()` methods without actually inserting or removing rows from the data provider is likely to result in inconsistent renderings, and the user agent is likely to disable the data grid.

3.18.2.7. Requirements for interactive user agents

This section only applies to interactive user agents.

If the `datagrid` element has a **`disabled`** attribute, then the user agent must disable the `datagrid`, preventing the user from interacting with it. The `datagrid` element should still continue to update itself when the data provider signals changes to the data, though. Obviously, conformance requirements stating that `datagrid` elements must react to users in particular ways do not apply when one is disabled.

If a row is openable, then the user should be able to toggle its open/closed state. When a row's open/closed state changes, the user agent must update the rendering to match the new state.

If a cell is a cell whose value can be cycled between multiple states, then the user must be able to activate the cell to cycle its value. When the user activates this "cycling" behaviour of a cell, then the datagrid must invoke the data provider's cycleCell() method, with a RowSpecification object representing the cell's row as the first argument and the cell's column index as the second. The datagrid must act as if the datagrid's updateCellChanged() method had been invoked with those same arguments immediately before the provider's method was invoked.

When a cell has a checkbox, the user must be able to set the checkbox's state. When the user changes the state of a checkbox in such a cell, the datagrid must invoke the data provider's setCellCheckedState() method, with a RowSpecification object representing the cell's row as the first argument, the cell's column index as the second, and the checkbox's new state as the third. The state should be represented by the number 1 if the new state is checked, 0 if the new state is unchecked, and -1 if the new state is indeterminate (which must only be possible if the cell has the indeterminate class set). The datagrid must act as if the datagrid's updateCellChanged() method had been invoked, specifying the same cell, immediately before the provider's method was invoked.

If a cell is editable, the user must be able to edit the data for that cell, and doing so must cause the user agent to invoke the editCell() method of the data provider with three arguments: a RowSpecification object representing the cell's row, the cell's column's index, and the new text entered by the user. The user agent must act as if the updateCellChanged() method had been invoked, with the same row and column specified, immediately before the provider's method was invoked.

3.18.2.8. The selection

This section only applies to interactive user agents. For other user agents, the selection attribute must return null.

```
interface DataGridSelection {
  readonly attribute unsigned long length;
  RowSpecification item(in unsigned long index);
  boolean isSelected(in RowSpecification row);
  void setSelected(in RowSpecification row, in boolean selected);

  void selectAll();
  void invert();
  void clear();
};
```

Each datagrid element must keep track of which rows are currently selected. Initially no rows are selected, but this can be changed via the methods described in this section.

The selection of a datagrid is represented by its **selection** DOM attribute, which must be a DataGridSelection object.

DataGridSelection objects represent the rows in the selection. In the selection the rows must be ordered in the natural order of the data provider (and not, e.g., the rendered order). Rows that are not rendered because one of their ancestors is closed must share the same selection state as their nearest rendered ancestor. Such rows are not considered part of the selection for the purposes of

iterating over the selection.

> **Note: This selection API doesn't allow for hidden rows to be selected because it is trivial to create a data provider that has infinite depth, which would then require the selection to be infinite if every row, including every hidden row, was selected.**

The **length** attribute must return the number of rows currently present in the selection. The **item(_index_)** method must return the _index_th row in the selection. If the argument is out of range (less than zero or greater than the number of selected rows minus one), then it must raise an INDEX_SIZE_ERR exception. [DOM3CORE]

The **isSelected()** method must return the selected state of the row specified by its argument. If the specified row exists and is selected, it must return true, otherwise it must return false.

The **setSelected()** method takes two arguments, _row_ and _selected_. When invoked, it must set the selection state of row _row_ to selected if _selected_ is true, and unselected if it is false. If _row_ is not a row in the data grid, the method must raise an INDEX_SIZE_ERR exception. If the specified row is not rendered because one of its ancestors is closed, the method must do nothing.

The **selectAll()** method must mark all the rows in the data grid as selected. After a call to selectAll(), the length attribute will return the number of rows in the data grid, not counting children of closed rows.

The **invert()** method must cause all the rows in the selection that were marked as selected to now be marked as not selected, and vice versa.

The **clear()** method must mark all the rows in the data grid to be marked as not selected. After a call to clear(), the length attribute will return zero.

If the datagrid element has a **multiple** attribute, then the user must be able to select any number of rows (zero or more). If the attribute is not present, then the user must only be able to select a single row at a time, and selecting another one must unselect all the other rows.

> **Note: This only applies to the user. Scripts can select multiple rows even when the _multiple_ attribute is absent.**

Whenever the selection of a datagrid changes, whether due to the user interacting with the element, or as a result of calls to methods of the selection object, a **select** event that bubbles but is not cancelable must be fired on the datagrid element. If changes are made to the selection via calls to the object's methods during the execution of a script, then the select events must be coalesced into one, which must then be fired when the script execution has completed.

> **Note: The _DataGridSelection_ interface has no relation to the _Selection_ interface.**

3.18.2.9. Columns and captions

This section only applies to interactive user agents.

Each datagrid element must keep track of which columns are currently being rendered. User agents should initially show all the columns except those with the initially-hidden class, but may allow users to hide or show columns. User agents should initially display the columns in the order given by the data provider, but may allow this order to be changed by the user.

If columns are not being used, as might be the case if the data grid is being presented in an icon view, or if an overview of data is being read in an aural context, then the text of the first column of each row should be used to represent the row.

If none of the columns have any captions (i.e. if the data provider does not provide a `getCaptionText()` method), then user agents may avoid showing the column headers at all. This may prevent the user from performing actions on the columns (such as reordering them, changing the sort column, and so on).

> *Note: Whatever the order used for rendering, and irrespective of what columns are being shown or hidden, the "first column" as referred to in this specification is always the column with index zero, and the "last column" is always the column with the index one less than the value returned by the `getColumnCount()` method of the data provider.*

If a column is sortable, then the user must be able to invoke it to sort the data. When the user does so, then the `datagrid` must invoke the data provider's `toggleColumnSortState()` method, with the column's index as the only argument. The `datagrid` must *then* act as if the `datagrid`'s `updateEverything()` method had been invoked.

3.18.3. The `command` element

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 URI (or IRI).

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](#) their respective namesake content attributes.

The **click()** method's behaviour depends on the value of the <u>type</u> attribute of the element, as follows:

> ↳ **If the <u>type</u> attribute has the value `checkbox`**
>
>> If the element has a <u>checked</u> attribute, the UA must remove that attribute. Otherwise, the UA must add a <u>checked</u> attribute, with the literal value `checked`. The UA must then [fire a `click` event](#) at the element.
>
> ↳ **If the <u>type</u> 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 <u>command</u> element, if that element has a <u>radiogroup</u> attribute whose value exactly matches the current element's (treating missing <u>radiogroup</u> attributes as if they were the empty string), and has a <u>checked</u> attribute, must remove that attribute and [fire a `click` event](#) at the element.
>>
>> Then, the element's <u>checked</u> 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.*

<div style="border:2px solid red; color:red; padding:8px;">
should change all the above so it actually is just trigged by a click event, then we could remove the shadowing click() method and rely on actual events.
</div>

<div style="border:2px solid red; color:red; padding:8px;">
Need to define the command="" attribute
</div>

> *Note: `command` elements are not rendered unless they [form part of a menu](#).*

3.18.4. The `menu` element

Categories
> [Prose content](#).
> If there is a <u>menu</u> element ancestor: [phrasing content](#).

Contexts in which this element may be used:
> Where [prose content](#) is expected.
> If there is a <u>menu</u> element ancestor: where [phrasing content](#) is expected.

Content model:
> Either: Zero or more <u>li</u> elements.
> Or: [Phrasing content](#).

Element-specific attributes:
> <u>type</u>
> <u>label</u>

```
        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 declaraing 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, prose 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.

3.18.4.1. Introduction

This section is non-normative.

...

3.18.4.2. 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 behaviour depends on what the node is, as follows:

↳ **An element that defines a command**

> Append the command to the menu. If the element is a command element with a default attribute, mark the command as being a default command.

↳ **An hr element**

↳ **An option element that has a value attribute set to the empty string, and has a disabled attribute, and whose textContent consists of a string of one or more hyphens (U+002D HYPHEN-MINUS)**

> Append a separator to the menu.

↳ **An li element**

> Iterate over the children of the li element.

↳ **A menu element with no label attribute**

↳ **A select element**

> Append a separator to the menu, then iterate over the children of the menu or select element, then append another separator.

↳ **A menu element with a label attribute**

↳ **An optgroup element**

> Append a submenu to the menu, using the value of the element's label attribute as the label of the menu. The submenu must be constructed by taking the element and creating a new menu for it using the complete process described in this section.

↳ **Any other node**

> Ignore the node.

Once all the nodes have been processed as described above, the user agent must the post-process the menu as follows:

1. Any menu item with no label, or whose label is the empty string, must be removed.

2. Any sequence of two or more separators in a row must be collapsed to a single separator.

3. Any separator at the start or end of the menu must be removed.

3.18.4.3. Context menus

The **contextmenu** attribute gives the element's context menu. The value must be the ID of a menu element in the DOM. If the node that would be obtained by the invoking the getElementById() method using the attribute's value as the only argument is null or not a menu element, then the element has no assigned context menu. Otherwise, the element's assigned context menu is the element so identified.

When an element's context menu is requested (e.g. by the user right-clicking the element, or

pressing a context menu key), the UA must fire a `contextmenu` event on the element for which the menu was requested.

> *Note: Typically, therefore, the firing of the `contextmenu` event will be the default action of a `mouseup` or `keyup` event. The exact sequence of events is UA-dependent, as it will vary based on platform conventions.*

The default action of the `contextmenu` event depends on whether the element has a context menu assigned (using the `contextmenu` attribute) or not. If it does not, the default action must be for the user agent to show its default context menu, if it has one.

If the element *does* have a context menu assigned, then the user agent must fire a `show` event on the relevant `menu` element.

The default action of *this* event is that the user agent must show a context menu built from the `menu` element.

The user agent may also provide access to its default context menu, if any, with the context menu shown. For example, it could merge the menu items from the two menus together, or provide the page's context menu as a submenu of the default menu.

If the user dismisses the menu without making a selection, nothing in particular happens.

If the user selects a menu item that represents a command, then the UA must invoke that command's Action.

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.

3.18.4.4. 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.

3.18.5. Commands

A **command** is the abstraction behind menu items, buttons, and links. Once a command is defined, other parts of the interface can refer to the same command, allowing many access points to a single feature to share aspects such as the disabled state.

Commands are defined to have the following *facets*:

Type

The kind of command: "command", meaning it is a normal command; "radio", meaning that triggering the command will, amongst other things, set the Checked State to true (and probably uncheck some other commands); or "checkbox", meaning that triggering the command will, amongst other things, toggle the value of the Checked State.

ID

The name of the command, for referring to the command from the markup or from script. If a command has no ID, it is an **anonymous command**.

Label

The name of the command as seen by the user.

Hint

A helpful or descriptive string that can be shown to the user.

Icon

A graphical image that represents the action.

Hidden State

Whether the command is hidden or not (basically, whether it should be shown in menus).

Disabled State

Whether the command can be triggered or not. If the Hidden State is true (hidden) then the Disabled State will be true (disabled) regardless.

Checked State

Whether the command is checked or not.

Action

The actual effect that triggering the command will have. This could be a scripted event handler, a URI to which to navigate, or a form submission.

Triggers

The list of elements that can trigger the command. The element defining a command is always in the list of elements that can trigger the command. For anonymous commands, only the element defining the command is on the list, since other elements have no way to refer to it.

Commands are represented by elements in the DOM. Any element that can define a command also implements the Command interface:

```
interface Command {
  readonly attribute DOMString commandType;
  readonly attribute DOMString id;
  readonly attribute DOMString label;
  readonly attribute DOMString title;
  readonly attribute DOMString icon;
  readonly attribute boolean hidden;
  readonly attribute boolean disabled;
  readonly attribute boolean checked;
  void click();
  readonly attribute HTMLCollection triggers;
  readonly attribute Command command;
};
```

The `Command` interface is implemented by any element capable of defining a command. (If an element can define a command, its definition will list this interface explicitly.) All the attributes of the `Command` interface are read-only. Elements implementing this interface may implement other interfaces that have attributes with identical names but that are mutable; in bindings that simply flatten all supported interfaces on the object, the mutable attributes must shadow the readonly attributes defined in the `Command` interface.

The **commandType** attribute must return a string whose value is either "`command`", "`radio`", or "`checked`", depending on whether the Type of the command defined by the element is "command", "radio", or "checked" respectively. If the element does not define a command, it must return null.

The **id** attribute must return the command's ID, or null if the element does not define a command or defines an anonymous command. This attribute will be shadowed by the `id` DOM attribute on the `HTMLElement` interface.

The **label** attribute must return the command's Label, or null if the element does not define a command or does not specify a Label. This attribute will be shadowed by the `label` DOM attribute on `option` and `command` elements.

The **title** attribute must return the command's Hint, or null if the element does not define a command or does not specify a Hint. This attribute will be shadowed by the `title` DOM attribute on the `HTMLElement` interface.

The **icon** attribute must return an absolute URI to the command's Icon. If the element does not specify an icon, or if the element does not define a command, then the attribute must return null. This attribute will be shadowed by the `icon` DOM attribute on `command` elements.

The **hidden** attribute must return true if the command's Hidden State is that the command is hidden, and false if it is that the command is not hidden. If the element does not define a command, the attribute must return false. This attribute will be shadowed by the `hidden` DOM attribute on `command` elements.

The **disabled** attribute must return true if the command's Disabled State is that the command is disabled, and false if the command is not disabled. This attribute is not affected by the command's Hidden State. If the element does not define a command, the attribute must return false. This attribute will be shadowed by the `disabled` attribute on `button`, `input`, `option`, and `command` elements.

The **checked** attribute must return true if the command's Checked State is that the command is checked, and false if it is that the command is not checked. If the element does not define a command, the attribute must return false. This attribute will be shadowed by the `checked` attribute on `input` and `command` elements.

The **click()** method must trigger the Action for the command. If the element does not define a command, this method must do nothing. This method will be shadowed by the `click()` method on HTML elements, and is included only for completeness.

The **triggers** attribute must return a list containing the elements that can trigger the command (the command's Triggers). The list must be live. While the element does not define a command, the list must be empty.

The **commands** attribute of the document's `HTMLDocument` interface must return an `HTMLCollection` rooted at the `Document` node, whose filter matches only elements that define commands and have IDs.

The following elements can define commands: `a`, `button`, `input`, `option`, `command`.

3.18.5.1. Using the `a` element to define a command

An `a` element with an `href` attribute defines a command.

The Type of the command is "command".

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the string given by the element's `textContent` DOM attribute.

The Hint of the command is the value of the `title` attribute of the `a` element. If the attribute is not present, the Hint is the empty string.

The Icon of the command is the absolute URI of the first image in the element. Specifically, in a depth-first search of the children of the element, the first element that is `img` element with a `src` attribute is the one that is used as the image. The URI must be taken from the element's `src` attribute. Relative URIs must be resolved relative to the base URI of the image element. If no image is found, then the Icon facet is left blank.

The Hidden State and Disabled State facets of the command are always false. (The command is always enabled.)

The Checked State of the command is always false. (The command is never checked.)

The Action of the command is to fire a `click` event at the element.

3.18.5.2. 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).

3.18.5.3. 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.

3.18.5.4. 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.

3.18.5.5. Using the `command` element to define a command

A `command` element defines a command.

The Type of the command is "radio" if the `command`'s `type` attribute is "radio", "checkbox" if the attribute's value is "checkbox", and "command" otherwise.

The ID of the command is the value of the `id` attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command.

The Label of the command is the value of the element's `label` attribute, if there is one, or the empty string if it doesn't.

The Hint of the command is the string given by the element's `title` attribute, if any, and the empty string if the attribute is absent.

The Icon for the command is the absolute URI resulting from resolving the value of the element's `icon` attribute as a URI relative to the element's base URI. If the element has no `icon` attribute then the command has no Icon.

The Hidden State of the command is true (hidden) if the element has a `hidden` attribute, and false otherwise.

The Disabled State of the command is true (disabled) if the element has either a `disabled` attribute or a `hidden` attribute (or both), and false otherwise.

The Checked State of the command is true (checked) if the element has a `checked` attribute, and false otherwise.

The Action of the command is to invoke the behaviour described in the definition of the `click()` method of the `HTMLCommandElement` interface.

3.19. Data Templates

3.19.1. Introduction

...examples...

3.19.2. The `datatemplate` element

Categories
> Metadata content.
> Prose content.

Contexts in which this element may be used:
> As the root element of an XML document.
> Where metadata content is expected.
> Where prose content is expected.

Content model:
> Zero or more `rule` elements.

Element-specific attributes:
> None.

DOM interface:

> No difference from `HTMLElement`.

The `datatemplate` element brings together the various rules that form a data template. The element doesn't itself do anything exciting.

3.19.3. 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, CDATA 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.

3.19.4. The `nest` element

Categories
> None.

Contexts in which this element may be used:

> As a child of a descendant of a <u>rule</u> element, regardless of the element's content model.

Content model:

> Empty.

Element-specific attributes:

> <u>filter</u>
> <u>mode</u>

DOM interface:

```
interface HTMLNestElement : HTMLElement {
          attribute DOMString filter;
          attribute DOMString mode;
};
```

The <u>nest</u> 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 <u>rule</u> in which the <u>nest</u> 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 <u>filter</u> attribute is not specified, then all elements, text nodes, CDATA 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 <u>filter</u> content attribute.

The **mode** DOM attribute must reflect the <u>mode</u> content attribute.

3.19.5. 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 <u>template</u> attribute, when specified, must be a URI 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 <u>datatemplate</u> element, otherwise, the root element must be a <u>datatemplate</u> element.

The **template** DOM attribute must reflect the <u>template</u> content attribute.

The **ref** attribute may be specified on any element on which the <u>template</u> attribute is specified. If it is specified, it must be a URI to an XML or HTML document, or a fragment identifier pointing at another part of the document.

When an element has a <u>template</u> attribute but no <u>ref</u> 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 <u>ref</u> content attribute.

The **registrationmark** attribute may be specified on any element that is a descendant of a <u>rule</u> element, except <u>nest</u> 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 <u>registrationmark</u> 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 <u>registrationmark</u> content attribute.

3.19.6. Processing model

3.19.6.1. The <u>originalContent</u> 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 <u>template</u> attribute is set to a usable value, and is unset when the attribute is removed.

> *Note: The <u>originalContent</u> DOM attribute can thus be used as an indicator of whether a template is currently being applied, just as the <u>templateElement</u> DOM attribute can.*

3.19.6.2. The <u>template</u> attribute

Setting: When an [HTML element](#) without a <u>template</u> attribute has its <u>template</u> attribute set, the user agent must fetch the specified file and parse it (without a [browsing context](#), and with scripting disabled) to obtain a DOM. If the URI is the same as the URI of the current document, 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 <u>template</u> 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 <u>template</u> attribute's value has a fragment identifier, and, in the DOM in question, it identifies a <u>datatemplate</u> element, then set the <u>templateElement</u> DOM attribute to that element.

   Otherwise, if the <u>template</u> attribute value does not have a fragment identifier, and the root element of the DOM in question is a <u>datatemplate</u> element, then set the <u>templateElement</u> 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 <u>originalContent</u> DOM attribute on the element to this new DocumentFragment object.

3. Jump to the steps below for <u>updating the generated content</u>.

If the resource has **failed to parse**, the user agent must <u>fire a simple event</u> with the name `error` at the element on which the `template` attribute was found.

Unsetting: When an <u>HTML element</u> 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 <u>act as if it was now set to its new value</u> (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.

3.19.6.3. The `ref` attribute

Setting: When an <u>HTML element</u> without a `ref` attribute has its `ref` attribute set, the user agent must fetch the specified file and parse it (without a <u>browsing context</u>, and with scripting disabled) to obtain a DOM. If the URI is the same as the URI of the current document, then the current document's DOM is 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 <u>failed to parse</u> 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 <u>failed to parse</u> steps below.

2. Jump to the steps below for <u>updating the generated content</u>.

If the resource has **failed to parse**, the user agent must <u>fire a simple event</u> 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.

3.19.6.4. 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.

3.19.6.5. 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.

3.19.6.6. 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 initialised 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 <u>mode</u> 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 <u>condition</u> attribute, and the attribute's value, when <u>evaluated as a selector</u>, 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 <u>Levenberg template node algorithm</u> 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 <u>Levenberg data node algorithm</u>, returning to whatever algorithm invoked it.

7. Let *condition* be the next <u>rule</u> element that is a child of the *rule container* element, after the *condition* element itself, or null if there are no more <u>rule</u> 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 <u>Levenberg template node algorithm</u>, returning to whatever algorithm invoked it.

2. If *template node* is a <u>nest</u> element, then run these substeps:

   1. If *source node* is not an element, then abort the current instance of the <u>Levenberg template node algorithm</u>, returning to whatever algorithm invoked it.

   2. If the *template node* has a <u>mode</u> 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 <u>Levenberg template node algorithm</u>, returning to whatever algorithm invoked it.

   5. If the *template node* element has a <u>filter</u> attribute, and the attribute's value, when <u>evaluated as a selector</u>, matches *child node*, then invoke the <u>Levenberg data node algorithm</u> 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 <u>registrationmark</u> 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 <u>dataNode</u> 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 <u>text expansion algorithm for templates</u> 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, CDATA block, or PI. Run these substeps instead:

1. Let *expanded* be the result of passing the node value of *template node* (the content of the text node, CDATA block, or PI) to the <u>text expansion algorithm for templates</u> 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 <u>Levenberg template node algorithm</u> 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**

3.20. Miscellaneous elements

3.20.1. 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:

> No difference from `HTMLElement`.

The `legend` element represents a title or explanatory caption for the rest of the contents of the `legend` element's parent element.

3.20.2. The `div` element

Categories

> None.

Contexts in which this element may be used:

> Where prose content is expected.

Content model:

> Prose content.

Element-specific attributes:

> None.

DOM interface:

> No difference from `HTMLElement`.

The `div` element represents nothing at all. It can be used with the `class`, `lang`/`xml:lang`, and `title` attributes to mark up semantics common to a group of consecutive elements.

> Allowing `div` elements to contain phrasing content makes it easy for authors to abuse `div`, using it with the `class=""` attribute to the point of not having any other elements in the markup. This is a disaster from an accessibility point of view, and it would be nice if we could somehow make such pages non-compliant without preventing people from using `div`s as the extension mechanism that they are, to handle things the spec can't otherwise do (like making new widgets).

4. 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.

4.1. Browsing contexts

A **browsing context** is a collection of one or more `Document` objects, and one or more <u>views</u>.

At any one time, one of the `Document`s in a <u>browsing context</u> is the **active document**. The collection of `Document`s is the <u>browsing context</u>'s <u>session history</u>.

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 <u>browsing context</u>. [DOM2VIEWS]

> *Note: The `document` attribute of an `AbstractView` object representing a <u>view</u> gives the `Document` object of the view's <u>browsing context</u>'s <u>active document</u>. <u>[DOM2VIEWS]</u>*

> *Note: Events that use the `UIEvent` interface are related to a specific <u>view</u> (the view in which the event happened); the `AbstractView` of that view is given in the event object's `view` attribute. <u>[DOM3EVENTS]</u>*

> *Note: A typical Web browser has one obvious <u>view</u> per <u>browsing context</u>: 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 <u>browsing context</u> associated with it. In particular, data mining tools are likely to never instantiate browsing contexts.*

The main <u>view</u> through which a user primarily interacts with a user agent is the **default view**.

> *Note: The <u>default view</u> of a `Document` is given by the `defaultView` attribute on the `Document` object's `DocumentView` interface. <u>[DOM3VIEWS]</u>*

When a <u>browsing context</u> 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 <u>HTML documents</u>. The `Document` must have a single child <u>html</u> node, which itself has a single child <u>body</u> node. If the <u>browsing context</u> is created specifically to be immediately navigated, then that initial navigation will have <u>replacement enabled</u>.

4.1.1. Nested browsing contexts

Certain elements (for example, <u>iframe</u> elements) can instantiate further <u>browsing contexts</u>. 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*.

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).

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.

4.1.2. 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.

4.1.3. 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.

4.1.4. 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 consists of 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 serialised so that for each unit of related browsing contexts there is only one script being executed at a time.

4.1.5. 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 that does not start with a U+005F LOW LINE character, or, a string that case-insensitively matches one of: `_self`, `_parent`, or `_top`. (Names starting with an underscore are reserved for special keywords.)

The rules for chosing a browsing context given a browsing context name are as follows. The rules assume that they are being applied in the context of a <u>browsing context</u>.

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 <u>*parent* browsing context</u> 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 <u>top-level browsing context</u> of the current one.

4. If the given browsing context name is not `_blank` and there exists a browsing context whose <u>name</u> is the same as the given browsing context name, and one of the following is true:

   - Either the <u>origin</u> of that browsing context's <u>active document</u> is the same as the <u>origin</u> of the current browsing context's <u>active document</u>,

   - Or that browsing context is an <u>auxiliary browsing context</u> and its <u>opener browsing context</u> is either the current browsing context or a browsing context that the user agent considers is closely enough related to the current browsing context,

   - Or that browsing context is not a <u>top-level browsing context</u>, and the <u>origin</u> of the <u>active document</u> of the <u>parent browsing context</u> of that browsing context is the same as the <u>origin</u> of the current browsing context's <u>active document</u>,

   ...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 user agent has been configured such that in this instance it will create a new browsing context

   > A new <u>auxiliary browsing context</u> must be created, with the <u>opener browsing context</u> 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 <u>navigated</u>, then the navigation will be done with <u>replacement enabled</u>.

   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.

4.2. The default view

The `AbstractView` object of default views must also implement the `Window` object.

```
interface Window {
  // the current browsing context
  readonly attribute Window window;
  readonly attribute Window self;
           attribute DOMString name;
  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;
  readonly attribute Storage globalStorage;
  Database openDatabase(in DOMString name, in DOMString version, in
DOMString displayName, in unsigned long estimatedSize);

  // modal 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();

  // other browsing contexts
  readonly attribute Window frames;
  readonly attribute unsigned long length;
  readonly attribute Window opener;
  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);

  // 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 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** which, when a document is active, are available on the Document's default view Window object. A Document object's list of added properties must be empty when the Document object is created.

Objects implementing the Window interface must also implement the EventTarget interface.

Note: Window objects also have an implicit [[Get]] method which returns nested browsing contexts.

4.2.1. Security

User agents must raise a security exception whenever any of the members of a Window object are accessed by scripts whose origin is not the same as the Window object's browsing context's active document's origin, with the following exceptions:

- The location object

- The postMessage() method

User agents must not allow scripts to override the <u>location</u> object's setter.

4.2.2. Constructors

All <u>Window</u> objects must provide the following constructors:

`Audio()`

`Audio(src)`

> When invoked as constructors, these must return a new <u>HTMLAudioElement</u> object (a new <u>audio</u> element). If the *src* argument is present, the object created must have its <u>src</u> content attribute set to the provided value, and the user agent must invoke the <u>load()</u> method on the object before returning.

`Image()`

`Image(in unsigned long w)`

`Image(in unsigned long w, in unsigned long h)`

> When invoked as corstructors, these must return a new <u>HTMLImageElement</u> object (a new <u>img</u> element). If the *h* argument is present, the new object's `height` content attribute must be set to *h*. If the *w* argument is present, the new object's `width` content attribute must be set to *w*.

`Option()`

`Option(in DOMString name)`

`Option(in DOMString name, in DOMString value)`

> When invoked as constructors, these must return a new `HTMLOptionElement` object (a new `option` element). need to define argument processing

And when constructors are invoked but without using the constructor syntax...?

4.2.3. APIs for creating and navigating browsing contexts by name

The **`open()`** method on <u>Window</u> objects provides a mechanism for <u>navigating</u> an existing <u>browsing context</u> or opening and navigating an <u>auxiliary browsing context</u>.

The method has four arguments, though they are all optional.

The first argument, *url*, gives a URI (or IRI) for a page to load in the browsing context. If no arguments are provided, then the *url* argument defaults to "`about:blank`". The argument must be resolved to an absolute URI by ...

The second argument, *target*, specifies the <u>name</u> of the browsing context that is to be navigated. It must be a <u>valid browsing context name</u>. 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 <u>replace</u> 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 <u>session history</u>). 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 chosing 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 URI given in *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 chosing a browsing context given a browsing context name.

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.*

4.2.4. Accessing other browsing contexts

In ECMAScript implementations, objects that implement the `Window` interface must have a **[[Get]]** method that, when invoked with a property name that is a number *i*, returns the *i*th child browsing context of the active `Document`, sorted in document order of the elements nesting those browsing contexts.

The **length** DOM attribute on the `Window` interface must return the number of child browsing contexts of the active `Document`.

4.3. Scripting

4.3.1. Running executable code

Various mechanisms can cause author-provided executable code to run in the context of a document. These mechanisms include, but are probably not limited to:

- Processing of `script` elements.

- Processing of inline `javascript:` URIs (e.g. the `src` attribute of `img` elements, or an `@import` rule in a CSS `style` element block).

- Event handlers, whether registered through the DOM using `addEventListener()`, by explicit event handler content attributes, by event handler DOM attributes, or otherwise.

- Processing of technologies like XBL or SVG that have their own scripting features.

User agents may provide a mechanism to enable or disable the execution of author-provided code.

When the user agent is configured such that author-provided code does not execute, or if the user agent is implemented so as to never execute author-provided code, it is said that **scripting is disabled**. When author-provided code *does* execute, **scripting is enabled**. A user agent with scripting disabled is a <u>user agent with no scripting support</u> for the purposes of conformance.

4.3.2. Origin

Access to certain APIs is granted or denied to scripts based on the **origin** of the script and the API being accessed.

The origin of a script depends on the context of that script:

If a script is in a <u>script</u> element

> The origin of the script is the origin of the Document to which the <u>script</u> element belongs.

If a script is a function or other code reference created by another script

> The origin of the script is the origin of the script that created it.

If a script is a <u>javascript: URI</u> in an attribute

> The origin is the origin of the Document of the element on which the attribute is found.

If a script is a <u>javascript: URI</u> in a style sheet

> The origin is the origin of the Document to which the style sheet applies.

If a script is a <u>javascript: URI</u> to which a <u>browsing context</u> is being <u>navigated</u>, the URI having been provided by the user (e.g. by using a *bookmarklet*)

> The origin is the origin of the Document of the <u>browsing context</u>'s <u>active document</u>.

If a script is a <u>javascript: URI</u> to which a <u>browsing context</u> is being <u>navigated</u>, the URI having been declared in markup

> The origin is the origin of the Document of the element (e.g. an <u>a</u> or <u>area</u> element) that declared the URI.

If a script is a <u>javascript: URI</u> to which a <u>browsing context</u> is being <u>navigated</u>, the URI having been provided by script

> The origin is the origin of the script that provided the URI.

The origin of scripts thus comes down to finding the origin of Document objects.

The origin of a Document or image that was served over the network and whose address uses a URI scheme with a server-based naming authority is the tuple consisting of the <scheme>, <host>/<ihost>, and <port> parts of the Document's full URI. [RFC3986] [RFC3987]

The origin of a Document or image that was generated from a data: URI found in another Document or in a script is the origin of the Document or script.

The origin of a Document or image that was generated from a data: URI from another source is a globally unique identifier assigned when the document is created.

The origin of a Document or image that was generated from a <u>javascript: URI</u> is the same as the origin of that javascript: URI.

The string representing the script's domain in IDNA format is obtained as follows: take the domain part of the script's <u>origin</u> tuple and apply the IDNA ToASCII algorithm and then the IDNA ToUnicode algorithm to each component of the domain name (with both the AllowUnassigned and UseSTD3ASCIIRules flags set both times). [RFC3490]

If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, or if the origin of the script has no domain part, then the string representing the script's domain in IDNA format cannot be obtained. (ToUnicode is defined to never fail.)

> It's been suggested that we should put IP addresses into the origin tuple, to mitigate DNS rebinding attacks. However that would kill multi-homed systems like GMail. Should we do something like have a DNS record say whether or not to include the IP in the origin for a host?

4.3.3. Unscripted same-origin checks

When two URIs are to be compared to determine if they have the **same scheme/host/port**, it means that the following algorithm must be invoked, where uri_1 and uri_2 are the two URIs.

1. First, both uri_1 and uri_2 must be normalized to obtain the two tuples ($scheme_1$, $host_1$, $port_1$) and ($scheme_2$, $host_2$, $port_2$), by applying the following subalgorithm to each URI:

   1. Let uri be the URI being normalized.

   2. Parse uri according to the rules described in RFC 3986 and RFC 3987. [RFC3986] [RFC3987]

   3. If uri does not use a server-based naming authority, then fail the overall algorithm — the two URIs do not have the same scheme/host/port.

   4. Let $scheme$ be the <scheme> component of the URI. If the UA doesn't support the given protocol, then fail the overall algorithm — the two URIs do not have the same scheme/host/port.

   5. Let $host$ be the <host>/<ihost> component of the URI.

   6. 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 fail the overall algorithm — the two URIs do not have the same scheme/host/port. [RFC3490]

   7. If no port is explicitly listed, then let $port$ be the default port for the protocol given by $scheme$. Otherwise, let $port$ be the <port> component of the URI.

   8. Return the tuple ($scheme$, $host$, $port$).

2. If $scheme_1$ is not case-insensitively identical to $scheme_2$, or if $host_1$ is not case-insensitively identical to $host_2$, or if $port_1$ is not identical to $port_2$, then fail the overall algorithm — the two URIs do not have the same scheme/host/port.

3. Otherwise, the two URIs do have the same scheme/host/port.

4.3.4. Security exceptions

> Define **security exception**.

4.3.5. The `javascript:` protocol

A URI using the `javascript:` protocol must, if evaluated, be evaluated using the in-context evaluation operation defined for `javascript:` URIs. [JSURI]

When a browsing context is <u>navigated</u> to a `javascript:` URI, and the <u>active document</u> of that browsing context has the same <u>origin</u> as the URI, the dereference context must be the <u>browsing context</u> being navigated.

When a browsing context is <u>navigated</u> to a `javascript:` URI, and the <u>active document</u> of that browsing context has a *different* <u>origin</u> than the URI, the dereference context must be an empty object.

Otherwise, the dereference context must the <u>browsing context</u> of the `Document` to which belongs the element for which the URI is being dereferenced, or to which the style sheet for which the URI is being dereferenced applies, whichever is appropriate.

URIs using the `javascript:` protocol should be evaluated when the resource for that URI is needed, unless <u>scripting is disabled</u> or the `Document` corresponding to the dereference context (as defined above), if any, has `designMode` enabled.

If the dereference by-product is void (there is no return value), then the URI must be treated in a manner equivalent to an HTTP resource with an HTTP 204 No Content response.

Otherwise, the URI must be treated in a manner equivalent to an HTTP resource with a 200 OK response whose <u>Content-Type metadata</u> is `text/html` and whose response body is the dereference by-product, converted to a string value.

> ***Note: Certain contexts, in particular <u>`img`</u> elements, ignore the <u>Content-Type metadata</u>.***

> So for example a `javascript:` URI for a <u>src</u> attribute of an <u>img</u> element would be evaluated in the context of the page as soon as the attribute is set; it would then be sniffed to determine the image type and decoded as an image.
>
> A `javascript:` URI in an `href` attribute of an <u>a</u> element would only be evaluated when the link was <u>followed</u>.
>
> The <u>src</u> attribute of an <u>iframe</u> element would be evaluated in the context of the <u>iframe</u>'s own <u>browsing context</u>; once evaluated, its return value (if it was not void) would replace that <u>browsing context</u>'s document, thus changing the variables visible in that <u>browsing context</u>.

4.3.6. Events

<div style="border:2px solid red; color:red; padding:8px">
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 <u>script</u> element section, which says scripts don't run when the document isn't active.
</div>

4.3.6.1. Event handler attributes

<u>HTML elements</u> 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 <u>event handler content attribute</u> and an <u>event handler</u>

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 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]*

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 <u>drag</u> event is targeted at or bubbles through the element.

ondragend

> Must be invoked whenever a <u>dragend</u> event is targeted at or bubbles through the element.

ondragenter

> Must be invoked whenever a <u>dragenter</u> event is targeted at or bubbles through the element.

ondragleave

> Must be invoked whenever a <u>dragleave</u> event is targeted at or bubbles through the element.

ondragover

> Must be invoked whenever a <u>dragover</u> event is targeted at or bubbles through the element.

ondragstart

> Must be invoked whenever a <u>dragstart</u> event is targeted at or bubbles through the element.

ondrop

> Must be invoked whenever a <u>drop</u> event is targeted at or bubbles through the element.

onerror

> Must be invoked whenever an <u>error</u> event is targeted at or bubbles through the element.

> *Note: The <u>onerror</u> handler is also used for <u>reporting script errors</u>.*

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 <u>load</u> event is targeted at or bubbles through the element.

onmessage

> Must be invoked whenever a <u>message</u> 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.

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.

When scripting is disabled, event handler attributes must do nothing.

When scripting is enabled, 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.

4.3.6.2. Event firing

maybe this should be moved higher up (terminology? conformance? DOM?) Also, the whole

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, 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).

Firing a `load` event means firing a simple event called load. **Firing an `error` event** means firing a simple event called error.

The default action of these event is to do nothing unless otherwise stated.

4.3.6.3. 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.

4.3.6.4. 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 URI to the resource in which the error occured, and the third must give the line number in that resource on which the error occured.

> If the function returns false, then the error should not be reported to the user. Otherwise, if the function returns another value (or does not return at all), the error should be reported to the user.

> Any exceptions thrown or errors caused by this function must be reported to the user immediately after the error that the function was called for, without calling the function again.

↳ **If the value is `null`**

> The error should not reported to the user.

↳ **If the value is anything else**

> The error should be reported to the user.

The initial value of `onerror` must be `undefined`.

4.4. User prompts

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 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 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.

The **print()** method, when invoked, should offer the user the opportunity to <u>obtain a physical form</u> of the document. The user agent may make the method wait for the user to either accept or decline before returning; if so, the user agent must <u>pause</u> while the method is waiting. (This does not, of course, preclude the user agent from *always* offering the user with the opportunity to convert the document to whatever media the user might want.)

4.5. Browser state

The **navigator** attribute of the <u>Window</u> interface must return an instance of the <u>ClientInformation</u> interface, which represents the identity and state of the user agent (the client), and allows Web pages to register themselves as potential protocol and content handlers:

```
interface ClientInformation {
  readonly attribute boolean onLine;
  void registerProtocolHandler(in DOMString protocol, in DOMString
uri, in DOMString title);
  void registerContentHandler(in DOMString mimeType, in DOMString
uri, in DOMString title);
};
```

4.5.1. 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 <u>an example of how these methods could be presented to the user</u> below.

The arguments to the methods have the following meanings:

protocol (<u>registerProtocolHandler()</u> **only**)

> A scheme, such as `ftp` or `fax`. The scheme must be treated case-insensitively by user agents for the purposes of comparing with the scheme part of URIs that they consider against the list of registered handlers.

> The *protocol* value, if it contains a colon (as in "`ftp:`"), will never match anything, since schemes don't contain colons.

mimeType (<u>registerContentHandler()</u> **only**)

> A MIME type, such as `model/vrml` or `text/richtext`. The MIME type must be treated case-insensitively by user agents for the purposes of comparing with MIME types of

documents that they consider against the list of registered handlers.

User agents must compare the given values only to the MIME type/subtype parts of content types, not to the complete type including parameters. Thus, if *mimeType* values passed to this method include characters such as commas or whitespace, or include MIME parameters, then the handler being registered will never be used.

uri

The URI of the page that will handle the requests. When the user agent uses this URI, it must replace the first occurrence of the exact literal string "`%s`" with an escaped version of the URI of the content in question (as defined below), and then fetch the resulting URI using the GET method (or equivalent for non-HTTP URIs).

To get the escaped version of the URI, first, the domain part of the URI (if any) must be converted to its punycode representation, and then, every character in the URI that is not in the ranges given in the next paragraph must be replaced by its UTF-8 byte representation, each byte being represented by a U+0025 (%) character and two digits in the range U+0030 (0) to U+0039 (9) and U+0041 (A) to U+0046 (F) giving the hexadecimal representation of the byte.

The ranges of characters that must not be escaped are: U+002D (-), U+002E (.), U+0030 (0) to U+0039 (9), U+0041 (A) to U+005A (Z), U+005F (_), U+0061 (a) to U+007A (z), and U+007E (~).

> If the user had visited a site that made the following call:
>
> ```
> navigator.registerContentHandler('application/x-soup',
> 'http://example.com/soup?url=%s', 'SoupWeb™')
> ```
>
> ...and then clicked on a link such as:
>
> ```
> <a href="http://www.example.net/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 URI:
>
> ```
> http://example.com/soup?url=http%3A%2F
> %2Fwww.example.net%2Fchickenk%C3%AFwi.soup
> ```
>
> This site could then fetch the `chickenkïwi.soup` file and do whatever it is that it does with soup (synthesise it and ship it to the user, or whatever).

title

A descriptive title of the handler, which the UA might use to remind the user what the site in question is.

User agents should raise [security exceptions](#) if the methods are called with *protocol* or *mimeType* values that the UA deems to be "privileged". For example, a site attempting to register a handler for `http` URIs or `text/html` content in a Web browser would likely cause an exception to be raised.

User agents must raise a `SYNTAX_ERR` exception if the *uri* argument passed to one of these methods does not contain the exact literal string "`%s`".

User agents must not raise any other exceptions (other than binding-specific exceptions, such as for an incorrect number of arguments in an ECMAScript implementation).

This section does not define how the pages registered by these methods are used, beyond the requirements on how to process the *uri* value (see above). To some extent, the processing model for navigating across documents defines some cases where these methods are relevant, but in general UAs may use this information wherever they would otherwise consider handing content to native plugins or helper applications.

UAs must not use registered content handlers to handle content that was returned as part of a non-GET transaction (or rather, as part of any non-idempotent transaction), as the remote site would not be able to fetch the same data.

4.5.1.1. 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 URIs. The mechanism described in this section can result in secret Intranet URIs being leaked, in the following manner:

1. The user registers a third-party content handler as the default handler for a content type.

2. The user then browses his corporate Intranet site and accesses a document that uses that content type.

3. The user agent contacts the third party and hands the third party the URI to the Intranet content.

No actual confidential file data is leaked in this manner, but the URIs themselves could contain confidential information. For example, the URI could be `https://www.corp.example.com /upcoming-aquisitions/samples.egf`, which might tell the third party that Example Corporation is intending to merge with Samples LLC. Implementors might wish to consider allowing administrators to disable this feature for certain subdomains, content types, or protocols.

Leaking secure URIs. User agents should not send HTTPS URIs to third-party sites registered as content handlers, in the same way that user agents do not send `Referer` headers from secure sites to third-party sites.

Leaking credentials. User agents must never send username or password information in the URIs that are escaped and included sent to the handler sites. User agents may even avoid attempting to pass to Web-based handlers the URIs of resources that are known to require authentication to access, as such sites would be unable to access the resources in question without prompting the user for credentials themselves (a practice that would require the user to know whether to trust the third-party handler, a decision many users are unable to make or even understand).

4.5.1.2. 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:                   |
|                                                              |
|     Kittens-at-work displayer                                |
|     http://kittens.example.org/?show=%s                      |
|                                                              |
| Do you trust the administrators of the "kittens.example.     |
| org" domain?                                                 |
|                                                              |
|                  ( Trust kittens.example.org )  (( Cancel )) |
|_____|
```

...where "Kittens at work" is the title of the page that invoked the method, "http://kittens.example.org/" is the URI of that page, "x-meow" is the string that was passed to the registerProtocolHandler() method as its first argument (*protocol*), "http://kittens.example.org/?show=%s" was the second argument (*uri*), and "Kittens-at-work displayer" was the third argument (*title*).

If the user clicks the Cancel button, then nothing further happens. If the user clicks the "Trust" button, then the handler is remembered.

When the user then attempts to fetch a URI that uses the "x-meow:" scheme, then it might display a dialog as follows:

```
||[ Unknown Protocol ]||||||||||||||||||||||||||||||||||||||||||
|                                                               |
| You have attempted to access:                                 |
|                                                               |
|     x-meow:S2l0dGVucyBhcmUgdGhlIGN1dGVzdCE%3D                 |
```

```
|                                                               |
| How would you like FerretBrowser to handle this resource?     |
|                                                               |
|   (o) Contact the FerretBrowser plugin registry to see if     |
|        there is an official way to handle this resource.      |
|                                                               |
|   ( ) Pass this URI to a local application:                   |
|        [ /no application selected/           ] ( Choose )     |
|                                                               |
|   ( ) Pass this URI to the "Kittens-at-work displayer"        |
|        application at "kittens.example.org".                  |
|                                                               |
|   [ ] Always do this for resources using the "x-meow"         |
|        protocol in future.                                    |
|                                                               |
|                                      ( Ok )  (( Cancel ))     |
|_____|
```

...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.

4.6. Offline Web applications

4.6.1. Introduction

...

4.6.2. Application caches

An **application cache** is a collection of resources. An application cache is identified by the URI 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 URI, 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 URI 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 browsing context 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 URIs, 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 URI 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 scheme/host/port 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 an `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 URIs 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 URI in the list can be flagged with multiple different types, and thus an entry can end up being categorised 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**: URIs, used as prefix match patterns, each of which is mapped to a fallback entry. Each namespace URI prefix, when parsed as a URI, has the same scheme/host/port as the manifest.

- Zero or more URIs 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.

4.6.3. The cache manifest syntax

4.6.3.1. 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.

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. If any other text is found on the first line, the user agent will ignore the entire file. The first line may optionally be preceded by a U+FEFF BYTE ORDER MARK (BOM) character.

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 URI, 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 URI reference or IRI reference, and then zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters. [RFC3986] [RFC3987]

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 URI reference or IRI reference, one or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters, another valid URI reference or IRI reference, and then zero or more U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters. [RFC3986] [RFC3987]

Manifests may contain sections more than once. Sections may be empty.

URIs 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 URI of the data line, and the corresponding fallback page being the second URI. All the other pages to be cached must be listed in explicit sections.

Opportunistic caching namespaces must have the same scheme/host/port as the manifest itself.

An opportunistic caching namespace must not be listed more than once.

URIs that the user agent is to put into the online whitelist must all be specified in online whitelist sections. (This is needed for any URI that the page is intending to use to communicate back to the server.)

URIs 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. (URIs in the online whitelist section may match opportunistic caching namespaces, however.)

Relative URIs must be given relative to the manifest's own URI.

URIs in manifests must not have fragment identifiers.

4.6.3.2. 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 bytestream 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. All U+0000 NULL characters must be replaced by U+FFFD REPLACEMENT CHARACTERs.

2. Let *explicit URIs* be an initially empty list of explicit entries.

3. Let *fallback URIs* be an initially empty mapping of opportunistic caching namespaces to fallback entries.

4. Let *online whitelist URIs* be an initially empty list of URIs for a online whitelist.

5. Let *input* be the decoded text of the manifest's bytestream.

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. If the first character in *line* is a U+0023 NUMBER SIGN (#) character, then jump back to the step labelled "start of line".

16. Drop any trailing U+0020 SPACE, or U+0009 CHARACTER TABULATION (tab) characters at the end of *line*.

17. 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 labelled "start of line".

18. 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 labelled "start of line".

19. 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 labelled "start of line".

20. This is either a data line or it is syntactically incorrect.

↳ **If *mode* is "explicit"**

If *line* is not a syntactically valid URI reference or IRI reference, then jump back to the step labelled "start of line".

Otherwise, resolve the URI reference or IRI reference to an absolute URI or IRI, and drop the fragment identifier, if any.

Now, if the resource's URI has a different <scheme> component than the manifest's URI, then jump back to the step labelled "start of line".

Otherwise, add this URI to the *explicit URIs*.

↳ **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 labelled "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*.

Strip any leading U+0020 SPACE or U+0009 CHARACTER TABULATION (tab) characters in *part two*.

If *part one* and *part two* are not both syntactically valid URI or IRI references, then jump back to the step labelled "start of line".

Resolve the URI or IRI references in *part one* and *part two* to absolute URIs or IRIs.

If the absolute URI or IRI corresponding to *part one* is already in the *fallback URIs* mapping as an [opportunistic caching namespace](), then jump back to the step labelled "start of line".

If the absolute URI or IRI corresponding to *part one* does not have the [same scheme/host/port]() as the manifest's URI, then jump back to the step labelled "start of line".

If the absolute URI or IRI corresponding to *part two* has a different <scheme> component than the manifest's URI, then jump back to the step labelled "start of line".

Otherwise, add the absolute URI or IRI corresponding to *part one* to the *fallback URIs* mapping as an [opportunistic caching namespace](), mapped to the absolute URI corresponding to *part two* as the [fallback entry]().

↳ **If *mode* is "online whitelist"**

If *line* is not a syntactically valid URI reference or IRI reference, then jump back to the step labelled "start of line".

Otherwise, resolve the URI reference or IRI reference to an absolute URI or IRI, and drop the fragment identifier, if any.

Now, if the resource's URI has a different <scheme> component than the manifest's URI, then jump back to the step labelled "start of line".

Otherwise, add this URI to the *online whitelist URIs*.

21. Jump back to the step labelled "start of line". (That step jumps to the next, and last, step when the end of the file is reached.)

22. Return the *explicit URIs* list, the *fallback URIs* mapping, and the *online whitelist URIs*.

Relative URI references and IRI references resolved to absolute URIs or IRIs in the above algorithm must use the manifest's URI as the Base URI from the Retrieval URI for the purposes reference resolution as defined by RFC 3986. [RFC3986]

> *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.*

4.6.4. 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:

<div style="border:2px solid red; color:red;">

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)

</div>

1. Let *manifest URI* be the URI of the manifest of the cache to be updated.

2. Let *cache group* be the group of application caches identified by *manifest URI*.

3. Let *cache* be the most recently updated application cache identified by *manifest URI* (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 URI of *manifest URI* in *cache*, and that resource is categorised 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 URI*, and let *manifest* be that resource.

   If the resource is labelled 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 parser for manifests fails when checking the magic signature), or if the resource is labelled with a MIME type other than `text/cache-manifest`, then run these substeps:

   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, optionally

alerting the user to the failure.

    3. Otherwise, jump to the last step in the overall set of steps of the update process.

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 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. Then, jump to the last step of 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 URI, 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 URIs with flags.

14. Add all the URIs in the list of explicit entries obtained by parsing *manifest* to *file list*, each flagged with "explicit entry".

15. Add all the URIs 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 URIs 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 URIs of implicit entries in *cache* to *file list*, each flagged with "implicit entry".

18. If this is an upgrade attempt, then add all the URIs of dynamic entries in *cache* to *file list*, each flagged with "dynamic entry".

19. If any URI is in *file list* more than once, then merge the entries into one entry for that URI, that entry having all the flags that the original entries had.

20. For each URI in *file list*, run the following steps:

    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 honour 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.

    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), then run these substeps:

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, optionally alerting the user to the failure.

3. Otherwise, jump to the last step in the overall set of steps of the update process.

4. Otherwise, the fetching succeeded. Store the resource in the *new cache*.

5. If the URI being processed was flagged as an "explicit entry" in *file list*, then categorise the entry as an explicit entry.

6. If the URI being processed was flagged as a "fallback entry" in *file list*, then categorise the entry as a fallback entry.

7. If the URI being processed was flagged as a "opportunistic entry" in *file list*, then categorise the entry as an opportunistically cached entry.

8. If the URI being processed was flagged as an "implicit entry" in *file list*, then categorise the entry as a implicit entry.

9. If the URI being processed was flagged as an "dynamic entry" in *file list*, then categorise the entry as a dynamic entry.

21. Store *manifest* in *new cache*, if it's not there already, and categorise this entry (whether newly added or not) as the manifest.

22. Store the list of opportunistic caching namespaces, and the URIs of the fallback entries that they map to, in the new cache.

23. Store the URIs that form the new online whitelist in the new cache.

24. If this is a cache attempt, then:

    Set the status of *cache group* to *idle*.

    Associate any `Document` objects that were flagged as candidates for this manifest URI'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.

25. Otherwise, this is an upgrade attempt:

    Set the status of *cache group* to *idle*.

    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.

26. Abort these steps. The following step is jumped to by various parts of the algorithm above when they have to cancel the update.

27. 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.

4.6.5. 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 URI **matches an opportunistic caching namespace** if there exists an application cache whose manifest's URI has the same scheme/host/port as the URI 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 URI being examined. If multiple opportunistic caching namespaces match the same URI, the one with the longest <path> component is the one that matches. A URI 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 URI.

When the **application cache selection algorithm** algorithm is invoked with a manifest URI, 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 optimisation, if the resource was loaded from an application cache, and the manifest URI of that cache doesn't match the manifest URI 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 URI of that application cache's manifest is the same as the manifest URI 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 URI of that application cache's manifest is *not* the same as the manifest URI 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 URI does not have the same scheme/host/port as the resource's own URI, 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 URI, and that application cache contains a resource with the URI of the manifest, and that resource is categorised as a manifest, then: store the resource in the matching cache with the most up to date version, categorised 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 URI's caches.

4. If there is already an application cache identified by this manifest URI, then that application cache does not yet contain a resource with the URI of the manifest, or it does but that resource is not yet categorised as a manifest: store the resource in that cache, categorised 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.

5. Otherwise, there is no matching application cache: create a new application cache identified by this manifest URI, store the resource in that cache, categorised 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.

4.6.5.1. 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 URI, 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 URI 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 URI has the <u>same scheme/host/port</u> as the manifest's URI, and the start of the resource's URI's <path> component is exactly matched by the <path> component of an <u>opportunistic caching namespace</u> in the <u>application cache</u>, then:

   Fetch the resource normally. If this results 4xx or 5xx status codes or equivalent, or if there were network errors, then instead fetch, from the cache, the resource of the <u>fallback entry</u> 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.

4.6.6. 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 uri);
  void remove(in DOMString uri);

  // 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 categorised 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 categorised 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 dynamic entries 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(uri)`** 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. If there is already a resource in in the application cache with which the `ApplicationCache` object is associated that has the address *uri*, then ensure that entry is categorised as a dynamic entry and return and abort these steps.

3. If *uri* has a different <scheme> component than the manifest's URI, then raise a [security exception](#).

4. Return, but do not abort these steps.

5. Fetch the resource referenced by *uri*.

6. If this results 4xx or 5xx status codes or equivalent, or if there were network errors, then abort these steps.

7. 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 categorise 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(*uri*)** method must remove the [dynamic entry](#) categorisation of any entry with the address *uri* in the [application cache](#) with which the `ApplicationCache` object is associated. If this removes the last categorisation 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 categorised 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 categorised as a <u>manifest</u>.

6. Unassociate *document* from *cache* and instead associate it with *new cache*.

The following are the <u>event handler DOM attributes</u> 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 <u>`error`</u> 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 <u>`progress`</u> 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.

4.6.7. 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 <u>fire a simple event</u> called **offline** at <u>the body element</u>.

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 <u>fire a simple event</u> called **online** at <u>the body element</u>.

4.7. Session history and navigation

4.7.1. 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 URI 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](#).*

URIs without assaciated [state objects](#) are added to the session history as the user (or script) navigates from page to page.

A **state object** is an object representing a user interface state.

Pages can [add](#) [state objects](#) between their entry in the session history and the next ("forward") entry. These are then [returned to the script](#) when the user (or script) goes back in the history, thus enabling authors to use the "navigation" metaphor even in one-page applications.

At any point, one of the entries in the session history is the **current entry**. This is the entry representing the [active document](#) of the [browsing context](#). The [current entry](#) is usually an entry for the [location](#) of the `Document`. However, it can also be one of the entries for [state objects](#) added to the history by that document.

Entries that consist of [state objects](#) share the same `Document` as the entry for the page that was active when they were added.

Contiguous entries that differ just by fragment identifier also share the same `Document`.

> *Note: All entries that share the same `Document` (and that are therefore merely different states of one particular document) are contiguous by definition.*

User agents may **discard** the DOMs 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 pages' DOMs and when they should cache them. See the section on the `load` and `unload` events for more details.

Entries that have had their DOM discarded must, for the purposes of the algorithms given below, act as if they had not. When the user or script navigates back or forwards to a page which has no in-memory DOM objects, any other entries that shared the same `Document` object with it must share the new object as well.

When state object entries are added, a URI can be provided. This URI is used to replace the state object entry if the `Document` is evicted.

When a user agent discards the DOM from an entry in the session history, it must also discard all the entries that share that `Document` but do not have an associated URI (i.e. entries that only have a

state object). Entries that shared that Document object but had a state object and have a different URI 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.

4.7.2. 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, as described below. The **specified entry** is the one whose index equals the index of the current entry plus *delta*.

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 preform 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.

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 there are any entries with state objects between the current entry and the specified entry (not inclusive), then the user agent must iterate through every entry between the current entry and the specified entry, starting with the entry closest to the current entry, and ending with the

one closest to the specified entry. For each entry, if the entry is a state object, the user agent must activate the state object.

4. If the specified entry has a different `Document` object than the current entry then the user agent must run the following substeps:

    1. 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.

    2. 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.

    3. 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.)

    4. 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.

    5. 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.

5. If the specified entry is a state object, the user agent must activate that state object.

6. If the specified entry has a URI 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 URI 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.

<div style="border:2px solid red; padding:8px; color:red;">how does the changing of the global attributes affect .watch() when seen from other Windows?</div>

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 first check the third argument. If a third argument is specified, then the user agent must verify that the third argument is a valid URI or IRI (as defined by RFC 3986 and 3987), and if so, that, after resolving it to an absolute URI, it is either identical to the document's URI, or that it differs from the document's URI only in the <query>, <abs_path>, and/or <fragment> parts, as applicable (the <query> and <abs_path> parts can only be the same if the document's URI uses a hierarchical <scheme>). If the verification fails (either because the argument is syntactically incorrect, or differs in a way not described as acceptable in the previous sentence) then the user agent must raise a security exception. [RFC3986] [RFC3987]

If the third argument passes its verification step, or if the third argument was omitted, then the user agent must remove from the session history any entries for that 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.

Then, the user agent must add a state object entry to the session history, after the current entry, with the specified *data* as the state object, the given *title* as the title, and, if the third argument is present, the given *url* as the URI of the entry.

Finally, the user agent must 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.

4.7.3. Activating state objects

When a state object in the session history is activated (which happens in the cases described above), the user agent must fire a **popstate** event in no namespace on the the body element using the PopStateEvent interface, with the state object in the state attribute. This event bubbles but is

not cancelable and has no default action.

```
interface PopStateEvent : Event {
  readonly attribute DOMObject state;
  void initPopStateEvent(in DOMString typeArg, in boolean
canBubbleArg, in boolean cancelableArg, in DOMObject statetArg);
  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.

Should we coalesce these events if they occur while the page is away? (e.g. during traversal -- see above)

4.7.4. 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 the URI of their document, and allow the current entry of the browsing context's session history to be changed, by adding or replacing entries in the history object.

```
interface Location {
  readonly attribute DOMString href;
  void assign(in DOMString url);
  void replace(in DOMString url);
  void reload();

  // URI decomposition attributes
          attribute DOMString protocol;
          attribute DOMString host;
          attribute DOMString hostname;
          attribute DOMString port;
          attribute DOMString pathname;
          attribute DOMString search;
          attribute DOMString hash;
};
```

In the ECMAScript DOM binding, objects implementing this interface must stringify to the same

value as the `href` attribute.

In the ECMAScript DOM binding, the `location` members of the `HTMLDocument` and `Window` interfaces behave as if they had a setter: user agents must treats attempts to set these `location` attribute as attempts at setting the `href` attribute of the relevant `Location` object instead.

The **`href`** attribute returns the address of the page represented by the associated `Document` object, as an absolute IRI reference.

On setting, the user agent must act as if the `assign()` method had been called with the new value as its argument.

When the **`assign(url)`** method is invoked, the UA must <u>navigate</u> the <u>browsing context</u> to the specified *url*.

When the **`replace(url)`** method is invoked, the UA must <u>navigate</u> to the specified *url* with <u>replacement enabled</u>.

Relative *url* arguments for `assign()` and `replace()` must be resolved relative to the base URI of the script that made the method call.

The `Location` interface also has the complement of <u>URI decomposition attributes</u>, **`protocol`**, **`host`**, **`port`**, **`hostname`**, **`pathname`**, **`search`**, and **`hash`**. These must follow the rules given for URI decomposition attributes, with the <u>input</u> being the address of the page represented by the associated `Document` object, as an absolute IRI reference (same as the `href` attribute), and the <u>common setter action</u> being the same as setting the `href` attribute to the new output value.

4.7.4.1. Security

User agents must raise a <u>security exception</u> whenever any of the members of a `Location` object are accessed by scripts whose <u>origin</u> is not the same as the `Location` object's associated `Document`'s origin, with the following exceptions:

* The `href` setter

User agents must not allow scripts to override the `href` attribute's setter.

4.7.5. 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.

4.8. Navigating across documents

Certain actions cause the browsing context to **navigate**. 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 also provide various ways for the user to explicitly cause a browsing context to navigate.

When a browsing context is navigated, the user agent must run the following steps:

1. Cancel any preexisting attempt to navigate the browsing context.

2. If the new resource is the same as the current resource, but a fragment identifier has been specified, then navigate to that fragment identifier and abort these steps.

3. 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.

4. 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.

5. 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 scheme/host/port as the URI in question, and that have this URI as one of their entries (excluding entries marked as manifest), and that already contain their manifest, categorised 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 specified 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:` URI). If this results in a redirect, return to step 2 with the new resource.

   > 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.

6. 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.

7. If the resource was not fetched from an application cache, and was to be fetched using HTTP

GET or equivalent, and its URI matches the opportunistic caching namespace of one or more application caches, then:

↳ **If the file was successfully downloaded**

> The user agent must cache the resource in all those application caches, categorised as opportunistically cached entries.

↳ **If the server returned a 4xx or 5xx status code or equivalent, or there were network errors**

> If the browsing context being navigated is a top-level browsing context, then the user agent must discard the failed load and instead use 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. For the purposes of session history (and features that depend on session history, e.g. bookmarking) the user agent must use the URI 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 URI of the fallback resource actually is.

8. 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*

9. Let *type* be the sniffed type of the resource.

10. 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.

11. 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.

12. 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

3. onunload

4. **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.

4.8.1. 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__.*

4.8.2. 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.

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 run the application cache selection algorithm with the value of that attribute as the manifest URI. Otherwise, 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 URIs 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 step 10 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`.

4.8.3. 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 tokenisation 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]

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.

4.8.4. 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.

4.8.5. 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`.

4.8.6. 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 <u>HTML document</u>, 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 <u>stopped parsing</u>.

Upon creation of the `Document` object, the user agent must run the <u>application cache selection algorithm</u> with no manifest.

After creating the `Document` object, but potentially before the page has been completely set up, the user agent must <u>update the session history with the new page</u>.

4.8.7. Navigating to a fragment identifier

When a user agent is supposed to navigate to a fragment identifier, then the user agent must <u>update the session history with the new page</u>, where "the new page" has the same `Document` as before but with the URI having the newly specified fragment identifier.

Part of that algorithm involves the user agent having to <u>scroll to the fragment identifier</u>, 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 <u>the indicated part of the document</u> is brought to the user's attention. If there is no indicated part, then the user agent must not scroll anywhere.

The **the indicated part of the document** is the one that the fragment identifier 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 <u>the indicated part of the document</u> is.

1. Let *fragid* be the <fragment> part of the URI. [RFC3987]

2. If *fragid* is the empty string, then the 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 <u>the indicated part of the document</u>; 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 <u>the indicated part of the document</u>; 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 <u>the indicated part of the document</u>, if that is an element; otherwise there is no *target element*. [SELECTORS]

4.9. Determining the type of a new resource in a browsing context

⚠**Warning!** *It is imperative that the rules in this section be followed exactly. When two user*

agents use different heuristics for content type detection, security problems can occur. For example, if a server believes a contributed file to be an image (and thus benign), but a Web browser believes the content to be HTML (and thus capable of executing script), the end user can be exposed to malicious content, making the user vulnerable to cookie theft attacks and other cross-site scripting attacks.

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

1. If the resource was fetched over an HTTP protocol, and there is no HTTP Content-Encoding header, but there is an HTTP Content-Type header and it has a value whose bytes exactly match one of the following three 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`

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

2. 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.

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

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

5. 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.

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

7. Otherwise, the sniffed type of the resource is *official type*.

4.9.1. 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 file match one of the following byte sets:

Bytes in Hexadecimal	Description
FE FF	UTF-16BE BOM or UTF-32LE BOM
FF FE	UTF-16LE BOM
00 00 FE FF	UTF-32BE BOM
EF BB BF	UTF-8 BOM

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

Should we remove UTF-32 from the above?

- Otherwise, if any of the first *n* bytes of the resource are in one of the following byte ranges:

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

  ...then the sniffed type of the resource is "application/octet-stream".

  maybe we should invoke the "Content-Type sniffing: image" section now, falling back on "application/octet-stream".

- Otherwise, the sniffed type of the resource is "text/plain".

4.9.2. 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 *pattern length* is smaller than *stream 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 $index_{stream}$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.

↳ **Otherwies, if the $index_{stream}$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), 0x0B (ASCII VT), 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. As a last-ditch effort, jump to the [text or binary](#) section.

Bytes in Hexadecimal		Sniffed type	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	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	The string "`<HTML`" in US-ASCII or compatible encodings, case-insensitively, possibly with leading spaces.
FF FF DF DF DF DF	*WS* 3C 48 45 41 44	text/html	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	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	The string "`%PDF-`", the PDF signature.
FF FF FF FF FF FF FF FF FF FF FF	25 21 50 53 2D 41 64 6F 62 65 2D	application/postscript	The string "`%!PS-Adobe-`", the PostScript signature.
FF FF FF FF FF FF	47 49 46 38 37 61	image/gif	The string "`GIF87a`", a GIF signature.
FF FF FF FF FF FF	47 49 46 38 39 61	image/gif	The string "`GIF89a`", a GIF signature.
FF FF FF FF FF FF FF FF	89 50 4E 47 0D 0A 1A 0A	image/png	The PNG signature.

Bytes in Hexadecimal		Sniffed type	Comment
Mask	Pattern		
FF FF FF	FF D8 FF	image/jpeg	A JPEG SOI marker followed by the first byte of another marker.
FF FF	42 4D	image/bmp	The string "BM", a BMP signature.

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).

4.9.3. Content-Type sniffing: image

If the first bytes of the file match one of the byte sequences in the first columns 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.

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*.

4.9.4. 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. 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.

6. 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 (step 5) in the overall algorithm in this section.

    3. Otherwise, increase *pos* by 1.

    4. Otherwise, return to step 2 in these substeps.

7. 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 step 5 in the overall algorithm in this section.

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

8. 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 step 5 in the overall algorithm in this section.

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

9. 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.

10. 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)

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

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

4.9.5. 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 Content-Type HTTP header contributes any data; 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 it 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. [HTTP]

For resources fetched from the filesystem, 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 a encoding or nothing.

1. Skip characters in *s* up to and including the first U+003B SEMICOLON (*;*) character.

2. Skip any U+0009, U+000A, U+000B, U+000C, U+000D, or U+0020 characters (i.e. spaces) that immediately follow the semicolon.

3. If the next six characters are not 'charset', return nothing.

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

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

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

7. 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***

   Return string between the two quotation marks.

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

Return the string between the two apostrophes.

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

> ↳ **If it is an unmatched U+0027 APOSTROPHE ("'")**

Return nothing.

> ↳ **Otherwise**

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

4.10. Client-side session and persistent storage of name/value pairs

4.10.1. Introduction

This section is non-normative.

This specification introduces two related mechanisms, similar to HTTP session cookies [RFC2965], for storing structured data on the client side.

The first is designed for scenarios where the user is carrying out a single transaction, but could be carrying out multiple transactions in different windows at the same time.

Cookies don't really handle this case well. For example, a user could be buying plane tickets in two different windows, using the same site. If the site used cookies to keep track of which ticket the user was buying, then as the user clicked from page to page in both windows, the ticket currently being purchased would "leak" from one window to the other, potentially causing the user to buy two tickets for the same flight without really noticing.

To address this, this specification introduces the `sessionStorage` DOM attribute. Sites can add data to the session storage, and it will be accessible to any page from that origin opened in that window.

> For example, a page could have a checkbox that the user ticks to indicate that he wants insurance:
>
> ```
> <label>
>  <input type="checkbox" onchange="sessionStorage.insurance =
> checked">
>  I want insurance on this trip.
> </label>
> ```
>
> A later page could then check, from script, whether the user had checked the checkbox or not:
>
> ```
> if (sessionStorage.insurance) { ... }
> ```
>
> If the user had multiple windows opened on the site, each one would have its own individual copy of the session storage object.

The second storage mechanism is designed for storage that spans multiple windows, and lasts beyond the current session. In particular, Web applications may wish to store megabytes of user data, such as entire user-authored documents or a user's mailbox, on the clientside for performance reasons.

Again, cookies do not handle this case well, because they are transmitted with every request.

The `globalStorage` DOM attribute is used to access a page's global 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:
>
> ```
> <p>
>   You have viewed this page
>   <span id="count">an untold number of</span>
>   time(s).
> </p>
> <script>
>   if (!globalStorage.pageLoadCount)
>     globalStorage.pageLoadCount = 0;
>   globalStorage.pageLoadCount =
> parseInt(globalStorage.pageLoadCount, 10) + 1;
>   document.getElementById('count').textContent =
> globalStorage.pageLoadCount;
> </script>
> ```

Each origin has its own separate storage area.

Storage areas (both session storage and global storage) store strings. To store structured data in a storage area, you must first convert it to a string.

4.10.2. The `Storage` interface

```
interface Storage {
  readonly attribute unsigned long length;
  DOMString key(in unsigned long index);
  DOMString getItem(in DOMString key);
  void setItem(in DOMString key, in DOMString data);
  void removeItem(in DOMString key);
};
```

Each `Storage` object provides access to a list of key/value pairs, which are sometimes called items. Keys 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 `globalStorage` 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.

When the setItem() method is invoked, events are fired on other HTMLDocument objects that can access the newly stored data, as defined in the sections on the sessionStorage and globalStorage attributes.

The **removeItem(*key*)** method must cause the key/value pair with the given *key* to be removed from the list associated with the object, if it exists. 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.

In the ECMAScript DOM binding, enumerating a Storage object must enumerate through the currently stored keys in the list the object is associated with. (It must not enumerate the values or the actual members of the interface). In the ECMAScript DOM binding, Storage objects must support dereferencing such that getting a property that is not a member of the object (i.e. is neither a member of the Storage interface nor of Object) must invoke the getItem() method with the property's name as the argument, and setting such a property must invoke the setItem() method with the property's name as the first argument and the given value as the second argument.

4.10.3. 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()` method is 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.

4.10.4. The `globalStorage` attribute

The **`globalStorage`** object provides a `Storage` object for origin.

User agents must have a set of global storage areas, one for each origin.

User agents should only expire data from the global storage areas for security reasons or when requested to do so by the user. User agents should always avoid deleting data while a script that could access that data is running. Data stored in global storage areas should be considered potentially user-critical. It is expected that Web applications will use the global storage areas for storing user-written documents.

When the `globalStorage` attribute is accessed, the user agent must check to see if it has allocated global 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 global storage area, and return it.

When the `setItem()` method is called on a `Storage` object *x* that is associated with a global storage area, then in every `HTMLDocument` object whose `Window` object's `globalStorage` attribute's `Storage` object is associated with the same storage area, other than *x*, a `storage` event must be fired, as described below.

4.10.5. 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 global storage).

When this happens, the user agent must fire a simple event called `storage` on the body element.

However, it is possible (indeed, for session storage areas, likely) that the target's `HTMLDocument` object is not an active document at that time. In such cases, the user agent must instead delay the firing of the event until such time as the `HTMLDocument` object in question becomes an active document again.

When there are multiple delayed `storage` events for the same `HTMLDocument` object, user agents must coalesce those events such that only one event fires when the document becomes active

again.

If the DOM of a page that has delayed `storage` events queued up is [discarded](#), then the delayed events are dropped as well.

4.10.6. Miscellaneous implementation requirements for storage areas

4.10.6.1. Disk space

User agents should limit the total amount of space allowed for a storage area based on the domain of the page setting the value.

User agents should not limit the total amount of space allowed on a per-storage-area basis, otherwise a site could just store data in any number of subdomains or ports, e.g. storing up to the limit in a1.example.com, a2.example.com, a3.example.com, etc, circumventing per-domain limits.

User agents may prompt the user when per-domain space quotas are reached, allowing the user to grant a site more space. This enables sites to store many user-created documents on the user's computer, for instance.

User agents should allow users to see how much space each domain is using.

If the storage area space limit is reached during a `setItem()` call, the user agent must raise an `INVALID_ACCESS_ERR` exception.

A mostly arbitrary limit of five megabytes per domain is recommended. Implementation feedback is welcome and will be used to update this suggestion in future.

4.10.6.2. Threads

Multiple browsing contexts must be able to access the global storage areas simultaneously in a predictable manner. Scripts must not be able to detect any concurrent script execution.

This is required to guarentee that the `length` attribute of a `Storage` object never changes while a script is executing, other than in a way that is predictable by the script itself.

There are various ways of implementing this requirement. One is that if a script running in one browsing context accesses a global storage area, the UA blocks scripts in other browsing contexts when they try to access the global 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.

4.10.7. Security and privacy

4.10.7.1. 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 global storage area 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 `globalStorage` object to scripts originating at the domain of the top-level document of the browsing context.

- Expiring stored data: user agents may automatically delete stored data after a period of time.

  For example, a user agent could treat third-party global 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 may present the persistent storage feature to the user in a way that does not distinguish it from HTTP session cookies. [RFC2965]

  This might encourage users to view persistent storage with healthy suspicion.

- Site-specific white-listing of access to global storage areas: user agents may allow sites to access session storage areas in an unrestricted manner, but require the user to authorise access to global storage areas.

- 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 this API 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 URIs, a technique already commonly used for innocuous purposes but easily repurposed for user tracking (even retroactively). This information can then be shared with other sites, using using visitors' IP addresses and other user-specific data (e.g. user-agent headers and configuration settings) to combine separate sessions into coherent user profiles.

4.10.7.2. Cookie resurrection

If the user interface for persistent storage presents data in the persistent storage feature separately from data in HTTP session cookies, then users are likely to delete data in one and not the other.

This would allow sites to use the two features as redundant backup for each other, defeating a user's attempts to protect his privacy.

4.10.7.3. DNS spoofing attacks

Because of the potential for DNS spoofing attacks, one cannot guarentee that a host claiming to be in a certain domain really is from that domain. 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 global storage areas.

4.10.7.4. Cross-directory attacks

Different authors sharing one host name, for example users hosting content on `geocities.com`, all share one persistent storage object. There is no feature to restrict the access by pathname. Authors on shared hosts are therefore recommended to avoid using the persistent storage feature, as it would be trivial for other authors to read from and write to the same storage area.

> *Note: Even if a path-restriction feature was made available, the usual DOM scripting security model would make it trivial to bypass this protection and access the data from any path.*

4.10.7.5. Implementation risks

The two primary risks when implementing this persistent storage feature are letting hostile sites read information from other domains, and letting hostile sites write information that is then read from other domains.

Letting third-party sites read data that is not supposed to be read from their domain causes *information leakage*, For example, a user's shopping wishlist on one domain could be used by another domain for 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 model described in this specification is important for user security.

4.11. Client-side database storage

4.11.1. Introduction

...

4.11.2. 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 <u>Database</u> 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.

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.

Otherwise, if the database 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 <u>Database</u> object representing the database associated with the <u>origin</u> of the <u>active document</u> of the <u>browsing context</u> of the <u>Window</u> object on which the method was called 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 optimise 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 {
  boolean 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.

4.11.3. 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 ommitted or null, then treat the *arguments* array as empty.

   The result is *the statement*.

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 filesystem will fail.

User agents should limit the total amount of space allowed for each origin, but may prompt the user and extend the limit if a database is reaching its quota. User agents should allow users to see how much space each database is using.

A mostly arbitrary limit of five megabytes per origin is recommended. Implementation feedback is welcome and will be used to update this suggestion in future.

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.

4.11.4. 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.

```
interface SQLResultSetRowList {
  readonly attribute unsigned long length;
  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.

4.11.5. 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 dependending upon.
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 localised to the user's language.

4.11.6. 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 occured 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, 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. Invoke the statement's result set callback 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 his 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 occured 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 occured in this transaction. If the error callback returned false, and the last error wasn't itself a failure when committing the transaction, then try to commit the transaction. If that fails, or if the callback couldn't be called (e.g. the method was called with only one argument), or if it didn't return false, then rollback the transaction. Any still-pending statements in the transaction are discarded.

4.11.7. Privacy

In contrast with the `globalStorage` feature, which intentionally allows data to be accessed across multiple domains, protocols, and ports (albeit in a controlled fashion), this database feature is limited to scripts running with the same origin as the database. Thus, it is expected that the privacy implications be equivalent to those already present in allowing scripts to communicate with their originating host.

User agents are encouraged to treat data stored in databases in the same way as cookies for the purposes of user interfaces, to reduce the risk of using this feature for cookie resurrection.

4.11.8. Security

4.11.8.1. 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.

4.11.8.2. 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.

4.12. Links

4.12.1. Hyperlink elements

The `a`, `area`, and `link` elements can, in certain situations described in the definitions of those elements, represent **hyperlinks**.

The **href** attribute on a hyperlink element must have a value that is a URI (or IRI). This URI is the *destination resource* of the hyperlink.

> **The *href* attribute on *a* and *area* elements is not required; when those elements do not have *href* attributes they do not represent hyperlinks.**
>
> **The *href* attribute on the *link* element is required, but whether a *link* element represents a hyperlink or not depends on the value of the *rel* attribute of that element.**

The **target** attribute, if present, must be a valid browsing context name. User agents use this name when following hyperlinks.

The **ping** attribute, if present, gives the URIs of the resources that are interested in being notified if the user follows the hyperlink. The value must be a space separated list of one or more URIs (or IRIs). 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 recognised by the UA, then the document has no particular relationship with the destination resource other than there being a hyperlink between the two.

The **media** attribute describes for which media the target document was designed. It is purely advisory. The value must be a valid media query. [MQ] The default, if the `media` attribute is omitted, is `all`.

The **hreflang** attribute on hyperlink elements, if present, gives the language of the linked resource. It is purely advisory. The value must be a valid RFC 3066 language code. [RFC3066] User agents must not consider this attribute authoritative — upon fetching the resource, user agents must only

use language information associated with the resource to determine its language, not metadata included in the link to the resource.

The `type` attribute, if present, gives the MIME type of the linked resource. It is purely advisory. The value must be a valid MIME type, optionally with parameters. [RFC2046] User agents must not consider the `type` attribute authoritative — upon fetching the resource, user agents must not use metadata included in the link to the resource to determine its type.

4.12.2. Following hyperlinks

When a user *follows a hyperlink*, the user agent must navigate a browsing context to the URI of the hyperlink.

The URI of the hyperlink is URI given by resolving the the `href` attribute of that hyperlink relative to the hyperlink's element. In the case of server-side image maps, the URI 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 chosing 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 chosing 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.

4.12.2.1. 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, treat each resulting token as a URI (resolving relative URIs according to element's base URI) and then should send a request to each of the resulting URIs. This may be done in parallel with the primary request, and is independent of the result of that request.

User agents should allow the user to adjust this behaviour, for example in conjunction with a setting that disables the sending of HTTP `Referer` headers. Based on the user's preferences, UAs may either ignore the `ping` attribute altogether, or selectively ignore URIs in the list (e.g. ignoring any third-party URIs).

For URIs that are HTTP URIs, the requests must be performed using the POST method (with an

empty entity body in the request). User agents must ignore any entity bodies returned in the responses, but must, unless otherwise specified by the user, honour the HTTP headers — in particular, HTTP cookie headers. [RFC2965]

> *Note: To save bandwidth, implementors might wish to consider omitting optional headers such as `Accept` from these requests.*

When the `ping` attribute is present, user agents should clearly indicate to the user that following the hyperlink will also cause secondary requests to be sent in the background, possibly including listing the actual target URIs.

> *The `ping` attribute is redundant with pre-existing technologies like HTTP redirects and JavaScript in allowing Web pages to track which off-site links are most popular or allowing advertisers to track click-through rates.*
>
> *However, the `ping` attribute provides these advantages to the user over those alternatives:*
>
> - *It allows the user to see the final target URI unobscured.*
>
> - *It allows the UA to inform the user about the out-of-band notifications.*
>
> - *It allows the paranoid user to disable the notifications without losing the underlying link functionality.*
>
> - *It allows the UA to optimise the use of available network bandwidth so that the target page loads faster.*
>
> *Thus, while it is possible to track users without this feature, authors are encouraged to use the `ping` attribute so that the user agent can improve the user experience.*

4.12.3. Link types

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

In this section, the term *referenced document* refers to the resource identified by the element representing the link, and the term *current document* refers to the resource within which the element representing the link finds itself.

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.

Link type	Effect on...		Brief description
	`link`	`a` and `area`	
bookmark	*not allowed*	Hyperlink	Gives the permalink for the nearest ancestor section.
contact	Hyperlink	Hyperlink	Gives a link to contact information for the current document.
external	*not allowed*	Hyperlink	Indicates that the referenced document is not part of the same site as the current document.
feed	Hyperlink	Hyperlink	Gives the address of a syndication feed for the current document.
first	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the first document in the series is the referenced document.
help	Hyperlink	Hyperlink	Provides a link to context-sensitive help.
icon	External Resource	*not allowed*	Imports an icon to represent the current document.
index	Hyperlink	Hyperlink	Gives a link to the document that provides a table of contents or index listing the current document.
last	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the last document in the series is the referenced document.
license	Hyperlink	Hyperlink	Indicates that the current document is covered by the copyright license described by the referenced document.
next	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the next document in the series is the referenced document.
nofollow	*not allowed*	Hyperlink	Indicates that the current document's original author or publisher does not endorse the referenced document.
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 pre-emptively cached.
prev	Hyperlink	Hyperlink	Indicates that the current document is a part of a series, and that the previous document in the series is the referenced document.
search	Hyperlink	Hyperlink	Gives a link to a resource that can be used to search through the current document and its related pages.
stylesheet	External Resource	*not allowed*	Imports a stylesheet.
sidebar	Hyperlink	Hyperlink	Specifies that the referenced document, if retrieved, is intended to be shown in the browser's sidebar (if it has one).

Link type	Effect on...		Brief description
	link	**a and area**	
tag	Hyperlink	Hyperlink	Gives a tag (identified by the given address) that applies to the current document.
up	Hyperlink	Hyperlink	Provides a link to a document giving the context for the current document.

Some of the types described below list synonyms for these values. These are to be handled as specified by user agents, but must not be used in documents.

4.12.3.1. Link type "alternate"

The alternate keyword may be used with link, a, and area elements. For link elements, if the rel attribute does not also contain the keyword stylesheet, it creates a hyperlink; but if it *does* also contains the keyword stylesheet, the alternate keyword instead modifies the meaning of the stylesheet keyword in the way described for that keyword, and the rest of this subsection doesn't apply.

The alternate keyword indicates that the referenced document is an alternate representation of the current document.

The nature of the referenced document is given by the media, hreflang, and type attributes.

If the alternate keyword is used with the media attribute, it indicates that the referenced document is intended for use with the media specified.

If the alternate keyword is used with the hreflang attribute, 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.

4.12.3.2. Link type "archives"

The archives keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The `archives` keyword indicates that the referenced document describes a collection of records, documents, or other materials of historical interest.

|| A blog's index page could link to an index of the blog's past posts with `rel="archives"`.

Synonyms: For historical reasons, user agents must also treat the keyword "`archive`" like the `archives` keyword.

4.12.3.3. Link type "`author`"

The `author` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a [hyperlink](#).

For `a` and `area` elements, the `author` keyword indicates that the referenced document provides further information about the author of the section that the element defining the hyperlink [applies](#) to.

For `link` elements, the `author` keyword indicates that the referenced document provides further information about the author for the page as a whole.

> **Note: The "referenced document" can be, and often is, a `mailto:` URI giving the e-mail address of the author. [MAILTO]**

Synonyms: For historical reasons, user agents must also treat `link`, `a`, and `area` elements that have a `rev` attribute with the value "`made`" as having the `author` keyword specified as a link relationship.

4.12.3.4. Link type "`bookmark`"

The `bookmark` keyword may be used with `a` and `area` elements.

The `bookmark` keyword gives a permalink for the nearest ancestor `article` element of the linking element in question, or of [the section the linking element is most closely associated with](#), if there are no ancestor `article` elements.

> The following snippet has three permalinks. A user agent could determine which permalink applies to which part of the spec by looking at where the permalinks are given.
>
> ```
>    ...
>   <body>
>    <h1>Example of permalinks</h1>
>    <div id="a">
>     <h2>First example</h2>
>     <p><a href="a.html" rel="bookmark">This</a> permalink applies to
>     only the content from the first H2 to the second H2. The DIV
> isn't
>     exactly that section, but it roughly corresponds to it.</p>
>    </div>
>    <h2>Second example</h2>
>    <article id="b">
>     <p><a href="b.html" rel="bookmark">This</a> permalink applies to
>     the outer ARTICLE element (which could be, e.g., a blog
> post).</p>
>     <article id="c">
>      <p><a href="c.html" rel="bookmark">This</a> permalink applies
> ```

```
to
    the inner ARTICLE element (which could be, e.g., a blog
comment).</p>
    </article>
  </article>
 </body>
 ...
```

4.12.3.5. Link type "`contact`"

The `contact` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a [hyperlink](#).

For `a` and `area` elements, the `contact` keyword indicates that the referenced document provides further contact information for the section that the element defining the hyperlink [applies](#) to.

User agents must treat any hyperlink in an `address` element as having the `contact` link type specified.

For `link` elements, the `contact` keyword indicates that the referenced document provides further contact information for the page as a whole.

4.12.3.6. Link type "`external`"

The `external` keyword may be used with `a` and `area` elements.

The `external` keyword indicates that the link is leading to a document that is not part of the site that the current document forms a part of.

4.12.3.7. Link type "`feed`"

The `feed` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a [hyperlink](#).

The `feed` keyword indicates that the referenced document is a syndication feed. If the `alternate` link type is also specified, then the feed is specifically the feed for the current document; otherwise, the feed is just a syndication feed, not necessarily associated with a particular Web page.

The first `link`, `a`, or `area` element in the document (in tree order) that creates a hyperlink with the link type `feed` must be treated as the default syndication feed for the purposes of feed autodiscovery.

> **Note: The _feed_ keyword is implied by the _alternate_ link type in certain cases (q.v.).**

The following two `link` elements are equivalent: both give the syndication feed for the current page:

```
<link rel="alternate" type="application/atom+xml" href="data.xml">

<link rel="feed alternate" href="data.xml">
```

The following extract offers various different syndication feeds:

```
<p>You can access the planets database using Atom feeds:</p>
<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>
```

4.12.3.8. Link type "*help*"

The help keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

For a and area elements, the help keyword indicates that the referenced document provides further help information for the parent of the element defining the hyperlink, and its children.

> In the following example, the form control has associated context-sensitive help. The user agent could use this information, for example, displaying the referenced document if the user presses the "Help" or "F1" key.
>
> ```
>     <p><label> Topic: <input name=topic> <a href="help/topic.html"
> rel="help">(Help)</a></label></p>
> ```

For link elements, the help keyword indicates that the referenced document provides help for the page as a whole.

4.12.3.9. Link type "*icon*"

The icon keyword may be used with link elements, for which it creates an external resource link.

The specified resource is an icon representing the page or site, and should be used by the user agent when representing the page in the user interface.

Icons could be auditory icons, visual icons, or other kinds of icons. If multiple icons are provided, the user agent must select the most appropriate icon according to the media attribute.

4.12.3.10. Link type "*license*"

The license keyword may be used with link, a, and area elements. For link elements, it creates a hyperlink.

The license keyword indicates that the referenced document provides the copyright license terms under which the current document is provided.

Synonyms: For historical reasons, user agents must also treat the keyword "copyright" like the license keyword.

4.12.3.11. Link type "*nofollow*"

The nofollow keyword may be used with a and area elements.

The nofollow keyword indicates that the link is not endorsed by the original author or publisher of the page.

4.12.3.12. Link type "*noreferrer*"

The `norefererr` 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.

4.12.3.13. Link type "`pingback`"

The `pingback` keyword may be used with `link` elements, for which it creates an external resource link.

For the semantics of the `pingback` keyword, see the Pingback 1.0 specification. [PINGBACK]

4.12.3.14. Link type "`prefetch`"

The `prefetch` keyword may be used with `link` elements, for which it creates an external resource link.

The `prefetch` keyword indicates that preemptively fetching and caching the specified resource is likely to be beneficial, as it is highly likely that the user will require this resource.

4.12.3.15. 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]*

4.12.3.16. 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.

4.12.3.17. 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**.

4.12.3.18. Link type "`tag`"

The `tag` keyword may be used with `link`, `a`, and `area` elements. For `link` elements, it creates a hyperlink.

The `tag` keyword indicates that the *tag* that the referenced document represents applies to the current document.

4.12.3.19. 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.

4.12.3.19.1. 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.

4.12.3.19.2. 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 occurance 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.

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 breadcrumbs. In the following example, the current page can be reached via two paths.
>
> ```
>     <nav>
>      <p>
>        <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>
    </p>
    <p>
     <a href="/" rel="index up up">Main</a> >
     <a href="/second-hand/" rel="up">Second hand</a> >
     <a>Dishwashers</a>
    </p>
   </nav>
```

Note: The `relList` DOM attribute (e.g. on the `a` element) does not currently represent multiple `up` keywords (the interface hides duplicates).

4.12.3.20. 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.

4.12.3.20.1. 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.

4.12.3.20.2. 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.

4.12.3.20.3. 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.

4.12.3.20.4. 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.

4.12.3.21. 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.

4.13. Interfaces for URI manipulation

An interface that has a complement of **URI 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 URI 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 URI decomposition attributes will define an **input**, which is a URI 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 URI 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 URI 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 URI given by the "component" column in the input URI 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*.

The rules for parsing and constructing URIs are described in RFC 3986 and RFC 3987. [RFC3986] [RFC3987]

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>/<ihost>	input is hierarchical and uses a server-based naming authority	—	—	Remove all leading U+002F SOLIDUS ("/") characters	—
`port`	<port>	input is hierarchical and uses a server-based naming authority	—	—	Remove any characters in the new value that are not in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE	The new value is not the empty string
`pathname`	<abs_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	U+003F QUESTION MARK ("?")	—	Remove one leading U+003F QUESTION MARK ("?") character, if any	—

Attribute	Component	Getter Condition	Prefix	Suffix	Setter Preprocessor	Setter Condition
`hash`	<fragment>	Fragment identifier is longer than zero characters	U+0023 NUMBER SIGN ("#")	—	Remove one leading U+0023 NUMBER SIGN ("#") character, if any	—

The **<hostport>** component is defined as being the <host>/<ihost> component, followed by a colon and the <port> component, but with the colon and <port> component omitted if the given port matches the default port for the protocol given by the <scheme> component.

5. Editing

This section describes various features that allow authors to enable users to edit documents and parts of documents interactively.

5.1. Introduction

This section is non-normative.

Would be nice to explain how these features work together.

5.2. 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, which is also the *invalid value default*. There is no *missing value default*.

If an HTML element has a `contenteditable` attribute set to the true state, or if its nearest ancestor HTML element with the `contenteditable` attribute set has its attribute set to the true state, or if it has no ancestors with the `contenteditable` attribute set 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 nearest ancestor HTML element with the `contenteditable` attribute set is not *editable*, or it has no ancestor with the `contenteditable` attribute set and the Document itself has `designMode` disabled, and the element is thus not editable.

The **`contentEditable`** DOM attribute, on getting, must return the string "`inherit`" if the content attribute isn't set, "`true`" if the attribute is set and has the true state, and "`false`" 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.

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.

5.2.1. User editing actions

There are several actions that the user agent should allow the user to perform while the user is interacting with an editing host. How exactly each action is triggered is not defined for every action, but when it is not defined, suggested key bindings are provided to guide implementors.

Move the caret

> User agents must allow users to move the caret to any position within an editing host, even into nested editable elements. This could be triggered as the default action of `keydown` events with various key identifiers and as the default action of `mousedown` events.

Change the selection

> User agents must allow users to change [the selection](#) within an editing host, even into nested editable elements. This could be triggered as the default action of `keydown` events with various key identifiers and as the default action of `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 behaviour is UA-dependent, but user agents must not, in response to a request to insert text, generate a DOM that is less conformant than the DOM prior to the request.

> User agents should allow users to insert new paragraphs into elements that 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 behaviour 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 behaviour is UA-dependent, but user agents must not, in response to a request to insert a line separator, generate a DOM that is less conformant than the DOM prior to the request.

Delete

UAs should offer a way for the user to delete text and elements, e.g. as the default action of `keydown` events whose identifiers are "U+0008" or "U+007F".

Five edge cases in particular need to be considered carefully when implementing this feature: backspacing at the start of an element, backspacing when the caret is immediately after an element, forward-deleting at the end of an element, forward-deleting when the caret is immediately before an element, and deleting a [selection](#) whose start and end points do not share a common parent node.

In any case, the exact behaviour is UA-dependent, but user agents must not, in response to a request to delete text or an element, generate a DOM that is less conformant than the DOM prior to the request.

Insert, and wrap text in, semantic elements

UAs should offer a way for the user to mark text as having [stress emphasis](#) and as being [important](#), and may offer the user the ability to mark text and paragraphs with other semantics.

UAs should similarly offer a way for the user to insert empty semantic elements (such as, again, `em`, `strong`, and others) to subsequently fill by entering text manually.

UAs should also offer a way to remove those semantics from marked up text, and to remove empty semantic element that have been inserted.

The exact behaviour is UA-dependent, but user agents must not, in response to a request to wrap semantics around some text or to insert or remove a semantic element, generate a DOM that is less conformant than the DOM prior to the request.

Select and move non-editable elements nested inside editing hosts

UAs should offer a way for the user to move images and other non-editable parts around the

content within an editing host. This may be done using the drag and drop mechanism. User agents must not, in response to a request to move non-editable elements nested inside editing hosts, generate a DOM that is less conformant than the DOM prior to the request.

Edit form controls nested inside editing hosts

When an editable form control is edited, the changes must be reflected in both its current value *and* its default value. For `input` elements this means updating the `defaultValue` DOM attribute as well as the `value` DOM attribute; for `select` elements it means updating the `option` elements' `defaultSelected` DOM attribute as well as the `selected` DOM attribute; for `textarea` elements this means updating the `defaultValue` DOM attribute as well as the `value` DOM attribute. (Updating the `default*` DOM attributes causes content 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.

5.2.2. Making entire documents editable

Documents have a **designMode**, which can be either enabled or disabled.

The `designMode` DOM attribute on the `Document` object takes 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.

When the `Document` has `designMode` enabled, event listeners registered on the document or any elements owned by the document must do nothing.

5.3. Drag and drop

This section defines an event-based drag-and-drop mechanism.

This specification does not define exactly what a *drag-and-drop operation* actually is.

On a visual medium with a pointing device, a drag operation could be the default action of a `mousedown` event that is followed by a series of `mousemove` events, and the drop could be triggered by the mouse being released.

On media without a pointing device, the user would probably have to explicitly indicate his intention to perform a drag-and-drop operation, stating what he wishes to drag and what he wishes to drop, respectively.

However it is implemented, drag-and-drop operations must have a starting point (e.g. where the mouse was clicked, or the start of the selection or element that was selected for the drag), may have

any number of intermediate steps (elements that the mouse moves over during a drag, or elements that the user picks as possible drop points as he cycles through possibilities), and must either have an end point (the element above which the mouse button was released, or the element that was finally selected), or be canceled. The end point must be the last element selected as a possible drop point before the drop occurs (so if the operation is not canceled, there must be at least one element in the middle step).

5.3.1. 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.

When a `DragEvent` object is created, a new `DataTransfer` object must be created and assigned to the `dataTransfer` context information field of the event object.

```
interface DataTransfer {
           attribute DOMString dropEffect;
           attribute DOMString effectAllowed;
  void clearData(in DOMString format);
  void setData(in DOMString format, in DOMString data);
  DOMString getData(in DOMString format);
  void setDragImage(in Element image, in long x, in long y);
  void addElement(in Element element);
};
```

`DataTransfer` objects can conceptually contain various kinds of data.

When a `DragEvent` event object is initialised, the `DataTransfer` object created for the event's `dataTransfer` member must be initialised as follows:

- The `DataTransfer` object must initially contain no data, no elements, and have no associated image.

- The `DataTransfer` object's `effectAllowed` attribute must be set to "`uninitialized`".

- The `dropEffect` attribute must be set to "`none`".

The **dropEffect** attribute controls the drag-and-drop feedback that the user is given during a drag-and-drop operation.

The attribute must ignore any attempts to set it to a value other than `none`, `copy`, `link`, and `move`. On getting, the attribute must return the last of those four values that it was set to.

The **effectAllowed** attribute is used in the drag-and-drop processing model to initialise the dropEffect attribute during the dragenter and dragover events.

The attribute must ignore any attempts to set it to a value other than `none`, `copy`, `copyLink`, `copyMove`, `link`, `linkMove`, `move`, `all`, and `uninitialized`. On getting, the attribute must return the last of those values that it was set to.

DataTransfer objects can hold pieces of data, each associated with a unique format. Formats are generally given by MIME types, with some values special-cased for legacy reasons.

The **clearData(*format*)** method must clear the DataTransfer object of any data associated with the given *format*. If *format* is the value "`Text`", then it must be treated as "`text/plain`". If the *format* is "`URL`", then it must be treated as "`text/uri-list`".

The **setData(*format*, *data*)** method must add *data* to the data stored in the DataTransfer object, labelled as being of the type *format*. This must replace any previous data that had been set for that format. If *format* is the value "`Text`", then it must be treated as "`text/plain`". If the *format* is "`URL`", then it must be treated as "`text/uri-list`".

The **getData(*format*)** method must return the data that is associated with the type *format*, if any, and must return the empty string otherwise. If *format* is the value "`Text`", then it must be treated as "`text/plain`". If the *format* is "`URL`", then the data associated with the "`text/uri-list`" format must be parsed as appropriate for `text/uri-list` data, and the first URI from the list must be returned. If there is no data with that format, or if there is but it has no URIs, then the method must return the empty string. [RFC2483]

The **setDragImage(*element*, *x*, *y*)** method sets which element to use to generate the drag feedback. The *element* argument can be any Element; if it is an img element, then the user agent should use the element's image (at its intrinsic size) to generate the feedback, otherwise the user agent should base the feedback on the given element (but the exact mechanism for doing so is not specified).

The **addElement(*element*)** method is an alternative way of specifying how the user agent is to render the drag feedback. It adds an element to the DataTransfer object.

5.3.2. Events fired during a drag-and-drop action

The following events are involved in the drag-and-drop model. Whenever the processing model described below causes one of these events to be fired, the event fired must use the DragEvent interface defined above, must have the bubbling and cancelable behaviours given in the table below, and must have the context information set up as described after the table, 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	dropEffec
dragstart	Source node	✓ Bubbles	✓ Cancelable	Contains source node	uninitialized	none

Event Name	Target	Bubbles?	Cancelable?	`dataTransfer`	`effectAllowed`	`dropEffec`
				unless a selection is being dragged, in which case it is empty		
drag	[Source node](#)	✓ Bubbles	✓ Cancelable	Empty	[Same as last event](#)	`none`
dragenter	[Immediate user selection](#) or [the body element](#)	✓ Bubbles	✓ Cancelable	Empty	[Same as last event](#)	[Based on `effectAllo` value](#)
dragleave	[Previous target element](#)	✓ Bubbles	—	Empty	[Same as last event](#)	`none`
dragover	[Current target element](#)	✓ Bubbles	✓ Cancelable	Empty	[Same as last event](#)	[Based on `effectAllo` value](#)
drop	[Current target element](#)	✓ Bubbles	✓ Cancelable	`getData()` returns data set in `dragstart` event	[Same as last event](#)	[Current drag operation](#)
dragend	[Source node](#)	✓ Bubbles	—	Empty	[Same as last event](#)	[Current drag operation](#)

The `dataTransfer` object's contents are empty except for `dragstart` events and `drop` events, for which the contents are set as described in the processing model, below.

The `effectAllowed` attribute must be set to "`uninitialized`" for `dragstart` events, and to whatever value the field had after the last drag-and-drop event was fired for all other events (only counting events fired by the user agent for the purposes of the drag-and-drop model described below).

The `dropEffect` attribute must be set to "`none`" for `dragstart`, `drag`, 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

effectAllowed	dropEffect
`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`

5.3.3. Drag-and-drop processing model

When the user attempts to begin a drag operation, the user agent must first determine what is being dragged. If the drag operation was invoked on a selection, then it is the selection that is being dragged. Otherwise, it is the first element, going up the ancestor chain, starting at the node that the user tried to drag, that has the DOM attribute `draggable` set to true. If there is no such element, then nothing is being dragged, the drag-and-drop operation is never started, and the user agent must not continue with this algorithm.

> Note: *img* elements and *a* elements with an *href* attribute have their *draggable* attribute set to true by default.

If the user agent determines that something can be dragged, a `dragstart` event must then be fired.

If it is a selection that is being dragged, then this event must be fired on the node that the user started the drag on (typically the text node that the user originally clicked). If the user did not specify a particular node, for example if the user just told the user agent to begin a drag of "the selection", then the event must be fired on the deepest node that is a common ancestor of all parts of the selection.

> 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 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.

5.3.3.1. When the drag-and-drop operation starts or ends in another document

The model described above is independent of which `Document` object the nodes involved are from; the events must be fired as described above and the rest of the processing model must be followed as described above, irrespective of how many documents are involved in the operation.

5.3.3.2. When the drag-and-drop operation starts or ends in another application

If the drag is initiated in another application, the source node is not a DOM node, and the user agent must use platform-specific conventions instead when the requirements above involve the source node. User agents in this situation must act as if the dragged data had been added to the `DataTransfer` object when the drag started, even though no `dragstart` event was actually fired; user agents must similarly use platform-specific conventions when deciding on what drag feedback to use.

If a drag is started in a document but ends in another application, then the user agent must instead replace the parts of the processing model relating to handling the _target_ according to platform-specific conventions.

In any case, scripts running in the context of the document must not be able to distinguish the case of a drag-and-drop operation being started or ended in another application from the case of a drag-and-drop operation being started or ended in another document from another domain.

5.3.4. 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`.

5.3.5. 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.

5.3.5.1. 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 clipbroad. 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).

5.3.5.2. 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.

5.3.5.3. 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 text from the keyboard (either as `text/plain` or `text/uri-list`). If the contents of the clipboard cannot be represented as text or URIs, then the paste operation must not have any effect.

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.

5.3.5.4. 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.

If the contents of the selection cannot be represented as text or URIs, then the paste operation must not have any effect.

5.3.6. Security risks in the drag-and-drop model

User agents must not make the data added to the `DataTransfer` object during the `dragstart` event available to scripts until the `drop` event, because otherwise, if a user were to drag sensitive information from one document to a second document, crossing a hostile third document in the process, the hostile document could intercept the data.

For the same reason, user agents must only consider a drop to be successful if the user specifically ended the drag operation — if any scripts end the drag operation, it must be considered unsuccessful (canceled) and the `drop` event must not be fired.

User agents should take care to not start drag-and-drop operations in response to script actions. For example, in a mouse-and-window environment, if a script moves a window while the user has his mouse button depressed, the UA would not consider that to start a drag. This is important because otherwise UAs could cause data to be dragged from sensitive sources and dropped into hostile documents without the user's consent.

5.4. Undo history

> There has got to be a better way of doing this, surely.

The user agent must associate an **undo transaction history** with each `HTMLDocument` object.

The undo transaction history is a list of entries. The entries are of two type: DOM changes and undo objects.

Each **DOM changes** entry in the undo transaction history consists of batches of one or more of the following:

- Changes to the content attributes of an `Element` node.

- Changes to the DOM attributes of a `Node`.

- Changes to the DOM hierarchy of nodes that are descendants of the `HTMLDocument` object (`parentNode`, `childNodes`).

Undo object entries consist of objects representing state that scripts running in the document are managing. For example, a Web mail application could use an undo object to keep track of the fact that a user has moved an e-mail to a particular folder, so that the user can undo the action and have the e-mail return to its former location.

Broadly speaking, DOM changes entries are handled by the UA in response to user edits of form controls and editing hosts on the page, and undo object entries are handled by script in response to higher-level user actions (such as interactions with server-side state, or in the implementation of a drawing tool).

5.4.1. The `UndoManager` interface

This API sucks. Seriously. It's a terrible API. Really bad. I hate it. Here are the requirements:

- Has to cope with cases where the server has undo state already when the page is loaded, that can be stuffed into the undo buffer onload.

- Has to support undo/redo.

- Has to cope with the "undo" action being "contact the server and tell it to undo", rather than it being the opposite of the "redo" action.

- Has to cope with some undo states expiring from the undo history (e.g. server can only remember one undelete action) but other states not expiring (e.g. client can undo arbitrary amounts of local edits).

To manage <u>undo object</u> entries in the <u>undo transaction history</u>, the <u>UndoManager</u> interface can be used:

```
interface UndoManager {
  unsigned long add(in DOMObject data, in DOMStrong title);
  void remove(in unsigned long index);
  void clearUndo();
  void clearRedo();
  DOMObject item(in unsigned long index);
  readonly attribute unsigned long length;
  readonly attribute unsigned long position;
};
```

The **undoManager** attribute of the <u>Window</u> interface must return the object implementing the <u>UndoManager</u> interface for that <u>Window</u> object's associated <u>HTMLDocument</u> object.

In the ECMAScript DOM binding, objects implementing this interface must also support being dereferenced using the square bracket notation, such that dereferencing with an integer index is equivalent to invoking the <u>item()</u> method with that index (e.g. `undoManager[1]` returns the same as `undoManager.item(1)`).

<u>UndoManager</u> objects represent their document's <u>undo transaction history</u>. Only <u>undo object</u> entries are visible with this API, but this does not mean that <u>DOM changes</u> entries are absent from the <u>undo transaction history</u>.

The **length** attribute must return the number of <u>undo object</u> entries in the <u>undo transaction history</u>.

The **item(*n*)** method must return the *n*th <u>undo object</u> entry in the <u>undo transaction history</u>.

The <u>undo transaction history</u> has a **current position**. This is the position between two entries in the <u>undo transaction history</u>'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 <u>undo object</u> entry nearest to the <u>undo position</u>, on the "redo" side. If there are no <u>undo object</u> entries on the "redo" side, then the attribute must return the same as the <u>length</u> attribute. If there are no <u>undo object</u> entries on the "undo" side of the <u>undo position</u>, the position attribute returns zero.

Note: Since the <u>undo transaction history</u> contains both <u>undo object</u> entries and <u>DOM changes</u> entries, but the `position` attribute only returns indices relative to <u>undo object</u> entries, it is possible for several "undo" or "redo" actions to be performed without the value of the `position` attribute changing.

The **add(*data, title*)** method's behaviour depends on the current state. Normally, it must insert the *data* object passed as an argument into the <u>undo transaction history</u> immediately before the <u>undo position</u>, optionally remembering the given *title* to use in the UI. If the method is called <u>during an undo operation</u>, however, the object must instead be added immediately *after* the <u>undo position</u>.

If the method is called and there is neither <u>an undo operation in progress</u> nor <u>a redo operation in progress</u> then any entries in the <u>undo transaction history</u> after the <u>undo position</u> 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 <u>undo object</u> 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. <u>DOM changes</u> entries are unaffected by this method.

The **clearUndo()** method must remove all entries in the <u>undo transaction history</u> before the <u>undo position</u>, be they <u>DOM changes</u> entries or <u>undo object</u> entries.

The **clearRedo()** method must remove all entries in the <u>undo transaction history</u> after the <u>undo position</u>, be they <u>DOM changes</u> entries or <u>undo object</u> 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.

5.4.2. Undo: moving back in the undo transaction history

When the user invokes an undo operation, or when the `execCommand()` method is called with the `undo` command, the user agent must perform an undo operation.

If the <u>undo position</u> is at the start of the <u>undo transaction history</u>, then the user agent must do nothing.

If the entry immediately before the <u>undo position</u> is a <u>DOM changes</u> entry, then the user agent must remove that <u>DOM changes</u> entry, reverse the DOM changes that were listed in that entry, and, if the changes were reversed with no problems, add a new <u>DOM changes</u> entry (consisting of the opposite of those DOM changes) to the <u>undo transaction history</u> on the other side of the <u>undo position</u>.

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 <u>DOM changes</u> entry, without doing anything else.

If the entry immediately before the <u>undo position</u> is an <u>undo object</u> entry, then the user agent must first remove that <u>undo object</u> entry from the <u>undo transaction history</u>, and then must fire an `undo` event on the Document object, using the <u>undo object</u> 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.

5.4.3. Redo: moving forward in the undo transaction history

When the user invokes a redo operation, or when the `execCommand()` method is called with the `redo` command, the user agent must perform a redo operation.

This is mostly the opposite of an undo operation, but the full definition is included here for completeness.

If the undo position is at the end of the undo transaction history, then the user agent must do nothing.

If the entry immediately after the undo position is a DOM changes entry, then the user agent must remove that DOM changes entry, reverse the DOM changes that were listed in that entry, and, if the changes were reversed with no problems, add a new DOM changes entry (consisting of the opposite of those DOM changes) to the undo transaction history on the other side of the undo position.

If the DOM changes cannot be redone (e.g. because the DOM state is no longer consistent with the changes represented in the entry), then the user agent must simply remove the DOM changes entry, without doing anything else.

If the entry immediately after the undo position is an undo object entry, then the user agent must first remove that undo object entry from the undo transaction history, and then must fire a `redo` event on the `Document` object, using the undo object entry's associated undo object as the event's data.

5.4.4. The `UndoManagerEvent` interface and the `undo` and `redo` events

```
interface UndoManagerEvent : Event {
  readonly attribute DOMObject data;
  void initUndoManagerEvent(in DOMString typeArg, in boolean
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.

5.4.5. Implementation notes

How user agents present the above conceptual model to the user is not defined. The undo interface could be a filtered view of the undo transaction history, it could manipulate the undo transaction history in ways not described above, and so forth. For example, it is possible to design a UA that

appears to have separate [undo transaction histories](#) for each form control; similarly, it is possible to design systems where the user has access to more undo information than is present in the offical (as described above) [undo transaction history](#) (such as providing a tree-based approach to document state). Such UI models should be based upon the single [undo transaction history](#) described in this section, however, such that to a script there is no detectable difference.

5.5. Command APIs

The **execCommand(***commandId, doShowUI, value***)** method on the [HTMLDocument](#) interface allows scripts to perform actions on the [current selection](#) or at the current caret position. Generally, these commands would be used to implement editor UI, for example having a "delete" button on a toolbar.

There are three variants to this method, with one, two, and three arguments respectively. The *doShowUI* and *value* parameters, even if specified, are ignored unless otherwise stated.

> ### *Note: In this specification, in fact, the doShowUI parameter is always ignored, regardless of its value. It is included for historical reasons only.*

When any of these methods are invoked, user agents must act as described in the list below.

For actions marked "**editing hosts only**", if the selection is not entirely within an [editing host](#), of if there is no selection and the caret is not inside an [editing host](#), then the user agent must do nothing.

If the *commandId* is undo

> The user agent must [move back one step](#) in its [undo transaction history](#), restoring the associated state. If there is no further undo information the user agent must do nothing. See the [undo history](#).

If the *commandId* is redo

> The user agent must [move forward one step](#) in its [undo transaction history](#), restoring the associated state. If there is no further undo (well, "redo") information the user agent must do nothing. See the [undo history](#).

If the *commandId* is selectAll

> The user agent must change the selection so that all the content in the currently focused [editing host](#) is selected. If no [editing host](#) is focused, then the content of the entire document must be selected.

If the *commandId* is unselect

> The user agent must change the selection so that nothing is selected.

> We need some sort of way in which the user can make a selection without risk of script clobbering it.

If the *commandId* is superscript

> *Editing hosts only.* The user agent must act as if the user had requested that the selection [be wrapped in the semantics](#) of the `sup` element (or unwrapped, or, if there is no selection, have that semantic inserted or removed — the exact behaviour is UA-defined).

If the *commandId* is subscript

> *Editing hosts only.* The user agent must act as if the user had requested that the selection [be](#)

wrapped in the semantics of the *sub* element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA).

If the *commandId* is `formatBlock`

Editing hosts only. This command changes the semantics of the blocks containing the selection.

If there is no selection, then, where in the description below refers to the selection, the user agent must act as if the selection was an empty range at the caret position.

If the *value* parameter is not specified or has a value other than one of the following literal strings:

- `<address>`
- `<aside>`
- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`
- `<nav>`
- `<p>`
- `<pre>`

...then the user agent must do nothing.

Otherwise, the user agent must, for every position in the selection, take the furthest prose content ancestor element of that position that contains only phrasing content, and, if that element is a descendant of the editing host, rename it (as if the `Element.renameNode()` method had been used) according to the *value*, by stripping the leading `<` character and the trailing `>` character and using the rest as the new tag name, using the HTML namespace.

If the *commandId* is `delete`

Editing hosts only. The user agent must act as if the user had performed a backspace operation.

If the *commandId* is `forwardDelete`

Editing hosts only. The user agent must act as if the user had performed a forward delete operation.

If the *commandId* is `insertLineBreak`

Editing hosts only. The user agent must act as if the user had requested a line separator.

If the *commandId* is `insertParagraph`

Editing hosts only. The user agent must act as if the user had performed a break block editing action.

If the *commandId* is `insertText`

Editing hosts only. The user agent must act as if the user had inserted text corresponding to the *value* parameter.

If the *commandId* is `vendorID-customCommandID`

User agents may implement vendor-specific extensions to this API. Vendor-specific extensions to the list of commands should use the syntax *vendorID-customCommandID* so as to prevent clashes between extensions from different vendors and future additions to this specification.

If the *commandId* is something else

> User agents must do nothing.

5.6. 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**.

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]*

5.6.1. 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.

```
interface Selection {
  readonly attribute Node anchorNode;
  readonly attribute long anchorOffset;
  readonly attribute Node focusNode;
  readonly attribute long focusOffset;
  readonly attribute boolean isCollapsed;
  void collapse(in Node parentNode, in long offset);
  void collapseToStart();
```

```
        void collapseToEnd();
        void selectAllChildren(in Node parentNode);
        void deleteFromDocument();
        readonly attribute long rangeCount;
        Range getRangeAt(in long index);
        void addRange(in Range range);
        void removeRange(in Range range);
        void removeAllRanges();
        DOMString toString();
    };
```

The Selection interface is represents a list of Range objects. The first item in the list has index 0, and the last item has index *count*-1, where *count* is the number of ranges in the list. [DOM2RANGE]

All of the members of the Selection interface are defined in terms of operations on the Range objects represented by this object. These operations can raise exceptions, as defined for the Range interface; this can therefore result in the members of the Selection interface raising exceptions as well, in addition to any explicitly called out below.

The **anchorNode** attribute must return the value returned by the startContainer attribute of the last Range object in the list, or null if the list is empty.

The **anchorOffset** attribute must return the value returned by the startOffset attribute of the last Range object in the list, or 0 if the list is empty.

The **focusNode** attribute must return the value returned by the endContainer attribute of the last Range object in the list, or null if the list is empty.

The **focusOffset** attribute must return the value returned by the endOffset attribute of the last Range object in the list, or 0 if the list is empty.

The **isCollapsed** attribute must return true if there are zero ranges, or if there is exactly one range and its collapsed attribute is itself true. Otherwise it must return false.

The **collapse(*parentNode, offset*)** method must raise a WRONG_DOCUMENT_ERR DOM exception if *parentNode*'s ownerDocument is not the HTMLDocument object with which the Selection object is associated. Otherwise it is, and the method must remove all the ranges in the Selection list, then create a new Range object, add it to the list, and invoke its setStart() and setEnd() methods with the *parentNode* and *offset* values as their arguments.

The **collapseToStart()** method must raise an INVALID_STATE_ERR DOM exception if there are no ranges in the list. Otherwise, it must invoke the collapse() method with the startContainer and startOffset values of the first Range object in the list as the arguments.

The **collapseToEnd()** method must raise an INVALID_STATE_ERR DOM exception if there are no ranges in the list. Otherwise, it must invoke the collapse() method with the endContainer and endOffset values of the last Range object in the list as the arguments.

The **selectAllChildren(*parentNode*)** method must invoke the collapse() method with the *parentNode* value as the first argument and 0 as the second argument, and must then invoke the selectNodeContents() method on the first (and only) range in the list with the *parentNode* value as the argument.

The **deleteFromDocument()** method must invoke the deleteContents() method on each

range in the list, if any, from first to last.

The **rangeCount** attribute must return the number of ranges in the list.

The **getRangeAt(*index*)** method must return the *index*th range in the list. If *index* is less than zero or greater or equal to the value returned by the rangeCount attribute, then the method must raise an INDEX_SIZE_ERR DOM exception.

The **addRange(*range*)** method must add the given *range* Range object to the list of selections, at the end (so the newly added range is the new last range). Duplicates are not prevented; a range may be added more than once in which case it appears in the list more than once, which (for example) will cause toString() to return the range's text twice.

The **removeRange(*range*)** method must remove the first occurrence of *range* in the list of ranges, if it appears at all.

The **removeAllRanges()** method must remove all the ranges from the list of ranges, such that the rangeCount attribute returns 0 after the removeAllRanges() method is invoked (and until a new range is added to the list, either through this interface or via user interaction).

The **toString()** method must return a concatenation of the results of invoking the toString() method of the Range object on each of the ranges of the selection, in the order they appear in the list (first to last).

In language bindings where this is supported, objects implementing the Selection interface must stringify to the value returned by the object's toString() method.

> In the following document fragment, the emphasised parts indicate the selection.
>
> ```
> <p>The cute girl likes the <cite>Oxford English
> Dictionary</cite>.</p>
> ```
>
> If a script invoked window.getSelection().toString(), the return value would be "the Oxford English".

Note: The Selection interface has no relation to the DataGridSelection interface.

5.6.2. 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. Communication

6.1. Event definitions

Messages in cross-document messaging and, by default, in server-sent DOM events, use the **message** event.

The following interface is defined for this event:

```
interface MessageEvent : Event {
  readonly attribute DOMString data;
```

```
    readonly attribute DOMString domain;
    readonly attribute DOMString uri;
    readonly attribute Window source;
    void initMessageEvent(in DOMString typeArg, in boolean
  canBubbleArg, in boolean cancelableArg, in DOMString dataArg, in
  DOMString domainArg, in DOMString uriArg, in Window sourceArg);
    void initMessageEventNS(in DOMString namespaceURI, in DOMString
  typeArg, in boolean canBubbleArg, in boolean cancelableArg, in
  DOMString dataArg, in DOMString domainArg, in DOMString uriArg, in
  Window sourceArg);
  };
```

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 **domain** attribute represents, in cross-document messaging, the domain of the document from which the message came.

The **uri** attribute represents, in cross-document messaging, the address of the document from which the message came.

The **source** attribute represents, in cross-document messaging, the Window from which the message came.

6.2. Server-sent DOM events

This section describes a mechanism for allowing servers to dispatch DOM events into documents that expect it. The event-source element provides a simple interface to this mechanism.

6.2.1. 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 add the URI specified in *src* to the list of event sources for that object. The same URI can be registered multiple times.

When the **removeEventSource(*src*)** method is invoked, the user agent must remove the URI specified in *src* from the list of event sources for that object. If the same URI has been registered multiple times, removing it must only remove one instance of that URI for each invocation of the removeEventSource() method.

Relative URIs must be resolved relative to ... .

6.2.2. 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 URI isn't removed from the list before they do so, fetch the resource identified by that URI.

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.

In general, the semantics of the transport protocol specified by the URIs for the event sources must be followed, including HTTP caching rules.

For HTTP connections, the `Accept` header may be included; if included, it must only contain formats of event framing that are supported by the user agent (one of which must be `application/x-dom-event-stream`, as described below).

Other formats of event framing may also be supported in addition to `application/x-dom-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.

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 `application/x-dom-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 of approximately five seconds.

HTTP 200 OK responses that have a Content-Type other than `application/x-dom-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 short delay.

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 URI instead of the previously specified URI for all subsequent requests for this event source. (It doesn't affect other event sources with the same URI 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 URI, but if the user agent needs to again request the resource at a later point, it must return to the previously specified URI 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 short delay of approximately five seconds.

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.

6.2.3. Parsing an event stream

This event stream format's MIME type is `application/x-dom-event-stream`.

The event stream format is (in pseudo-BNF):

```
<stream>          ::= <bom>? <event>*
<event>           ::= [ <comment> | <command> | <field> ]* <newline>
<comment>         ::= ';' <any-char>* <newline>
<command>         ::= ':' <any-char>* <newline>
<field>           ::= <name> [ ':' <space>? <any-char>* ]? <newline>
<name>            ::= <name-start-char> <name-char>*

# 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
<name-start-char> ::= a single Unicode character other than
                      ':', ';', U+000D CARRIAGE RETURN and U+000A LINE
FEED
```

```
<name-char>          ::= a single Unicode character other than
                           ':', 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. User agents must treat those three variants as equivalent line terminators.

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, in blocks separated by blank lines. Comment lines (those starting with the character ';') and command lines (those starting with the character ':') must be ignored.

Command lines are reserved for future extensions.

For each non-blank, non-comment, non-command line, the field name must first be taken. This is everything on the line up to but not including the first colon (':') or the line terminator, whichever comes first. Then, if there was a colon, the data for that line must be taken. This is everything after the colon, ignoring a single space after the colon if there is one, up to the end of the line. If there was no colon the data is the empty string.

> Examples:
>
> ```
> Field name: Field data
>
> This is a blank field
>
> 1. These two lines: have the same data
> 2. These two lines:have the same data
>
> 1. But these two lines:  do not
> 2. But these two lines: do not
> ```

If a field name occurs multiple times in a block, the value for that field in that black must consist of the data parts for each of those lines, concatenated with U+000A LINE FEED characters between them (regardless of what the line terminators used in the stream actually are).

> For example, the following block:
>
> ```
> Test: Line 1
> Foo:  Bar
> Test: Line 2
> ```
>
> ...is treated as having two fields, one called Test with the value "Line 1\nLine 2" (where \n represents a newline), and one called Foo with the value " Bar" (note the leading space character).

A block thus consists of all the name-value pairs for its fields. Command lines have no effect on blocks and are not considered part of a block.

Note: Since any random stream of characters matches the above format, there is no

need to define any error handling.

6.2.4. Interpreting an event stream

Once the fields have been parsed, they are interpreted as follows (these are case-sensitive exact comparisons):

`Event` field

> This field gives the name of the event. For example, `load`, `DOMActivate`, `updateTicker`. If there is no field with this name, the name <u>`message`</u> must be used.

`Namespace` field

> This field gives the DOM3 namespace for the event. (For normal DOM events this would be null.) If it isn't specified the event namespace is null.

`Class` field

> This field gives is the interface used for the event, for instance `Event`, `UIEvent`, `MutationEvent`, `KeyboardEvent`, etc. For compatibility with DOM3 Events, the values `UIEvents`, `MouseEvents`, `MutationEvents`, and `HTMLEvents` are valid values and must be treated respectively as meaning the interfaces `UIEvent`, `MouseEvent`, `MutationEvent`, and `Event`. (This value can therefore be used as the argument to `createEvent()`.)
>
> If the value is not specified but the `Namespace` is null and the `Event` field exactly matches one of the events specified by DOM3 Events in <u>section 1.4.2 "Complete list of event types"</u>, then the interface used must default to the interface relevant for that event type. <u>[DOM3EVENTS]</u>
>
> > For example:
> >
> >   Event: click
> >
> > ...would cause `Class` to be treated as `MouseEvent`.
>
> If the `Namespace` is null and the `Event` field is <u>`message`</u> (including if it was not specified explicitly), then the <u>`MessageEvent`</u> interface must be used.
>
> Otherwise, the `Event` interface must be used.
>
> It is quite possible to give the wrong class for an event. This is equivalent to creating an event in the DOM using the DOM Event APIs, but using the wrong interface for it.

`Bubbles` field

> This field specifies whether the event is to bubble. If it is specified and has the value `No`, the event must not bubble. If it is specified and has any other value (including `no` or `NO`) then the event must bubble.
>
> If it is not specified but the `Namespace` field is null and the `Event` field exactly matches one of the events specified by DOM3 Events in <u>section 1.4.2 "Complete list of event types"</u>, then the event must bubble if the DOM3 Events spec specifies that that event bubbles, and musn't bubble if it specifies it does not. <u>[DOM3EVENTS]</u>
>
> > For example:
> >
> >   Event: load

...would cause `Bubbles` to be treated as `No`.

Otherwise, the event must bubble.

`Cancelable` field

This field specifies whether the event can have its default action prevented. If it is specified and has the value `No`, the event must not be cancelable. If it is specified and has any other value (including `no` or `NO`) then the event must be cancelable.

If it is not specified, but the `Namespace` field is null and the `Event` field exactly matches one of the events specified by DOM3 Events in section 1.4.2 "Complete list of event types", then the event must be cancelable if the DOM3 Events specification specifies that it is, and must not be cancelable otherwise. [DOM3EVENTS]

> For example:
>
> ```
> Event: load
> ```
>
> ...would cause `Cancelable` to be treated as `No`.

Otherwise, the event must be cancelable.

`Target` field

This field gives the node that the event is to be dispatched on.

If the object for which the event source is being processed is not a Node, but the `Target` field is nonetheless specified, then the event must be dropped.

Otherwise, if field is specified and its value starts with a `#` character, then the remainder of the value represents an ID, and the event must be dispatched on the same node as would be obtained by the `getElementById()` method on the `ownerDocument` of the node whose event source is being processed.

> For example,
>
> ```
> Target: #test
> ```
>
> ...would target the element with ID `test`.

Otherwise, if the field is specified and its value is the literal string `"Document"`, then the event must be dispatched at the `ownerDocument` of the node whose event source is being processed.

Otherwise, the field (whether specified or not) is ignored and the event must be dispatched at the object itself.

Other fields depend on the interface specified (or possibly implied) by the `Class` field. If the specified interface has an attribute that exactly matches the name of the field, and the value of the field can be converted (using the type conversions defined in ECMAScript) to the type of the attribute, then it must be used. Any attributes (other than the `Event` interface attributes) that do not have matching fields are initialised to zero, null, false, or the empty string.

> For example:
>
> ```
> Event: click
> Class: MouseEvent
> ```

```
button: 2
```

...would result in a 'click' event using the `MouseEvent` interface that has `button` set to `2` but `screenX`, `screenY`, etc, set to 0, false, or null as appropriate.

If a field does not match any of the attributes on the event, it must be ignored.

For example:

```
Event: keypress
Class: MouseEvent
keyIdentifier: 0
```

...would result in a `MouseEvent` event with its fields all at their default values, with the event name being `keypress`. The `keyIdentifier` field would be ignored. (If the author had not included the `Class` field explicitly, it would have just worked, since the class would have defaulted as described above.)

Once a blank line or the end of the file is reached, an event of the type and namespace given by the `Event` and `Namespace` fields respectively must be synthesized and dispatched to the appropriate node as described by the fields above. No event must be dispatched until a blank line has been received or the end of the file reached.

The event must be dispatched as if using the DOM `dispatchEvent()` method. Thus, if the `Event` field was omitted, leaving the name as the empty string, or if the name had invalid characters, then the dispatching of the event fails.

Events fired from event sources do not have user-agent default actions.

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 `event-source` element, which would then bubble up the DOM, and whose `data` attribute would contain the string `YHOO\n-2\n10` (where `\n` again represents a newline).

This could be used as follows:

```
<event-source src="http://stocks.example.com/ticker.php"
              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 and therefore fires four events. The first block has just a comment, and will fire a `message` event with all the fields set to the empty string or null. The second block has two fields with names "load" and "Target" respectively; since there is no "`load`" member on the `MessageEvent` object that field is ignored, leaving the event as a second `message` event with all the fields set to the empty string or null, but this time the event

is targetted at an element with ID "image1". The third block is empty (no lines between two blank lines), and the fourth block has only two comments, so they both yet again fire `message` events with all the fields set to the empty string or null.

```
; test

load
Target: #image1


; if any more events follow this block, they will not be affected
by
; the "Target" and "load" fields above.
```

6.2.5. 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 URI in the `src` attribute of the `event-source` 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 `event-source` 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 `event-source` functionality on a per-page basis.

6.3. Network connections

To enable Web applications to communicate with each other in local area networks, and to maintain bidirectional communications with their originating server, this specification introduces the `Connection` interface.

The `Window` interface provides three constructors for creating `Connection` objects: `TCPConnection()`, for creating a direct (possibly encrypted) link to another node on the Internet using TCP/IP; `LocalBroadcastConnection()`, for creating a connection to any listening peer on a local network (which could be a local TCP/IP subnet using UDP, a Bluetooth PAN, or another kind of network infrastructure); and `PeerToPeerConnection()`, for a direct peer-to-peer connection (which could again be over TCP/IP, Bluetooth, IrDA, or some other type of network).

> *Note: This interface does not allow for raw access to the underlying network. For example, this interface could not be used to implement an IRC client without proxying messages through a custom server.*

6.3.1. Introduction

This section is non-normative.

An introduction to the client-side and server-side of using the direct connection APIs.

An example of a party-line implementation of a broadcast service, and direct peer-to-peer chat for direct local connections.

6.3.2. The `Connection` interface

```
interface Connection {
  readonly attribute DOMString network;
  readonly attribute DOMString peer;
  readonly attribute int readyState;
           attribute EventListener onopen;
           attribute EventListener onread;
           attribute EventListener onclose;
  void send(in DOMString data);
  void disconnect();
};
```

`Connection` objects must also implement the `EventTarget` interface. [DOM3EVENTS]

When a `Connection` object is created, the UA must try to establish a connection, as described in the sections below describing each connection type.

The **network** attribute represents the name of the network connection (the value depends on the kind of connection being established). The **peer** attribute identifies the remote host for direct (non-broadcast) connections.

The `network` attribute must be set as soon as the `Connection` object is created, and keeps the same value for the lifetime of the object. The `peer` attribute must initially be set to the empty string and must be updated once, when the connection is established, after which point it must keep the same value for the lifetime of the object.

The **readyState** attribute represents the state of the connection. When the object is created it must be set to 0. It can have the following values:

0 Connecting

> The connection has not yet been established.

1 Connected

> The connection is established and communication is possible.

2 Closed

> The connection has been closed.

Once a connection is established, the `readyState` attribute's value must be changed to 1, and the `open` event must be fired on the `Connection` object.

When data is received, the `read` event will be fired on the `Connection` object.

When the connection is closed, the `readyState` attribute's value must be changed to 2, and the `close` event must be fired on the `Connection` object.

The **onopen**, **onread**, and **onclose** attributes must, when set, register their new value as an event

listener for their respective events (namely <u>open</u>, <u>read</u>, and <u>close</u>), and unregister their previous value if any.

The **send()** method transmits data using the connection. If the connection is not yet established, it must raise an INVALID_STATE_ERR exception. If the connection *is* established, then the behaviour depends on the connection type, as described below.

The **disconnect()** method must close the connection, if it is open. If the connection is already closed, it must do nothing. Closing the connection causes a <u>close</u> event to be fired and the <u>readyState</u> attribute's value to change, as [described above](#).

6.3.3. Connection Events

All the events described in this section are events in no namespace, which do not bubble, are not cancelable, and have no default action.

The **open** event is fired when the connection is established. UAs must use the normal Event interface when firing this event.

The **close** event is fired when the connection is closed (whether by the author, calling the <u>disconnect()</u> method, or by the server, or by a network error). UAs must use the normal Event interface when firing this event as well.

> ### Note: No information regarding why the connection was closed is passed to the application in this version of this specification.

The **read** event is fired when when data is received for a connection. UAs must use the ConnectionReadEvent interface for this event.

```
interface ConnectionReadEvent : Event {
  readonly attribute DOMString data;
  readonly attribute DOMString source;
  void initConnectionReadEvent(in DOMString typeArg, in boolean
canBubbleArg, in boolean cancelableArg, in DOMString dataArg);
  void initConnectionReadEventNS(in DOMString namespaceURI, in
DOMString typeArg, in boolean canBubbleArg, in boolean cancelableArg,
in DOMString dataArg);
};
```

The **initConnectionReadEvent()** and **initConnectionReadEventNS()** 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 data that was transmitted from the peer.

The **source** attribute represents the name of the peer. This is primarily useful on broadcast connections; on direct connections it is equal to the <u>peer</u> attribute on the Connection object.

Events that would be fired during script execution (e.g. between the connection object being created — and thus the connection being established — and the current script completing; or, during the execution of a <u>read</u> event handler) must be buffered, and those events queued up and each one individually fired after the script has completed.

6.3.4. TCP connections

The **TCPConnection(*subdomain, port, secure*)** constructor on the <u>Window</u> interface returns a new object implementing the <u>Connection</u> interface, set up for a direct connection to a specified host on the page's domain.

When this constructor is invoked, the following steps must be followed.

First, if the domain part of the script's <u>origin</u> is not a host name (e.g. it is an IP address) then the UA must raise a <u>security exception</u>.

Then, if the *subdomain* argument is null or the empty string, the target host is the domain part of the script's <u>origin</u>. Otherwise, the *subdomain* argument is prepended to the domain part of the script's origin with a dot separating the two strings, and that is the target host.

If either:

  * the target host is not a valid host name, or

  * the *port* argument is neither equal to 80, nor equal to 443, nor greater than or equal to 1024 and less than or equal to 65535,

...then the UA must raise a <u>security exception</u>.

Otherwise, the user agent must verify that the <u>the string representing the script's domain in IDNA format</u> can be obtained without errors. If it cannot, then the user agent must raise a <u>security exception</u>.

The user agent may also raise a <u>security exception</u> at this time if, for some reason, permission to create a direct TCP connection to the relevant host is denied. Reasons could include the UA being instructed by the user to not allow direct connections, or the UA establishing (for instance using UPnP) that the network topology will cause connections on the specified port to be directed at the wrong host.

If no exceptions are raised by the previous steps, then a new <u>Connection</u> object must be created, its <u>peer</u> attribute must be set to a string consisting of the name of the target host, a colon (U+003A COLON), and the port number as decimal digits, and its <u>network</u> attribute must be set to the same value as the <u>peer</u> attribute.

This object must then be returned.

The user agent must then begin trying to establish a connection with the target host and specified port. (This typically would begin in the backgound, while the script continues to execute.)

If the *secure* boolean argument is set to true, then the user agent must establish a secure connection with the target host and specified port using TLS or another protocol, negotiated with the server. [RFC2246] If this fails the user agent must act as if it had <u>closed the connection</u>.

Once a secure connection is established, or if the *secure* boolean argument is not set to true, then the user agent must continue to connect to the server using the protocol described in the section entitled <u>clients connecting over TCP</u>. All data on connections made using TLS must be sent as "application data".

Once the connection is established, the UA must act as described in the section entitled <u>sending and receiving data over TCP</u>.

User agents should allow multiple TCP connections to be established per host. In particular, user agents should not apply per-host HTTP connection limits to connections established with the <u>TCPConnection</u> constructor.

6.3.5. Broadcast connections

The **LocalBroadcastConnection()** constructor on the <u>Window</u> interface returns a new object implementing the <u>Connection</u> interface, set up to broadcast on the local network.

When this constructor is invoked, a new <u>Connection</u> object must be created.

The <u>network</u> attribute of the object must be set to <u>the string representing the script's domain in IDNA format</u>. If this string cannot be obtained, then the user agent must raise a <u>security exception</u> exception when the constructor is called.

The <u>peer</u> attribute must be set to the empty string.

The object must then be returned, unless, for some reason, permission to broadcast on the local network is to be denied. In the latter case, a <u>security exception</u> must be raised instead. User agents may deny such permission for any reason, for example a user preference.

If the object is returned (i.e. if no exception is raised), the user agent must the begin broadcasting and listening on the local network, in the background, as described below. The user agent may define "the local network" in any way it considers appropriate and safe; for instance the user agent may ask the user which network (e.g. Bluetooth, IrDA, Ethernet, etc) the user would like to broadcast on before beginning broadcasting.

UAs may broadcast and listen on multiple networks at once. For example, the UA could broadcast on both Bluetooth and Wifi at the same time.

As soon as the object is returned, the connection <u>has been established</u>, which implies that the <u>open</u> event must be fired. Broadcast connections are never closed.

6.3.5.1. Broadcasting over TCP/IP

Should we drop this altogether? Letting people fill the local network with garbage seems unwise.

We need to register a UDP port for this. For now this spec refers to port 18080/udp.

> ***Note: Since this feature requires that the user agent listen to a particular port, some platforms might prevent more than one user agent per IP address from using this feature at any one time.***

On TCP/IP networks, broadcast connections transmit data using UDP over port 18080.

When the <u>send(*data*)</u> method is invoked on a <u>Connection</u> object that was created by the <u>LocalBroadcastConnection()</u> constructor, the user agent must follow these steps:

1. Create a string consisting of the value of the <u>network</u> attribute of the <u>Connection</u> object, a U+0020 SPACE character, a U+0002 START OF TEXT character, and the *data* argument.

2. Encode the string as UTF-8.

3. If the resulting byte stream is longer than 65487 bytes, raise an INDEX_SIZE_ERR DOM exception and stop.

4. Create a UDP packet whose data is the byte stream, with the source and destination ports being 18080, and with appropriate length and checksum fields. Transmit this packet to IPv4

address 255.255.255.255 or IPv6 address ff02::1, as appropriate.   ***IPv6 applications will also have to enable reception from this address.***

When a broadcast connection is opened on a TCP/IP network, the user agent should listen for UDP packets on port 18080.

When the user agent receives a packet on port 18080, the user agent must attempt to decode that packet's data as UTF-8. If the data is not fully correct UTF-8 (i.e. if there are decoding errors) then the packet must be ignored. Otherwise, the user agent must check to see if the decoded string contains a U+0020 SPACE character. If it does not, then the packet must again be ignored (it might be a peer discovery packet from a `PeerToPeerConnection()` constructor). If it does then the user agent must split the string at the first space character. All the characters before the space are then known as *d*, and all the characters after the space are known as *s*. If *s* is not at least one character long, or if the first character of *s* is not a U+0002 START OF TEXT character, then the packet must be ignored. (This allows for future extension of this protocol.)

Otherwise, for each `Connection` object that was created by the `LocalBroadcastConnection()` constructor and whose `network` attribute exactly matches *d*, a `read` event must be fired on the `Connection` object. The string *s*, with the first character removed, must be used as the `data`, and the source IP address of the packet as the `source`.

Making the source IP available means that if two or more machines in a private network can be made to go to a hostile page simultaneously, the hostile page can determine the IP addresses used locally (i.e. on the other side of any NAT router). Is there some way we can keep link-local IP addresses secret while still allowing for applications to distinguish between multiple participants?

6.3.5.2. Broadcasting over Bluetooth

Does anyone know enough about Bluetooth to write this section?

6.3.5.3. Broadcasting over IrDA

Does anyone know enough about IrDA to write this section?

6.3.6. Peer-to-peer connections

The **`PeerToPeerConnection()`** constructor on the `Window` interface returns a new object implementing the `Connection` interface, set up for a direct connection to a user-specified host.

When this constructor is invoked, a new `Connection` object must be created.

The `network` attribute of the object must be set to the string representing the script's domain in IDNA format. If this string cannot be obtained, then the user agent must raise a security exception exception when the constructor is called.

The `peer` attribute must be set to the empty string.

The object must then be returned, unless, for some reason, permission to establish peer-to-peer connections is generally disallowed, for example due to administrator settings. In the latter case, a security exception must be raised instead.

The user agent must then, typically while the script resumes execution, find a remote host to establish a connection to. To do this it must start broadcasting and listening for peer discovery messages and listening for incoming connection requests on all the supported networks. How this is performed depends on the type of network and is described below.

The UA should inform the user of the clients that are detected, and allow the user to select one to connect to. UAs may also allow users to explicit specify hosts that were not detected, e.g. by having the user enter an IP address.

If an incoming connection is detected before the user specifies a target host, the user agent should ask the user to confirm that this is the host they wish to connect to. If it is, the connection should be accepted and the UA will act as the *server* in this connection. (Which UA acts as the server and which acts as the client is not discernible at the DOM API level.)

If no incoming connection is detected and if the user specifies a particular target host, a connection should be established to that host, with the UA acting as the *client* in the connection.

No more than one connection must be established per `Connection` object, so once a connection has been established, the user agent must stop listening for further connections (unless, or until such time as, another `Connection` object is being created).

If at any point the user cancels the connection process or the remote host refuses the connection, then the user agent must act as if it had [closed the connection](), and stop trying to connect.

6.3.6.1. Peer-to-peer connections over TCP/IP

> Should we replace this section with something that uses Rendez-vous/zeroconf or equivalent?

> We need to register ports for this. For now this spec refers to port 18080/udp and 18080/tcp.

> ***Note: Since this feature requires that the user agent listen to a particular port, some platforms might prevent more than one user agent per IP address from using this feature at any one time.***

When using TCP/IP, broadcasting peer discovery messages must be done by creating UDP packets every few seconds containing as their data the value of the connection's `network` attribute, encoded as UTF-8, with the source and destination ports being set to 18080 and appropriate length and checksum fields, and sending these packets to address (in IPv4) 255.255.255.255 or (in IPv6) ff02::1, as appropriate.

Listening for peer discovery messages must be done by examining incoming UDP packets on port 18080.   ***IPv6 applications will also have to enable reception from the ff02::1 address.***   If their payload is exactly byte-for-byte equal to a UTF-8 encoded version of the value of the connection's `network` attribute, then the source address of that packet represents the address of a host that is ready to accept a peer-to-peer connection, and it should therefore be offered to the user.

Incoming connection requests must be listened for on TCP port 18080. If an incoming connection is received, the UA must act as a *server*, as described in the section entitled [servers accepting connections over TCP]().

If no incoming connection requests are accepted and the user instead specifies a target host to connect to, the UA acts as a *client*: the user agent must attempt to connect to the user-specified host on port 18080, as described in the section entitled [clients connecting over TCP]().

Once the connection is established, the UA must act as described in the section entitled sending and receiving data over TCP.

Note: This specification does not include a way to establish secure (encrypted) peer-to-peer connections at this time. **If you can see a good way to do this, let me know.**

6.3.6.2. Peer-to-peer connections over Bluetooth

Does anyone know enough about Bluetooth to write this section?

6.3.6.3. Peer-to-peer connections over IrDA

Does anyone know enough about IrDA to write this section?

6.3.7. The common protocol for TCP-based connections

The same protocol is used for `TCPConnection` and `PeerToPeerConnection` connection types. This section describes how such connections are established from the client and server sides, and then describes how data is sent and received over such connections (which is the same for both clients and servers).

6.3.7.1. Clients connecting over TCP

This section defines the client-side requirements of the protocol used by the `TCPConnection` and `PeerToPeerConnection` connection types.

If a TCP connection to the specified target host and port cannot be established, for example because the target host is a domain name that cannot be resolved to an IP address, or because packets cannot be routed to the host, the user agent should retry creating the connection. If the user agent gives up trying to connect, the user agent must act as if it had closed the connection.

Note: No information regarding the state of the connection is passed to the application while the connection is being established in this version of this specification.

Once a TCP/IP connection to the remote host is established, the user agent must transmit the following sequence of bytes, represented here in hexadecimal form:

```
0x48 0x65 0x6C 0x6C 0x6F 0x0A
```

Note: This represents the string "Hello" followed by a newline, encoded in UTF-8.

The user agent must then read all the bytes sent from the remote host, up to the first 0x0A byte (inclusive). That string of bytes is then compared byte-for-byte to the following string of bytes:

```
0x57 0x65 0x6C 0x63 0x6F 0x6E 0x65 0x0A
```

Note: This says "Welcome".

If the server sent back a string in any way different to this, then the user agent must close the connection and give up trying to connect.

Otherwise, the user agent must then take the string representing the script's domain in IDNA format, encode it as UTF-8, and send that to the remote host, followed by a 0x0A byte (a U+000A LINE FEED in UTF-8).

The user agent must then read all the bytes sent from the remote host, up to the first 0x0A byte (inclusive). That string of bytes must then be compared byte-for-byte to the string that was just sent to the server (the one with the IDNA domain name and ending with a newline character). If the server sent back a string in any way different to this, then the user agent must close the connection and give up trying to connect.

Otherwise, the connection has been established (and events and so forth get fired, as described above).

If at any point during this process the connection is closed prematurely, then the user agent must close the connection and give up trying to connect.

6.3.7.2. Servers accepting connections over TCP

This section defines the server side of the protocol described in the previous section. For authors, it should be used as a guide for how to implement servers that can communicate with Web pages over TCP. For UAs these are the requirements for the server part of `PeerToPeerConnection`s.

Once a TCP/IP connection from a remote host is established, the user agent must transmit the following sequence of bytes, represented here in hexadecimal form:

```
0x57 0x65 0x6C 0x63 0x6F 0x6E 0x65 0x0A
```

> **Note: This says "Welcome" and a newline in UTF-8.**

The user agent must then read all the bytes sent from the remote host, up to the first 0x0A byte (inclusive). That string of bytes is then compared byte-for-byte to the following string of bytes:

```
0x48 0x65 0x6C 0x6C 0x6F 0x0A
```

> **Note: "Hello" and a newline.**

If the remote host sent back a string in any way different to this, then the user agent must close the connection and give up trying to connect.

Otherwise, the user agent must then take the string representing the script's domain in IDNA format, encode it as UTF-8, and send that to the remote host, followed by a 0x0A byte (a U+000A LINE FEED in UTF-8).

The user agent must then read all the bytes sent from the remote host, up to the first 0x0A byte (inclusive). That string of bytes must then be compared byte-for-byte to the string that was just sent to that host (the one with the IDNA domain name and ending with a newline character). If the remote host sent back a string in any way different to this, then the user agent must close the connection and give up trying to connect.

Otherwise, the connection has been established (and events and so forth get fired, as described above).

Note: For author-written servers (as opposed to the server side of a peer-to-peer connection), the script's domain would be replaced by the hostname of the server. Alternatively, such servers might instead wait for the client to send its domain string, and then simply echo it back. This would allow connections from pages on any domain, instead of just pages originating from the same host. The client compares the two strings to ensure they are the same before allowing the connection to be used by author script.

If at any point during this process the connection is closed prematurely, then the user agent must close the connection and give up trying to connect.

6.3.7.3. Sending and receiving data over TCP

When the `send(data)` method is invoked on the connection's corresponding `Connection` object, the user agent must take the *data* argument, replace any U+0000 NULL and U+0017 END OF TRANSMISSION BLOCK characters in it with U+FFFD REPLACEMENT CHARACTER characters, then transmit a U+0002 START OF TEXT character, this new *data* string and a single U+0017 END OF TRANSMISSION BLOCK character (in that order) to the remote host, all encoded as UTF-8.

When the user agent receives bytes on the connection, the user agent must buffer received bytes until it receives a 0x17 byte (a U+0017 END OF TRANSMISSION BLOCK character). If the first buffered byte is not a 0x02 byte (a U+0002 START OF TEXT character encoded as UTF-8) then all the data up to the 0x17 byte, inclusive, must be dropped. (This allows for future extension of this protocol.) Otherwise, all the data from (but not including) the 0x02 byte and up to (but not including) the 0x17 byte must be taken, interpreted as a UTF-8 string, and a `read` event must be fired on the `Connection` object with that string as the `data`. If that string cannot be decoded as UTF-8 without errors, the packet should be ignored.

Note: This protocol does not yet allow binary data (e.g. an image or media data) to be efficiently transmitted. A future version of this protocol might allow this by using the prefix character U+001F INFORMATION SEPARATOR ONE, followed by binary data which uses a particular byte (e.g. 0xFF) to encode byte 0x17 somehow (since otherwise 0x17 would be treated as transmission end by down-level UAs).

6.3.8. Security

Need to write this section.

If you have an unencrypted page that is (through a man-in-the-middle attack) changed, it can access a secure service that is using IP authentication and then send that data back to the attacker. Ergo we should probably stop unencrypted pages from accessing encrypted services, on the principle that the actual level of security is zero. Then again, if we do that, we prevent insecure sites from using SSL as a tunneling mechanism.

Should consider dropping the subdomain-only restriction. It doesn't seem to add anything, and prevents cross-domain chatter.

6.3.9. Relationship to other standards

Should have a section talking about the fact that we blithely ignoring IANA's port assignments

here.

<div style="border: 2px solid red; color: red;">
Should explain why we are not reusing HTTP for this. (HTTP is too heavy-weight for such a simple need; requiring authors to implement an HTTP server just to have a party line is too much of a barrier to entry; cannot rely on prebuilt components; having a simple protocol makes it much easier to do RAD; HTTP doesn't fit the needs and doesn't have the security model needed; etc)
</div>

6.4. 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.

6.4.1. Processing model

When a script invokes the **`postMessage(message)`** method on a `Window` object, the user agent must create an event that uses the `MessageEvent` interface, with the event name `message`, which bubbles, is cancelable, and has no default action. The `data` attribute must be set to the value passed as the *message* argument to the `postMessage()` method, the `domain` attribute must be set to the [domain of the document](#) that the script that invoked the methods is associated with, the `uri` attribute must be set to the URI of that document, and the `source` attribute must be set to the `Window` object of the default view of the browsing context with which that document is associated.

The event must then be dispatched at the `Document` object that is the [active document](#) of the `Window` object on which the method was invoked.

The `postMessage()` method must only return once the event dispatch has been completely processed by the target document (i.e. all three of the capture, target, and bubble phases have been done, and event listeners have been executed as appropriate).

⚠**Warning!** *Authors should check the `domain` attribute to ensure that messages are only accepted from domains that they expect to receive messages from. Otherwise, bugs in the author's message handling code could be exploited by hostile sites.*

For example, if document A contains an `object` element that contains document B, and script in document A calls `postMessage()` on document B, then a message event will be fired on that element, marked as originating from document A. The script in document A might look like:

```
var o = document.getElementsByTagName('object')[0];
o.contentWindow.postMessage('Hello world');
```

To register an event handler for incoming events, the script would use `addEventListener()` (or similar mechanisms). For example, the script in document B might look like:

```
document.addEventListener('message', receiver, false);
function receiver(e) {
  if (e.domain == 'example.com') {
    if (e.data == 'Hello world') {
```

```
              e.source.postMessage('Hello');
          } else {
            alert(e.data);
          }
        }
      }
```

This script first checks the domain is the expected domain, and then looks at the message, which it either displays to the user, or responds to by sending a message back to the document which sent the message in the first place.

⚠**Warning!** *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.*

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. Repetition templates

See WF2 for now

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 serialised, as discussed in the section on that topic.

The U+0000 NULL character must not appear anywhere in a document.

Note: Space characters before the root `html` 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 `html` element. Thus, space characters around the root element do not round-trip. It is suggested that newlines be inserted after the DOCTYPE and any comments that aren't in the root 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 four different kinds of **elements**: void elements, CDATA elements, RCDATA elements, and normal elements.

Void elements

> base, link, meta, hr, br, img, embed, param, area, col, input

CDATA elements

> style, script

RCDATA elements

> title, textarea

Normal elements

> All other allowed HTML elements are normal elements.

Tags are used to delimit the start and end of elements in the markup. CDATA, RCDATA, and normal

elements have a start tag to indicate where they begin, and an end tag to indicate where they end. The start and end tags of certain normal elements can be omitted, as described later. Those that cannot be omitted must not be omitted. Void elements only have a start tag; end tags must not be specified for void elements.

The contents of the element must be placed between just after the start tag (which might be implied, in certain cases) and just before the end tag (which again, might be implied in certain cases). The exact allowed contents of each individual element depends on the content model of that element, as described earlier in this specification. Elements must not contain content that their content model disallows. In addition to the restrictions placed on the contents by those content models, however, the four types of elements have additional *syntactic* requirements.

Void elements can't have any contents (since there's no end tag, no content can be put between the start tag and the end tag.)

CDATA elements can have text, though it has restrictions described below.

RCDATA elements can have text and character entity references, but the text must not contain an ambiguous ampersand. There are also further restrictions described below.

Normal elements can have text, character entity references, other elements, and comments, but the text must not contain the character U+003C LESS-THAN SIGN (<) or 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+0061 LATIN SMALL LETTER A .. U+007A LATIN SMALL LETTER Z, or, in uppercase, U+0041 LATIN CAPITAL LETTER A .. U+005A LATIN CAPITAL LETTER Z, and U+002D HYPHEN-MINUS (-). In the HTML syntax, tag names may be written with any mix of lower- and uppercase letters that, when converted to all-lowercase, matches the element's tag name; tag names are case-insensitive.

8.1.2.1. Start tags

Start tags must have the following format:

1. The first character of a start tag must be a U+003C LESS-THAN SIGN (<).

2. The next few characters of a start tag must be the element's tag name.

3. If there are to be any attributes in the next step, there must first be one or more space characters.

4. Then, the start tag may have a number of attributes, the syntax for which is described below. Attributes may be separated from each other by one or more space characters.

5. After the attributes, there may be one or more space characters. (Some attributes are required to be followed by a space. See the attributes section below.)

6. Then, if the element is one of the void elements, then there may be a single U+002F SOLIDUS (/) character. This character has no effect except to appease the markup gods. As this character is therefore just a symbol of faith, atheists should omit it.

7. Finally, start tags must be closed by a U+003E GREATER-THAN SIGN (>) character.

8.1.2.2. End tags

End tags must have the following format:

1. The first character of an end tag must be a U+003C LESS-THAN SIGN (<).

2. The second character of an end tag must be a U+002F SOLIDUS (/).

3. The next few characters of an end tag must be the element's tag name.

4. After the tag name, there may be one or more space characters.

5. Finally, end tags must be closed by a U+003E GREATER-THAN SIGN (>) character.

8.1.2.3. Attributes

Attributes for an element are expressed inside the element's start tag.

Attributes have a name and a value. **Attribute names** must consist of one character other than the space characters, U+003E GREATER-THAN SIGN (>), and U+002F SOLIDUS (/), followed by zero or more characters other than the space characters, U+003E GREATER-THAN SIGN (>), U+002F SOLIDUS (/), and U+003D EQUALS SIGN (=). In the HTML syntax, attribute names may be written with any mix of lower- and uppercase letters that, when converted to all-lowercase, matches the attribute's name; attribute names are case-insensitive.

Attribute values are a mixture of text and character entity references, except with the additional restriction that the text cannot contain 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 or U+003E GREATER-THAN SIGN (>) characters, and must not, furthermore, start with either a literal U+0022 QUOTATION MARK (") character or a literal U+0027 APOSTROPHE (') character.

In the following example, the `value` attribute is given with the unquoted attribute value syntax:

```
<input value=yes>
```

If an attribute using the unquoted attribute syntax is to be followed by another attribute or by one of the optional U+002F SOLIDUS (/) characters allowed in step 6 of the start tag syntax above, then there must be a space character separating the two.

Single-quoted attribute value syntax

The <u>attribute name</u>, followed by zero or more <u>space characters</u>, followed by a single U+003D EQUALS SIGN character, followed by zero or more <u>space characters</u>, followed by a single U+0027 APOSTROPHE (`'`) character, followed by the <u>attribute value</u>, which, in addition to the requirements given above for attribute values, must not contain any literal U+0027 APOSTROPHE (`'`) characters, and finally followed by a second single U+0027 APOSTROPHE (`'`) character.

> In the following example, the `type` attribute is given with the single-quoted attribute value syntax:
>
> ```
> <input type='checkbox'>
> ```

Double-quoted attribute value syntax

The <u>attribute name</u>, followed by zero or more <u>space characters</u>, followed by a single U+003D EQUALS SIGN character, followed by zero or more <u>space characters</u>, followed by a single U+0022 QUOTATION MARK (`"`) character, followed by the <u>attribute value</u>, which, in addition to the requirements given above for attribute values, must not contain any literal U+0022 QUOTATION MARK (`"`) characters, and finally followed by a second single U+0022 QUOTATION MARK (`"`) character.

> In the following example, the `name` attribute is given with the double-quoted attribute value syntax:
>
> ```
> <input name="be evil">
> ```

8.1.2.4. Optional tags

Certain tags can be **omitted**.

An `html` element's start tag may be omitted if the first thing inside the `html` element is not a <u>space character</u> or a <u>comment</u>.

An `html` element's end tag may be omitted if the `html` element is not immediately followed by a <u>space character</u> or a <u>comment</u>.

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 <u>space character</u> or a <u>comment</u>.

A `body` element's start tag may be omitted if the first thing inside the `body` element is not a <u>space character</u> or a <u>comment</u>, 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 <u>space character</u> or a <u>comment</u>.

A `li` element's end tag may be omitted if the `li` element is immediately followed by another `li` element or if there is no more content in the parent element.

A `dt` element's end tag may be omitted if the `dt` element is immediately followed by another `dt` element or a `dd` element.

A `dd` element's end tag may be omitted if the `dd` element is immediately followed by another `dd`

element or a `dt` element, or if there is no more content in the parent element.

A `p` element's end tag may be omitted if the `p` element is immediately followed by an `address`, `blockquote`, `dl`, `fieldset`, `form`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `hr`, `menu`, `ol`, `p`, `pre`, `table`, or `ul` element, or if there is no more content in the parent element.

An `optgroup` element's end tag may be omitted if the `optgroup` element is immediately followed by another `optgroup` element, or if there is no more content in the parent element.

An `option` element's end tag may be omitted if the `option` element is immediately followed by another `option` element, or if there is no more content in the parent element.

A `colgroup` element's start tag may be omitted if the first thing inside the `colgroup` element is a `col` element, and if the element is not immediately preceded by another `colgroup` element whose end tag has been omitted.

A `colgroup` element's end tag may be omitted if the `colgroup` element is not immediately followed by a [space character](#) or a [comment](#).

A `thead` element's end tag may be omitted if the `thead` element is immediately followed by a `tbody` or `tfoot` element.

A `tbody` element's start tag may be omitted if the first thing inside the `tbody` element is a `tr` element, and if the element is not immediately preceded by a `tbody`, `thead`, or `tfoot` element whose end tag has been omitted.

A `tbody` element's end tag may be omitted if the `tbody` element is immediately followed by a `tbody` or `tfoot` element, or if there is no more content in the parent element.

A `tfoot` element's end tag may be omitted if the `tfoot` element is immediately followed by a `tbody` element, or if there is no more content in the parent element.

A `tr` element's end tag may be omitted if the `tr` element is immediately followed by another `tr` element, or if there is no more content in the parent element.

A `td` element's end tag may be omitted if the `td` element is immediately followed by a `td` or `th` element, or if there is no more content in the parent element.

A `th` element's end tag may be omitted if the `th` element is immediately followed by a `td` or `th` element, or if there is no more content in the parent element.

However, a start tag must never be omitted if it has any attributes.

8.1.2.5. Restrictions on content models

For historical reasons, certain elements have extra restrictions beyond even the restrictions given by their content model.

A `p` element must not contain `blockquote`, `dl`, `menu`, `ol`, `pre`, `table`, or `ul` elements, even though these elements are technically allowed inside `p` elements according to the content models described in this specification. (In fact, if one of those elements is put inside a `p` element in the markup, it will instead imply a `p` element end tag before it.)

An `optgroup` element must not contain `optgroup` elements, even though these elements are technically allowed to be nested according to the content models described in this specification. (If an `optgroup` element is put inside another in the markup, it will in fact imply an `optgroup` end tag

before it.)

A `table` element must not contain `tr` elements, even though these elements are technically allowed inside `table` elements according to the content models described in this specification. (If a `tr` element is put inside a `table` in the markup, it will in fact imply a `tbody` start tag before it.)

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 occurences 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+000B LINE TABULATION, 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 (in CDATA and RCDATA elements) and character entity references (in RCDATA elements) 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.

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 valid Unicode characters other than U+0000. Text should not contain control characters other than space characters. Extra constraints are placed on what is and what is not allowed in text based on where the text is to be put, as described in the other sections.

8.1.3.1. Newlines

Newlines in HTML may be represented either as U+000D CARRIAGE RETURN (CR) characters,

U+000A LINE FEED (LF) characters, or pairs of U+000D CARRIAGE RETURN (CR), U+000A LINE FEED (LF) characters in that order.

8.1.4. Character entity references

In certain cases described in other sections, text may be mixed with **character entity references**. These can be used to escape characters that couldn't otherwise legally be included in text.

Character entity references must start with a U+0026 AMPERSAND (&). Following this, there are three possible kinds of character entity references:

Named entities

> The ampersand must be followed by one of the names given in the entities section, using the same case. The name must be one that is terminated by a U+003B SEMICOLON (;) character.

Decimal numeric entities

> The ampersand must be followed by a U+0023 NUMBER SIGN (#) character, followed by one or more digits in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, representing a base-ten integer that itself is a valid Unicode code point that 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 entities

> The ampersand must be followed by a U+0023 NUMBER SIGN (#) character, which must be followed by either a U+0078 LATIN SMALL LETTER X or a U+0058 LATIN CAPITAL LETTER X character, which must then be followed by one or more digits in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F, and U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F, representing a base-sixteen integer that itself is a valid Unicode code point that 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 not the last character in the file, that is not followed by a space character, that is not followed by a start tag that has not been omitted, and that is not followed by another U+0026 AMPERSAND (&) character.

8.1.5. Comments

Comments must start with the four character sequence U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS (<!--). Following this sequence, the comment may have text, with the additional restriction that the text must not contain two consecutive U+002D HYPHEN-MINUS (-) characters, nor end with a U+002D HYPHEN-MINUS (-) character. Finally, the comment must be ended by the three character sequence U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN (-->).

8.2. Parsing HTML documents

This section only applies to user agents, data mining tools, and conformance checkers.

The rules for parsing XML documents (and thus XHTML documents) into DOM trees are covered by the XML and Namespaces in XML specifications, and are out of scope of this specification. [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 serialisation of HTML5.

This specification defines the parsing rules for HTML documents, whether they are syntactically valid or not. Certain points in the parsing algorithm are said to be **parse errors**. The error handling for parse errors is well-defined: user agents must either act as described below when encountering such problems, or must abort processing at the first error that they encounter for which they do not wish to apply the rules described below.

Conformance checkers must report at least one parse error condition to the user if one or more parse error conditions exist in the document and must not report parse error conditions if none exist in the document. Conformance checkers may report more than one parse error condition if more than one parse error conditions exist in the document. Conformance checkers are not required to recover from parse errors.

Note: Parse errors are only errors with the syntax of HTML. In addition to checking for parse errors, conformance checkers will also verify that the document obeys all the other conformance requirements described in this specification.

8.2.1. Overview of the parsing model

The input to the HTML parsing process consists of a stream of Unicode characters, which is passed through a tokenisation stage (lexical analysis) followed by a tree construction stage (semantic analysis). The output is a `Document` object.

Note: Implementations that do not support scripting do not have to actually create a DOM `Document` object, but the DOM tree in such cases is still used as the model for the rest of the specification.

In the common case, the data handled by the tokenisation stage comes from the network, but it can also come from script, e.g. using the `document.write()` API.

There is only one set of state for the tokeniser stage and the tree construction stage, but the tree construction stage is reentrant, meaning that while the tree construction stage is handling one token, the tokeniser might be resumed, causing further tokens to be emitted and processed before the first

token's processing is complete.

> In the following example, the tree construction stage will be called upon to handle a "p" start tag
> token while handling the "script" start tag token:
>
> ```
> ...
> <script>
>  document.write('<p>');
> </script>
>  ...
> ```

8.2.2. The input stream

The stream of Unicode characters that consists the input to the tokenisation stage will be initially
seen by the user agent as a stream of bytes (typically coming over the network or from the local file
system). The bytes encode the actual characters according to a particular *character encoding*, which
the user agent must use to decode the bytes into characters.

> **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	Description
FE FF	UTF-16BE BOM
FF FE	UTF-16LE BOM
EF BB BF	UTF-8 BOM

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, 0x0B, 0x0C, 0x0D, 0x20 (case-insensitive ASCII '<meta' followed by a space)**

   > 1. Advance the *position* pointer so that it points at the next 0x09, 0x0A, 0x0B, 0x0C, 0x0D, or 0x20 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. Examine the attribute's name:
   >
   >   ↪ **If it is 'charset'**
   >
   >   > If the attribute's value is a supported character encoding, then return the given encoding, with confidence *tentative*, and abort all these steps. Otherwise, do nothing with this attribute, and continue looking for other attributes.
   >
   >   ↪ **If it is 'content'**
   >
   >   > The attribute's value is now parsed.
   >
   >   > 1. Apply the algorithm for extracting an encoding from a Content-Type, giving the attribute's value as the string to parse.
   >   >
   >   > 2. If an encoding was returned, and it is the name of

a supported character encoding, then return that encoding, with the [confidence](#) *tentative*, and abort all these steps.

3. Otherwise, skip this 'content' attribute and continue on with any other attributes.

    ↳ **Any other name**

Do nothing with that attribute.

4. Return to step 1 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), 0x0B (ASCII VT), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), 0x3E (ASCII '>'), 0x3C (ASCII '<') byte.

2. If the pointer points to a 0x3C (ASCII '<') byte, then return to the first step in the overall "two step" algorithm.

3. 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), 0x0B (ASCII VT), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), or 0x2F (ASCII '/') then advance *position* to the next byte and start over.

2. If the byte at *position* is 0x3C (ASCII '<'), then move *position* back to the previous byte, and stop looking for an attribute. There isn't one.

3. If the byte at *position* is 0x3E (ASCII '>'), then stop looking for an attribute. There isn't one.

4. Otherwise, the byte at *position* is the start of the attribute name. Let *attribute name* and *attribute value* be the empty string.

5. *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 labelled *value*.

↳ **If it is 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0B (ASCII VT), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space)**

Jump to the step below labelled *spaces*.

↳ **If it is 0x2F (ASCII '/'), 0x3C (ASCII '<'), or 0x3E (ASCII '>')**

Stop looking for an attribute. 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.)

6. Advance *position* to the next byte and return to the previous step.

7. *Spaces.* If the byte at *position* is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0B (ASCII VT), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space) then advance *position* to the next byte, then, repeat this step.

8. If the byte at *position* is *not* 0x3D (ASCII '='), stop looking for an attribute. Move *position* back to the previous byte. The attribute's name is the value of *attribute name*, its value is the empty string.

9. Advance *position* past the 0x3D (ASCII '=') byte.

10. *Value.* If the byte at *position* is one of 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0B (ASCII VT), 0x0C (ASCII FF), 0x0D (ASCII CR), or 0x20 (ASCII space) then advance *position* to the next byte, then, repeat this step.

11. 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 stop looking for an attribute. 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 0x3C (ASCII '<'), or 0x3E (ASCII '>')**

Stop looking for an attribute. 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*).

↳ **Anything else**

Append the Unicode character with the same codepoint as the value of the byte at *position*) to *attribute value*.

12. Process the byte at *position* as follows:

↳ **If it is 0x09 (ASCII TAB), 0x0A (ASCII LF), 0x0B (ASCII VT), 0x0C (ASCII FF), 0x0D (ASCII CR), 0x20 (ASCII space), 0x3C (ASCII '<'), or 0x3E (ASCII '>')**

Stop looking for an attribute. 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*.

13. 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*. Due to its use in legacy content, `windows-1252` is recommended as a default in predominantly Western demographics. In non-legacy environments, the more comprehensive `UTF-8` encoding is recommended instead. Since these encodings can in many cases be distinguished by inspection, a user agent may heuristically decide which to use as a default.

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 a user agent would otherwise use the ISO-8859-1 encoding, it must instead use the Windows-1252 encoding.

Note: This requirement 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.

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 leading U+FEFF BYTE ORDER MARK characters must not be stripped by the encoding layer.

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.

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.

U+000D CARRIAGE RETURN (CR) characters, and U+000A LINE FEED (LF) characters, are treated specially. Any CR characters that are followed by LF characters must be removed, and any CR characters not followed by LF characters must be converted to LF characters. Thus, newlines in HTML DOMs are represented by LF characters, and there are never any CR characters in the input to the tokenisation stage.

The **next input character** is the first character in the input stream that has not yet been **consumed**. Initially, the *next input character* is the first character in the input.

The **insertion point** is the position (just before a character or just before the end of the input stream) where content inserted using `document.write()` is actually inserted. The insertion point is relative to the position of the character immediately after it, it is not an absolute offset into the input stream. Initially, the insertion point is uninitialised.

The "EOF" character in the tables below is a conceptual character representing the end of the input stream. If the parser is a script-created parser, then the end of the input stream is reached when an **explicit "EOF" character** (inserted by the `document.close()` method) is consumed. Otherwise, the "EOF" 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 UTF-16, 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 encoding to the new encoding, set the confidence to *confident*, and abort these steps.

4. Otherwise, navigate to the document again, with replacement enabled, 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.

Note: While the invocation of this algorithm is not a parse error, it is still indicative of non-conforming content.

8.2.3. Tokenisation

Implementations must act as if they used the following state machine to tokenise HTML. The state machine must start in the data state. Most states consume a single character, which may have various side-effects, and either switches the state machine to a new state to *reconsume* the same character, or switches it to a new state (to consume the next character), or repeats the same state (to consume the next character). Some states have more complicated behaviour and can consume several characters before switching to another state.

The exact behaviour of certain states depends on a **content model flag** that is set after certain tokens are emitted. The flag has several states: *PCDATA*, *RCDATA*, *CDATA*, and *PLAINTEXT*. Initially it must be in the PCDATA state. In the RCDATA and CDATA states, a further **escape flag** is used to control the behaviour of the tokeniser. It is either true or false, and initially must be set to the false state.

The output of the tokenisation step is a series of zero or more of the following tokens: DOCTYPE, start tag, end tag, comment, character, end-of-file. DOCTYPE tokens have a name, a public identifier, a system identifier, and a correctness flag. When a DOCTYPE token is created, its name, public identifier, and system identifier must be marked as missing, and the correctness flag must be set to *correct* (its other state is *incorrect*). Start and end tag tokens have a tag name and a list of attributes, each of which has a name and a value. Comment and character tokens have data.

When a token is emitted, it must immediately be handled by the tree construction stage. The tree construction stage can affect the state of the content model flag, and can insert additional characters into the stream. (For example, the `script` element can result in scripts executing and using the dynamic markup insertion APIs to insert characters into the stream being tokenised.)

When an end tag token is emitted, the content model flag must be switched to the PCDATA state.

When an end tag token is emitted with attributes, that is a [parse error](#).

A **permitted slash** is a U+002F SOLIDUS character that is immediately followed by a U+003E GREATER-THAN SIGN, if, and only if, the current token being processed is a start tag token whose tag name is one of the following: `base`, `link`, `meta`, `hr`, `br`, `img`, `embed`, `param`, `area`, `col`, `input`

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 is as follows:

Data state

Consume the [next input character](#):

    ↳ **U+0026 AMPERSAND (&)**

When the [content model flag](#) is set to one of the PCDATA or RCDATA states: switch to the [entity data state](#).
Otherwise: treat it as per the "anything else" entry below.

    ↳ **U+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](#).

Entity data state

(This cannot happen if the [content model flag](#) is set to the CDATA state.)

Attempt to [consume an entity](#).

If nothing is returned, emit a U+0026 AMPERSAND character token.

Otherwise, emit the character token that was returned.

Finally, switch to the data state.

Tag open state

The behaviour of this state depends on the content model flag.

If the content model flag is set to the RCDATA or CDATA states

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.

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+000B LINE TABULATION
- 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.

Tag name state

> Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

> Switch to the before attribute name state.

↪ **U+003E GREATER-THAN SIGN (>)**

> Emit the current tag token. Switch to the data state.

↪ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

> Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the current tag token's tag name. Stay in the tag name state.

↪ **EOF**

> Parse error. Emit the current tag token. Reconsume the EOF character in the data state.

↪ **U+002F SOLIDUS (/)**

> Parse error unless this is a permitted slash. Switch to the before attribute name state.

↪ **Anything else**

> Append the current input character to the current tag token's tag name. Stay in the tag name state.

Before attribute name state

> Consume the next input character:

↳ **U+0009 CHARACTER TABULATION**

↳ **U+000A LINE FEED (LF)**

↳ **U+000B LINE TABULATION**

↳ **U+000C FORM FEED (FF)**

↳ **U+0020 SPACE**

> Stay in the before attribute name state.

↳ **U+003E GREATER-THAN SIGN (>)**

> Emit the current tag token. Switch to the data state.

↳ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

> Start a new attribute in the current tag token. Set that attribute's name to the lowercase version of the current input character (add 0x0020 to the character's code point), and its value to the empty string. Switch to the attribute name state.

↳ **U+002F SOLIDUS (/)**

> Parse error unless this is a permitted slash. Stay in the before attribute name state.

↳ **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.

Attribute name state

> Consume the next input character:

↳ **U+0009 CHARACTER TABULATION**

↳ **U+000A LINE FEED (LF)**

↳ **U+000B LINE TABULATION**

↳ **U+000C FORM FEED (FF)**

↳ **U+0020 SPACE**

> Switch to the after attribute name state.

↳ **U+003D EQUALS SIGN (=)**

> Switch to the before attribute value state.

↳ **U+003E GREATER-THAN SIGN (>)**

> Emit the current tag token. Switch to the data state.

↳ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

> Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the current attribute's name. Stay in the attribute name state.

↳ **U+002F SOLIDUS (/)**

> Parse error unless this is a permitted slash. Switch to the before attribute name state.

↳ **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).

After attribute name state

Consume the [next input character](#):

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the [after attribute name state](#).

↪ **U+003D EQUALS SIGN (=)**

Switch to the [before attribute value state](#).

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the [data state](#).

↪ **U+0041 LATIN CAPITAL LETTER A through to U+005A LATIN CAPITAL LETTER Z**

Start a new attribute in the current tag token. Set that attribute's name to the lowercase version of the current input character (add 0x0020 to the character's code point), and its value to the empty string. Switch to the [attribute name state](#).

↪ **U+002F SOLIDUS (/)**

[Parse error](#) unless this is a [permitted slash](#). Switch to the [before attribute name state](#).

↪ **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](#).

Before attribute value state

Consume the [next input character](#):

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the [before attribute value state](#).

↪ **U+0022 QUOTATION MARK (")**

Switch to the [attribute value (double-quoted) state](#).

↪ **U+0026 AMPERSAND (&)**

Switch to the [attribute value (unquoted) state](#) and reconsume this input character.

↪ **U+0027 APOSTROPHE (')**

Switch to the [attribute value (single-quoted) state](#).

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the [data state](#).

↪ **EOF**

[Parse error](#). Emit the current tag token. Reconsume the character in the [data state](#).

↪ **Anything else**

Append the current input character to the current attribute's value. Switch to the [attribute value (unquoted) state](#).

Attribute value (double-quoted) state

Consume the [next input character](#):

↪ **U+0022 QUOTATION MARK (")**

Switch to the [before attribute name state](#).

↪ **U+0026 AMPERSAND (&)**

Switch to the [entity in attribute value state](#).

↪ **EOF**

[Parse error](#). Emit the current tag token. Reconsume the character in the [data state](#).

↪ **Anything else**

Append the current input character to the current attribute's value. Stay in the [attribute value (double-quoted) state](#).

Attribute value (single-quoted) state

Consume the [next input character](#):

↪ **U+0027 APOSTROPHE (')**

Switch to the [before attribute name state](#).

↪ **U+0026 AMPERSAND (&)**

Switch to the [entity in attribute value state](#).

↪ **EOF**

[Parse error](#). Emit the current tag token. Reconsume the character in the [data state](#).

↪ **Anything else**

Append the current input character to the current attribute's value. Stay in the [attribute value (single-quoted) state](#).

Attribute value (unquoted) state

Consume the [next input character](#):

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Switch to the [before attribute name state](#).

↪ **U+0026 AMPERSAND (&)**

Switch to the [entity in attribute value state](#).

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the [data state](#).

↪ **EOF**

[Parse error](#). Emit the current tag token. Reconsume the character in the [data state](#).

> ↳ **Anything else**
>> Append the current input character to the current attribute's value. Stay in the attribute value (unquoted) state.

Entity in attribute value state

Attempt to consume an entity.

If nothing is returned, append a U+0026 AMPERSAND character to the current attribute's value.

Otherwise, append the returned character token to the current attribute's value.

Finally, switch back to the attribute value state that you were in when were switched into this state.

Bogus comment state

(This can only happen if the content model flag is set to the PCDATA state.)

Consume every character up to the first U+003E GREATER-THAN SIGN character (>) or the end of the file (EOF), whichever comes first. Emit a comment token whose data is the concatenation of all the characters starting from and including the character that caused the state machine to switch into the bogus comment state, up to and including the last consumed character before the U+003E character, if any, or up to the end of the file otherwise. (If the comment was started by the end of the file (EOF), the token is empty.)

Switch to the data state.

If the end of the file was reached, reconsume the EOF character.

Markup declaration open state

(This can only happen if the content model flag is set to the PCDATA state.)

If the next two characters are both U+002D HYPHEN-MINUS (-) characters, consume those two characters, create a comment token whose data is the empty string, and switch to the comment 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, is is a parse error. Switch to the bogus comment state. The next character that is consumed, if any, is the first character that will be in the comment.

Comment 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.

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.

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.

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.

Comment end state

Consume the next input character:

↳ **U+003E GREATER-THAN SIGN (>)**

Emit the comment token. Switch to the data state.

↳ **U+002D HYPHEN-MINUS (-)**

Parse error. Append a U+002D HYPHEN-MINUS (-) character to the comment token's data. Stay in the comment end state.

↳ **EOF**

Parse error. Emit the comment token. Reconsume the EOF character in the data state.

↳ **Anything else**

Parse error. Append two U+002D HYPHEN-MINUS (-) characters and the input character to the comment token's data. Switch to the comment state.

DOCTYPE state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Switch to the before DOCTYPE name state.

↪ **Anything else**

Parse error. Reconsume the current character in the before DOCTYPE name state.

Before DOCTYPE name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the before DOCTYPE name state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Create a new DOCTYPE token. Set its correctness flag to *incorrect*. Emit the token. Switch to the data state.

↪ **EOF**

Parse error. Create a new DOCTYPE token. Set its correctness flag to *incorrect*. Emit the token. Reconsume the EOF character in the data state.

↪ **Anything else**

Create a new DOCTYPE token. Set the token's name name to the current input character. Switch to the DOCTYPE name state.

DOCTYPE name state

First, consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Switch to the after DOCTYPE name state.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current DOCTYPE token. Switch to the data state.

↪ **EOF**

Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. 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.

After DOCTYPE name state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

> Stay in the after DOCTYPE name state.

↪ **U+003E GREATER-THAN SIGN (>)**

> Emit the current DOCTYPE token. Switch to the data state.

↪ **EOF**

> Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. 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. Switch to the bogus DOCTYPE state.

Before DOCTYPE public identifier state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

> Stay in the before DOCTYPE public identifier state.

↪ **U+0022 QUOTATION MARK (")**

> Set the DOCTYPE token's public identifier to the empty string, then switch to the DOCTYPE public identifier (double-quoted) state.

↪ **U+0027 APOSTROPHE (')**

> Set the DOCTYPE token's public identifier to the empty string, then switch to the DOCTYPE public identifier (single-quoted) state.

↪ **U+003E GREATER-THAN SIGN (>)**

> Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**

> Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

> Parse error. Switch to the bogus DOCTYPE state.

DOCTYPE public identifier (double-quoted) state

Consume the next input character:

↪ **U+0022 QUOTATION MARK (")**

Switch to the after DOCTYPE public identifier state.

↪ **EOF**

Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. 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.

DOCTYPE public identifier (single-quoted) state

Consume the next input character:

↪ **U+0027 APOSTROPHE (')**

Switch to the after DOCTYPE public identifier state.

↪ **EOF**

Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. 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.

After DOCTYPE public identifier state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the after DOCTYPE public identifier state.

↪ **U+0022 QUOTATION MARK (")**

Set the DOCTYPE token's system identifier to the empty string, then switch to the DOCTYPE system identifier (double-quoted) state.

↪ **U+0027 APOSTROPHE (')**

Set the DOCTYPE token's system identifier to the empty string, 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 correctness flag to *incorrect*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Switch to the bogus DOCTYPE state.

Before DOCTYPE system identifier state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the before DOCTYPE system identifier state.

↪ **U+0022 QUOTATION MARK (")**

Set the DOCTYPE token's system identifier to the empty string, then switch to the DOCTYPE system identifier (double-quoted) state.

↪ **U+0027 APOSTROPHE (')**

Set the DOCTYPE token's system identifier to the empty string, then switch to the DOCTYPE system identifier (single-quoted) state.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. Emit that DOCTYPE token. Switch to the data state.

↪ **EOF**

Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. Emit that DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**

Parse error. Switch to the bogus DOCTYPE state.

DOCTYPE system identifier (double-quoted) state

Consume the next input character:

↪ **U+0022 QUOTATION MARK (")**

Switch to the after DOCTYPE system identifier state.

↪ **EOF**

Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. 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.

DOCTYPE system identifier (single-quoted) state

Consume the next input character:

↪ **U+0027 APOSTROPHE (')**

Switch to the after DOCTYPE system identifier state.

↪ **EOF**

Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. 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.

After DOCTYPE system identifier state

Consume the next input character:

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

Stay in the after DOCTYPE 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 correctness flag to *incorrect*. Emit that
> DOCTYPE token. Reconsume the EOF character in the data state.

> ↪ **Anything else**
>> Parse error. Switch to the bogus DOCTYPE state.

Bogus DOCTYPE state

Consume the next input character:

↪ **U+003E GREATER-THAN SIGN (>)**
> Set the DOCTYPE token's correctness flag to *incorrect*. Emit that DOCTYPE
> token. Switch to the data state.

↪ **EOF**
> Parse error. Set the DOCTYPE token's correctness flag to *incorrect*. Emit that
> DOCTYPE token. Reconsume the EOF character in the data state.

↪ **Anything else**
> Stay in the bogus DOCTYPE state.

8.2.3.1. Tokenising entities

This section defines how to **consume an entity**. This definition is used when parsing entities in text
and in attributes.

The behaviour depends on the identity of the next character (the one immediately after the U+0026
AMPERSAND character):

↪ **U+0009 CHARACTER TABULATION**

↪ **U+000A LINE FEED (LF)**

↪ **U+000B LINE TABULATION**

↪ **U+000C FORM FEED (FF)**

↪ **U+0020 SPACE**

↪ **U+003C LESS-THAN SIGN**

↪ **U+0026 AMPERSAND**

↪ **EOF**
> Not an entity. 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 behaviour further depends on the character after the U+0023 NUMBER SIGN:
>
> ↪ **U+0078 LATIN SMALL LETTER X**
> ↪ **U+0058 LATIN CAPITAL LETTER X**
>> Consume the X.
>>
>> Follow the steps below, but using the range of characters U+0030 DIGIT
>> ZERO through to U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A
>> through to U+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 ('•')

Number		Unicode character
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 zero, if the number is higher than 0x10FFFF, or if it's one of the surrogate characters (characters in the range 0xD800 to 0xDFFF), 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 entities 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 entity 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 entity name (as given by the second column of the entities table).
>
> > If the markup contains `I'm &notit; I tell you`, the entity is parsed as "not", as in, `I'm ¬it; I tell you`. But if the markup was `I'm ∉ I tell you`, the entity would be parsed as "notin;", resulting in `I'm ∉ I tell you`.

8.2.4. Tree construction

The input to the tree construction stage is a sequence of tokens from the tokenisation stage. The tree construction stage is associated with a DOM Document object when a parser is created. The "output" of this stage consists of dynamically modifying or extending that document's DOM tree.

Tree construction passes through several phases. Initially, UAs must act according to the steps described as being those of the initial phase.

This specification does not define when an interactive user agent has to render the `Document` available to the user, or when it has to begin accepting user input.

When the steps below require the UA to **append a character** to a node, the UA must collect it and all subsequent consecutive characters that would be appended to that node, and insert one `Text` node whose data is the concatenation of all those characters.

DOM mutation events must not fire for changes caused by the UA parsing the document. (Conceptually, the parser is not mutating the DOM, it is constructing it.) This includes the parsing of any content inserted using `document.write()` and `document.writeln()` calls. [DOM3EVENTS]

> *Note: Not all of the tag names mentioned below are conformant tag names in this specification; many are included to handle legacy content. They still form part of the algorithm that implementations are required to implement to claim conformance.*

> *Note: The algorithm described below places no limit on the depth of the DOM tree generated, or on the length of tag names, attribute names, attribute values, text nodes, etc. While implementators are encouraged to avoid arbitrary limits, it is recognised that practical concerns will likely force user agents to impose nesting depths.*

8.2.4.1. The initial phase

Initially, the tree construction stage must handle each token emitted from the tokenisation stage as follows:

↳ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

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 recognise 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 not set; the `systemId` attribute set to the system identifier given in the DOCTYPE token, or the empty string if the system identifier was not set; 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 correctness flag is set to *incorrect*.
- The name is set to anything other than "HTML".
- The public identifier is set to: "+//Silmaril//dtd html Pro v0r11 19970101//EN"
- The public identifier is set to: "-//AdvaSoft Ltd//DTD HTML 3.0 asWedit + extensions//EN"
- The public identifier is set to: "-//AS//DTD HTML 3.0 asWedit + extensions//EN"
- The public identifier is set to: "-//IETF//DTD HTML 2.0 Level 1//EN"
- The public identifier is set to: "-//IETF//DTD HTML 2.0 Level 2//EN"
- The public identifier is set to: "-//IETF//DTD HTML 2.0 Strict Level 1//EN"
- The public identifier is set to: "-//IETF//DTD HTML 2.0 Strict Level 2//EN"
- The public identifier is set to: "-//IETF//DTD HTML 2.0 Strict//EN"
- The public identifier is set to: "-//IETF//DTD HTML 2.0//EN"
- The public identifier is set to: "-//IETF//DTD HTML 2.1E//EN"
- The public identifier is set to: "-//IETF//DTD HTML 3.0//EN"
- The public identifier is set to: "-//IETF//DTD HTML 3.0//EN//"
- The public identifier is set to: "-//IETF//DTD HTML 3.2 Final//EN"
- The public identifier is set to: "-//IETF//DTD HTML 3.2//EN"
- The public identifier is set to: "-//IETF//DTD HTML 3//EN"
- The public identifier is set to: "-//IETF//DTD HTML Level 0//EN"
- The public identifier is set to: "-//IETF//DTD HTML Level 0//EN//2.0"
- The public identifier is set to: "-//IETF//DTD HTML Level 1//EN"
- The public identifier is set to: "-//IETF//DTD HTML Level 1//EN//2.0"
- The public identifier is set to: "-//IETF//DTD HTML Level 2//EN"
- The public identifier is set to: "-//IETF//DTD HTML Level 2//EN//2.0"
- The public identifier is set to: "-//IETF//DTD HTML Level 3//EN"
- The public identifier is set to: "-//IETF//DTD HTML Level 3//EN//3.0"
- The public identifier is set to: "-//IETF//DTD HTML Strict Level 0//EN"
- The public identifier is set to: "-//IETF//DTD HTML Strict Level 0//EN//2.0"
- The public identifier is set to: "-//IETF//DTD HTML Strict Level 1//EN"
- The public identifier is set to: "-//IETF//DTD HTML Strict Level 1//EN//2.0"
- The public identifier is set to: "-//IETF//DTD HTML Strict Level 2//EN"
- The public identifier is set to: "-//IETF//DTD HTML Strict Level 2//EN//2.0"
- The public identifier is set to: "-//IETF//DTD HTML Strict Level 3//EN"
- The public identifier is set to: "-//IETF//DTD HTML Strict Level 3//EN//3.0"
- The public identifier is set to: "-//IETF//DTD HTML Strict//EN"
- The public identifier is set to: "-//IETF//DTD HTML Strict//EN//2.0"
- The public identifier is set to: "-//IETF//DTD HTML Strict//EN//3.0"
- The public identifier is set to: "-//IETF//DTD HTML//EN"
- The public identifier is set to: "-//IETF//DTD HTML//EN//2.0"
- The public identifier is set to: "-//IETF//DTD HTML//EN//3.0"
- The public identifier is set to: "-//Metrius//DTD Metrius Presentational//EN"
- The public identifier is set to: "-//Microsoft//DTD Internet Explorer 2.0 HTML Strict//EN"
- The public identifier is set to: "-//Microsoft//DTD Internet Explorer 2.0 HTML//EN"
- The public identifier is set to: "-//Microsoft//DTD Internet Explorer 2.0 Tables//EN"
- The public identifier is set to: "-//Microsoft//DTD Internet Explorer 3.0 HTML Strict//EN"
- The public identifier is set to: "-//Microsoft//DTD Internet Explorer

```
3.0 HTML//EN"
```
- The public identifier is set to: "`-//Microsoft//DTD Internet Explorer
  3.0 Tables//EN`"
- The public identifier is set to: "`-//Netscape Comm. Corp.//DTD HTML//EN`"
- The public identifier is set to: "`-//Netscape Comm. Corp.//DTD Strict
  HTML//EN`"
- The public identifier is set to: "`-//O'Reilly and Associates//DTD HTML
  2.0//EN`"
- The public identifier is set to: "`-//O'Reilly and Associates//DTD HTML
  Extended 1.0//EN`"
- The public identifier is set to: "`-//Spyglass//DTD HTML 2.0 Extended//EN`"
- The public identifier is set to: "`-//SQ//DTD HTML 2.0 HoTMetaL +
  extensions//EN`"
- The public identifier is set to: "`-//Sun Microsystems Corp.//DTD HotJava
  HTML//EN`"
- The public identifier is set to: "`-//Sun Microsystems Corp.//DTD HotJava
  Strict HTML//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML 3 1995-03-24//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML 3.2 Draft//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML 3.2 Final//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML 3.2//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML 3.2S Draft//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML 4.0 Frameset//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML 4.0 Transitional//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML Experimental
  19960712//EN`"
- The public identifier is set to: "`-//W3C//DTD HTML Experimental
  970421//EN`"
- The public identifier is set to: "`-//W3C//DTD W3 HTML//EN`"
- The public identifier is set to: "`-//W3O//DTD W3 HTML 3.0//EN`"
- The public identifier is set to: "`-//W3O//DTD W3 HTML 3.0//EN//`"
- The public identifier is set to: "`-//W3O//DTD W3 HTML Strict 3.0//EN//`"
- The public identifier is set to: "`-//WebTechs//DTD Mozilla HTML 2.0//EN`"
- The public identifier is set to: "`-//WebTechs//DTD Mozilla HTML//EN`"
- 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 is set to: "`-//W3C//DTD
  HTML 4.01 Frameset//EN`"
- The system identifier is missing and the public identifier is set to: "`-//W3C//DTD
  HTML 4.01 Transitional//EN`"

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 is set to: "`-//W3C//DTD XHTML 1.0 Frameset//EN`"
- The public identifier is set to: "`-//W3C//DTD XHTML 1.0 Transitional//EN`"
- The system identifier is not missing and the public identifier is set to:
  "`-//W3C//DTD HTML 4.01 Frameset//EN`"
- The system identifier is not missing and the public identifier is set to:
  "`-//W3C//DTD HTML 4.01 Transitional//EN`"

The name, system identifier, and public identifier strings must be compared to the values
given in the lists above in a case-insensitive manner.

Then, switch to the root element phase of the tree construction stage.

↳ **A start tag token**
↳ **An end tag token**

↳ **A character token that is not one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

↳ **An end-of-file token**

> Parse error.
>
> Set the document to quirks mode.
>
> Then, switch to the root element phase of the tree construction stage and reprocess the current token.

8.2.4.2. The root element phase

After the initial phase, as each token is emitted from the tokenisation stage, it must be processed as described in this section.

↳ **A DOCTYPE token**

> Parse error. Ignore the token.

↳ **A comment token**

> Append a Comment node to the Document object with the data attribute set to the data given in the comment token.

↳ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

> Ignore the token.

↳ **A character token that is *not* one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

↳ **A start tag token**

↳ **An end tag token**

↳ **An end-of-file token**

> If the token is a start tag token with the tag name "html", and it has an attribute "application", then run the application cache selection algorithm with the value of that attribute as the manifest URI. Otherwise, run the application cache selection algorithm with no manifest.
>
> Create an HTMLElement node with the tag name html, in the HTML namespace. Append it to the Document object. Switch to the main phase and reprocess the current token.

> Should probably make end tags be ignored, so that "</head><!-- --><html>" puts the comment 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.

8.2.4.3. The main phase

After the root element phase, each token emitted from the tokenisation stage must be processed as described in *this* section. This is by far the most involved part of parsing an HTML document.

The tree construction stage in this phase has several pieces of state: a stack of open elements, a list

of active formatting elements, a `head` element pointer, a `form` element pointer, and an insertion mode.

We could just fold insertion modes and phases into one concept (and duplicate the two rules common to all insertion modes into all of them).

8.2.4.3.1. THE STACK OF OPEN ELEMENTS

Initially the **stack of open elements** contains just the `html` root element node created in the last phase before switching to *this* phase (or, in the fragment case, the `html` element created as part of that algorithm). That's the topmost node of the stack. It never gets popped off the stack. (This stack grows downwards.)

The **current node** is the bottommost node in this stack.

Elements in the stack fall into the following categories:

Special

> The following HTML elements have varying levels of special parsing rules: `address`, `area`, `base`, basefont, bgsound, `blockquote`, `body`, `br`, center, `col`, `colgroup`, dd, dir, `div`, `dl`, `dt`, `embed`, fieldset, form, frame, frameset, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `head`, `hr`, `iframe`, image, `img`, input, isindex, `li`, `link`, listing, `menu`, `meta`, noembed, noframes, `noscript`, `ol`, optgroup, option, `p`, `param`, plaintext, `pre`, `script`, select, spacer, `style`, `tbody`, textarea, `tfoot`, `thead`, `title`, `tr`, `ul`, and wbr.

Scoping

> The following HTML elements introduce new scopes for various parts of the parsing: button, `caption`, `html`, marquee, `object`, `table`, `td` and `th`.

Formatting

> The following HTML elements are those that end up in the list of active formatting elements: `a`, `b`, big, `em`, `font`, `i`, nobr, s, `small`, strike, `strong`, tt, and u.

Phrasing

> All other elements found while parsing an HTML document.

Still need to add these new elements to the lists: `event-source`, `section`, `nav`, `article`, `aside`, `header`, `footer`, `datagrid`, `command`

The stack of open elements is said to **have an element in scope** or **have an element in *table scope*** when the following algorithm terminates in a match state:

1. Initialise *node* to be the current node (the bottommost node of the stack).

2. If *node* is the target node, terminate in a match state.

3. Otherwise, if *node* is a `table` element, terminate in a failure state.

4. Otherwise, if the algorithm is the "has an element in scope" variant (rather than the "has an element in table scope" variant), and *node* is one of the following, terminate in a failure state:

   - `caption`
   - `td`
   - `th`

- button
- marquee
- object

5. Otherwise, if *node* is an `html` element, terminate in a failure state. (This can only happen if the *node* is the topmost node of the <u>stack of open elements</u>, and prevents the next step from being invoked if there are no more elements in the stack.)

6. Otherwise, set *node* to the previous entry in the <u>stack of open elements</u> and return to step 2. (This will never fail, since the loop will always terminate in the previous step if the top of the stack is reached.)

Nothing happens if at any time any of the elements in the <u>stack of open elements</u> 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 <u>closing misnested formatting elements</u>), the stack is manipulated in a random-access fashion.*

8.2.4.3.2. THE LIST OF ACTIVE FORMATTING ELEMENTS

Initially the **list of active formatting elements** is empty. It is used to handle mis-nested <u>formatting element tags</u>.

The list contains elements in the <u>formatting</u> category, and scope markers. The scope markers are inserted when entering buttons, `object` elements, marquees, table cells, and table captions, and are used to prevent formatting from "leaking" into tables, buttons, `object` elements, and marquees.

When the steps below require the UA to **reconstruct the active formatting elements**, the UA must perform the following steps:

1. If there are no entries in the <u>list of active formatting elements</u>, then there is nothing to reconstruct; stop this algorithm.

2. If the last (most recently added) entry in the <u>list of active formatting elements</u> is a marker, or if it is an element that is in the <u>stack of open elements</u>, then there is nothing to reconstruct; stop this algorithm.

3. Let *entry* be the last (most recently added) element in the <u>list of active formatting elements</u>.

4. If there are no entries before *entry* in the <u>list of active formatting elements</u>, then jump to step 8.

5. Let *entry* be the entry one earlier than *entry* in the <u>list of active formatting elements</u>.

6. If *entry* is neither a marker nor an element that is also in the <u>stack of open elements</u>, go to step 4.

7. Let *entry* be the element one later than *entry* in the <u>list of active formatting elements</u>.

8. Perform a shallow clone of the element *entry* to obtain *clone*. [DOM3CORE]

9. Append *clone* to the <u>current node</u> and push it onto the <u>stack of open elements</u> so that it is the new <u>current node</u>.

10. Replace the entry for *entry* in the list with an entry for *clone*.

11. If the entry for *clone* in the list of active formatting elements is not the last entry in the list, return to step 7.

This has the effect of reopening all the formatting elements that were opened in the current body, cell, or caption (whichever is youngest) that haven't been explicitly closed.

> *Note: The way this specification is written, the list of active formatting elements always consists of elements in chronological order with the least recently added element first and the most recently added element last (except for while steps 8 to 11 of the above algorithm are being executed, of course).*

When the steps below require the UA to **clear the list of active formatting elements up to the last marker**, the UA must perform the following steps:

1. Let *entry* be the last (most recently added) entry in the list of active formatting elements.

2. Remove *entry* from the list of active formatting elements.

3. If *entry* was a marker, then stop the algorithm at this point. The list has been cleared up to the last marker.

4. Go to step 1.

8.2.4.3.3. Creating and inserting HTML elements

When the steps below require the UA to **create an element for a token**, the UA must create a node implementing the interface appropriate for the element type corresponding to the tag name of the token (as given in the section of this specification that defines that element, e.g. for an `a` element it would be the `HTMLAnchorElement` interface), with the tag name being the name of that element, with the node being in the HTML namespace, and with the attributes on the node being those given in the given token.

When the steps below require the UA to **insert an HTML element** for a token, the UA must first create an element for the token, and then append this node to the current node, and push it onto the stack of open elements so that it is the new current node.

The steps below may also require that the UA insert an HTML element in a particular place, in which case the UA must create an element for the token and then insert or append the new node in the location specified. (This happens in particular during the parsing of tables with invalid content.)

The interface appropriate for an element that is not defined in this specification is `HTMLElement`.

The **generic CDATA parsing algorithm** and the **generic RCDATA parsing algorithm** consist of the following steps. These algorithms are always invoked in response to a start tag token, and are always passed a *context node*, typically the current node, which is used as the place to insert the resulting element node.

1. Create an element for the token.

2. Append the new element to the given *context node*.

3. If the algorithm that was invoked is the generic CDATA parsing algorithm, switch the tokeniser's content model flag to the CDATA state; otherwise the algorithm invoked was the generic RCDATA 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 tokenising.

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, this is a parse error.

8.2.4.3.4. CLOSING ELEMENTS THAT HAVE IMPLIED END TAGS

When the steps below require the UA to **generate implied end tags**, then, if the current node is a `dd` element, a `dt` element, an `li` element, a `p` element, a `tbody` element, a `td` element, a `tfoot` element, a `th` element, a `thead` element, a `tr` element, the UA must act as if an end tag with the respective tag name had been seen and then generate implied end tags again.

The step that requires the UA to generate implied end tags but lists an element to exclude from the process, then the UA must perform the above steps as if that element was not in the above list.

8.2.4.3.5. THE ELEMENT POINTERS

Initially the **head element pointer** and the **form element pointer** are both null.

Once a `head` element has been parsed (whether implicitly or explicitly) the head element pointer gets set to point to this node.

The form element pointer points to the last `form` element that was opened and whose end tag has not yet been seen. It is used to make form controls associate with forms in the face of dramatically bad markup, for historical reasons.

8.2.4.3.6. THE INSERTION MODE

Initially the **insertion mode** is "before head". It can change to "in head", "in head noscript", "after head", "in body", "in table", "in caption", "in column group", "in table body", "in row", "in cell", "in select", "after body", "in frameset", and "after frameset" during the course of the parsing, as described below. It affects how certain tokens are processed.

If the tree construction stage is switched from the main phase to the trailing end phase and back again, the various pieces of state are not reset; the UA must act as if the state was maintained.

When the steps below require the UA to **reset the insertion mode appropriately**, it means the UA must follow these steps:

1. Let *last* be false.

2. Let *node* be the last node in the stack of open elements.

3. If *node* is the first node in the stack of open elements, then set *last* to true. If the *context* element of the HTML fragment parsing algorithm is neither a `td` element nor a `th` element, then 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, 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 a `head` element, then switch the [insertion mode](#) to "[in body](#)" ("[in body](#)"! *not "[in head](#)"*!) and abort these steps. ([fragment case](#))

12. If *node* is a `body` element, then switch the [insertion mode](#) to "[in body](#)" and abort these steps.

13. If *node* is a `frameset` element, then switch the [insertion mode](#) to "[in frameset](#)" and abort these steps. ([fragment case](#))

14. If *node* is an `html` element, then: if the [head element pointer](#) is null, switch the [insertion mode](#) to "[before head](#)", otherwise, switch the [insertion mode](#) to "[after head](#)". In either case, abort these steps. ([fragment case](#))

15. If *last* is true, then set the [insertion mode](#) to "[in body](#)" and abort these steps. ([fragment case](#))

16. Let *node* now be the node before *node* in the [stack of open elements](#).

17. Return to step 3.

8.2.4.3.7. HOW TO HANDLE TOKENS IN THE MAIN PHASE

Tokens in the main phase must be handled as follows:

↳ **A DOCTYPE token**

> [Parse error](#). Ignore the token.

↳ **A start tag whose tag name is "html"**

> If this start tag token was not the first start tag token, then it is a [parse error](#).

> For each attribute on the token, check to see if the attribute is already present on the top element of the [stack of open elements](#). If it is not, add the attribute and its corresponding value to that element.

↳ **An end-of-file token**

> [Generate implied end tags.](#)

> If there are more than two nodes on the [stack of open elements](#), or if there are two nodes but the second node is not a `body` node, this is a [parse error](#).

> Otherwise, if the parser was originally created as part of the [HTML fragment parsing algorithm](#), and there's more than one element in the [stack of open elements](#), and the second node on the [stack of open elements](#) is not a `body` node, then this is a [parse](#)

error. (fragment case)

Stop parsing.

> This fails because it doesn't imply HEAD and BODY tags. We should probably expand out the insertion modes and merge them with phases and then put the three things here into each insertion mode instead of trying to factor them out so carefully.

↳ **Anything else**

Depends on the insertion mode:

↳ **If the insertion mode is "before head"**

Handle the token as follows:

↳ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**
Append the character to the current node.

↳ **A comment token**
Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↳ **A start tag whose tag name is "head"**
Create an element for the token.

Set the `head` element pointer to this new element node.

Append the new element to the current node and push it onto the stack of open elements.

Change the insertion mode to "in head".

↳ **A start tag token whose tag name is one of: "base", "link", "meta", "script", "style", "title"**
Act as if a start tag token with the tag name "head" and no attributes had been seen, then reprocess the current token.

> *Note: This will result in a `head` element being generated, and with the current token being reprocessed in the "in head" insertion mode.*

↳ **An end tag whose tag name is one of: "head", "body", "html", "p", "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.

> Do we really want to ignore end tags here?

↳ **A character token that is *not* one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION,**

U+000C FORM FEED (FF), or U+0020 SPACE

↪ **Any other start tag token**

Act as if a start tag token with the tag name "head" and no attributes had been seen, then reprocess the current token.

> *Note: This will result in an empty `head` element being generated, with the current token being reprocessed in the "after head" insertion mode.*

↪ **If the insertion mode is "in head"**

Handle the token as follows.

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character to the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A start tag whose tag name is one of: "base", "link"**

Insert an HTML element for the token.

↪ **A start tag whose tag name is "meta"**

Insert an HTML element for the token.

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 to extract 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 parsing algorithm, with the head element pointer as the *context node*, unless that's null, in which case use the current node (fragment cose).

↪ **A start tag whose tag name is "noscript", if scripting is enabled:**
↪ **A start tag whose tag name is "style"**

Follow the generic CDATA parsing algorithm, with the current node as the *context node*.

↪ **A start tag whose tag name is "noscript", if scripting is disabled:**

Insert a `noscript` element for the token.

Change the insertion mode to "in head noscript".

↪ **A start tag whose tag name is "script"**

Create an element for the token.

Mark the element as being "parser-inserted". This ensures that, if the script is external, any `document.write()` calls in the script will execute in-line, instead of blowing the document away, as would happen in most other cases.

Switch the tokeniser's content model flag to the CDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token, or until it stops tokenising.

If this process resulted in a collection of character tokens, append a single `Text` node to the `script` element node whose contents is the concatenation of all those tokens' characters.

The tokeniser's content model flag will have switched back to the PCDATA state.

If the next token is not an end tag token with the tag name "script", then this is a parse error; mark the `script` element as "already executed". Otherwise, the token is the `script` element's end tag, so ignore it.

If the parser was originally created 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 "scripts that will execute as soon as the parser resumes" 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 script that will execute as soon as the parser resumes, then:

> ↳ **If the tree construction stage is being called reentrantly, say from a call to `document.write()`:**
>   Abort the processing of any nested invokations of the

tokeniser, yielding control back to the caller. (Tokenisation will resume when the caller returns to the "outer" tree construction stage.)

↳ **Otherwise:**

Follow these steps:

1. Let *the script* be the script that will execute as soon as the parser resumes. There is no longer a script that will execute as soon as the parser resumes.

2. Pause until the script has completed loading.

3. Let the insertion point be just before the next input character.

4. Execute the script.

5. Let the insertion point be undefined again.

6. If there is once again a script that will execute as soon as the parser resumes, then repeat these steps from step 1.

↳ **An end tag whose tag name is "head"**

Pop the current node (which will be the `head` element) off the stack of open elements.

Change the insertion mode to "after head".

↳ **An end tag whose tag name is one of: "body", "html", "p", "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.

<div style="border: 2px solid red; color: red;">
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?
</div>

↳ **If the insertion mode is "in head noscript"**

↳ **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+000B LINE TABULATION,
> > > U+000C FORM FEED (FF), or U+0020 SPACE

> > ↳ **A comment token**

> > ↳ **A start tag whose tag name is one of: "link", "meta", "style"**
> > > Process the token as if the insertion mode had been "in head".

> > ↳ **An end tag whose tag name is one of: "p", "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.

> ↳ **If the insertion mode is "after head"**
> > Handle the token as follows:

> > ↳ **A character token that is one of one of U+0009 CHARACTER
> > TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION,
> > U+000C FORM FEED (FF), or U+0020 SPACE**
> > > Append the character to the current node.

> > ↳ **A comment token**
> > > Append a Comment node to the current node with the data
> > > attribute set to the data given in the comment token.

> > ↳ **A start tag whose tag name is "body"**
> > > Insert a body element for the token.
> > >
> > > Change the insertion mode to "in body".

> > ↳ **A start tag whose tag name is "frameset"**
> > > Insert a frameset element for the token.
> > >
> > > Change the insertion mode to "in frameset".

> > ↳ **A start tag token whose tag name is one of: "base", "link", "meta",
> > "script", "style", "title"**
> > > Parse error.
> > >
> > > Push the node pointed to by the head element pointer onto the
> > > stack of open elements.
> > >
> > > Process the token as if the insertion mode had been "in head".
> > >
> > > Pop the current node (which will be the node pointed to by the
> > > head element pointer) off the stack of open elements.

> > ↳ **Anything else**
> > > Act as if a start tag token with the tag name "body" and no
> > > attributes had been seen, and then reprocess the current token.

> ↳ **If the insertion mode is "in body"**

Handle the token as follows:

↪ **A character token**

Reconstruct the active formatting elements, if any.

Append the token's character to the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A start tag token whose tag name is one of: "base", "link", "meta", "script", "style"**

Process the token as if the insertion mode had been "in head".

↪ **A start tag whose tag name is "title"**

Parse error. Process the token as if the insertion mode had been "in head".

↪ **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 tag whose tag name is "body"**

If the second element in the stack of open elements is not a `body` element, this is a parse error. Ignore the token. (fragment case)

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.

Change the insertion mode to "after body".

↪ **An end tag whose tag name is "html"**

Act as if an end tag with tag name "body" had been seen, then, if that token wasn't ignored, reprocess the current token.

> *Note: The fake end tag token here can only be*
> *ignored in the fragment case.*

↪ **A start tag whose tag name is one of: "address", "blockquote", "center", "dir", "div", "dl", "fieldset", "listing", "menu", "ol", "p", "ul"**

This doesn't match browsers.

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 "pre"**

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, ignore the token with a parse error.

Otherwise:

If the stack of open elements has a p element in scope, then act as if an end tag with the tag name p had been seen.

Insert an HTML element for the token, and set the form element pointer to point to the element created.

↳ **A start tag whose tag name is "li"**

If the stack of open elements has a p element in scope, then act as if an end tag with the tag name p had been seen.

Run the following algorithm:

1. Initialise *node* to be the current node (the bottommost node of the stack).

2. If *node* is an li element, then pop all the nodes from the current node up to *node*, including *node*, then stop this algorithm. If more than one node is popped, then this is a parse error.

3. If *node* is not in the formatting category, and is not in the phrasing category, and is not an address or div element, then stop this algorithm.

4. Otherwise, set *node* to the previous entry in the stack of open elements and return to step 2.

Finally, insert an li element.

↳ **A start tag whose tag name is one of: "dd", "dt"**

If the stack of open elements has a p element in scope, then act as if an end tag with the tag name p had been seen.

Run the following algorithm:

1. Initialise *node* to be the current node (the bottommost node

of the stack).

2. If *node* is a <u>dd</u> or <u>dt</u> element, then pop all the nodes from the <u>current node</u> up to *node*, including *node*, then stop this algorithm. If more than one node is popped, then this is a <u>parse error</u>.

3. If *node* is not in the <u>formatting</u> category, and is not in the <u>phrasing</u> category, and is not an <u>address</u> or <u>div</u> element, then stop this algorithm.

4. Otherwise, set *node* to the previous entry in the <u>stack of open elements</u> and return to step 2.

Finally, <u>insert an HTML element</u> with the same tag name as the token's.

↪ **A start tag whose tag name is "plaintext"**

If the <u>stack of open elements</u> <u>has a p element in scope</u>, then act as if an end tag with the tag name <u>p</u> had been seen.

<u>Insert an HTML element</u> for the token.

Switch the <u>content model flag</u> 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 <u>content model flag</u> out of the PLAINTEXT state.*

↪ **An end tag whose tag name is one of: "address", "blockquote", "center", "dir", "div", "dl", "fieldset", "listing", "menu", "ol", "pre", "ul"**

If the <u>stack of open elements</u> <u>has an element in scope</u> with the same tag name as that of the token, then <u>generate implied end tags</u>.

Now, if the <u>current node</u> is not an element with the same tag name as that of the token, then this is a <u>parse error</u>.

If the <u>stack of open elements</u> <u>has an element in scope</u> with the same tag name as that of the token, then pop elements from this stack until an element with that tag name has been popped from the stack.

↪ **An end tag whose tag name is "form"**

If the <u>stack of open elements</u> <u>has an element in scope</u> with the same tag name as that of the token, then <u>generate implied end tags</u>.

Now, if the <u>current node</u> is not an element with the same tag name as that of the token, then this is a <u>parse error</u>.

Otherwise, if the <u>current node</u> is an element with the same tag name as that of the token pop that element from the stack.

In any case, set the form element pointer to null.

↳ **An end tag whose tag name is "p"**

If the stack of open elements has a p element in scope, then generate implied end tags, except for p elements.

If the current node is not a p element, then this is a parse error.

If the stack of open elements has a p element in scope, then pop elements from this stack until the stack no longer has a p element in scope.

Otherwise, act as if a start tag with the tag name p had been seen, then reprocess the current token.

↳ **An end tag whose tag name is one of: "dd", "dt", "li"**

If the stack of open elements has an element in scope whose tag name matches the tag name of the token, then generate implied end tags, except for elements with the same tag name as the token.

If the current node is not an element with the same tag name as the token, then this is a parse error.

If the stack of open elements has an element in scope whose tag name matches the tag name of the token, then pop elements from this stack until an element with that tag name has been popped from the stack.

↳ **A start tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

If the stack of open elements has a p element in scope, then act as if an end tag with the tag name p had been seen.

Insert an HTML element for the token.

↳ **An end tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

If the stack of open elements has in scope an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then generate implied end tags.

Now, if the current node is not an element with the same tag name as that of the token, then this is a parse error.

If the stack of open elements has in scope an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then pop elements from the stack until an element with one of those tag names has been popped from the stack.

↳ **A start tag whose tag name is "a"**

If the list of active formatting elements contains an element whose tag name is "a" between the end of the list and the last marker on the list (or the start of the list if there is no marker on the list), then this is a parse error; act as if an end tag with the tag name "a" had been seen, then remove that element from the list of active

formatting elements and the stack of open elements if the end tag didn't already remove it (it might not have if the element is not in table scope).

> In the non-conforming stream
> `<a href="a">a<table><a href="b">b</table>x`, the first a element would be closed upon seeing the second one, and the "x" character would be inside a link to "b", not to "a". This is despite the fact that the outer a element is not in table scope (meaning that a regular `</a>` end tag at the start of the table wouldn't close the outer a element).

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Add that element to the list of active formatting elements.

↪ **A start tag whose tag name is one of: "b", "big", "em", "font", "i", "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. Abort these steps. The token is ignored.

   Otherwise, if there is such a node, but that node is not in the stack of open elements, then this is a parse error; remove the element from the list, and abort these steps.

Otherwise, there is a *formatting element* and that element is in the stack and is in scope. If the element is not the current node, this is a parse error. In any case, proceed with the algorithm as written in the following steps.

2. Let the *furthest block* be the topmost node in the stack of open elements that is lower in the stack than the *formatting element*, and is not an element in the phrasing or formatting categories. There might not be one.

3. If there is no *furthest block*, then the UA must skip the subsequent steps and instead just pop all the nodes from the bottom of the stack of open elements, from the current node up to 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 prior to *node* in the stack of open elements.

   2. If *node* is not in the list of active formatting elements, then remove *node* from the stack of open elements and then go back to step 1.

   3. Otherwise, if *node* is the *formatting element*, then go to the next step in the overall algorithm.

   4. Otherwise, if *last node* is the *furthest block*, then move the aforementioned bookmark to be immediately after the *node* in the list of active formatting elements.

   5. If *node* has any children, perform a shallow clone of *node*, replace the entry for *node* in the list of active formatting elements with an entry for the clone, replace the entry for *node* in the stack of open elements with an entry for the clone, and let *node* be the clone.

   6. Insert *last node* into *node*, first removing it from its previous parent node if any.

7. Let *last node* be *node*.

8. Return to step 1 of this inner set of steps.

8. Insert whatever *last node* ended up being in the previous step into the *common ancestor* node, first removing it from its previous parent node if any.

9. Perform a shallow clone of the *formatting element*.

10. Take all of the child nodes of the *furthest block* and append them to the clone created in the last step.

11. Append that clone to the *furthest block*.

12. Remove the *formatting element* from the list of active formatting elements, and insert the clone into the list of active formatting elements at the position of the aforementioned bookmark.

13. Remove the *formatting element* from the stack of open elements, and insert the clone into the stack of open elements immediately after (i.e. in a more deeply nested position than) the position of the *furthest block* in that stack.

14. Jump back to step 1 in this series of steps.

Note: The way these steps are defined, only elements in the _formatting_ category ever get cloned by this algorithm.

Note: Because of the way this algorithm causes elements to change parents, it has been dubbed the "adoption agency algorithm" (in contrast with other possibly algorithms for dealing with misnested content, which included the "incest algorithm", the "secret affair algorithm", and the "Heisenberg algorithm").

↳ **A start tag whose tag name is "button"**

If the stack of open elements has a `button` element in scope, then this is a parse error; act as if an end tag with the tag name "button" had been seen, then reprocess the token.

Otherwise:

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Insert a marker at the end of the list of active formatting elements.

↳ **A start tag token whose tag name is one of: "marquee", "object"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Insert a marker at the end of the <u>list of active formatting elements</u>.

↪ **An end tag token whose tag name is one of: "button", "marquee", "object"**

If the <u>stack of open elements</u> <u>has in scope</u> an element whose tag name is the same as the tag name of the token, then <u>generate implied end tags</u>.

Now, if the <u>current node</u> is not an element with the same tag name as the token, then this is a <u>parse error</u>.

Now, if the <u>stack of open elements</u> <u>has an element in scope</u> whose tag name matches the tag name of the token, then pop elements from the stack until that element has been popped from the stack, and <u>clear the list of active formatting elements up to the last marker</u>.

↪ **A start tag whose tag name is "xmp"**

<u>Reconstruct the active formatting elements</u>, if any.

Follow the <u>generic CDATA parsing algorithm</u>, with the <u>current node</u> as the *context node*.

↪ **A start tag whose tag name is "table"**

If the <u>stack of open elements</u> <u>has a p element in scope</u>, then act as if an end tag with the tag name p had been seen.

<u>Insert an HTML element</u> for the token.

Change the <u>insertion mode</u> to "<u>in table</u>".

↪ **A start tag whose tag name is one of: "area", "basefont", "bgsound", "br", "embed", "img", "param", "spacer", "wbr"**

<u>Reconstruct the active formatting elements</u>, if any.

<u>Insert an HTML element</u> for the token. Immediately pop the <u>current node</u> off the <u>stack of open elements</u>.

↪ **A start tag whose tag name is "hr"**

If the <u>stack of open elements</u> <u>has a p element in scope</u>, then act as if an end tag with the tag name p had been seen.

<u>Insert an HTML element</u> for the token. Immediately pop the <u>current node</u> off the <u>stack of open elements</u>.

↪ **A start tag whose tag name is "image"**

<u>Parse error</u>. Change the token's tag name to "img" and reprocess it. (Don't ask.)

↪ **A start tag whose tag name is "input"**

<u>Reconstruct the active formatting elements</u>, if any.

<u>Insert an input element</u> for the token.

If the <u>form element pointer</u> is not null, then associate the input element with the form element pointed to by the <u>form element pointer</u>.

Pop that `input` element off the [stack of open elements](#).

↪ **A start tag whose tag name is "isindex"**

[Parse error](#).

If the [`form` element pointer](#) is not null, then ignore the token.

Otherwise:

Act as if a start tag token with the tag name "form" had been seen.

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](#).

If the <u>form</u> <u>element pointer</u> is not null, then associate the textarea element with the form element pointed to by the <u>form element pointer</u>.

Append the new element to the <u>current node</u>.

Switch the tokeniser's <u>content model flag</u> 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 tokenising.

If this process resulted in a collection of character tokens, append a single Text node, whose contents is the concatenation of all those tokens' characters, to the new element node.

The tokeniser's <u>content model flag</u> 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 <u>parse error</u>.

↪ **A start tag whose tag name is one of: "iframe", "noembed", "noframes"**

↪ **A start tag whose tag name is "noscript", if <u>scripting is enabled</u>:**
Follow the <u>generic CDATA parsing algorithm</u>, with the <u>current node</u> as the *context node*.

↪ **A start tag whose tag name is "select"**
<u>Reconstruct the active formatting elements</u>, if any.

<u>Insert an HTML element</u> for the token.

Change the <u>insertion mode</u> to "<u>in select</u>".

↪ **An end tag whose tag name is "br"**
<u>Parse error</u>. Act as if a start tag token with the tag name "br" had been seen. Ignore the end tag token.

↪ **A start or end tag whose tag name is one of: "caption", "col", "colgroup", "frame", "frameset", "head", "option", "optgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

↪ **An end tag whose tag name is one of: "area", "basefont", "bgsound", "br", "embed", "hr", "iframe", "image", "img", "input", "isindex", "noembed", "noframes", "param", "select", "spacer", "table", "textarea", "wbr"**

↪ **An end tag whose tag name is "noscript", if <u>scripting is enabled</u>:**
<u>Parse error</u>. Ignore the token.

↪ **A start or end tag whose tag name is one of: "event-source", "section", "nav", "article", "aside", "header", "footer", "datagrid",**

"command"

> Work in progress!

↪ **A start tag token not covered by the previous entries**
> Reconstruct the active formatting elements, if any.
>
> Insert an HTML element for the token.
>
> *Note: This element will be a phrasing element.*

↪ **An end tag token not covered by the previous entries**
> Run the following algorithm:
>
> 1. Initialise *node* to be the current node (the bottommost node of the stack).
>
> 2. If *node* has the same tag name as the end tag token, then:
>
>   1. Generate implied end tags.
>
>   2. If the tag name of the end tag token does not match the tag name of the current node, this is a parse error.
>
>   3. Pop all the nodes from the current node up to *node*, including *node*, then stop this algorithm.
>
> 3. Otherwise, if *node* is in neither the formatting category nor the phrasing category, then this is a parse error. Stop this algorithm. The end tag token is ignored.
>
> 4. Set *node* to the previous entry in the stack of open elements.
>
> 5. Return to step 2.

↪ **If the insertion mode is "in table"**

> ↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**
> > Append the character to the current node.
>
> ↪ **A comment token**
> > Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.
>
> ↪ **A start tag whose tag name is "caption"**
> > Clear the stack back to a table context. (See below.)
> >
> > Insert a marker at the end of the list of active formatting elements.
> >
> > Insert an HTML element for the token, then switch the insertion mode to "in caption".
>
> ↪ **A start tag whose tag name is "colgroup"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in column group".

↳ **A start tag whose tag name is "col"**

Act as if a start tag token with the tag name "colgroup" had been seen, then reprocess the current token.

↳ **A start tag whose tag name is one of: "tbody", "tfoot", "thead"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in table body".

↳ **A start tag whose tag name is one of: "td", "th", "tr"**

Act as if a start tag token with the tag name "tbody" had been seen, then reprocess the current token.

↳ **A start tag whose tag name is "table"**

Parse error. Act as if an end tag token with the tag name "table" had been seen, then, if that token wasn't ignored, reprocess the current token.

> *Note: The fake end tag token here can only be ignored in the 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:

Generate implied end tags.

Now, if the current node is not a `table` element, then this is a parse error.

Pop elements from this stack until a `table` element has been popped from the stack.

Reset the insertion mode appropriately.

↳ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↳ **Anything else**

Parse error. Process the token as if the insertion mode was "in body", with the following exception:

If the current node is a `table`, `tbody`, `tfoot`, `thead`, or `tr` element, then, whenever a node would be inserted into the current node, it must instead be inserted into the *foster parent element*.

The **foster parent element** is the parent element of the last `table` element in the stack of open elements, if there is a `table` element and it has such a parent element. If there is no `table` element in the stack of open elements (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 the new node must be inserted immediately *before* the last `table` element in the stack of open elements in the foster parent element; otherwise, the new node must be *appended* to the foster parent element.

When the steps above require the UA to **clear the stack back to a table context**, it means that the UA must, while the current node is not a `table` element or an `html` element, pop elements from the stack of open elements. If this causes any elements to be popped from the stack, then this is a parse error.

> *Note: The current node being an `html` element after this process is a fragment case.*

↳ **If the insertion mode is "in caption"**

　　↳ **An end tag whose tag name is "caption"**

　　　　If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (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 <u>fragment case</u>.

> ↳ **An end tag whose tag name is one of: "body", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**
> > <u>Parse error</u>. Ignore the token.

> ↳ **Anything else**
> > Process the token as if the <u>insertion mode</u> was "<u>in body</u>".

↳ **If the <u>insertion mode</u> is "in column group"**

> ↳ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**
> > <u>Append the character</u> to the <u>current node</u>.

> ↳ **A comment token**
> > Append a `Comment` node to the <u>current node</u> with the `data` attribute set to the data given in the comment token.

> ↳ **A start tag whose tag name is "col"**
> > <u>Insert a `col` element</u> for the token. Immediately pop the <u>current node</u> off the <u>stack of open elements</u>.

> ↳ **An end tag whose tag name is "colgroup"**
> > If the <u>current node</u> is the root `html` element, then this is a <u>parse error</u>, ignore the token. (<u>fragment case</u>)
> >
> > Otherwise, pop the <u>current node</u> (which will be a `colgroup` element) from the <u>stack of open elements</u>. Switch the <u>insertion mode</u> to "<u>in table</u>".

> ↳ **An end tag whose tag name is "col"**
> > <u>Parse error</u>. Ignore the token.

> ↳ **Anything else**
> > Act as if an end tag with the tag name "colgroup" had been seen, and then, if that token wasn't ignored, reprocess the current token.
> >
> > *Note: The fake end tag token here can only be ignored in the <u>fragment case</u>.*

↳ **If the <u>insertion mode</u> is "in table body"**

> ↳ **A start tag whose tag name is "tr"**
> > <u>Clear the stack back to a table body context</u>. (See below.)
> >
> > <u>Insert a `tr` element</u> for the token, then switch the <u>insertion mode</u> to "<u>in row</u>".

> ↳ **A start tag whose tag name is one of: "th", "td"**
> > <u>Parse error</u>. 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 <u>stack of open elements</u> does not <u>have an element in table scope</u> with the same tag name as the token, this is a <u>parse error</u>.

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 as if the insertion mode was "in table".

When the steps above require the UA to **clear the stack back to a table body context**, it means that the UA must, while the current node is not a `tbody`, `tfoot`, `thead`, or `html` element, pop elements from the stack of open elements. If this causes any elements to be popped from the stack, then this is a parse error.

> *Note: The current node being an `html` element after this process is a fragment case.*

↳ **If the insertion mode is "in row"**

↳ **A start tag whose tag name is one of: "th", "td"**
Clear the stack back to a table row context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in cell".

Insert a marker at the end of the list of active formatting elements.

↳ **An end tag whose tag name is "tr"**
If the stack of open elements does not have an element in table scope with the same tag name as the token, this is a parse error. Ignore the token. (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 as if the insertion mode was "in table".

When the steps above require the UA to **clear the stack back to a table row context**, it means that the UA must, while the current node is not a `tr` element or an `html` element, pop elements from the stack of open elements. If this causes any elements to be popped from the stack, then this is a parse error.

> *Note: The current node being an `html` element after this process is a fragment case.*

↪ **If the insertion mode is "in cell"**

↪ **An end tag whose tag name is one of: "td", "th"**

> If the stack of open elements does not have an element in table scope with the same tag name as that of the token, then this is a parse error and the token must be ignored.
>
> Otherwise:
>
> Generate implied end tags, except for elements with the same tag name as the token.
>
> Now, if the current node is not an element with the same tag name as the token, then this is a parse error.
>
> Pop elements from this stack until an element with the same tag name as the token has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in row". (The current node will be a `tr` element at this point.)

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

If the stack of open elements does *not* have a `td` or `th` element in table scope, then this is a parse error; ignore the token. (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 as if the insertion mode was "in body".

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".

↪ **If the insertion mode is "in select"**

Handle the token as follows:

↪ **A character token**

Append the token's character to the current node.

↪ **A comment token**

Append a `Comment` node to the current node with the `data` attribute set to the data given in the comment token.

↪ **A start tag 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.

↳ **An end tag whose tag name is one of: "caption", "table", "tbody", "tfoot", "thead", "tr", "td", "th"**

[Parse error](#).

If the [stack of open elements](#) [has an element in table scope](#) with the same tag name as that of the token, then act as if an end tag with the tag name "select" had been seen, and reprocess the token. Otherwise, ignore the token.

↳ **Anything else**

[Parse error](#). Ignore the token.

↳ **If the <u>insertion mode</u> is "after body"**

Handle the token as follows:

↳ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Process the token as it would be processed if the <u>insertion mode</u> was "<u>in body</u>".

↳ **A comment token**

Append a `Comment` node to the first element in the <u>stack of open elements</u> (the `html` element), with the `data` attribute set to the data given in the comment token.

↳ **An end tag whose tag name is "html"**

If the parser was originally created as part of the <u>HTML fragment parsing algorithm</u>, this is a <u>parse error</u>; ignore the token. (The element will be an `html` element in this case.) (<u>fragment case</u>)

Otherwise, switch to <u>the trailing end phase</u>.

↳ **Anything else**

<u>Parse error</u>. Set the <u>insertion mode</u> to "<u>in body</u>" and reprocess the token.

↳ **If the <u>insertion mode</u> is "in frameset"**

Handle the token as follows:

↳ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

<u>Append the character</u> to the <u>current node</u>.

↳ **A comment token**

Append a `Comment` node to the <u>current node</u> with the `data` attribute set to the data given in the comment token.

↳ **A start tag whose tag name is "frameset"**

<u>Insert a `frameset` element</u> for the token.

↳ **An end tag whose tag name is "frameset"**

If the <u>current node</u> is the root `html` element, then this is a <u>parse error</u>; ignore the token. (<u>fragment case</u>)

Otherwise, pop the <u>current node</u> from the <u>stack of open elements</u>.

If the parser was *not* originally created as part of the <u>HTML fragment parsing algorithm</u> (<u>fragment case</u>), and the <u>current node</u> is no longer a `frameset` element, then change the <u>insertion mode</u> to "<u>after frameset</u>".

↳ **A start tag whose tag name is "frame"**

<u>Insert an HTML element</u> for the token. Immediately pop the <u>current node</u> off the <u>stack of open elements</u>.

↳ **A start tag whose tag name is "noframes"**

Process the token as if the insertion mode had been "in body".

↳ **Anything else**

Parse error. Ignore the token.

↳ **If the insertion mode is "after frameset"**

Handle the token as follows:

↳ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character to the current node.

↳ **A comment token**

Append a Comment node to the current node with the data attribute set to the data given in the comment token.

↳ **An end tag whose tag name is "html"**

Switch to the trailing end phase.

↳ **A start tag whose tag name is "noframes"**

Process the token as if the insertion mode had been "in body".

↳ **Anything else**

Parse error. Ignore the token.

This doesn't handle UAs that don't support frames, or that do support frames but want to show the NOFRAMES content. Supporting the former is easy; supporting the latter is harder.

8.2.4.4. The trailing end phase

After the main phase, as each token is emitted from the tokenisation stage, it must be processed as described in this section.

↳ **A DOCTYPE token**

Parse error. Ignore the token.

↳ **A comment token**

Append a Comment node to the Document object with the data attribute set to the data given in the comment token.

↳ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Process the token as it would be processed in the main phase.

↳ **A character token that is *not* one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

↳ **A start tag token**

↳ **An end tag token**

Parse error. Switch back to the main phase and reprocess the token.

↳ **An end-of-file token**

Stop parsing.

8.2.5. The End

Once the user agent **stops parsing** the document, the user agent must follow the steps in this section.

First, 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 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?

8.3. Namespaces

The **HTML namespace** is: `http://www.w3.org/1999/xhtml`

8.4. Serialising HTML fragments

The following steps form the **HTML fragment serialisation 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 serialises the children of the node being serialised, not the node itself.*

1. Let *s* be a string, and initialise it to the empty string.

2. For each child node *child* of *the node*, in tree order, append the appropriate string from the following list to *s*:

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

   > Append a U+003C LESS-THAN SIGN (<) character, followed by the element's tag name. (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, 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 serialise an element's attributes in the same order.

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

If the child 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 the child node is a `pre` or textarea element, append a U+000A LINE FEED (LF) character.

Append the value of running the HTML fragment serialisation algorithm on the *child* 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 the child node is a `Text` or `CDATASection` node**

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

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

↳ **If the child node is a `Comment`**

Append the literal string `<!--` (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS), followed by the value of the *child* node's `data` DOM attribute, followed by the literal string `-->` (U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN).

↳ **If the child node is a `DocumentType`**

Append the literal string `<!DOCTYPE` (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+0044 LATIN CAPITAL LETTER D, U+004F LATIN CAPITAL LETTER O, U+0043 LATIN CAPITAL LETTER C, U+0054 LATIN CAPITAL LETTER T, U+0059 LATIN CAPITAL LETTER Y, U+0050 LATIN CAPITAL LETTER P, U+0045 LATIN CAPITAL LETTER E), followed by a space (U+0020 SPACE), followed by the value of the *child* node's `name` DOM attribute, followed by the literal string `>` (U+003E GREATER-THAN SIGN).

Other nodes types (e.g. `Attr`) cannot occur as children of elements. If they do, 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 occurances of the "`&`" character by the string "`&`", any occurances of the "`<`" character by the string "`<`", any occurances of the "`>`" character by the string "`>`", and any occurances of the "`"`" character by the string "`"`".

> *Note: Entity reference nodes are __assumed to be expanded__ by the user agent, and are therefore not covered in the algorithm above.*

> *Note: 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 serialised 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 serialising 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 *context*, 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 tokenisation 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 *RCDATA*.

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

      Set the content model flag to *CDATA*.

   ↳ **If it is a `noscript` element**

      If scripting is enabled, set the content model flag to *CDATA*. Otherwise, set the content model flag to *PCDATA*.

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

      Set the content model flag to *PLAINTEXT*.

> ↪ **Otherwise**
>> Set the content model flag to *PCDATA*.

4. Switch the HTML parser's tree construction stage to the main phase.

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

6. Append the element *root* to the `Document` node created above.

7. Set up the parser's stack of open elements so that it contains just the single element *root*.

8. Reset the parser's insertion mode appropriately.

   Note: The parser will reference the context node as part of that algorithm.

9. Set the parser's `form` element pointer to the nearest node to the *context* 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.

10. Place into the input stream for the HTML parser just created the *input*.

11. Start the parser and let it run until it has consumed all the characters just inserted into the input stream.

12. Return all the child nodes of *root*, preserving the document order.

8.6. Entities

This table lists the entity names that are supported by HTML, and the code points to which they refer. It is referenced by the previous sections.

Entity Name	Character
`AElig;`	U+00C6
`AElig`	U+00C6
`AMP;`	U+0026
`AMP`	U+0026
`Aacute;`	U+00C1
`Aacute`	U+00C1
`Acirc;`	U+00C2
`Acirc`	U+00C2
`Agrave;`	U+00C0
`Agrave`	U+00C0
`Alpha;`	U+0391
`Aring;`	U+00C5
`Aring`	U+00C5
`Atilde;`	U+00C3
`Atilde`	U+00C3

Entity Name	Character
Auml;	U+00C4
Auml	U+00C4
Beta;	U+0392
COPY;	U+00A9
COPY	U+00A9
Ccedil;	U+00C7
Ccedil	U+00C7
Chi;	U+03A7
Dagger;	U+2021
Delta;	U+0394
ETH;	U+00D0
ETH	U+00D0
Eacute;	U+00C9
Eacute	U+00C9
Ecirc;	U+00CA
Ecirc	U+00CA
Egrave;	U+00C8
Egrave	U+00C8
Epsilon;	U+0395
Eta;	U+0397
Euml;	U+00CB
Euml	U+00CB
GT;	U+003E
GT	U+003E
Gamma;	U+0393
Iacute;	U+00CD
Iacute	U+00CD
Icirc;	U+00CE
Icirc	U+00CE
Igrave;	U+00CC
Igrave	U+00CC
Iota;	U+0399
Iuml;	U+00CF
Iuml	U+00CF
Kappa;	U+039A
LT;	U+003C
LT	U+003C

Entity Name	Character
Lambda;	U+039B
Mu;	U+039C
Ntilde;	U+00D1
Ntilde	U+00D1
Nu;	U+039D
OElig;	U+0152
Oacute;	U+00D3
Oacute	U+00D3
Ocirc;	U+00D4
Ocirc	U+00D4
Ograve;	U+00D2
Ograve	U+00D2
Omega;	U+03A9
Omicron;	U+039F
Oslash;	U+00D8
Oslash	U+00D8
Otilde;	U+00D5
Otilde	U+00D5
Ouml;	U+00D6
Ouml	U+00D6
Phi;	U+03A6
Pi;	U+03A0
Prime;	U+2033
Psi;	U+03A8
QUOT;	U+0022
QUOT	U+0022
REG;	U+00AE
REG	U+00AE
Rho;	U+03A1
Scaron;	U+0160
Sigma;	U+03A3
THORN;	U+00DE
THORN	U+00DE
TRADE;	U+2122
Tau;	U+03A4
Theta;	U+0398
Uacute;	U+00DA

Entity Name	Character
Uacute	U+00DA
Ucirc;	U+00DB
Ucirc	U+00DB
Ugrave;	U+00D9
Ugrave	U+00D9
Upsilon;	U+03A5
Uuml;	U+00DC
Uuml	U+00DC
Xi;	U+039E
Yacute;	U+00DD
Yacute	U+00DD
Yuml;	U+0178
Zeta;	U+0396
aacute;	U+00E1
aacute	U+00E1
acirc;	U+00E2
acirc	U+00E2
acute;	U+00B4
acute	U+00B4
aelig;	U+00E6
aelig	U+00E6
agrave;	U+00E0
agrave	U+00E0
alefsym;	U+2135
alpha;	U+03B1
amp;	U+0026
amp	U+0026
and;	U+2227
ang;	U+2220
apos;	U+0027
aring;	U+00E5
aring	U+00E5
asymp;	U+2248
atilde;	U+00E3
atilde	U+00E3
auml;	U+00E4
auml	U+00E4

Entity Name	Character
bdquo;	U+201E
beta;	U+03B2
brvbar;	U+00A6
brvbar	U+00A6
bull;	U+2022
cap;	U+2229
ccedil;	U+00E7
ccedil	U+00E7
cedil;	U+00B8
cedil	U+00B8
cent;	U+00A2
cent	U+00A2
chi;	U+03C7
circ;	U+02C6
clubs;	U+2663
cong;	U+2245
copy;	U+00A9
copy	U+00A9
crarr;	U+21B5
cup;	U+222A
curren;	U+00A4
curren	U+00A4
dArr;	U+21D3
dagger;	U+2020
darr;	U+2193
deg;	U+00B0
deg	U+00B0
delta;	U+03B4
diams;	U+2666
divide;	U+00F7
divide	U+00F7
eacute;	U+00E9
eacute	U+00E9
ecirc;	U+00EA
ecirc	U+00EA
egrave;	U+00E8
egrave	U+00E8

Entity Name	Character
empty;	U+2205
emsp;	U+2003
ensp;	U+2002
epsilon;	U+03B5
equiv;	U+2261
eta;	U+03B7
eth;	U+00F0
eth	U+00F0
euml;	U+00EB
euml	U+00EB
euro;	U+20AC
exist;	U+2203
fnof;	U+0192
forall;	U+2200
frac12;	U+00BD
frac12	U+00BD
frac14;	U+00BC
frac14	U+00BC
frac34;	U+00BE
frac34	U+00BE
frasl;	U+2044
gamma;	U+03B3
ge;	U+2265
gt;	U+003E
gt	U+003E
hArr;	U+21D4
harr;	U+2194
hearts;	U+2665
hellip;	U+2026
iacute;	U+00ED
iacute	U+00ED
icirc;	U+00EE
icirc	U+00EE
iexcl;	U+00A1
iexcl	U+00A1
igrave;	U+00EC
igrave	U+00EC

Entity Name	Character
image;	U+2111
infin;	U+221E
int;	U+222B
iota;	U+03B9
iquest;	U+00BF
iquest	U+00BF
isin;	U+2208
iuml;	U+00EF
iuml	U+00EF
kappa;	U+03BA
lArr;	U+21D0
lambda;	U+03BB
lang;	U+3008
laquo;	U+00AB
laquo	U+00AB
larr;	U+2190
lceil;	U+2308
ldquo;	U+201C
le;	U+2264
lfloor;	U+230A
lowast;	U+2217
loz;	U+25CA
lrm;	U+200E
lsaquo;	U+2039
lsquo;	U+2018
lt;	U+003C
lt	U+003C
macr;	U+00AF
macr	U+00AF
mdash;	U+2014
micro;	U+00B5
micro	U+00B5
middot;	U+00B7
middot	U+00B7
minus;	U+2212
mu;	U+03BC
nabla;	U+2207

Entity Name	Character
nbsp;	U+00A0
nbsp	U+00A0
ndash;	U+2013
ne;	U+2260
ni;	U+220B
not;	U+00AC
not	U+00AC
notin;	U+2209
nsub;	U+2284
ntilde;	U+00F1
ntilde	U+00F1
nu;	U+03BD
oacute;	U+00F3
oacute	U+00F3
ocirc;	U+00F4
ocirc	U+00F4
oelig;	U+0153
ograve;	U+00F2
ograve	U+00F2
oline;	U+203E
omega;	U+03C9
omicron;	U+03BF
oplus;	U+2295
or;	U+2228
ordf;	U+00AA
ordf	U+00AA
ordm;	U+00BA
ordm	U+00BA
oslash;	U+00F8
oslash	U+00F8
otilde;	U+00F5
otilde	U+00F5
otimes;	U+2297
ouml;	U+00F6
ouml	U+00F6
para;	U+00B6
para	U+00B6

Entity Name	Character
part;	U+2202
permil;	U+2030
perp;	U+22A5
phi;	U+03C6
pi;	U+03C0
piv;	U+03D6
plusmn;	U+00B1
plusmn	U+00B1
pound;	U+00A3
pound	U+00A3
prime;	U+2032
prod;	U+220F
prop;	U+221D
psi;	U+03C8
quot;	U+0022
quot	U+0022
rArr;	U+21D2
radic;	U+221A
rang;	U+3009
raquo;	U+00BB
raquo	U+00BB
rarr;	U+2192
rceil;	U+2309
rdquo;	U+201D
real;	U+211C
reg;	U+00AE
reg	U+00AE
rfloor;	U+230B
rho;	U+03C1
rlm;	U+200F
rsaquo;	U+203A
rsquo;	U+2019
sbquo;	U+201A
scaron;	U+0161
sdot;	U+22C5
sect;	U+00A7
sect	U+00A7

Entity Name	Character
shy;	U+00AD
shy	U+00AD
sigma;	U+03C3
sigmaf;	U+03C2
sim;	U+223C
spades;	U+2660
sub;	U+2282
sube;	U+2286
sum;	U+2211
sup1;	U+00B9
sup1	U+00B9
sup2;	U+00B2
sup2	U+00B2
sup3;	U+00B3
sup3	U+00B3
sup;	U+2283
supe;	U+2287
szlig;	U+00DF
szlig	U+00DF
tau;	U+03C4
there4;	U+2234
theta;	U+03B8
thetasym;	U+03D1
thinsp;	U+2009
thorn;	U+00FE
thorn	U+00FE
tilde;	U+02DC
times;	U+00D7
times	U+00D7
trade;	U+2122
uArr;	U+21D1
uacute;	U+00FA
uacute	U+00FA
uarr;	U+2191
ucirc;	U+00FB
ucirc	U+00FB
ugrave;	U+00F9

Entity Name	Character
ugrave	U+00F9
uml;	U+00A8
uml	U+00A8
upsih;	U+03D2
upsilon;	U+03C5
uuml;	U+00FC
uuml	U+00FC
weierp;	U+2118
xi;	U+03BE
yacute;	U+00FD
yacute	U+00FD
yen;	U+00A5
yen	U+00A5
yuml;	U+00FF
yuml	U+00FF
zeta;	U+03B6
zwj;	U+200D
zwnj;	U+200C

9. WYSIWYG editors

WYSIWYG editors are authoring tools with a predominantly presentation-driven user interface.

9.1. Presentational markup

9.1.1. WYSIWYG signature

WYSIWYG editors must include a <u>meta</u> element in the <u>head</u> element whose <u>name</u> attribute has the value <u>generator</u> and whose <u>content</u> attribute's value ends with the string "(WYSIWYG editor)". Non-WYSIWYG authoring tools must not include this string in their generator string.

> This entire section will probably be dropped. The intent of this section was to allow a way for WYSIWYG editors, which aren't going to use semantic markup, to still write conforming documents, while not letting it be ok for hand-coding authors to not use semantic markup. We still need some sort of solution to this, but it's not clear what it is.

9.1.2. The `font` element

Categories
> [Phrasing content](#).

Contexts in which this element may be used:

> Where phrasing content is expected.

Content model:

> Transparent.

Element-specific attributes:

> style

DOM interface:

```
interface HTMLFontElement : HTMLElement {
  readonly attribute CSSStyleDeclaration style;
};
```

> This entire section will probably be dropped. The intent of this section was to allow a way for WYSIWYG editors, which don't have enough information to use the "real" "semantic" elements, to still make HTML pages without abusing those semantic elements (since abusing elements is even worse than not using them in the first place). We have still got to find a solution to this, while not letting it be ok for hand-coding authors to abuse the style="" attribute.

The font element doesn't represent anything. It must not be used except by WYSIWYG editors, which may use it to achieve presentational affects. Even WYSIWYG editors, however, should make every effort to use appropriate semantic markup and avoid the use of media-specific presentational markup.

Conformance checkers must consider this element to be non-conforming if it is used on a page lacking the WYSIWYG signature.

> The following would be syntactically legal (as the output from a WYSIWYG editor, though not anywhere else):
>
> ```
> <!DOCTYPE HTML>
> <html>
>  <head>
>   <title></title>
>   <meta name="generator" content="Sample Editor 1.0 (WYSIWYG
> editor)">
>  </head>
>  <body>
>   <font style="display: block; border: solid">
>    <h1>Hello.</h1>
>   </font>
>   <p>
>    <font style="color: orange; background: white">How</font>
>    <font style="color: yellow; background: white">do</font>
>    <font style="color: green; background: white"><em>you</em>
> </font>
>    <font style="color: blue; background: white">do?</font>
>   </p>
>  </body>
> </html>
> ```

> The first <u>font</u> element is conformant because <u>h1</u> and <u>p</u> elements are both allowed in <u>body</u> elements. the next four are allowed because text and <u>em</u> elements are allowed in <u>p</u> elements.

The **style** attribute, if specified, must contain only a list of zero or more semicolon-separated (;) CSS declarations. [CSS21]

> We probably need to move this attribute to more elements, maybe even all of them, though if we do that we really should find a way to strongly discourage its use (and the use of its DOM attribute) for non-WYSIWYG authors.

The declarations specified must be parsed and treated as the body of a declaration block whose selector matches just that <u>font</u> element. For the purposes of the CSS cascade, the attribute must be considered to be a 'style' attribute at the author level.

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 <u>style</u> attribute on the element (if there isn't one already) and then change its value to be a value representing the serialised form of the CSSStyleDeclaration object. [CSSOM]

10. Rendering

> 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** of a document (printing) and what this means for the UA

> Must define that in CSS, tag names in HTML documents, and class names in quirks mode documents, are case-insensitive.

10.1. Rendering and the DOM

> This section is wrong. mediaMode will end up on Window, I think. All views implement Window.

Any object implement the AbstractView interface must also implement the MediaModeAbstractView interface.

```
interface MediaModeAbstractView {
  readonly attribute DOMString mediaMode;
};
```

The **mediaMode** attribute on objects implementing the MediaModeAbstractView interface must

return the string that represents the canvas' current rendering mode (`screen`, `print`, etc). This is a lowercase string, as defined by the CSS specification. [CSS21]

Some user agents may support multiple media, in which case there will exist multiple objects implementing the `AbstractView` interface. Only the default view implements the `Window` interface. The other views can be reached using the `view` attribute of the `UIEvent` inteface, during event propagation. There is no way currently to enumerate all the views.

10.2. Rendering and menus/toolbars

10.2.1. 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.

11. 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.

11.1. Localisation

If you wish to create localised versions of an HTML application, the best solution is to preprocess the files on the server, and then use HTTP content negotation to serve the appropriate language.

11.2. Declarative 2D vector graphics and animation

Embedding vector graphics into XHTML documents is the domain of SVG.

11.3. Declarative 3D scenes

Embedding 3D imagery into XHTML documents is the domain of X3D, or technologies based on X3D that are namespace-aware.

11.4. Timers

This section is expected to be moved to the Window Object specification in due course.

```
interface WindowTimers {
  // timers
```

```
   long setTimeout(in TimeoutHandler handler, in long timeout);
   long setTimeout(in TimeoutHandler handler, in long timeout,
arguments...);
   long setTimeout(in DOMString code, in long timeout);
   long setTimeout(in DOMString code, in long timeout, in DOMString
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(arguments...);
};
```

The `WindowTimers` interface must be obtainable from any `Window` object using binding-specific casting methods.

The `setTimeout` and `setInterval` methods allow authors to schedule timer-based events.

The **setTimeout(*handler*, *timeout*[, *arguments...*])** method takes a reference to a `TimeoutHandler` object and a length of time in milliseconds. It must return a handle to the timeout created, and then asynchronously wait *timeout* milliseconds and then invoke `handleEvent()` on the *handler* object. If any *arguments...* were provided, they must be passed to the *handler* as arguments to the `handleEvent()` function.

In the ECMAScript DOM binding, the ECMAScript native `Function` type must implement the `TimeoutHandler` interface such that invoking the `handleEvent()` method of that interface on the object from another language binding invokes the function itself, with the arguments passed to `handleEvent()` as the arguments passed to the function. In the ECMAScript DOM binding itself, however, the `handleEvent()` method of the interface is not directly accessible on `Function` objects. Such functions must be called in the scope of the browsing context in which they were created.

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 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.)

11.5. Events

In the ECMAScript DOM binding, the ECMAScript native `Function` type must implement the `EventListener` interface such that invoking the `handleEvent()` method of that interface on the object from another language binding invokes the function itself, with the `event` argument as its only argument. In the ECMAScript binding itself, however, the `handleEvent()` method of the interface is not directly accessible on `Function` objects. Such functions, when invoked, must be called in the scope of the [browsing context](#) that they were created in.

References

This section will be written in a future draft.

Acknowledgements

Thanks to Aankhen, Aaron Boodman, Aaron Leventhal, Adam Roben, Adrian Sutton, Agustín Fernández, Alexey Feldgendler, Andrew Gove, Andrew Sidwell, Anne van Kesteren, Anthony Hickson, Antti Koivisto, Asbjørn Ulsberg, Ben Godfrey, Ben Meadowcroft, Benjamin Hawkes-Lewis, Bert Bos, Billy Wong, Bjoern Hoehrmann, Boris Zbarsky, Brad Fults, Brad Neuberg, Brady Eidson, Brendan Eich, Brett Wilson, Brian Campbell, Carlos Perelló Marín, Chao Cai, 윤석찬 (Channy Yun), Charl van Niekerk, Charles Iliya Krempeaux, Charles McCathieNevile, Christian Biesinger, Christian Johansen, Chriswa, Daniel Brumbaugh Keeney, Daniel Peng, Daniel Spång, Darin Alder, Darin Fisher, Dave Singer, Dave Townsend, David Baron, David Flanagan, David Håsäther, David Hyatt, Derek Featherstone, DeWitt Clinton, Dimitri Glazkov, dolphinling, Doron Rosenberg, Doug Kramer, Eira Monstad, Elliotte Harold, Erik Arvidsson, Evan Martin, fantasai, Franck 'Shift' Quélain, Garrett Smith, Geoffrey Sneddon, Håkon Wium Lie, Henri Sivonen, Henrik Lied, Ignacio Javier, Ivo Emanuel Gonçalves, J. King, James Graham, James M Snell, James Perrett, Jan-Klaas Kollhof, Jasper Bryant-Greene, Jeff Cutsinger, Jeff Walden, Jens Bannmann, Jeroen van der Meer, Joel Spolsky, John Boyer, John Bussjaeger, John Harding, Johnny Stenback, Jon Perlow, Jonathan Worent, Jorgen Horstink, Josh Levenberg, Joshua Randall, Jukka K. Korpela, Kai Hendry, Kornel Lesinski, 黒澤剛志 (KUROSAWA Takeshi), Kristof Zelechovski, Lachlan Hunt, Larry Page, Lars Gunther, Laurens Holst, Lenny Domnitser, Léonard Bouchet, Leons Petrazickis, Logan, Loune, Maciej Stachowiak, Malcolm Rowe, Mark Nottingham, Mark Rowe, Mark Schenk, Martijn Wargers, Martin Atkins, Martin Dürst, Martin Honnen, Mathieu Henri, Matthew Mastracci, Matthew Raymond, Matthew Thomas, Mattias Waldau, Max Romantschuk, Michael 'Ratt' Iannarelli, Michael A. Nachbaur, Michael A. Puls II, Michael Gratton, Michael Powers, Michel Fortin, Michiel van der Blonk, Mihai Şucan, Mike Brown, Mike Dierken, Mike Dixon, Mike Schinkel, Mike Shaver, Mikko Rantalainen, Neil Deakin, Olav Junker Kjær, Peter Kasting, Philip Taylor, Rachid Finge, Rajas Moonka, Ralph Giles, Rimantas Liubertas, Robert O'Callahan, Robert Sayre, Roman Ivanov, S. Mike Dierken, Sam Ruby, Sam Weinig, Scott Hess, Sean Knapp, Shaun Inman, Silvia Pfeiffer, Simon Pieters, Stefan Haustein, Stephen Ma, Steve Runyon, Steven Garrity, Stewart Brodie, Stuart

Parmenter, Tantek Çelik, Thomas Broyer, Thomas O'Connor, Tim Altman, Tyler Close, Vladimir Vukićević, Wakaba, William Swanson, Øistein E. Andersen, and everyone on the WHATWG mailing list for their useful and substantial comments.

Special thanks to Richard Williamson for creating the first implementation of `canvas` in Safari, from which the canvas feature was designed.

Special thanks also to the Microsoft employees who first implemented the event-based drag-and-drop mechanism, `contenteditable`, and other features first widely deployed by the Windows Internet Explorer browser.

Special thanks and $10,000 to David Hyatt who came up with a broken implementation of the adoption agency algorithm that the editor had to reverse engineer and fix before using it in the parsing section.

Thanks also the Microsoft blogging community for some ideas, to the attendees of the W3C Workshop on Web Applications and Compound Documents for inspiration, and to the #mrt crew, the #mrt.no crew, and the cabal for their ideas and support.