# Web Applications 1.0
## Working Draft — 23 March 2007

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!

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

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

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

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

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

## Stability

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

Known issues are usually marked like this. There are some spec-wide issues that have not yet been addressed: case-sensitivity is a very poorly handled topic right now

## 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.

## 1.2. Structure of this specification

*This section is non-normative.*

This specification is divided into the following important sections:

**The DOM (page 25)**
The DOM, or Document Object Model, provides a base for the rest of the specification.

**The Semantics (page 47)**
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 (page 257)**
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 **(page 315)**: HTML documents can provide a number of mechanisms for users to modify content, which are described in this section.
The Communication APIs **(page 341)**: 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.
Miscellaneous APIs **(page 363)**: APIs that could not be placed into other parts of the specification live here.

**The Language Syntax (page 365)**
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 **(page 437)**, rendering rules **(page 439)** for Web browsers, and listing areas that are out of scope **(page 441)** for this specification.

### 1.2.1. How to read this specification

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

## 1.3. Conformance requirements

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

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

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

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

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

**Web browsers and other interactive user agents**

Web browsers that support XHTML **(page 18)** must process elements and attributes from the HTML namespace **(page 434)** found in XML documents **(page 25)** 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` **(page 210)** 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` **(page 210)** element as an opaque element that forms part of the transform.

Web browsers that support HTML **(page 18)** 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* (page 16)*.***

> 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 **(page 266)**) 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` **(page 96)** element is not a quote, conformance checkers do not have to check that `blockquote` **(page 96)** elements only contain quoted material).

Conformance checkers must check that the input document conforms when scripting is disabled **(page 266)**, and should also check that the input document conforms when scripting is enabled **(page 266)**. (This is only a "SHOULD" and not a "MUST" requirement because it has been proven to be impossible. [HALTINGPROBLEM])

> ***The term "validation" specifically refers to a subset of conformance checking that only verifies that a document complies with the requirements given by an SGML or XML DTD. Conformance checkers that only perform validation are non-conforming, as there are many conformance requirements described in this specification that cannot be checked by SGML or XML DTDs.***
>
> ***To put it another way, there are three types of conformance criteria:***
>
> 1. ***Criteria that can be expressed in a DTD.***
> 2. ***Criteria that cannot be expressed by a DTD, but can still be checked by a machine.***
> 3. ***Criteria that can only be checked by a human.***
>
> ***A conformance checker must check for the first two. A simple DTD-based validator only checks for the first class of errors and is therefore not a conforming conformance checker according to this specification.***

**Data mining tools**

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

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

**Authoring tools and markup generators**

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

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` **(page 101)** 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 **(page 437)**.

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 **(page 373)** inspired by SGML (referred to as **HTML5**). Implementations may support only one of these two formats, although supporting both is encouraged.

XHTML **(page 18)** documents (XML documents **(page 25)** using elements from the HTML namespace **(page 434)**) that use the new features described in this specification and that are served over the wire (e.g. by HTTP) must be sent using an XML MIME type such as `application/xml` or `application/xhtml+xml` and must not be served as `text/html`. [RFC3023]

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

HTML documents **(page 18)**, if they are served over the wire (e.g. by HTTP) must be labelled with the `text/html` MIME type.

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

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

### 1.3.1. Dependencies

This specification relies on several other underlying specifications.

**XML**

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

**XML Base**

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

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

**DOM**

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

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

**ECMAScript**

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

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

### 1.3.2. Features defined in other specifications

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

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

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

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

*This section is non-normative.*

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

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

### 1.3.4. Relationship to XHTML2

*This section is non-normative.*

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

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

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

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

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

*This section is non-normative.*

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

## 1.4. Terminology

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

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

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

The term HTML documents **(page 25)** is sometimes used in contrast with XML documents **(page 25)** to mean specifically documents that were parsed using an HTML parser **(page 373)** (as opposed to using an XML parser or created purely through the DOM).

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

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

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

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

An element is said to have been **inserted into a document** when its root element **(page 21)** changes and is now the document's root element **(page 21)**.

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

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

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

| | |
|---|---|
| **`xml`** | `http://www.w3.org/XML/1998/namespace` |
| **`html`** | `http://www.w3.org/1999/xhtml` |
| **`svg`** | `http://www.w3.org/2000/svg` |

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

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

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

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

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

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

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

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

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

### 1.4.1. HTML vs XHTML

*This section is non-normative.*

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

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

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

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

The second concrete syntax uses XML, and is known as "XHTML5". When a document is transmitted with an XML MIME type, such as `application/`

`xhtml+xml`, then it is processed by an XML processor by Web browsers, and treated as an "XHTML5" document. Generally speaking, authors are discouraged from trying to use XML on the Web, because XML has much stricter syntax rules than the "HTML5" variant described above, and is relatively newer and therefore less mature.

The "DOM5 HTML", "HTML5", and "XHTML5" representations cannot all represent the same content. For example, namespaces cannot be represented using "HTML5", but they are supported in "DOM5 HTML" and "XHTML5". Similarly, documents that use the `noscript` **(page 215)** feature can be represented using "HTML5", but cannot be represented with "XHTML5" and "DOM5 HTML". Comments that contain the string "`-->`" can be represented in "DOM5 HTML" but not in "HTML5" and "XHTML5". And so forth.

# 2. The Document Object Model

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

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

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

> For example, a `p` **(page 108)** element represents a paragraph.

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

## 2.1. Documents

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

`Document` objects are assumed to be **XML documents** unless they are flagged as being **HTML documents** when they are created. Whether a document is an HTML document **(page 25)** or an XML document **(page 25)** 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* (page 25)*, but can be made into an HTML document* (page 25) *by calling `document.open()`* (page 37) *on it.***

All `Document` objects (in user agents implementing this specification) must also implement the `HTMLDocument` **(page 25)** interface, available using binding-specific methods. (This is the case whether or not the document in question is an HTML document **(page 25)** or indeed whether it contains any HTML elements **(page 20)** 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` **(page 25)** and `SVGDocument`.

***Note: Because the `HTMLDocument`* (page 25) *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 (page 27)
  readonly attribute Location location (page 292);
  readonly attribute DOMString URL (page 27);
           attribute DOMString domain (page 27);
  readonly attribute DOMString referrer (page 27);
           attribute DOMString cookie (page 27);

  // DOM tree accessors (page 35)
           attribute DOMString title (page 35);
           attribute HTMLElement (page 27) body (page 35);
  readonly attribute HTMLCollection (page 31) images (page 36);
  readonly attribute HTMLCollection (page 31) links (page 36);
  readonly attribute HTMLCollection (page 31) forms (page 36);
  readonly attribute HTMLCollection (page 31) anchors (page 36);
  NodeList getElementsByName (page 36)(in DOMString elementName);
  NodeList getElementsByClassName (page 36)(in DOMString[] classNames);

  // Dynamic markup insertion (page 37)
           attribute DOMString innerHTML (page 37);
  void open (page 37)();
  void open (page 37)(in DOMString type);
  void open (page 37)(in DOMString type, in DOMString replace);
  void open (page 37)(in DOMString url, in DOMString name, in DOMString
features);
  void open (page 37)(in DOMString url, in DOMString name, in DOMString
features, in bool replace);
  void close (page 39)();
  void write (page 37)(in DOMString text);
  void writeln (page 37)(in DOMString text);

  // Interaction (page 78)
  readonly attribute Element activeElement (page 78);
  readonly attribute boolean hasFocus (page 78);

  // Commands (page 249)
  readonly attribute HTMLCollection (page 31) commands (page 251);

  // Editing (page 315)
           attribute boolean designMode (page 318);
  boolean execCommand (page 334)(in DOMString commandID);
  boolean execCommand (page 334)(in DOMString commandID, in boolean
doShowUI);
  boolean execCommand (page 334)(in DOMString commandID, in boolean
doShowUI, in DOMString value);
  Selection (page 337) getSelection (page 337)();

  // Cross-document messaging (page 360)
  void postMessage (page 360)(in DOMString message);

};
```

***Note: This specification requires that implementations also implement some version of the Window object specification, so all HTMLDocument*** **(page 25)** ***objects also implement the DocumentWindow object and thus the DocumentView object.***

Since the `HTMLDocument` **(page 25)** interface holds methods and attributes related to a number of disparate features, the members of this interface are described in various different sections.

### 2.1.1. Resource metadata management

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

The **domain** attribute must be initialised to the document's domain upon the creation of the Document object. On getting, the attribute must return its current value. On setting, if the new value is an allowed value (as defined below), the attribute's value must be changed to the new value. If the new value is not an allowed value, then a security exception **(page 267)** must be raised instead.

A new value is an allowed value for the document.domain **(page 27)** attribute if it is equal to the attribute's current value, or if the new value, prefixed by a U+002E FULL STOP ("."), exactly matches the end of the current value.

> ***Note: The domain* (page 27) *attribute is used to enable pages on different hosts of a domain to access each others' DOMs.***

The **referrer** attribute must return either the URI of the page which navigated **(page 257)** the browsing context **(page 19)** to the current document (if any), or the empty string (if there is no such originating page, or if the UA has been configured not to report referrers).

> ***Note: In the case of HTTP, the referrer* (page 27) *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 **(page 27)** attribute must cause the user agent to act as it would when processing cookies if it had just attempted to fetch the document's address over HTTP, and had received a response with a Set-Cookie header whose value was the specified value, as per RFC 2109 sections 4.3.1, 4.3.2, and 4.3.3. [RFC2109]

## 2.2. Elements

The nodes representing HTML elements **(page 20)** in the DOM must implement, and expose to scripts, the interfaces listed for them in the relevant sections of this specification. This includes XHTML **(page 18)** elements in XML documents **(page 25)**, even when those documents are in another context (e.g. inside an XSLT transform).

The basic interface, from which all the HTML elements **(page 20)**' interfaces inherit, and which must be used by elements that have no additional requirements, is the HTMLElement **(page 27)** interface.

```
interface HTMLElement : Element {
  // DOM tree accessors (page 35)
  NodeList getElementsByClassName (page 37)(in DOMString[] classNames);
  // Dynamic markup insertion (page 37)
           attribute DOMString innerHTML (page 37);

  // Metadata attributes
           attribute DOMString id (page 72);
           attribute DOMString title (page 72);
           attribute DOMString lang (page 73);
           attribute DOMString dir (page 73);
           attribute DOMString className (page 74);
  readonly attribute DOMTokenList (page 33) classList (page 74);

  // Interaction (page 78)
           attribute long tabindex (page 79);
  void click (page 78)();
  void focus (page 78)();
  void blur (page 78)();

  // Commands (page 249)
           attribute HTMLMenuElement contextMenu (page 248);

  // Editing (page 315)
           attribute boolean draggable (page 328);
           attribute DOMString contenteditable (page 315);

  // event handler DOM attributes (page 270)
           attribute EventListener onabort (page 270);
           attribute EventListener onbeforeunload (page 270);
           attribute EventListener onblur (page 270);
           attribute EventListener onchange (page 270);
           attribute EventListener onclick (page 270);
           attribute EventListener oncontextmenu (page 270);
           attribute EventListener ondblclick (page 270);
           attribute EventListener ondrag (page 270);
           attribute EventListener ondragend (page 270);
           attribute EventListener ondragenter (page 270);
           attribute EventListener ondragleave (page 270);
           attribute EventListener ondragover (page 271);
           attribute EventListener ondragstart (page 271);
           attribute EventListener ondrop (page 271);
           attribute EventListener onerror (page 271);
           attribute EventListener onfocus (page 271);
           attribute EventListener onkeydown (page 271);
           attribute EventListener onkeypress (page 271);
           attribute EventListener onkeyup (page 271);
           attribute EventListener onload (page 271);
           attribute EventListener onmessage (page 271);
           attribute EventListener onmousedown (page 271);
           attribute EventListener onmousemove (page 271);
           attribute EventListener onmouseout (page 271);
           attribute EventListener onmouseover (page 272);
           attribute EventListener onmouseup (page 272);
           attribute EventListener onmousewheel (page 272);
           attribute EventListener onresize (page 272);
           attribute EventListener onscroll (page 272);
           attribute EventListener onselect (page 272);
           attribute EventListener onsubmit (page 272);
           attribute EventListener onunload (page 272);

};
```

As with the `HTMLDocument` **(page 25)** interface, the `HTMLElement` **(page 27)** interface holds methods and attributes related to a number of disparate features, and the members of this interface are therefore described in various different sections of this specification.

### 2.2.1. Reflecting content attributes in DOM attributes

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

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

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

If a reflecting DOM attribute is a boolean attribute, then the DOM attribute must return true if the attribute is set, and false if it is absent. On setting, the content attribute must be removed if the DOM attribute is set to false, and must be set to have the same value as its name if the DOM attribute is set to true. (This corresponds to the rules for boolean content attributes **(page 48)**.)

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 **(page 49)** 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 **(page 48)** 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 **(page 48)** 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 **(page 48)** in base ten and then that string must be used as the new content attribute value.

If a reflecting DOM attribute is of the type `DOMTokenList` **(page 33)**, then on getting it must return a `DOMTokenList` **(page 33)** object whose underlying string is the element's corresponding content attribute. When the `DOMTokenList` **(page 33)** object mutates its underlying string, the attribute must itself be immediately mutated. When the attribute is absent, then the string represented by the `DOMTokenList` **(page 33)** 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` **(page 33)** attributes are always read-only. The same `DOMTokenList` **(page 33)** object must be returned every time for each attribute.

If a reflecting DOM attribute has the type `HTMLElement` **(page 27)**, or an interface that descends from `HTMLElement` **(page 27)**, 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` **(page 71)** attribute, then the content attribute must be set to the value of that `id` **(page 71)** attribute. Otherwise, the DOM attribute must be set to the empty string.

## 2.3. Common DOM interfaces

### 2.3.1. Collections

The `HTMLCollection` **(page 31)**, `HTMLFormControlsCollection` **(page 31)**, and `HTMLOptionsCollection` **(page 32)** interfaces represent various lists of DOM nodes. Collectively, objects implementing these interfaces are called **collections**.

When a collection **(page 30)** is created, a filter and a root are associated with the collection.

> For example, when the `HTMLCollection` **(page 31)** object for the `document.images` **(page 36)** attribute is created, it is associated with a filter that selects only `img` **(page 148)** elements, and rooted at the root of the document.

The collection then **represents** a live **(page 22)** 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 **(page 21)**.

***Note: The `rows`* (page 193) *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` **(page 31)** interface represents a generic collection of elements.

```
interface HTMLCollection {
  readonly attribute unsigned long length (page 31);
  Element item (page 31)(in unsigned long index);
  Element namedItem (page 31)(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` **(page 118)**, `applet`, `area` **(page 186)**, `form`, `img` **(page 148)**, or `object` **(page 153)** element with a `name` attribute equal to *key*, or,
- It is an HTML element of any kind with an `id` **(page 71)** attribute equal to *key*. (Non-HTML elements, even if they have IDs, are not searched for the purposes of `namedItem()` **(page 31)**.)

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

In ECMAScript implementations, objects that implement the `HTMLCollection` **(page 31)** interface must also have a [[Get]] method that, when invoked with a property name that is a number, acts like the `item()` **(page 31)** method would when invoked with that argument, and when invoked with a property name that is a string, acts like the `namedItem()` **(page 31)** method would when invoked with that argument.

### *2.3.1.2. HTMLFormControlsCollection*

The `HTMLFormControlsCollection` **(page 31)** interface represents a collection of form controls.

```
interface HTMLFormControlsCollection {
  readonly attribute unsigned long length (page 31);
  HTMLElement (page 27) item (page 31)(in unsigned long index);
  Object namedItem (page 31)(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` **(page 71)** 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` **(page 71)** 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` **(page 31)** object, further filtered so that the only nodes in the `NodeList` object are those that have either an `id` **(page 71)** attribute or a `name` attribute equal to *key*. The nodes in the `NodeList` object must be sorted in tree order **(page 21)**.
4. Return that `NodeList` object.

In the ECMAScript DOM binding, objects implementing the `HTMLFormControlsCollection` **(page 31)** interface must support being dereferenced using the square bracket notation, such that dereferencing with an integer index is equivalent to invoking the `item()` **(page 31)** method with that index, and such that dereferencing with a string index is equivalent to invoking the `namedItem()` **(page 31)** method with that index.

### *2.3.1.3. HTMLOptionsCollection*

The `HTMLOptionsCollection` **(page 32)** interface represents a list of `option` elements.

```
interface HTMLOptionsCollection {
           attribute unsigned long length (page 32);
  HTMLOptionElement item (page 33)(in unsigned long index);
  Object namedItem (page 33)(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` **(page 32)** is rooted, where *n* is the difference between the two numbers (new value minus old value). If the new value is lower, then the last *n* nodes in the collection must be removed from their parent nodes, where *n* is the difference between the two numbers (old value minus new value).

> ***Note: Setting `length`* (page 32) *never removes or adds any `optgroup` elements, and never adds new children to existing `optgroup` elements (though it can remove children from them).***

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

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

1. If, at the time the method is called, there is exactly one node in the collection that has either an `id` **(page 71)** 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` **(page 71)** attribute or a `name` attribute equal to *key*, then return null and stop the algorithm.
3. Otherwise, create a `NodeList` object representing a live view of the `HTMLOptionsCollection` **(page 32)** object, further filtered so that the only nodes in the `NodeList` object are those that have either an `id` **(page 71)** attribute or a `name` attribute equal to *key*. The nodes in the `NodeList` object must be sorted in tree order **(page 21)**.
4. Return that `NodeList` object.

In the ECMAScript DOM binding, objects implementing the `HTMLOptionsCollection` **(page 32)** interface must support being dereferenced using the square bracket notation, such that dereferencing with an integer index is equivalent to invoking the `item()` **(page 33)** method with that index, and such that dereferencing with a string index is equivalent to invoking the `namedItem()` **(page 33)** 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` **(page 33)** interface represents an interface to an underlying string that consists of an unordered set of space-separated tokens **(page 62)**.

Which string underlies a particular `DOMTokenList` **(page 33)** object is defined when the object is created. It might be a content attribute (e.g. the string that underlies the `classList` **(page 74)** object is the `class` **(page 73)** attribute), or it might be an anonymous string (e.g. when a `DOMTokenList` **(page 33)** object is passed to an author-implemented callback in the `datagrid` **(page 219)** APIs).

```
interface DOMTokenList {
  boolean has (page 33)(in DOMString token);
  void add (page 34)(in DOMString token);
  void remove(in DOMString token);
};
```

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

1. If the *token* argument contains any spaces, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.
2. Otherwise, split the underlying string on spaces **(page 62)** 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 **(page 62)** 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` **(page 33)** object's underlying string then stop the algorithm.
4. Otherwise, if the last character of the `DOMTokenList` **(page 33)** object's underlying string is not a space character **(page 47)**, 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` **(page 33)** object's underlying string.

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

1. If the *token* argument contains any spaces **(page 47)**, then raise an `INVALID_CHARACTER_ERR` exception and stop the algorithm.
2. Otherwise, remove the given *token* from the underlying string **(page 62)**.

In the ECMAScript DOM binding, objects implementing the `DOMTokenList` **(page 33)** 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` **(page 34)** 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` **(page 79)** element, or null otherwise.

**The `head` element** of a document is the first `head` **(page 80)** element that is a child of the `html` element **(page 35)**, if there is one, or null otherwise.

**The `title` element** of a document is the first `title` **(page 80)** element that is a child of the `head` element **(page 35)**, if there is one, or null otherwise.

The **`title`** attribute must, on getting, return a concatenation of the data of all the child text nodes **(page 22)** of the `title` element **(page 35)**, in tree order, or the empty string if the `title` element **(page 35)** is null.

On setting, the following algorithm must be run:

1. If the `head` element **(page 35)** is null, then the attribute must do nothing. Stop the algorithm here.
2. If the `title` element **(page 35)** is null, then a new `title` **(page 80)** element must be created and appended to the `head` element **(page 35)**.
3. The children of the `title` element **(page 35)** (if any) must all be removed.
4. A single `Text` node whose data is the new value being assigned must be appended to the `title` element **(page 35)**.

**The body element** of a document is the first child of the `html` element **(page 35)** that is either a `body` **(page 94)** 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 **(page 35)** of the document (either a `body` **(page 94)** element, a `frameset` element, or null). On setting, the following algorithm must be followed:

1. If the new value is not a `body` **(page 94)** 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 **(page 35)**, do nothing. Abort these steps.

3. Otherwise, if the body element **(page 35)** 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 **(page 35)** as its two arguments respectively, then abort these steps.

4. Otherwise, the the body element **(page 35)** is null. Append the new value to the root element.

The **`images`** attribute must return an `HTMLCollection` **(page 31)** rooted at the `Document` node, whose filter matches only `img` **(page 148)** elements.

The **`links`** attribute must return an `HTMLCollection` **(page 31)** rooted at the `Document` node, whose filter matches only `a` **(page 118)** elements with `href` **(page 273)** attributes and `area` **(page 186)** elements with `href` **(page 273)** attributes.

The **`forms`** attribute must return an `HTMLCollection` **(page 31)** rooted at the `Document` node, whose filter matches only `form` elements.

The **`anchors`** attribute must return an `HTMLCollection` **(page 31)** rooted at the `Document` node, whose filter matches only `a` **(page 118)** elements with `name` attributes.

The **`getElementsByName(`*`name`*`)`** method a string *name*, and must return a live `NodeList` containing all the `a` **(page 118)**, `applet`, `button`, `form`, `iframe` **(page 150)**, `img` **(page 148)**, `input`, `map` **(page 185)**, `meta` **(page 86)**, `object` **(page 153)**, `select`, and `textarea` elements in that document that have a `name` attribute whose value is equal to the *name* argument.

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

HTML, XHTML, SVG and MathML elements define which classes they are in by having an attribute in the per-element partition with the name `class` containing a space-separated list of classes to which the element belongs. Other specifications may also allow elements in their namespaces to be labelled as being in specific classes. UAs must not assume that all attributes of the name `class` for elements in any namespace work in this way, however, and must not assume that such attributes, when used as global attributes, label other elements as being in specific classes.

> Given the following XHTML fragment:
>
> ```
> <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('ccc bbb')` would return the same thing.
>
> A call to `getElementsByClassName(['aaa bbb'])` would return no nodes; none of the elements above are in the "aaa bbb" class.
>
> A call to `getElementsByClassName([''])` would also return no nodes, since none of the nodes are in the "" class (indeed, in HTML, it is impossible to specify that an element is in the "" class).

The **`getElementsByClassName()`** method on the `HTMLElement` **(page 27)** interface must return the nodes that the `HTMLDocument` **(page 25)** `getElementsByClassName()` **(page 36)** method would return, excluding any elements that are not descendants of the `HTMLElement` **(page 27)** object on which the method was invoked.

## 2.5. Dynamic markup insertion

The `document.write()` **(page 37)** family of methods and the `innerHTML` **(page 37)** family of DOM attributes enable script authors to dynamically insert markup into the document.

bz argues that innerHTML should be called something else on XML documents and XML elements. Is the sanity worth the migration pain?

Because these APIs interact with the parser, their behaviour varies depending on whether they are used with HTML documents **(page 25)** (and the HTML parser **(page 373)**) or XHTML in XML documents **(page 25)** (and the XML parser). The following table cross-references the various versions of these APIs.

| | **`document.write()`** | **`innerHTML`** |
|---|---|---|
| **For documents that are HTML documents (page 25)** | `document.write()` in HTML **(page 39)** | `innerHTML` in HTML **(page 39)** |
| **For documents that are XML documents (page 25)** | `document.write()` in XML **(page 43)** | `innerHTML` in XML **(page 43)** |

Regardless of the parsing mode, the **`document.writeln(`*`s`*`)`** method must call the `document.write()` **(page 37)** method with the same argument *s*, and then call the `document.write()` **(page 37)** 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 **(page 38)**, and the insertion point **(page 382)** associated with that parser's input stream **(page 375)** is not undefined (that is, it *does* point to somewhere in the input stream), then the method does nothing. Abort these steps.

   ***Note: This basically causes `document.open()`*** **(page 37)** ***to be ignored when it's called in an inline script found during the parsing of data sent over the network, while still letting it have an effect when called asynchronously or on a document that is itself being spoon-fed using these APIs.***

4. Otherwise, if the document has an active parser, then stop that parser, and throw away any pending content in the input stream. what about if it doesn't, because it's either like a text/plain, or Atom, or PDF, or XHTML, or image document, or something?
5. Remove all child nodes of the document.
6. Create a new HTML parser **(page 373)** and associate it with the document. This is a **script-created parser** (meaning that it can be closed by the `document.open()` **(page 37)** and `document.close()` **(page 39)** methods, and that the tokeniser will wait for an explicit call to `document.close()` **(page 39)** before emitting an end-of-file token).
7. Mark the document as being an HTML document **(page 25)** (it might already be so-marked).
8. If *type* does not have the value "`text/html`", then act as if the tokeniser had emitted a `pre` **(page 112)** element start tag, then set the HTML parser **(page 373)**'s tokenisation **(page 382)** stage's content model flag **(page 382)** to *PLAINTEXT*.
9. If *replace* is false, then: remove all the entries in the browsing context **(page 19)**'s session history **(page 288)** after the current entry **(page 289)** in its `Document`'s `History` **(page 289)** object, add a new entry whose address is the same as the current entry **(page 289)**'s at the end of the list, and then advance to that page as if the `history.forward()` **(page 291)** method had been invoked.
10. Finally, set the insertion point **(page 382)** to point at just before the end of the input stream **(page 375)** (which at this point will be empty).

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

When called with three or more arguments, the `open()` **(page 37)** method on the `HTMLDocument` **(page 25)** object must call the `open()` **(page 312)** method on the `WindowHTML` **(page 287)** interface of the object returned by the `defaultView` attribute of the `DocumentView` interface of the `HTMLDocument` **(page 25)** object, with the same arguments as the original call to the `open()` **(page 37)** method. If the `defaultView` attribute of the `DocumentView` interface of the `HTMLDocument` **(page 25)** 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 **(page 38)** 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 **(page 382)** at the insertion point **(page 382)** of the parser's input stream **(page 375)**.

### 2.5.2. Dynamic markup insertion in HTML

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

1. If the insertion point **(page 382)** is undefined, the `open()` **(page 37)** method must be called (with no arguments) on the `document` object. The insertion point **(page 382)** will point at just before the end of the (empty) input stream **(page 375)**.
2. The string *s* must be inserted into the input stream **(page 375)** just before the insertion point **(page 382)**.
3. If there is a script that will execute as soon as the parser resumes **(page 214)**, then the method must now return without further processing of the input stream **(page 375)**.
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` **(page 210)** start tag token is emitted by the tokeniser).

   ***Note: If the `document.write()`*** **(page 39)** ***method was called from script executing inline (i.e. executing because the parser parsed a set of `script`*** **(page 210)** ***tags), then this is a reentrant invocation of the parser*** **(page 375)*****.***

5. Finally, the method must return.

In HTML, the **`innerHTML`** DOM attribute of all `HTMLElement` **(page 27)** and `HTMLDocument` **(page 25)** 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` **(page 39)** DOM attribute must return the result of running the following algorithm:

1. Let *s* be a string, and initialise it to the empty string.

2. For each child node *child*, in tree order **(page 21)**, append the appropriate string from the following list to *s*:

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

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

   For each attribute that the element has, append a U+0020 SPACE character, the attribute's name (which again will be all lowercase), a U+003D EQUALS SIGN (=) character, a U+0022 QUOTATION MARK (") character, the attribute's value, escaped as described below **(page 41)**, and a second U+0022 QUOTATION MARK (") character.

   While the exact order of attributes is UA-defined, and may depend on factors such as the order that the attributes were given in the original markup, the sort order must be stable, such that consecutive calls to `innerHTML` **(page 39)** serialise an element's attributes in the same order.

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

   If the child node is an `Element` with a tag name that is one of `area` **(page 186)**, `base` **(page 81)**, `basefont`, `bgsound`, `br` **(page 110)**, `col` **(page 196)**, `embed` **(page 151)**, `frame`, `hr` **(page 109)**, `img` **(page 148)**, `input`, `link` **(page 82)**, `meta` **(page 86)**, `param` **(page 157)**, `spacer`, or `wbr`, then continue on to the next child node at this point.

   Otherwise, append the value of the *child* element's `innerHTML` **(page 39)** DOM attribute (thus recursing into this algorithm for that element), followed by a U+003C LESS-THAN SIGN (<) character, a U+002F SOLIDUS (/) character, the element's tag name again (which is again all lowercase), and finally a U+003E GREATER-THAN SIGN (>) character.

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

   If one of the ancestors of the child node is a `style` **(page 91)**, `script` **(page 210)**, `xmp`, `iframe` **(page 150)**, `noembed`, `noframes`, or `noscript` **(page 215)** 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 **(page 41)**.

   ↪ **If the child node is a `Comment`**

   Append the literal string `<!--` (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS), followed by the value of the *child* node's `data` DOM

attribute, followed by the literal string `-->` (U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS, U+003E GREATER-THAN SIGN).

↪ **If the child node is a `DocumentType`**

Append the literal string `<!DOCTYPE` (U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+0044 LATIN CAPITAL LETTER D, U+004F LATIN CAPITAL LETTER O, U+0043 LATIN CAPITAL LETTER C, U+0054 LATIN CAPITAL LETTER T, U+0059 LATIN CAPITAL LETTER Y, U+0050 LATIN CAPITAL LETTER P, U+0045 LATIN CAPITAL LETTER E), followed by a space (U+0020 SPACE), followed by the value of the *child* node's `name` DOM attribute, followed by the literal string `>` (U+003E GREATER-THAN SIGN).

Other nodes types (e.g. `Attr`) cannot occur as children of elements. If they do, the `innerHTML` **(page 39)** attribute must raise an `INVALID_STATE_ERR` exception.

3. The result of the algorithm is the string *s*.

**Escaping a string** (for the purposes of the algorithm above) consists of replacing any occurances of the "`&`" character by the string "`&amp;`", any occurances of the "`<`" character by the string "`&lt;`", any occurances of the "`>`" character by the string "`&gt;`", and any occurances of the "`"`" character by the string "`&quot;`".

***Note: Entity reference nodes are assumed to be expanded* (page 18) *by the user agent, and are therefore not covered in the algorithm above.***

***Note: If the element's contents are not conformant, it is possible that the roundtripping through `innerHTML`* (page 39) *will not work. For instance, if the element is a `textarea` element to which a `Comment` node has been appended, then assigning `innerHTML`* (page 39) *to itself will result in the comment being displayed in the text field. Similarly, if, as a result of DOM manipulation, the element contains a comment that contains the literal string "`-->`", then when the result of serialising the element is parsed, the comment will be truncated at that point and the rest of the comment will be interpreted as markup. Another example would be making a `script`* (page 210) *element contain a text node with the text string "`</script>`".***

On setting, if the node is a document, the `innerHTML` **(page 39)** DOM attribute must run the following algorithm:

1. Otherwise, if the document has an active parser, then stop that parser, and throw away any pending content in the input stream. what about if it doesn't, because it's either like a text/plain, or Atom, or PDF, or XHTML, or image document, or something?

2. The user agent must remove the children nodes of the `Document` whose `innerHTML` **(page 39)** attribute is being set.

3. The user agent must create a new HTML parser **(page 373)**, in its initial state, and associate it with the `Document` node.

4. The user agent must place into the input stream **(page 375)** for the HTML parser **(page 373)** just created the string being assigned into the `innerHTML` **(page 39)** attribute.

5. The user agent must start the parser and let it run until it has consumed all the characters just inserted into the input stream. (The `Document` node will have been populated with elements and a `load` event will have fired on its body element **(page 35)**.)

Otherwise, if the node is an element, then setting the `innerHTML` **(page 39)** DOM attribute must cause the following algorithm to run instead:

1. The user agent must create a new `Document` node, and mark it as being an HTML document **(page 25)**.

   The user agent must create a new HTML parser **(page 373)**, and associate it with the just created `Document` node.

   > ***Note: Parts marked* `innerHTML` case *in algorithms in the parser section are parts that only occur if the parser was created for the purposes of handling the setting of an element's* `innerHTML` (page 39) *attribute. The algorithms have been annotated with such markings for informational purposes only; such markings have no normative weight. If it is possible for a condition described as an* `innerHTML` *case* (page 42) *to occur even when the parser wasn't created for the purposes of handling an element's* `innerHTML` (page 39) *attribute, then that is an error in the specification.***

2. The user agent must set the HTML parser **(page 373)**'s tokenisation **(page 382)** stage's content model flag **(page 382)** according to the name of the element whose `innerHTML` **(page 39)** attribute is being set, as follows:

   ↪ **If it is a `title` (page 80) or `textarea` element**

   Set the content model flag **(page 382)** to *RCDATA*.

   ↪ **If it is a `style` (page 91), `script` (page 210), `xmp`, `iframe` (page 150), `noembed`, `noframes`, or `noscript` (page 215) element**

   Set the content model flag **(page 382)** to *CDATA*.

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

   Set the content model flag **(page 382)** to *PLAINTEXT*.

   ↪ **Otherwise**

   Set the content model flag **(page 382)** to *PCDATA*.

3. The user agent must switch the HTML parser **(page 373)**'s tree construction **(page 394)** stage to the main phase **(page 396)**.

4. Let *root* be a new `html` **(page 79)** element with no attributes.
5. The user agent must append the element *root* to the `Document` node created above.
6. The user agent must set up the parser's stack of open elements **(page 397)** so that it contains just the single element *root*.
7. The user agent must reset the parser's insertion mode appropriately **(page 401)**.
8. The user agent must set the parser's `form` element pointer **(page 400)** to the nearest node to the element whose `innerHTML` **(page 39)** attribute is being set that is a `form` element (going straight up the ancestor chain, and including the element itself, if it is a `form` element), or, if there is no such `form` element, to null.
9. The user agent must place into the input stream **(page 375)** for the HTML parser **(page 373)** just created the string being assigned into the `innerHTML` **(page 39)** attribute.
10. The user agent must start the parser and let it run until it has consumed all the characters just inserted into the input stream.
11. The user agent must remove the children of the element whose `innerHTML` **(page 39)** attribute is being set.
12. The user agent must move all the child nodes of the *root* element to the element whose `innerHTML` **(page 39)** attribute is being set, preserving their order.

### 2.5.3. Dynamic markup insertion in XML

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

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

On getting, the `innerHTML` **(page 43)** DOM attribute on `HTMLElement` **(page 27)**s and `HTMLDocument` **(page 25)**s, in an XML context, must return a namespace-well-formed XML representation of the element or document. User agents may adjust prefixes and namespace declarations in the serialisation (and indeed might be forced to do so in some cases to obtain namespace-well-formed XML). [XML] [XMLNS]

On setting, if the node is a document, the `innerHTML` **(page 39)** DOM attribute on must run the following algorithm:

1. The user agent must remove the children nodes of the `Document` whose `innerHTML` **(page 43)** attribute is being set.
2. The user agent must create a new XML parser.
3. If the `innerHTML` **(page 43)** attribute is being set on an element, the user agent must feed the parser just created the string corresponding to the start tag of that element, declaring all the namespace prefixes that are in scope on that element

in the DOM, as well as declaring the default namespace (if any) that is in scope on that element in the DOM.

4. The user agent must feed the parser just created the string being assigned into the `innerHTML` **(page 39)** attribute.
5. If the `innerHTML` **(page 43)** attribute is being set on an element, the user agent must feed the parser the string corresponding to the end tag of that element.
6. If the parser found a well-formedness error, the attribute's setter must raise a `SYNTAX_ERR` exception and abort these steps.
7. Otherwise, the user agent must take the children of the document, if the attribute is being set on a `Document` node, or of the document's root element, if the attribute is being set on an `Element` node, and append them to the node whose `innerHTML` **(page 39)** attribute is being set, preserving their order.

## 2.6. APIs in HTML documents

For HTML documents **(page 25)**, and for HTML elements **(page 20)** in HTML documents **(page 25)**, 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 **(page 25)** or to elements that are not in the HTML namespace **(page 434)** despite being in HTML documents **(page 25)**.

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

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

**`Document.createElement()`**

The canonical form of HTML markup is all-lowercase; thus, this method will lowercase the argument before creating the requisite element. Also, the element created must be in the HTML namespace **(page 434)**.

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

**`Element.setAttributeNode()`**

When an `Attr` node is set on an HTML element **(page 20)**, 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 **(page 20)**, 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 **(page 20)**, 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 **(page 434)**, then the new qualified name must be lowercased before the rename takes place.

# 3. Semantics and structure of HTML elements

## 3.1. [TBW] 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 **(page 47)** that are space characters **(page 47)**. The step **skip Zs characters** means that the user agent must collect a sequence of characters **(page 47)** 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. **(page 47)**
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. **(page 47)**
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. **(page 47)**
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` (page 134) and `meter` (page 129) 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 **(page 52)** 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 **(page 51)**, 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 **(page 52)** 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 **(page 51)**, 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 **(page 49)**, to obtain *number*. This step cannot fail (*string* is guarenteed to be a valid floating point number **(page 49)**).
6. Return *number*.

### *3.2.3.5. [TBW] Percentages and dimensions*

**valid non-negative percentages**, **rules for parsing dimension values** (only used by height/width on img, embed, object — maybe they should do the same as canvas, then this wouldn't even be needed)

*3.2.3.6. Lists of integers*

A **valid list of integers** is a number of valid integers **(page 48)** separated by U+002C COMMA characters, with no other characters (e.g. no space characters **(page 47)**). 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 $0 \leq h \leq 23$, the minute values ($m$) in the range $0 \leq m \leq 59$, and the second value ($s$) being in the range $0 \leq h < 60$. [GREGORIAN]

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

> The following are some examples of dates written as valid datetimes **(page 55)**.
>
> **`"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 **(page 55)**. 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 **(page 47)** 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 **(page 47)** 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* **(page 54)**.
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 **(page 47)** 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 **(page 47)** that are either U+0054 LATIN CAPITAL LETTER T characters or space characters **(page 47)**. 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 **(page 47)** 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 **(page 47)** 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 **(page 47)** 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. **(page 47)**

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 **(page 47)** 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 $0 \le timezone_{hours} \le 23$, 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 **(page 47)** 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 $0 \le timezone_{minutes} \le 59$, then fail.

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

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

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

29. Return *time* and *timezone*.

#### *3.2.4.2. 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 **(page 58)**, user agents must use the following algorithm. A date or time string **(page 58)** 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 **(page 47)**; for the "in attributes" variant: skip whitespace **(page 47)**.
5. Collect a sequence of characters **(page 47)** 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 **(page 47)** 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* **(page 54)**.
    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 **(page 47)** 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 **(page 47)**; for the "in attributes" variant: skip whitespace **(page 47)**.
    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 **(page 47)**; for the "in attributes" variant: skip whitespace **(page 47)**.
    19. Collect a sequence of characters **(page 47)** 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 **(page 47)** 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 **(page 47)** 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 **(page 47)**; for the "in attributes" variant: skip whitespace **(page 47)**.

    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 **(page 47)** 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 **(page 47)** 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 **(page 47)**; for the "in attributes" variant: skip whitespace **(page 47)**.

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. Tokens

A **set of space-separated tokens** is a set of zero or more words separated by one or more space characters **(page 47)**, where words consist of any string of one or more characters, none of which are space characters **(page 47)**.

A string containing a set of space-separated tokens **(page 62)** may have leading or trailing space characters **(page 47)**.

An **unordered set of space-separated tokens** is a set of space-separated tokens **(page 62)** where none of the words are duplicated.

An **ordered set of unique space-separated tokens** is a set of space-separated tokens **(page 62)** 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 **(page 47)**
5. While *position* is not past the end of *input*:
    1. Collect a sequence of characters **(page 47)** that are not space characters **(page 47)**.
    2. Add the string collected in the previous step to *tokens*.
    3. Skip whitespace **(page 47)**
6. Return *tokens*.

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

1. Let *input* be the string being modified.
2. Let *token* be the token being removed. It will not contain any space characters **(page 47)**.
3. Let *output* be the output string, initially empty.
4. Let *position* be a pointer into *input*, initially pointing at the start of the string.
5. If *position* is beyond the end of *input*, set the string being modified to *output*, and abort these steps.
6. If the character at *position* is a space character **(page 47)**:
    1. Append the character at *position* to the end of *output*.
    2. Increment *position* so it points at the next character in *input*.
    3. Return to step 5 in the overall set of steps.
7. Otherwise, the character at *position* is the first character of a token. Collect a sequence of characters **(page 47)** that are not space characters **(page 47)**, and let that be *s*.
8. If *s* is exactly equal to *token*, then:
    1. Skip whitespace **(page 47)** (in *input*).
    2. Remove any space characters **(page 47)** 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.***

### 3.2.6. 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` (page 315) 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.***

### 3.2.7. 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` **(page 71)** 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` **(page 71)** 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` **(page 112)** 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` **(page 99)** 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` **(page 134)** 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* (page 22)*, and entity reference nodes are treated as if they were expanded in place* (page 18)*.***

The space characters **(page 47)** 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 **(page 22)** and text nodes **(page 22)** consisting of just sequences of those characters are considered **inter-element whitespace**.

Inter-element whitespace **(page 66)**, comment nodes, and processing instruction nodes must be ignored when establishing whether an element matches its content model or not, and must be ignored when following algorithms that define document and element semantics.

An element *A* is said to be **preceeded or followed** by a second element *B* if *A* and *B* have the same parent node and there are no other element nodes or text nodes (other than inter-element whitespace **(page 66)**) between them.

Authors must only use elements in the HTML namespace **(page 20)** 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` **(page 79)** element as being allowed where subdocument fragments are allowed in a compound document. Together, these two definitions mean that placing an XHTML `html` **(page 79)** element as a child of an SVG `foreignObject` element is conforming.

### 3.3.3. Kinds of elements

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

- Metadata elements **(page 80)**
- Sectioning elements **(page 94)**
- Block-level elements **(page 67)**
- Strictly inline-level content **(page 67)**

- Structured inline-level elements **(page 68)**
- Interactive elements **(page 69)**
- Form control elements

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

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

### *3.3.3.1. Block-level elements*

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

There are several kinds of block-level elements:

- Some can only contain other block-level elements: `blockquote` **(page 96)**, `section` **(page 94)**, `article` **(page 96)**, `header` **(page 99)**.

- Some can only contain inline-level content **(page 67)**: `p` **(page 108)**, `h1` **(page 98)**-`h6` **(page 98)**, `address` **(page 101)**.

- Some can contain either block-level elements or inline-level content **(page 67)** (but not both): `nav` **(page 95)**, `aside` **(page 98)**, `footer` **(page 100)**, `div` **(page 255)**.

- Finally, some have very specific content models: `ul` **(page 113)**, `ol` **(page 112)**, `dl` **(page 115)**, `table` **(page 192)**, `script` **(page 210)**.

There are also elements that seem to be block-level but aren't, such as `body` **(page 94)**, `li` **(page 114)**, `dt` **(page 117)**, `dd` **(page 117)**, and `td` **(page 200)**. These elements are allowed only in specific places, not simply anywhere that block-level elements are allowed.

Some block-level elements play multiple roles. For instance, the `script` **(page 210)** elements is allowed inside `head` **(page 80)** elements and can also be used as inline-level content **(page 67)**. Similarly, the `ul` **(page 113)**, `ol` **(page 112)**, `dl` **(page 115)**, `table` **(page 192)**, and `blockquote` **(page 96)** elements play dual roles as both block-level and inline-level elements.

### *3.3.3.2. Inline-level content*

Inline-level content consists of text and various elements to annotate the text, as well as some embedded content **(page 68)** (such as images or sound clips).

Inline-level content comes in various types:

**Strictly inline-level content**

Text, embedded content **(page 68)**, and elements that annotate the text without introducing structural grouping. For example: `a` **(page 118)**, `meter` **(page 129)**, `img` **(page 148)**. Elements used in contexts allowing only strictly inline-level content must not have any descendants that are anything other than strictly inline-level content.

**Structured inline-level elements**

Block-level elements that can also be used as inline-level content. For example: `ol` **(page 112)**, `blockquote` **(page 96)**, `table` **(page 192)**.

Some elements are defined to have as a content model **significant inline content**. This means that at least one descendant of the element must be significant text **(page 68)** or embedded content **(page 68)**.

Unless an element's content model explicitly states that it must contain significant inline content **(page 68)**, simply having no text nodes **(page 22)** and no elements satisfies an element whose content model is some kind of inline content.

**Significant text**, for the purposes of determining the presence of significant inline content **(page 68)**, consists of any character other than those falling in the Unicode categories Zs, Zl, Zp, Cc, and Cf. [UNICODE]

> The following three paragraphs are non-conforming because their content model is not satisfied (they all count as empty).
>
> ```
> <p></p>
> <p><em>&#x00A0;</em></p>
> <p>
>  <ol>
>   <li></li>
>  </ol>
> </p>
> ```

**Embedded content** consists of elements that introduce content from other resources into the document, for example `img` **(page 148)**. Embedded content elements can have **fallback content**: content that is to be used when the external resource cannot be used (e.g. because it is of an unsupported format). The element definitions state what the fallback is, if any.

#### *3.3.3.3. 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 zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

#### *3.3.3.4. Determining if a particular element contains block-level elements or inline-level content*

Some elements are defined to have content models that allow either block-level elements **(page 67)** or inline-level content **(page 67)**, but not both. For example, the `aside` **(page 98)** and `li` **(page 114)** elements.

To establish whether such an element is being used as a block-level container or as an inline-level container, for example in order to determine if a document conforms to these requirements, user agents must look at the element's child nodes. If any of the child nodes are not allowed in block-level contexts, then the element is being used for inline-level content **(page 67)**. If all the child nodes are allowed in a block-level context, then the element is being used for block-level elements **(page 67)**.

Whenever this search would examine a transparent **(page 68)** element, the element's own child nodes must be examined instead, potentially recursing further if any of those are themselves transparent.

> For instance, in the following (non-conforming) XML fragment, the `li` **(page 114)** element is being used as an inline-level element container, because the `meta` **(page 86)** element is not allowed in a block-level context. (It doesn't matter, for the purposes of determining whether it is an inline-level or block-level context, that the `meta` **(page 86)** element is not allowed in inline-level contexts either.)
>
> ```
> <ol>
>  <li>
>   <p> Hello World </p>
>   <meta title="this is an invalid example"/>
>  </li>
> </ol>
> ```
>
> In the following fragment, the `aside` **(page 98)** element is being used as a block-level container, because even though all the elements it contains could be considered inline-level elements, there are no nodes that can only be considered inline-level.
>
> ```
> <aside>
>  <ol>
>   <li> ... </li>
>  </ol>
>  <ul>
>   <li> ... </li>
>  </ul>
> </aside>
> ```
>
> On the other hand, in the following similar fragment, the `aside` **(page 98)** element is an inline-level container, because the text ("Foo") can only be considered inline-level.
>
> ```
> <aside>
>  <ol>
>   <li> ... </li>
>  </ol>
>  Foo
> </aside>
> ```

##### *3.3.3.5. Interactive elements*

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

Certain elements in HTML can be activated, for instance `a` **(page 118)** 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 **(page 273)** 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 **(page 19)**; 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()` (page 78) 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.***

*3.3.3.6. Paragraphs*

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

Paragraphs can be represented by several elements. The `address` **(page 101)** element always represents a paragraph of contact information for its section, the `aside` **(page 98)**, `nav` **(page 95)**, `footer` **(page 100)**, `li` **(page 114)**, and `dd` **(page 117)** elements represent paragraphs with various specific semantics when they are used as inline-level content containers **(page 68)**, the `figure` **(page 146)** element represents a paragraph in the form of embedded content **(page 68)**, and the `p` **(page 108)** element represents all the other kinds of paragraphs, for which there are no dedicated elements.

## 3.4. Global attributes

The following attributes are common to and may be specified on all HTML elements **(page 20)** (even those not defined in this specification):

**Global attributes:**

`class` **(page 73)**
`contenteditable` **(page 315)**

`contextmenu` **(page 247)**
`dir` **(page 73)**
`draggable` **(page 327)**
`id` **(page 71)**
`lang` **(page 72)**
`tabindex` **(page 78)**
`title` **(page 72)**

In addition, the following event handler content attributes **(page 269)** may be specified on any HTML element:

- `onabort` **(page 270)**
- `onbeforeunload` **(page 270)**
- `onblur` **(page 270)**
- `onchange` **(page 270)**
- `onclick` **(page 270)**
- `oncontextmenu` **(page 270)**
- `ondblclick` **(page 270)**
- `ondrag` **(page 270)**
- `ondragend` **(page 270)**
- `ondragenter` **(page 270)**
- `ondragleave` **(page 270)**
- `ondragover` **(page 271)**
- `ondragstart` **(page 271)**
- `ondrop` **(page 271)**
- `onerror` **(page 271)**
- `onfocus` **(page 271)**
- `onkeydown` **(page 271)**
- `onkeypress` **(page 271)**
- `onkeyup` **(page 271)**
- `onload` **(page 271)**
- `onmessage` **(page 271)**
- `onmousedown` **(page 271)**
- `onmousemove` **(page 271)**
- `onmouseout` **(page 271)**
- `onmouseover` **(page 272)**
- `onmouseup` **(page 272)**
- `onmousewheel` **(page 272)**
- `onresize` **(page 272)**
- `onscroll` **(page 272)**
- `onselect` **(page 272)**
- `onsubmit` **(page 272)**
- `onunload` **(page 272)**

### 3.4.1. The `id` attribute

The `id` **(page 71)** attribute represents its element's unique identifier. The value must be unique in the subtree within which the element finds itself and must contain at least one character.

If the value is not the empty string, user agents must associate the element with the given value (exactly) for the purposes of ID matching within the subtree the element finds itself (e.g. for selectors in CSS or for the `getElementById()` method in the DOM).

Identifiers are opaque strings. Particular meanings should not be derived from the value of the `id` **(page 71)** 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` **(page 71)** attribute.

The **`id`** DOM attribute must reflect **(page 29)** the `id` **(page 71)** content attribute.

### 3.4.2. The `title` attribute

The `title` **(page 72)** 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` **(page 72)** attribute of the nearest ancestor with a `title` **(page 72)** attribute set is also relevant to this element. Setting the attribute overrides this, explicitly stating that the advisory information of any ancestors is not relevant to this element. Setting the attribute to the empty string indicates that the element has no advisory information.

Some elements, such as `link` **(page 82)** and `dfn` **(page 125)**, define additional semantics for the `title` **(page 72)** attribute beyond the semantics described above.

The **`title`** DOM attribute must reflect **(page 29)** the `title` **(page 72)** content attribute.

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

The `lang` **(page 72)** 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` **(page 72)** 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` **(page 72)** attribute may only be used on elements of HTML documents **(page 25)**. Authors must not use the `lang` **(page 72)** attribute in XML documents **(page 25)**.

The `xml:lang` **(page 72)** attribute may only be used on elements of XML documents **(page 25)**. Authors must not use the `xml:lang` **(page 72)** attribute in HTML documents **(page 25)**.

To determine the language of a node, user agents must look at the nearest ancestor element (including the element itself if the node is an element) that has a `lang` **(page 72)** or `xml:lang` **(page 72)** attribute set. That specifies the language of the node.

If both the `xml:lang` **(page 72)** attribute and the `lang` **(page 72)** attribute are set on an element, user agents must use the `xml:lang` **(page 72)** attribute, and the `lang` **(page 72)** attribute must be ignored **(page 21)** for the purposes of determining the element's language.

If no explicit language is given for the root element **(page 21)**, then language information from a higher-level protocol (such as HTTP), if any, must be used as the final fallback language. In the absence of any language information, the default value is unknown (the empty string).

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

The **`lang`** DOM attribute must reflect **(page 29)** the `lang` **(page 72)** content attribute.

### 3.4.4. The `dir` attribute

The `dir` **(page 73)** attribute specifies the element's text directionality. The attribute is an enumerated attribute **(page 63)** 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 must reflect **(page 29)** the `dir` **(page 73)** content attribute.

### 3.4.5. Classes

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

The attribute, if specified, must have a value that is an unordered set of space-separated tokens **(page 62)** representing the various classes that the element belongs to.

The classes that an HTML element has assigned to it consists of all the classes returned when the value of the `class` **(page 73)** attribute is split on spaces **(page 62)**.

> ***Note: Assinging classes to an element affects class matching in selectors in CSS, the `getElementsByClassName()`*** **(page 36)** ***method in the DOM, and other such features.***

Authors may use any value in the `class` **(page 73)** attribute, but are encouraged to use the predefined values defined in this specification where appropriate.

Classes that are not defined in this specification can be defined as extensions **(page 76)**, as described below.

Authors should bear in mind that using the `class` **(page 73)** attribute does not convey any additional meaning to the element unless the classes used have been defined by this specification or registered in the Wiki. There is no semantic difference between an element *with* a class attribute and one *without*. Authors that use classes that are not so defined should make sure, therefore, that their documents make as much sense once all `class` **(page 73)** attributes have been removed as they do with the attributes present. User agents should not derive particular meanings from `class` **(page 73)** attribute values that are neither defined by this specification nor registered in the Wiki.

The **`className`** and **`classList`** DOM attributes must both reflect **(page 29)** the `class` **(page 73)** content attribute.

### *3.4.5.1. Predefined class names*

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

| Class name | Applicable elements | Brief description |
|---|---|---|
| `copyright` **(page 75)** | `p` **(page 108)**, `span` **(page 141)** | Indicates that the element contains the copyright for the document. |
| `error` **(page 75)** | `p` **(page 108)**, `section` **(page 94)**, `span` **(page 141)**, `strong` **(page 123)** | Indicates that the element contains an error message, e.g. in an error message to the user in an interactive application, or an error notification in the report of a conformance checker. |
| `example` **(page 75)** | `aside` **(page 98)**, `figure` **(page 146)**, `p` **(page 108)**, `section` **(page 94)**, `span` **(page 141)** | Indicates that the element contains some sample material or a worked example illustrating the neighboring content. |
| `issue` **(page 75)** | `aside` **(page 98)**, `p` **(page 108)**, `span` **(page 141)** | Indicates that the element contains a point in dispute or a problem with the neighboring content that needs resolving. (Thus, this is typically only found in works-in-progress or drafts, not in completed documents.) |
| `note` **(page 75)** | `aside` **(page 98)**, `p` **(page 108)**, `span` **(page 141)** | Indicates that the element contains an explanatory note or clarification. |
| `search` **(page 76)** | `aside` **(page 98)**, `body` **(page 94)**, `form`, `p` **(page 108)**, `section` **(page 94)**, `span` **(page 141)** | Indicates that the element's primary purpose is to provide the user with an interface to perform a search (e.g. of the site's pages or of a database). |
| `warning` **(page 76)** | `article` **(page 96)**, `aside` **(page 98)**, `body` **(page 94)**, `figure` **(page 146)**, `p` **(page 108)**, `section` **(page 94)**, `span` | Indicates that the element contains a warning or admonition. |

| Class name | Applicable elements | Brief description |
|---|---|---|
| | **(page 141)**, strong **(page 123)** | |

3.4.5.1.1. CLASS NAME "COPYRIGHT"

The copyright **(page 75)** class name indicates that the element contains the copyright for the document.

It must only be used on the following elements: p **(page 108)**, span **(page 141)**

3.4.5.1.2. CLASS NAME "ERROR"

The error **(page 75)** class name indicates that the element contains an error message, e.g. in an error message to the user in an interactive application, or an error notification in the report of a conformance checker.

It must only be used on the following elements: p **(page 108)**, section **(page 94)**, span **(page 141)**, strong **(page 123)**

It cannot be used on the body **(page 94)** element; if the whole document is an error message then the appropriate metadata should be included at the network layer (e.g. with HTTP using the 4xx and 5xx status codes).

3.4.5.1.3. CLASS NAME "EXAMPLE"

The example **(page 75)** class name indicates that the element contains some sample material or a worked example illustrating the neighboring content.

It must only be used on the following elements: aside **(page 98)**, figure **(page 146)**, p **(page 108)**, span **(page 141)**

3.4.5.1.4. CLASS NAME "ISSUE"

The issue **(page 75)** class name indicates that the element contains a point in dispute or a problem with the neighboring content that needs resolving. (Thus, this is typically only found in works-in-progress or drafts, not in completed documents.)

It must only be used on the following elements: aside **(page 98)**, p **(page 108)**, span **(page 141)**

3.4.5.1.5. CLASS NAME "NOTE"

The note **(page 75)** class name indicates that the element contains an explanatory note or clarification.

It must only be used on the following elements: aside **(page 98)**, p **(page 108)**, span **(page 141)**

#### 3.4.5.1.6. Class name "search"

The `search` **(page 76)** class name indicates that the element's primary purpose is to provide the user with an interface to perform a search (e.g. of the site's pages or of a database).

It must only be used on the following elements: `aside` **(page 98)**, `body` **(page 94)**, `form`, `p` **(page 108)**, `section` **(page 94)**, `span` **(page 141)**

#### 3.4.5.1.7. Class name "warning"

The `warning` **(page 76)** class name indicates that the element contains a warning or admonition.

It must only be used on the following elements: `article` **(page 96)**, `aside` **(page 98)**, `body` **(page 94)**, `figure` **(page 146)**, `p` **(page 108)**, `section` **(page 94)**, `span` **(page 141)**, `strong` **(page 123)**

### *3.4.5.2. Other classes*

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

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

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

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

  **Metadata elements**
  : Meaning any element described as being a metadata element **(page 80)**.

  **Sectioning elements**
  : Meaning any element described as being a sectioning element **(page 94)**.

  **Block-level elements**
  : Meaning any element described as being a block-level element **(page 67)**, but only when that element is actually being used **(page 68)** as a block-level element, and not, say, as a structured inline-level element.

  **Strictly inline-level elements**
  : Meaning any element described as being strictly inline-level content **(page 67)**.

  **Structured inline-level elements**
  : Meaning any element described as being a structured inline-level element **(page 68)**.

  **Interactive**
  : Meaning any element described as being an interactive element **(page 69)**.

  **Embedded content elements**
  : Meaning any element described as being embedded content **(page 68)**.

**All elements**
Meaning any element.

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

**Brief description**
A short description of what the keyword's meaning is.

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

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

**Status**
One of the following:

**Proposal**
The keyword has not received wide peer review and approval. Someone has proposed it and is using it.

**Accepted**
The keyword has received wide peer review and approval. It has a specification that unambiguously defines how to handle pages that use the keyword, including when they use them in incorrect ways. Pages should use the keyword to mark up the semantic that it describes.

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

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

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

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

## 3.5. Interaction

### 3.5.1. Activation

The **click()** method must fire a `click` event **(page 273)** at the element, whose default action is the firing of a further `DOMActivate` event at the same element, whose own default action is to go through all the elements the `DOMActivate` event bubbled through (starting at the target node and going towards the `Document` node), looking for an element with an activation behavior **(page 19)**; 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. [WIP] Focus management*

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

Some elements, most notably `area` **(page 186)**, can correspond to more than one distinct focusable area. When such an element is focused using the `focus()` **(page 78)** 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 the root element.

The **`hasFocus`** attribute must return true if the document, one of its nested browsing contexts **(page 19)**, or any element in the document or its browsing contexts currently has the system focus.

This section on the `tabindex` attribute needs to be checked for backwards-compatibility.

#### *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` **(page 78)** attribute, if specified, must have a value that is a valid integer **(page 48)**.

If the attribute is specified, it must be parsed using the rules for parsing integers **(page 49)**. If parsing the value returns an error, the attribute is ignored for the purposes of focus management (as if it wasn't specified).

A positive integer or zero specifies the index of the element in the current scope's tab order. Elements with the same index are sorted in tree order **(page 21)** 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` **(page 78)** 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` **(page 78)** content attribute. If the attribute is not present (or has an invalid value) then the DOM attribute must return the UA's default value for that element, which will be either 0 (for elements in the tab order) or -1 (for elements not in the tab order).

## 3.6. The root element

### 3.6.1. The `html` element

**Contexts in which this element may be used:**

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

**Content model:**

A `head` **(page 80)** element followed by a `body` **(page 94)** element.

**Element-specific attributes:**

None (but see prose).

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `html` **(page 79)** element represents the root of an HTML document.

Though it has absolutely no effect and no meaning, the `html` **(page 79)** element, in HTML documents **(page 25)**, 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 **(page 25)**.

> ***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* (page 373)*, the attribute ends up in the null namespace, not the "`http://www.w3.org/2000/xmlns/`" namespace like namespace declaration attributes in XML do.***

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

## 3.7. Document metadata

Document metadata is represented by **metadata elements** in the document's `head` **(page 80)** element.

### 3.7.1. The `head` element

**Contexts in which this element may be used:**

As the first element in an `html` **(page 79)** element.

**Content model:**

In any order, optionally one `meta` **(page 86)** element with a `charset` **(page 90)** attribute, exactly one `title` **(page 80)** element, optionally one `base` **(page 81)** element, and zero or more other metadata elements **(page 80)** (in particular, `link` **(page 82)**, `meta` **(page 86)**, `style` **(page 91)**, and `script` **(page 210)**).

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `head` **(page 80)** element collects the document's metadata.

### 3.7.2. The `title` element

Metadata element **(page 80)**.

**Contexts in which this element may be used:**

In a `head` **(page 80)** element containing no other `title` **(page 80)** elements.

**Content model:**

Text (for details, see prose).

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `title` **(page 80)** element represents the document's title or name. Authors should use titles that identify their documents even when they are used out of context, for example in a user's history or bookmarks, or in search results. The document's title is often different from its first header, since the first header does not have to stand alone when taken out of context.

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

The `title` **(page 80)** element must not contain any elements.

The string to use as the document's title is given by the `document.title` **(page 35)** DOM attribute. User agents should use the document's title when referring to the document in their user interface.

### 3.7.3. The `base` element

Metadata element **(page 80)**.

**Contexts in which this element may be used:**

In a `head` **(page 80)** element, after the `meta` **(page 86)** element with the `charset` **(page 90)** attribute, if any, but before any other elements, and only if there are no other `base` **(page 81)** elements in the document.

**Content model:**

Empty.

**Element-specific attributes:**

`href` **(page 82)** (HTML only)
`target` **(page 82)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLBaseElement : HTMLElement (page 27) {
            attribute DOMString href (page 82);
            attribute DOMString target (page 82);
};
```

The `base` **(page 81)** 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 **(page 19)** for the purposes of following hyperlinks **(page 274)**.

There must be no more than one `base` **(page 81)** element per document.

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

Authors must not use the `href` attribute on the `base` **(page 81)** element in XML documents **(page 25)**. Authors should instead use the `xml:base` attribute. [XMLBASE]

User agents must use the value of the `href` attribute on the first `base` **(page 81)** element with an `href` attribute in the document as the document entity's base URI for the purposes of section 5.1.1 of RFC 2396 ("Establishing a Base URI": "Base URI within Document Content"). [RFC2396] Note that this base URI from RFC 2396 is referred to by the algorithm given in XML Base, which is a normative part of this specification **(page 18)**.

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

The **`target`** attribute, if specified, must contain a valid browsing context name **(page 313)**. User agents use this name when following hyperlinks **(page 274)**.

The **`href`** and **`target`** DOM attributes must reflect **(page 29)** the content attributes of the same name.

### 3.7.4. The `link` element

Metadata element **(page 80)**.

**Contexts in which this element may be used:**

In a `head` **(page 80)** element.

**Content model:**

Empty.

**Element-specific attributes:**

`href` **(page 83)** (required)
`rel` **(page 83)** (required)
`media` **(page 84)**
`hreflang` **(page 84)**
`type` **(page 84)**
Also, the `title` **(page 85)** attribute has special semantics on this element.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLLinkElement : HTMLElement (page 27) {
           attribute boolean disabled (page 85);
           attribute DOMString href (page 85);
           attribute DOMString rel (page 85);
  readonly attribute DOMTokenList relList (page 85);
           attribute DOMString media (page 85);
           attribute DOMString hreflang (page 85);
           attribute DOMString type (page 85);
};
```

The `LinkStyle` interface must also be implemented by this element, the styling processing model **(page 93)** defines how. [CSSOM]

The `link` **(page 82)** element allows authors to indicate explicit relationships between their document and other resources.

The destination of the link is given by the **`href`** attribute, which must be present and must contain a URI (or IRI). If the `href` **(page 83)** 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 an unordered set of space-separated tokens **(page 62)**. The allowed values and their meanings **(page 276)** are defined in a later section. If the `rel` **(page 83)** 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` **(page 82)** 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 **(page 273)**. The link types section **(page 276)** 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 **(page 274)** created using the `link` **(page 82)** 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` **(page 82)** element in the document:

- The relationship between this document and the resource (given by the `rel` **(page 83)** attribute)
- The title of the resource (given by the `title` **(page 85)** attribute).
- The URI of the resource (given by the `href` **(page 83)** attribute).
- The language of the resource (given by the `hreflang` **(page 84)** attribute).
- The optimum media for the resource (given by the `media` **(page 84)** attribute).

User agents may also include other information, such as the type of the resource (as given by the `type` **(page 84)** 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 **(page 83)** then the `media` **(page 84)** attribute is purely advisory, and describes for which media the document in question was designed.

However, if the link is an external resource link **(page 83)**, then the `media` **(page 84)** 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` **(page 84)** attribute is omitted, is `all`, meaning that by default links apply to all media.

The **`hreflang`** attribute on the `link` **(page 82)** element has the same semantics as the `hreflang` attribute on hyperlink elements **(page 274)**.

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 **(page 83)**, 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` **(page 84)** attribute authoritative — upon fetching the resource, user agents must only use the Content-Type information associated with the resource **(page 265)** to determine its type, not metadata included in the link to the resource.

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

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

The **`title`** attribute gives the title of the link. With one exception, it is purely advisory. The value is text. The exception is for style sheet links, where the `title` **(page 85)** attribute defines alternative style sheet sets **(page 93)**.

***Note: The `title`* (page 85) *attribute on `link`* (page 82) *elements differs from the global `title`* (page 72) *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` **(page 82)** 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` **(page 81)** element or `xml:base` attributes). [RFC2616] [RFC2068]

The DOM attributes **`href`**, **`rel`**, **`media`**, **`hreflang`**, and **`type`** each must reflect **(page 29)** the respective content attributes of the same name.

The DOM attribute **`relList`** must reflect **(page 29)** the `rel` **(page 83)** content attribute.

The DOM attribute **`disabled`** only applies to style sheet links. When the `link` **(page 82)** element defines a style sheet link, then the `disabled` **(page 85)** attribute behaves as defined for the alternative style sheets DOM **(page 93)**. For all other `link` **(page 82)** elements it always return false and does nothing on setting.

### 3.7.5. The `meta` element

Metadata element **(page 80)**.

**Contexts in which this element may be used:**
In a `head` **(page 80)** element.

**Content model:**
Empty.

**Element-specific attributes:**
`name` **(page 86)**
`http-equiv` **(page 88)**
`content` **(page 86)**
`charset` **(page 90)** (HTML **(page 25)** only)

**Predefined classes that apply to this element:**
None.

**DOM interface:**

```
interface HTMLMetaElement : HTMLElement (page 27) {
           attribute DOMString content (page 87);
           attribute DOMString name (page 87);
           attribute DOMString httpEquiv (page 87);
};
```

The `meta` **(page 86)** element represents various kinds of metadata that cannot be expressed using the `title` **(page 80)**, `base` **(page 81)**, `link` **(page 82)**, `style` **(page 91)**, and `script` **(page 210)** elements.

The `meta` **(page 86)** element can represent document-level metadata with the `name` **(page 86)** attribute, pragma directives with the `http-equiv` **(page 88)** 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` **(page 90)** attribute.

Exactly one of the `name` **(page 86)**, `http-equiv` **(page 88)**, and `charset` **(page 90)** attributes must be specified.

If either `name` **(page 86)** or `http-equiv` **(page 88)** is specified, then the `content` **(page 86)** attribute must also be specified. Otherwise, it must be omitted.

The `charset` **(page 90)** attribute may only be specified in HTML documents **(page 18)**, it must not be used in XML documents **(page 18)**. If the `charset` **(page 90)** attribute is specified, the element must be the first element in the `head` element **(page 35)** 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` **(page 86)** element has a **`name`** attribute, it sets document metadata. Document metadata is expressed in terms of name/value pairs, the `name` **(page 86)** attribute on the `meta` **(page 86)** element giving the name, and the `content` **(page 86)** 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` **(page 86)** element has no `content` **(page 86)** attribute, then the value part of the metadata name/value pair is the empty string.

The DOM attributes **`name`** and **`content`** must reflect **(page 29)** the respective content attributes of the same name. The DOM attribute **`httpEquiv`** must reflect the content attribute `http-equiv` **(page 88)**.

### *3.7.5.1. Standard metadata names*

This specification defines a few names for the `name` **(page 86)** attribute of the `meta` **(page 86)** 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 **(page 437)** on the value used with this metadata name.

**dns**
: The value must be an ordered set of unique space-separated tokens **(page 62)**, 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 **(page 62)**), 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` **(page 82)** element, not the `meta` **(page 86)** element.

### *3.7.5.3. Pragma directives*

When the **`http-equiv`** attribute is specified on a `meta` **(page 86)** element, the element is a pragma directive.

The **`http-equiv`** attribute is an enumerated attribute **(page 63)**. 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 **(page 89)** | `refresh` |
| Default style **(page 90)** | `default-style` |

When a `meta` **(page 86)** element is inserted into the document, if its `http-equiv` **(page 88)** 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` **(page 86)** element in the Refresh state **(page 89)** 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` **(page 86)** element has no `content` **(page 86)** 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` **(page 86)** attribute.
4. Let *position* point at the first character of *input*.
5. Skip whitespace **(page 47)**.
6. Collect a sequence of characters **(page 47)** 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 **(page 48)**. 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 **(page 47)** in the range U+0030 DIGIT ZERO to U+0039 DIGIT NINE and U+002E FULL STOP ("`.`"). Ignore any collected characters.
8. Skip whitespace **(page 47)**.
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 **(page 47)**.
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 **(page 47)**.

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 **(page 47)**.
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 **(page 47)** 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` **(page 86)** element.
22. Set a timer so that in *time* seconds, if the user has not canceled the redirect, the user agent navigates **(page 257)** to *url*, with replacement enabled **(page 259)**.

For `meta` **(page 86)** elements in the Refresh state **(page 89)**, the `content` **(page 86)** attribute must have a value consisting either of:

- just a valid non-negative integer **(page 48)**, or
- a valid non-negative integer **(page 48)**, followed by a U+003B SEMICOLON (`;`), followed by one or more space characters **(page 47)**, 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 and establishing the document's character encoding*

The `meta` **(page 86)** element may also be used to provide UAs with character encoding information for HTML **(page 18)** 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:

- When serialised **(page 365)**, the `charset` **(page 90)** attribute and its value must be contained completely in the first 512 bytes of the file.

- The attribute value must be serialised without the use of character entity references of any kind.
- The value must be a valid character encoding name. [IANACHARSET]
- The character encoding name given must be the name of the character encoding used to serialise the file.
- 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.

If the encoding is one of UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, or UTF-32LE, then authors can use a BOM at the start of the file to indicate the character encoding.

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

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

### 3.7.6. The `style` element

Metadata element **(page 80)**.

**Contexts in which this element may be used:**
In a `head` **(page 80)** element.
At the start of `article` **(page 96)**, `aside` **(page 98)**, `div` **(page 255)**, and `section` **(page 94)** elements.

**Content model:**
Depends on the value of the `type` **(page 92)** attribute.

**Element-specific attributes:**
`media` **(page 92)**
`type` **(page 92)**
`scoped` **(page 92)**
Also, the `title` **(page 92)** attribute has special semantics on this element.

**Predefined classes that apply to this element:**
None.

**DOM interface:**

```
interface HTMLStyleElement : HTMLElement (page 27) {
           attribute boolean disabled (page 92);
           attribute DOMString media (page 92);
           attribute DOMString type (page 92);
           attribute boolean scoped (page 92);
};
```

The `LinkStyle` interface must also be implemented by this element, the styling processing model **(page 93)** defines how. [CSSOM]

The `style` **(page 91)** element allows authors to embed style information in their documents. The `style` **(page 91)** element is one of several inputs to the styling processing model **(page 93)**.

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` **(page 92)** attribute is omitted, is `all`, meaning that by default styles apply to all media.

The **`scoped`** attribute is a boolean attribute **(page 48)**. If the attribute is present, then the user agent must only apply the specified style information to the `style` **(page 91)** element's parent element (if any), and that element's child nodes. Otherwise, the specified styles must, if applied, be applied to the entire document.

The **`title`** attribute on `style` **(page 91)** elements defines alternative style sheet sets **(page 93)**. If the `style` **(page 91)** element has no `title` **(page 92)** attribute, then it has no title; the `title` **(page 72)** attribute of ancestors does not apply to the `style` **(page 91)** element.

> ***Note: The `title`* (page 92) *attribute on `style`* (page 91) *elements, like the `title`* (page 85) *attribute on `link`* (page 82) *elements, differs from the global `title`* (page 72) *attribute in that a `style`* (page 91) *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` **(page 91)** element itself is evaluated. For styling languages that consist of pure text, user agents must evaluate `style` **(page 91)** elements by passing the concatenation of the contents of all the text nodes **(page 22)** that are direct children of the `style` **(page 91)** element (not any other nodes such as comments or elements), in tree order **(page 21)**, to the style system. For XML-based styling languages, user agents must pass all the children nodes of the `style` **(page 91)** 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 **(page 29)** the respective content attributes of the same name.

The DOM **`disabled`** attribute behaves as defined for the alternative style sheets DOM **(page 93)**.

### 3.7.7. Styling

The `link` **(page 82)** and `style` **(page 91)** 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` **(page 91)** and `link` **(page 82)** elements implement the `LinkStyle` interface. [CSSOM]

For `style` **(page 91)** 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` **(page 82)** elements that do not represent external resource links that contribute to the styling processing model **(page 283)** (i.e. that do not have a `stylesheet` **(page 283)** keyword in their `rel` **(page 83)** attribute), and `link` **(page 82)** elements whose specified resource has not yet been downloaded, or is not in a supported styling language, must have their `LinkStyle` interface's `sheet` attribute return null.

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

**The content type (`type` DOM attribute)**
: The content type must be the same as the style's specified type. For `style` **(page 91)** elements, this is the same as the `type` **(page 92)** content attribute's value, or `text/css` if that is omitted. For `link` **(page 82)** elements, this is the Content-Type metadata of the specified resource **(page 265)**.

**The location (`href` DOM attribute)**
: For `link` **(page 82)** elements, the location must be the URI given by the element's `href` **(page 83)** content attribute. For `style` **(page 91)** 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` **(page 82)** and `style` **(page 91)** 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

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

Some elements are scoped to their nearest ancestor sectioning element. For example, `address` **(page 101)** elements apply just to their section. For such elements *x*, the elements that apply to a sectioning element *e* are all the *x* elements whose nearest sectioning element is *e*.

### 3.8.1. The `body` element

Sectioning element **(page 94)**.

**Contexts in which this element may be used:**
  As the second element in an `html` **(page 79)** element.

**Content model:**
  Zero or more block-level elements **(page 67)**.

**Element-specific attributes:**
  None.

**Predefined classes that apply to this element:**
  `search` **(page 76)**, `warning` **(page 76)**

**DOM interface:**
  No difference from `HTMLElement` **(page 27)**.

The `body` **(page 94)** element represents the main content of the document.

The `body` **(page 94)** element potentially has a heading. See the section on headings and sections **(page 102)** for further details.

In conforming documents, there is only one `body` **(page 94)** element. The `document.body` **(page 35)** DOM attribute provides scripts with easy access to a document's `body` **(page 94)** element.

> ***Note: Some DOM operations (for example, parts of the drag and drop* (page 319) *model) are defined in terms of "the body element* (page 35)*". This refers to a particular element in the DOM, as per the definition of the term, and not any arbitrary `body`* (page 94) *element.***

### 3.8.2. The `section` element

Sectioning **(page 94)** block-level element **(page 67)**.

**Contexts in which this element may be used:**
  Where block-level elements **(page 67)** are expected.

**Content model:**

Zero or more `style` **(page 91)** elements, followed by zero or more block-level elements **(page 67)**.

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

`error` **(page 75)**, `example` **(page 75)**, `search` **(page 76)**, `warning` **(page 76)**

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `section` **(page 94)** element represents a generic document or application section. A section, in this context, is a thematic grouping of content, typically with a header, possibly with a footer.

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

Each `section` **(page 94)** element potentially has a heading. See the section on headings and sections **(page 102)** for further details.

### 3.8.3. The `nav` element

Sectioning **(page 94)** block-level element **(page 67)**.

**Contexts in which this element may be used:**

Where block-level elements **(page 67)** are expected.

**Content model:**

Zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `nav` **(page 95)** element represents a section of a page that links to other pages or to parts within the page: a section with navigation links.

When used as an inline-level content **(page 68)** container, the element represents a paragraph **(page 70)**.

Each `nav` **(page 95)** element potentially has a heading. See the section on headings and sections **(page 102)** for further details.

### 3.8.4. The `article` element

Sectioning **(page 94)** block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**
Zero or more `style` **(page 91)** elements, followed by zero or more block-level elements **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
`warning` **(page 76)**

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `article` **(page 96)** 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`* (page 96) *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* (page 94) *to a parent `body`* (page 94) *element still covers the `article`* (page 96) *as well.***

When `article` **(page 96)** elements are nested, the inner `article` **(page 96)** 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` **(page 96)** elements nested within the `article` **(page 96)** element for the Web log entry.

Author information associated with an `article` **(page 96)** element (q.v. the `address` **(page 101)** element) does not apply to nested `article` **(page 96)** elements.

Each `article` **(page 96)** element potentially has a heading. See the section on headings and sections **(page 102)** for further details.

### 3.8.5. The `blockquote` element

Sectioning **(page 94)** block-level element **(page 67)**, and structured inline-level element **(page 68)**.

**Contexts in which this element may be used:**

Where block-level elements **(page 67)** are expected.
Where structured inline-level elements **(page 68)** are allowed.

**Content model:**

Zero or more block-level elements **(page 67)**.

**Element-specific attributes:**

`cite` **(page 97)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLQuoteElement : HTMLElement (page 27) {
           attribute DOMString cite (page 97);
};
```

> ***Note: The `HTMLQuoteElement` (page 97) interface is also used by the `q` (page 120) element.***

The `blockquote` **(page 96)** element represents a section that is quoted from another source.

Content inside a `blockquote` **(page 96)** must be quoted from another source, whose URI, if it has one, should be cited in the **`cite`** attribute.

If the `cite` **(page 97)** attribute is present, it must be a URI (or IRI). User agents should allow users to follow such citation links.

If a `blockquote` **(page 96)** element is preceeded or followed **(page 66)** by a `p` **(page 108)** element that contains a single `cite` **(page 121)** element and is itself not preceeded or followed **(page 66)** by another `blockquote` **(page 96)** element and does not itself have a `q` **(page 120)** element descendant, then, the citation given by that `cite` **(page 121)** element gives the source of the quotation contained in the `blockquote` **(page 96)** element.

Each `blockquote` **(page 96)** element potentially has a heading. See the section on headings and sections **(page 102)** for further details.

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

> ***Note: The best way to represent a conversation is not with the `cite` (page 121) and `blockquote` (page 96) elements, but with the `dialog` (page 110) element.***

### 3.8.6. The `aside` element

Sectioning **(page 94)** block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**
Zero or more `style` **(page 91)** elements, followed by either zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
`example` **(page 75)**, `issue` **(page 75)**, `note` **(page 75)**, `search` **(page 76)**, `warning` **(page 76)**

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `aside` **(page 98)** element represents a section of a page that consists of content that is tangentially related to the content around the `aside` **(page 98)** element, and which could be considered separate from that content. Such sections are often represented as sidebars in printed typography.

When used as an inline-level content **(page 68)** container, the element represents a paragraph **(page 70)**.

Each `aside` **(page 98)** element potentially has a heading. See the section on headings and sections **(page 102)** for further details.

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

Block-level elements **(page 67)**.

**Contexts in which these elements may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**
Significant **(page 68)** strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

These elements define headers for their sections.

The semantics and meaning of these elements are defined in the section on headings and sections **(page 102)**.

These elements have a **rank** given by the number in their name. The `h1` **(page 98)** element is said to have the highest rank, the `h6` **(page 98)** element has the lowest rank, and two elements with the same name have equal rank.

These elements must not be empty **(page 68)**.

### 3.8.8. The `header` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected and there are no `header` **(page 99)** ancestors.

**Content model:**
Zero or more block-level elements **(page 67)**, including at least one descendant `h1` **(page 98)**, `h2` **(page 98)**, `h3` **(page 98)**, `h4` **(page 98)**, `h5` **(page 98)**, or `h6` **(page 98)** element, but no sectioning element descendants, no `header` **(page 99)** element descendants, and no `footer` **(page 100)** element descendants.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `header` **(page 99)** element represents the header of a section. Headers may contain more than just the section's heading — for example it would be reasonable for the header to include version history information.

`header` **(page 99)** elements must not contain any `header` **(page 99)** elements, `footer` **(page 100)** elements, or any sectioning elements (such as `section` **(page 94)**) as descendants.

`header` **(page 99)** elements must have at least one `h1` **(page 98)**, `h2` **(page 98)**, `h3` **(page 98)**, `h4` **(page 98)**, `h5` **(page 98)**, or `h6` **(page 98)** element as a descendant.

For the purposes of document summaries, outlines, and the like, `header` **(page 99)** elements are equivalent to the highest ranked **(page 99)** `h1` **(page 98)**-`h6` **(page 98)** element descendant (the first such element if there are multiple elements with that rank **(page 99)**).

Other heading elements indicate subheadings or subtitles.

Here are some examples of valid headers. In each case, the emphasised text represents the text that would be used as the header in an application extracting header data and ignoring subheadings.

```
<header>
 <h1>The reality dysfunction</h1>
 <h2>Space is not the only void</h2>
</header>
<header>
 <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 **(page 102)** defines how `header` **(page 99)** elements are assigned to individual sections.

The rank **(page 99)** of a `header` **(page 99)** element is the same as for an `h1` **(page 98)** element (the highest rank).

### 3.8.9. The `footer` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**
Either zero or more block-level elements **(page 67)**, but with no `h1` **(page 98)**, `h2` **(page 98)**, `h3` **(page 98)**, `h4` **(page 98)**, `h5` **(page 98)**, `h6` **(page 98)**, `header` **(page 99)**, or `footer` **(page 100)** elements as descendants, and with no sectioning elements **(page 94)** as descendants; or, inline-level content **(page 67)** (but not both).

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
: No difference from `HTMLElement` **(page 27)**.

The `footer` **(page 100)** element represents the footer for the section it applies **(page 94)** to. A footer typically contains information about its section such as who wrote it, links to related documents, copyright data, and the like.

`footer` **(page 100)** elements must not contain any `footer` **(page 100)**, `header` **(page 99)**, `h1` **(page 98)**, `h2` **(page 98)**, `h3` **(page 98)**, `h4` **(page 98)**, `h5` **(page 98)**, or `h6` **(page 98)** elements, or any of the sectioning elements (such as `section` **(page 94)**), as descendants.

When used as an inline-level content **(page 68)** container, the element represents a paragraph **(page 70)**.

Contact information for the section given in a `footer` **(page 100)** should be marked up using the `address` **(page 101)** element.

### 3.8.10. The `address` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**
: Where block-level elements **(page 67)** are expected.

**Content model:**
: Inline-level content **(page 67)**.

**Element-specific attributes:**
: None.

**Predefined classes that apply to this element:**
: None.

**DOM interface:**
: No difference from `HTMLElement` **(page 27)**.

The `address` **(page 101)** element represents a paragraph **(page 70)** of contact information for the section it applies **(page 94)** 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` **(page 101)** element must not be used to represent arbitrary addresses (e.g. postal addresses), unless those addresses are contact information for the section. (The p **(page 108)** element is the appropriate element for marking up such addresses.)

The address **(page 101)** element must not contain information other than contact information.

> For example, the following is non-conforming use of the address **(page 101)** element:
>
> ```
> <ADDRESS>Last Modified: 1999/12/24 23:37:50</ADDRESS>
> ```

Typically, the address **(page 101)** element would be included with other information in a footer **(page 100)** element.

To determine the contact information for a sectioning element (such as a document's body **(page 94)** element, which would give the contact information for the page), UAs must collect all the address **(page 101)** elements that apply **(page 94)** to that sectioning element and its ancestor sectioning elements. The contact information is the collection of all the information given by those elements.

> ***Note: Contact information for one sectioning element, e.g. a aside*** **(page 98)** ***element, does not apply to its ancestor elements, e.g. the page's body*** **(page 94)*****.***

### 3.8.11. Headings and sections

The h1 **(page 98)**-h6 **(page 98)** elements and the header **(page 99)** element are headings.

The first heading in a sectioning element gives the header for that section. Subsequent headers of equal or higher rank **(page 99)** start new (implied) sections, headers of lower rank **(page 99)** start subsections that are part of the previous one.

Sectioning elements other than blockquote **(page 96)** are always considered subsections of their nearest ancestor sectioning element, regardless of what implied sections other headings may have created. However, blockquote **(page 96)** elements *are* associated with implied sections. Effectively, blockquote **(page 96)** 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` **(page 94)** section)
    1. Bar (heading starting implied section)
        1. Bla (heading of explicit `blockquote` **(page 96)** section)

        Baz (paragraph)
    2. Quux (heading starting implied section)
    3. Thud (heading of explicit `section` **(page 94)** section)

    Grunt (paragraph)

Notice how the `blockquote` **(page 96)** nests inside an implicit section while the `section` **(page 94)** 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 **(page 99)**, but authors are strongly encouraged to either use only `h1` **(page 98)** elements, or to use elements of the appropriate rank **(page 99)** for the section's nesting level.

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

For example, the following is correct:

```
<body>
 <h4>Apples</h4>
 <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*

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

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

To derive the tree of sections from the document tree, a hypothetical tree is used, consisting of a view of the document tree containing only the `h1` **(page 98)**-`h6` **(page 98)** and `header` **(page 99)** elements, and the sectioning elements other than `blockquote` **(page 96)**. Descendants of `h1` **(page 98)**-`h6` **(page 98)**, `header` **(page 99)**, and `blockquote` **(page 96)** elements must be removed from this view.

The hypothetical tree must be rooted at the root element **(page 21)** or at a sectioning element. In particular, while the sections inside `blockquote` **(page 96)**s do not contribute to the document's tree of sections, `blockquote` **(page 96)**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 **(page 21)** and, for each `h1` **(page 98)**-`h6` **(page 98)** or `header` **(page 99)** element that is not the first element of its parent sectioning element, inserting a new sectioning element, as follows:

↪ **If the element is a `header` (page 99) element, or if it is an `h1` (page 98)-`h6` (page 98) node of rank (page 99) equal to or higher than the first element in the parent sectioning element (assuming that is also an `h1` (page 98)-`h6` (page 98) node), or if the first element of the parent sectioning element is a sectioning element:**

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

↪ **Otherwise:**

> Move the current heading element, and all subsequent siblings up to but excluding the next sectioning element, `header` **(page 99)** element, or `h1` **(page 98)**-`h6` **(page 98)** of equal or higher rank **(page 99)**, whichever comes first, into the new sectioning element, then insert the new sectioning element where the current header was.

The outline is then the resulting hypothetical tree. The ranks **(page 99)** of the headers become irrelevant at this point: each sectioning element in the hypothetical tree contains either no or one heading element child. If there is one, then it gives the section's heading, of there isn't, the section has no heading.

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

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

> Selecting the first section of the document therefore always takes the user to the top of the document, regardless of where the first header in the `body` **(page 94)** 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*

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

1. If the node has an ancestor that is a `header` **(page 99)** element, then the associated heading is the most distant such ancestor. The associated section is that `header` **(page 99)**'s associated section (i.e. repeat this algorithm for that `header` **(page 99)**).

2. If the node has an ancestor that is an `h1` **(page 98)**-`h6` **(page 98)** element, then the associated heading is the most distant such ancestor. The associated section is that heading's section (i.e. repeat this algorithm for that heading element).

3. If the node is an `h1` **(page 98)**-`h6` **(page 98)** element or a `header` **(page 99)** element, then the associated heading is the element itself. The UA must then generate the hypothetical section tree **(page 104)** described in the previous section, rooted at the nearest section ancestor (or the root element **(page 21)** if there is no such ancestor). If the parent of the heading in that hypothetical tree is an element in the real document tree, then that element is the associated section. Otherwise, there is no associated section element.

4. If the node is a sectioning element, then the associated section is itself. The UA must then generate the hypothetical section tree **(page 104)** 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` **(page 98)**-`h6` **(page 98)** element or a `header` **(page 99)** element, then that element is the associated heading. Otherwise, there is no associated heading element.

5. If the node is a `footer` **(page 100)** or `address` **(page 101)** element, then the associated section is the nearest ancestor sectioning element, if there is one. The node's associated heading is the same as that sectioning element's associated heading (i.e. repeat this algorithm for that sectioning element). If there is no ancestor sectioning element, the element has no associated section nor an associated heading.

6. Otherwise, the node is just a normal node, and the document has to be examined more closely to determine its section and heading. Create a view rooted at the nearest ancestor sectioning element (or the root element **(page 21)** if there is none) that has just `h1` **(page 98)**-`h6` **(page 98)** elements, `header` **(page 99)** elements, the node itself, and sectioning elements other than `blockquote` **(page 96)** 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` **(page 98)** or `header` **(page 99)** element, then return that element as the answer.
5. If *n* is pointing at an `h2` **(page 98)**-`h6` **(page 98)** 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` **(page 98)**-`h6` **(page 98)** element, and either *c* is still null, or *c* is a heading of lower rank **(page 99)** than this one, then set *c* to be this element, and continue going backwards through the previous siblings.
7. If *n* is pointing at a sectioning element, then from this point on top-level heading candidates are being searched for. (Specifically, we are looking for the nearest top-level header for the current section.) Continue going backwards through the previous siblings.

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

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

12. Otherwise, if the answer from that same earlier step is an `h1` **(page 98)**-`h6` **(page 98)** element or a `header` **(page 99)** element, then the associated heading is that element and the associated section is that heading element's associated section (i.e. repeat this algorithm for that heading).

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

For the following fragment:

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

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

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

## 3.9. Prose

### 3.9.1. The `p` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**
Significant **(page 68)** inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
`copyright` **(page 75)**, `error` **(page 75)**, `example` **(page 75)**, `issue` **(page 75)**, `note` **(page 75)**, `search` **(page 76)**, `warning` **(page 76)**

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `p` **(page 108)** element represents a paragraph **(page 70)**.

`p` **(page 108)** elements can contain a mixture of strictly inline-level content **(page 67)**, such as text, images, hyperlinks, etc, and structured inline-level elements **(page 68)**, such as lists, tables, and block quotes. `p` **(page 108)** elements must not be empty **(page 68)**.

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` **(page 108)** element should not be used when a more specific element is more appropriate.

The following example is technically correct:

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

However, it would be better marked-up as:

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

Or:

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

### 3.9.2. The `hr` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**
Empty.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `hr` **(page 109)** element represents a paragraph **(page 70)**-level thematic break, e.g. a scene change in a story, or a transition to another topic within a section of a reference book.

### 3.9.3. The `br` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Empty.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `br` **(page 110)** element represents a line break.

`br` **(page 110)** elements must be empty. Any content inside `br` **(page 110)** elements must not be considered part of the surrounding text.

`br` **(page 110)** 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` **(page 110)** element:
>
> ```
> <p>P. Sherman<br>
> 42 Wallaby Way<br>
> Sydney</p>
> ```

`br` **(page 110)** elements must not be used for separating thematic groups in a paragraph.

> The following examples are non-conforming, as they abuse the `br` **(page 110)** 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>
> ```

### 3.9.4. The `dialog` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**

Zero or more pairs of `dt` **(page 117)** and `dd` **(page 117)** elements.

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `dialog` **(page 110)** element represents a conversation.

Each part of the conversation must have an explicit talker (or speaker) given by a `dt` **(page 117)** element, and a discourse (or quote) given by a `dd` **(page 117)** element.

This example demonstrates this using an extract from Abbot and Costello's famous sketch, *Who's on first*:

```
<dialog>
 <dt>Costello
 <dd> Look, you gotta first baseman?
 <dt> Abbott
 <dd> Certainly.
 <dt> Costello
 <dd> Who's playing first?
 <dt> Abbott
 <dd> That's right.
 <dt> Costello
 <dd> When you pay off the first baseman every month, who gets the
money?
 <dt> Abbott
 <dd> Every dollar of it.
</dialog>
```

***Note: Text in a `dt`*** **(page 117)** ***element in a `dialog`*** **(page 110)** ***element is implicitly the source of the text given in the following `dd`*** **(page 117)** ***element, and the contents of the `dd`*** **(page 117)** ***element are implicitly a quote from that speaker. There is thus no need to include `cite`*** **(page 121)*****, `q`*** **(page 120)*****, or `blockquote`*** **(page 96)** ***elements in this markup. Indeed, a `q`*** **(page 120)** ***element inside a `dd`*** **(page 117)** ***element in a conversation would actually imply the person talking were themselves quoting someone else. See the `cite`*** **(page 121)*****, `q`*** **(page 120)*****, and `blockquote`*** **(page 96)** ***elements for other ways to cite or quote.***

## 3.10. Preformatted text

### 3.10.1. The `pre` element

Block-level element **(page 67)**, and structured inline-level element **(page 68)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.
Where structured inline-level elements **(page 68)** are allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `pre` **(page 112)** element represents a block of preformatted text, in which structure is represented by typographic conventions rather than by elements.

Some examples of cases where the `pre` **(page 112)** element could be used:

- Including an e-mail, with paragraphs indicated by blank lines, lists indicated by lines prefixed with a bullet, and so on.
- Including fragments of computer code, with structure indicated according to the conventions of that language.
- Displaying ASCII art.

If, ignoring text nodes **(page 22)** consisting only of whitespace **(page 66)**, the only child of a `pre` **(page 112)** is a `code` **(page 137)** element, then the `pre` **(page 112)** element represents a block of computer code.

If, ignoring text nodes **(page 22)** consisting only of whitespace **(page 66)**, the only child of a `pre` **(page 112)** is a `samp` **(page 138)** element, then the `pre` **(page 112)** element represents a block of computer output.

## 3.11. Lists

### 3.11.1. The `ol` element

Block-level element **(page 67)**, and structured inline-level element **(page 68)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.
Where structured inline-level elements **(page 68)** are allowed.

**Content model:**

Zero or more `li` **(page 114)** elements.

**Element-specific attributes:**

`start` **(page 113)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLOListElement : HTMLElement (page 27) {
           attribute long start (page 113);
};
```

The `ol` **(page 112)** element represents an ordered list of items (which are represented by `li` **(page 114)** elements).

The **`start`** attribute, if present, must be a valid integer **(page 48)** giving the ordinal value of the first list item.

If the `start` **(page 113)** attribute is present, user agents must parse it as an integer **(page 49)**, 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` **(page 114)** element child nodes of the `ol` **(page 112)** element, in tree order **(page 21)**.

The first item in the list has the ordinal value given by the `ol` **(page 112)** element's `start` **(page 113)** attribute, unless that `li` **(page 114)** element has a `value` **(page 115)** attribute with a value that can be successfully parsed, in which case it has the ordinal value given by that `value` **(page 115)** attribute.

Each subsequent item in the list has the ordinal value given by its `value` **(page 115)** 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 **(page 29)** the value of the `start` **(page 113)** content attribute.

### 3.11.2. The `ul` element

Block-level element **(page 67)**, and structured inline-level element **(page 68)**.

**Contexts in which this element may be used:**

Where block-level elements **(page 67)** are expected.
Where structured inline-level elements **(page 68)** are allowed.

**Content model:**

Zero or more `li` **(page 114)** elements.

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `ul` **(page 113)** element represents an unordered list of items (which are represented by `li` **(page 114)** elements).

The items of the list are the `li` **(page 114)** element child nodes of the `ul` **(page 113)** element.

### 3.11.3. The `li` element

**Contexts in which this element may be used:**

Inside `ol` **(page 112)** elements.
Inside `ul` **(page 113)** elements.
Inside `menu` **(page 245)** elements.

**Content model:**

When the element is a child of an `ol` **(page 112)** or `ul` **(page 113)** element and the grandchild of an element that is being used as an inline-level content container **(page 68)**, or, when the element is a child of a `menu` **(page 245)** element: inline-level content **(page 67)**.
Otherwise: zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**

If the element is a child of an `ol` **(page 112)** element: `value` **(page 115)**
If the element is not the child of an `ol` **(page 112)** element: None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLLIElement : HTMLElement (page 27) {
           attribute long value (page 115);
};
```

The `li` **(page 114)** element represents a list item. If its parent element is an `ol` **(page 112)**, `ul` **(page 113)**, or `menu` **(page 245)** 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` **(page 114)** element.

When the list item is the child of an `ol` **(page 112)** or `ul` **(page 113)** element, the content model of the item depends on the way that parent element was used. If it was used as structured inline content (i.e. if *that* element's parent was used as an inline-level content **(page 68)** container), then the `li` **(page 114)** element must only contain inline-level content **(page 67)**. Otherwise, the element may be used either for inline content **(page 67)** or block-level elements **(page 67)**.

When the list item is the child of a `menu` **(page 245)** element, the `li` **(page 114)** element must contain only inline-level content **(page 67)**.

When the list item is not the child of an `ol` **(page 112)**, `ul` **(page 113)**, or `menu` **(page 245)** element, e.g. because it is an orphaned node not in the document, it may contain either for inline content **(page 67)** or block-level elements **(page 67)**.

When used as an inline-level content **(page 68)** container, the list item represents a single paragraph **(page 70)**.

The **`value`** attribute, if present, must be a valid integer **(page 48)** giving the ordinal value of the first list item.

If the `value` **(page 115)** attribute is present, user agents must parse it as an integer **(page 49)**, in order to determine the attribute's value. If the attribute's value cannot be converted to a number, it must be treated as if the attribute was absent. The attribute has no default value.

The `value` **(page 115)** attribute is processed relative to the element's parent `ol` **(page 112)** element (q.v.), if there is one. If there is not, the attribute has no effect.

The **`value`** DOM attribute must reflect **(page 29)** the value of the `value` **(page 115)** content attribute.

### 3.11.4. The `dl` element

Block-level element **(page 67)**, and structured inline-level element **(page 68)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.
Where structured inline-level elements **(page 68)** are allowed.

**Content model:**
Zero or more groups each consisting of one or more `dt` **(page 117)** elements followed by one or mode `dd` **(page 117)** elements.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `dl` **(page 115)** 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` **(page 117)** elements) followed by one or more values (`dd` **(page 117)** 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` **(page 115)** 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` **(page 115)** element is empty, it contains no groups.

If a `dl` **(page 115)** element contains non-whitespace **(page 66)** text nodes **(page 22)**, or elements other than `dt` **(page 117)** and `dd` **(page 117)**, then those elements or text nodes **(page 22)** do not form part of any groups in that `dl` **(page 115)**, and the document is non-conforming.

If a `dl` **(page 115)** element contains only `dt` **(page 117)** elements, then it consists of one group with names but no values, and the document is non-conforming.

If a `dl` **(page 115)** element contains only `dd` **(page 117)** elements, then it consists of one group with values but no names, and the document is non-conforming.

***Note: The `dl`* (page 115) *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`* (page 110) *element.***

### 3.11.5. The `dt` element

**Contexts in which this element may be used:**

Before `dd` **(page 117)** or `dt` **(page 117)** elements inside `dl` **(page 115)** elements.
Before a `dd` **(page 117)** element inside a `dialog` **(page 110)** element.

**Content model:**

Strictly inline-level content **(page 67)**.

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `dt` **(page 117)** element represents the term, or name, part of a term-description group in a description list (`dl` **(page 115)** element), and the talker, or speaker, part of a talker-discourse pair in a conversation (`dialog` **(page 110)** element).

***Note: The `dt`* (page 117) *element itself, when used in a `dl`* (page 115) *element, does not indicate that its contents are a term being defined, but this can be indicated using the `dfn`* (page 125) *element.***

### 3.11.6. The `dd` element

**Contexts in which this element may be used:**

After `dt` **(page 117)** or `dd` **(page 117)** elements inside `dl` **(page 115)** elements.
After a `dt` **(page 117)** element inside a `dialog` **(page 110)** element.

**Content model:**

When the element is a child of a `dl` **(page 115)** element and the grandchild of an element that is being used as an inline-level content container **(page 68)**: inline-level content **(page 67)**.
Otherwise: zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `dd` **(page 117)** element represents the description, definition, or value, part of a term-description group in a description list (`dl` **(page 115)** element), and the discourse, or quote, part in a conversation (`dialog` **(page 110)** element).

The content model of a `dd` **(page 117)** element depends on the way its parent element is being used. If the parent element is a `dl` **(page 115)** element that is being used as structured inline content (i.e. if the `dl` **(page 115)** element's parent element is being used as an inline-level content **(page 68)** container), then the `dd` **(page 117)** element must only contain inline-level content **(page 67)**.

Otherwise, the element may be used either for inline content **(page 67)** or block-level elements **(page 67)**.

## 3.12. Phrase elements

### 3.12.1. The `a` element

Interactive **(page 69)**, strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

Where strictly inline-level content **(page 67)** is allowed, if there are no ancestor interactive elements **(page 69)**.

**Content model:**

When used in an element whose content model is only strictly inline-level content **(page 67)**: only significant **(page 68)** strictly inline-level content **(page 67)**, but there must be no interactive **(page 69)** descendants.
Otherwise: any significant **(page 68)** inline-level content **(page 67)**, but there must be no interactive **(page 69)** descendants.

**Element-specific attributes:**

`href` **(page 273)**
`target` **(page 274)**
`ping` **(page 274)**
`rel` **(page 274)**
`media` **(page 274)**
`hreflang` **(page 274)**
`type` **(page 274)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLAnchorElement : HTMLElement (page 27) {
            attribute DOMString href (page 120);
            attribute DOMString target (page 120);
```

```
           attribute DOMString ping (page 120);
           attribute DOMString rel (page 120);
  readonly attribute DOMTokenList relList (page 120);
           attribute DOMString media (page 120);
           attribute DOMString hreflang (page 120);
           attribute DOMString type (page 120);
};
```

The `Command` **(page 250)** interface must also be implemented by this element.

If the `a` **(page 118)** element has an `href` **(page 273)** attribute, then it represents a hyperlink **(page 273)**.

If the `a` **(page 118)** element has no `href` **(page 273)** attribute, then the element is a placeholder for where a link might otherwise have been placed, if it had been relevant.

The `target` **(page 274)**, `ping` **(page 274)**, `rel` **(page 274)**, `media` **(page 274)**, `hreflang` **(page 274)**, and `type` **(page 274)** attributes must be omitted if the `href` **(page 273)** 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` **(page 118)** 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 **(page 274)** created using the `a` **(page 118)** element. The `href` **(page 273)**, `target` **(page 274)** and `ping` **(page 274)** attributes decide how the link is followed. The `rel` **(page 274)**, `media` **(page 274)**, `hreflang` **(page 274)**, and `type` **(page 274)** 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 **(page 19)** of `a` **(page 118)** elements that represent hyperlinks is to run the following steps:

1. If the `DOMActivate` event in question is not trusted (i.e. a `click()` **(page 78)** method call was the reason for the event being dispatched), and the `a` **(page 118)** element's `target` **(page 274)** attribute is ... then raise an `INVALID_ACCESS_ERR` exception and abort these steps.

2. If the target of the `DOMActivate` event is an `img` **(page 148)** element with an `ismap` **(page 149)** 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` **(page 148)** 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 **(page 274)** defined by the `a` **(page 118)** element. If the steps above defined a *hyperlink suffix* **(page 120)**, 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`*** **(page 118)** ***elements to be clicked, or focussed and activated by the keyboard. This will cause*** **(page 69)** ***the aforementioned activation behavior*** **(page 19)** ***to be invoked.***

The `a` **(page 118)** element must not be empty **(page 68)**.

The DOM attributes **`href`**, **`ping`**, **`target`**, **`rel`**, **`media`**, **`hreflang`**, and **`type`**, must each reflect **(page 29)** the respective content attributes of the same name.

The DOM attribute **`relList`** must reflect **(page 29)** the `rel` **(page 274)** content attribute.

### 3.12.2. The `q` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
When used in an element whose content model is only strictly inline-level content **(page 67)**: only strictly inline-level content **(page 67)**.
Otherwise: any inline-level content **(page 67)**.

**Element-specific attributes:**
`cite` **(page 121)**

**Predefined classes that apply to this element:**
None.

**DOM interface:**
The `q` **(page 120)** element uses the `HTMLQuoteElement` **(page 97)** interface.

The `q` **(page 120)** element represents a part of a paragraph quoted from another source.

Content inside a `q` **(page 120)** element must be quoted from another source, whose URI, if it has one, should be cited in the **`cite`** attribute.

If the `cite` **(page 121)** attribute is present, it must be a URI (or IRI). User agents should allow users to follow such citation links.

If a `q` **(page 120)** element is contained (directly or indirectly) in a paragraph **(page 70)** that contains a single `cite` **(page 121)** element and has no other `q` **(page 120)** element descendants, then, the citation given by that `cite` **(page 121)** element gives the source of the quotation contained in the `q` **(page 120)** element.

### 3.12.3. The `cite` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `cite` **(page 121)** 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`*** **(page 120)** ***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` **(page 121)** ***element can apply to*** `blockquote` **(page 96)** ***and*** `q` **(page 120)** ***elements in certain cases described in the definitions of those elements.***

### 3.12.4. The `em` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
When used in an element whose content model is only strictly inline-level content **(page 67)**: only strictly inline-level content **(page 67)**.
Otherwise: any inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `em` **(page 122)** 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` **(page 122)** 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

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
When used in an element whose content model is only strictly inline-level content **(page 67)**: only strictly inline-level content **(page 67)**.
Otherwise: any inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
`error` **(page 75)**, `warning` **(page 76)**

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `strong` **(page 123)** 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` **(page 123)** elements; each `strong` **(page 123)** element increases the importance of its contents.

Changing the importance of a piece of text with the `strong` **(page 123)** 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

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
When used in an element whose content model is only strictly inline-level content **(page 67)**: only strictly inline-level content **(page 67)**.
Otherwise: any inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `small` **(page 124)** 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`*** **(page 124)** ***element does not "de-emphasise" or lower the importance of text emphasised by the `em`*** **(page 122)** ***element or marked as important with the `strong`*** **(page 123)** ***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` **(page 124)** 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` **(page 124)** 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

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
When used in an element whose content model is only strictly inline-level content **(page 67)**: only strictly inline-level content **(page 67)**.
Otherwise: any inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `m` **(page 125)** 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` **(page 125)** element is highlighting parts of a document that are matching some search string. If someone looked at a document, and the server knew that the user was searching for the word "kitten", then the server might return the document with one paragraph modified as follows:
>
> ```
> <p>I also have some <m>kitten</m>s who are visiting me
> these days. They're really cute. I think they like my garden!</p>
> ```

### 3.12.8. The `dfn` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed, if there are no ancestor `dfn` **(page 125)** elements.

**Content model:**
Strictly inline-level content **(page 67)**, but there must be no descendant `dfn` **(page 125)** elements.

**Element-specific attributes:**
None, but the `title` **(page 126)** attribute has special semantics on this element.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `dfn` **(page 125)** element represents the defining instance of a term. The paragraph **(page 70)**, description list group **(page 115)**, or section **(page 94)** that contains the `dfn` **(page 125)** element contains the definition for the term given by the contents of the `dfn` **(page 125)** element.

`dfn` **(page 125)** elements must not be nested.

**Defining term**: If the `dfn` **(page 125)** 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 **(page 22)**, and that child element is an `abbr` **(page 127)** element with a `title` **(page 127)** attribute, then the exact value of *that* attribute is the term being defined. Otherwise, it is the exact `textContent` **(page 19)** of the `dfn` **(page 125)** element that gives the term being defined.

If the `title` **(page 126)** attribute of the `dfn` **(page 125)** element is present, then it must only contain the term being defined.

There must only be one `dfn` **(page 125)** element per document for each term defined (i.e. there must not be any duplicate terms **(page 126)**).

> ***Note: The `title`* (page 72) *attribute of ancestor elements does not affect `dfn`* (page 125) *elements.***

The `dfn` **(page 125)** element enables automatic cross-references. Specifically, any `span` **(page 141)**, `abbr` **(page 127)**, `code` **(page 137)**, `var` **(page 137)**, `samp` **(page 138)**, or `i` **(page 141)** element that has a non-empty `title` **(page 72)** attribute whose value exactly equals the term **(page 126)** of a `dfn` **(page 125)** element in the same document, or which has no `title` **(page 72)** attribute but whose `textContent` **(page 19)** exactly equals the term **(page 126)** of a `dfn` **(page 125)** element in the document, and that has no interactive elements **(page 69)** or `dfn` **(page 125)** 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` **(page 125)** 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` **(page 127)** element in the second paragraph to the `dfn` **(page 125)** 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

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None, but the `title` **(page 127)** attribute has special semantics on this element.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `abbr` **(page 127)** 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` **(page 127)** 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` **(page 127)** attribute may be omitted if there is a `dfn` **(page 125)** element in the document whose defining term **(page 126)** is the abbreviation (the `textContent` **(page 19)** of the `abbr` **(page 127)** 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` **(page 127)** attribute has been omitted. Because of the way `dfn` **(page 125)** elements are defined, the second `abbr` **(page 127)** 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

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

Strictly inline-level content **(page 67)**.

**Element-specific attributes:**

`datetime` **(page 128)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLTimeElement : HTMLElement (page 27) {
           attribute DOMString datetime (page 128);
  readonly attribute DOMTimeStamp date (page 129);
  readonly attribute DOMTimeStamp time (page 129);
  readonly attribute DOMTimeStamp timezone (page 129);
};
```

The `time` **(page 127)** element represents a date and/or a time.

The **`datetime`** attribute, if present, must contain a date or time string **(page 58)** that identifies the date or time being specified.

If the `datetime` **(page 128)** 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` **(page 19)** according to the rules for parsing date or time strings in content **(page 58)** successfully extracts a date or time.

The **`datetime`** DOM attribute must reflect **(page 29)** the `datetime` **(page 128)** content attribute.

User agents, to obtain the **date**, **time**, and **timezone** represented by a `time` **(page 127)** element, must follow the following steps:

1. If the `datetime` **(page 128)** attribute is present, then parse it according to the rules for parsing date or time strings in content **(page 58)**, and let the result be *result*.
2. Otherwise, parse the element's `textContent` **(page 19)** according to the rules for parsing date or time strings in content **(page 58)**, and let the result be *result*.
3. If *result* is empty (because the parsing failed), then the date **(page 128)** is unknown, the time **(page 128)** is unknown, and the timezone **(page 128)** is unknown.
4. Otherwise: if *result* contains a date, then that is the date **(page 128)**; if *result* contains a time, then that is the time **(page 128)**; and if *result* contains a

timezone, then the timezone is the element's timezone **(page 128)**. (A timezone can only be present if both a date and a time are also present.)

The **`date`** DOM attribute must return null if the date **(page 128)** is unknown, and otherwise must return the time corresponding to midnight UTC (i.e. the first second) of the given date **(page 128)**.

The **`time`** DOM attribute must return null if the time **(page 128)** is unknown, and otherwise must return the time corresponding to the given time **(page 128)** of 1970-01-01, with the timezone UTC.

The **`timezone`** DOM attribute must return null if the timezone **(page 128)** is unknown, and otherwise must return the time corresponding to 1970-01-01 00:00 UTC in the given timezone **(page 128)**, 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 `time` **(page 127)** element's `date` **(page 129)** attribute would have the value 1,158,969,600,000ms, and the `time` **(page 129)** and `timezone` **(page 129)** 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 `time` **(page 127)** element's `date` **(page 129)** attribute would have the value 1,159,056,000,000ms, the `time` **(page 129)** attribute would have the value 18,000,000ms, and the `timezone` **(page 129)** attribute would return -25,200,000ms. To obtain the actual time, the three attributes can be added together, obtaining 1,159,048,800,000, which is the specified date and time in UTC.
>
> Finally, in the following snippet:
>
> ```
> <p>Many people get up at <time>08:00</time>.</p>
> ```
>
> ...the `time` **(page 127)** element's `date` **(page 129)** attribute would have the value null, the `time` **(page 129)** attribute would have the value 28,800,000ms, and the `timezone` **(page 129)** attribute would return null.

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

### 3.12.11. The `meter` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

Strictly inline-level content **(page 67)**.

**Element-specific attributes:**

`value` **(page 130)**
`min` **(page 130)**
`low` **(page 130)**
`high` **(page 130)**
`max` **(page 130)**
`optimum` **(page 130)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLMeterElement : HTMLElement (page 27) {
           attribute long value (page 134);
           attribute long min (page 134);
           attribute long max (page 134);
           attribute long low (page 134);
           attribute long high (page 134);
           attribute long optimum (page 134);
};
```

The `meter` **(page 129)** 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`* (page 129) *element should not be used to indicate progress (as in a progress bar). For that role, HTML provides a separate `progress`* (page 134) *element.***

There are six attributes that determine the semantics of the gauge represented by the element.

The **`min`** attribute specifies the lower bound of the range, and the **`max`** attribute specifies the upper bound. The **`value`** attribute specifies the value to have the gauge indicate as the "measured" value.

The other three attributes can be used to segment the gauge's range into "low", "medium", and "high" parts, and to indicate which part of the gauge is the "optimum" part. The **`low`** attribute specifies the range that is considered to be the "low" part, and the **`high`** attribute specifies the range that is considered to be the "high" part. The **`optimum`** attribute gives the position that is "optimum"; if that is higher than the "high" value then this indicates that the higher the value, the better; if it's lower than the "low" mark then it indicates that lower values are better, and naturally if it is in between then it indicates that neither high nor low values are good.

**Authoring requirements**: The recommended way of giving the value is to include it as contents of the element, either as two numbers (the higher number represents the maximum, the other number the current value), or as a percentage or similar (using one of the characters such as "%"), or as a fraction.

The `value` **(page 130)**, `min` **(page 130)**, `low` **(page 130)**, `high` **(page 130)**, `max` **(page 130)**, and `optimum` **(page 130)** attributes are all optional. When present, they must have values that are valid floating point numbers **(page 49)**.

> The following examples all represent a measurement of three quarters (of the maximum of whatever is being measured):
>
> ```
> <meter>75%</meter>
> <meter>750‰</meter>
> <meter>3/4</meter>
> <meter>6 blocks used (out of 8 total)</meter>
> <meter>max: 100; current: 75</meter>
> <meter><object data="graph75.png">0.75</object></meter>
> <meter min="0" max="100" value="75"></meter>
> ```

**User agent requirements**: User agents must parse the `min` **(page 130)**, `max` **(page 130)**, `value` **(page 130)**, `low` **(page 130)**, `high` **(page 130)**, and `optimum` **(page 130)** attributes using the rules for parsing floating point number values **(page 49)**.

If the `value` **(page 130)** attribute has been omitted, the user agent must also process the `textContent` **(page 19)** of the element according to the steps for finding one or two numbers of a ratio in a string **(page 51)**. 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` **(page 130)** 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` **(page 130)** attribute is specified and a value could be parsed out of it, the maximum value is that value.

Otherwise, if the `max` **(page 130)** 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` **(page 130)** or `value` **(page 130)** attributes *were* specified, then the maximum value is 1.

Otherwise, none of the `max` **(page 130)**, `min` **(page 130)**, and `value` **(page 130)** attributes were specified. If the result of processing the `textContent` **(page 19)** 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 **(page 51)**; and finally, if there were two numbers parsed out of the `textContent` **(page 19)**, 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` **(page 130)** attribute is specified and a value could be parsed out of it, then that value is the actual value.

If the `value` **(page 130)** attribute is not specified but the `max` **(page 130)** attribute *is* specified and the result of processing the `textContent` **(page 19)** of the element was one number with no associated denominator punctuation character, then that number is the actual value.

If neither of the `value` **(page 130)** and `max` **(page 130)** attributes are specified, then, if the result of processing the `textContent` **(page 19)** 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` **(page 19)** 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` **(page 130)** 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` **(page 130)** 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` **(page 130)** 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.

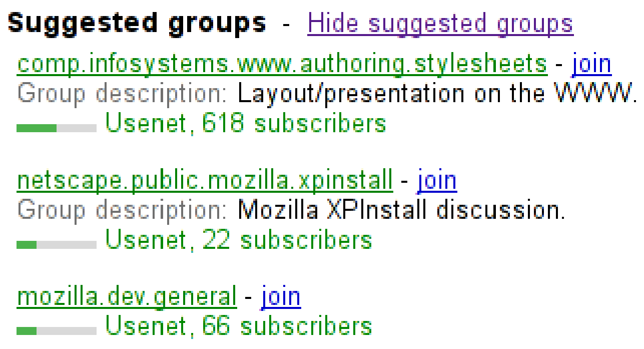
**UA requirements for showing the gauge**: When representing a `meter` **(page 129)** element to the user, the UA should indicate the relative position of the actual value to the minimum and maximum values, and the relationship between the actual value and the three regions of the gauge.

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:


**Suggested groups** - Hide suggested groups

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

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

mozilla.dev.general - join
Usenet, 66 subscribers

The **`min`**, **`max`**, **`value`**, **`low`**, **`high`**, and **`optimum`** DOM attributes must reflect the elements' content attributes of the same name. When the relevant content attributes are absent, the DOM attributes must return zero. The value parsed from the `textContent` **(page 19)** never affects the DOM values.

Would be cool to have the `value` **(page 134)** DOM attribute update the `textContent` **(page 19)** in-line...

### 3.12.12. The `progress` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

Strictly inline-level content **(page 67)**.

**Element-specific attributes:**

`value` **(page 135)**
`max` **(page 135)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLProgressElement : HTMLElement (page 27) {
           attribute float value (page 136);
           attribute float max (page 136);
  readonly attribute float position (page 137);
};
```

The `progress` **(page 134)** 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` **(page 135)** and `value` **(page 135)** attributes, when present, must have values that are valid floating point numbers **(page 49)**. The `max` **(page 135)** attribute, if present, must have a value greater than zero. The `value` **(page 135)** attribute, if present, must have a value equal to or greater than zero, and less than or equal to the value of the `max` **(page 135)** attribute, if present.

**User agent requirements**: User agents must parse the `max` **(page 135)** and `value` **(page 135)** attributes' values according to the rules for parsing floating point number values **(page 49)**.

If the `value` **(page 135)** attribute is omitted, then user agents must also parse the `textContent` **(page 19)** of the `progress` **(page 134)** element in question using the steps for finding one or two numbers of a ratio in a string **(page 51)**. 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` **(page 135)** attribute is omitted, and the `value` **(page 135)** is omitted, and the results of parsing the `textContent` **(page 19)** 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` **(page 135)** 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` **(page 135)** attribute is absent but the `value` **(page 135)** attribute is present, or, if the `max` **(page 135)** 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` **(page 19)** 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` **(page 19)** contained two numbers, the maximum value is the higher of the two values; otherwise, the maximum value is 1.
6. If the `value` **(page 135)** 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` **(page 135)** attribute is absent and the `max` **(page 135)** attribute is present, then, if the `textContent` **(page 19)** 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` **(page 135)** attribute is absent and the `max` **(page 135)** 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` **(page 19)** 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` **(page 134)** 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` **(page 19)** never affects the DOM values.

Would be cool to have the `value` **(page 136)** DOM attribute update the `textContent` **(page 19)** in-line...

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

### 3.12.13. The `code` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
When used in an element whose content model is only strictly inline-level content **(page 67)**: only strictly inline-level content **(page 67)**.
Otherwise: any inline-level content **(page 67)**.

**Element-specific attributes:**
None, but the `title` **(page 72)** attribute has special semantics on this element when used with the `dfn` **(page 125)** element.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `code` **(page 137)** element represents a fragment of computer code. This could be an XML element name, a filename, a computer program, or any other string that a computer would recognise.

***Note: See the `pre`* (page 112) *element for more detais.***

The following example shows how a block of code could be marked up using the `pre` **(page 112)** and `code` **(page 137)** elements.

```
<pre><code>var i: Integer;
begin
   i := 1;
end.</code></pre>
```

### 3.12.14. The `var` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None, but the `title` **(page 72)** attribute has special semantics on this element when used with the `dfn` **(page 125)** element.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `var` **(page 137)** 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

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

When used in an element whose content model is only strictly inline-level content **(page 67)**: only strictly inline-level content **(page 67)**.
Otherwise: any inline-level content **(page 67)**.

**Element-specific attributes:**

None, but the `title` **(page 72)** attribute has special semantics on this element when used with the `dfn` **(page 125)** element.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `samp` **(page 138)** element represents (sample) output from a program or computing system.

***Note: See the `pre`* (page 112) *and `kbd`* (page 139) *elements for more detais.***

> This example shows the `samp` **(page 138)** 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` **(page 138)** and `kbd` **(page 139)** 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

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `kbd` **(page 139)** element represents user input (typically keyboard input, although it may also be used to represent other input, such as voice commands).

When the `kbd` **(page 139)** element is nested inside a `samp` **(page 138)** element, it represents the input as it was echoed by the system.

When the `kbd` **(page 139)** element *contains* a `samp` **(page 138)** element, it represents input based on system output, for example invoking a menu item.

When the `kbd` **(page 139)** element is nested inside another `kbd` **(page 139)** element, it represents an actual key or other single unit of input as appropriate for the input mechanism.

Here the `kbd` **(page 139)** 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` **(page 139)** element marks up a block of input, with the inner `kbd` **(page 139)** elements representing each individual step of the input, and the `samp` **(page 138)** 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 `sup` and `sub` elements

Strictly inline-level content **(page 67)**.

**Contexts in which these elements may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `sup` **(page 140)** element represents a superscript and the `sub` **(page 140)** element represents a subscript.

These elements must only be used to mark up typographical conventions with specific meanings, not for typographical presentation for presentation's sake. For example, it would be inappropriate for the `sup` **(page 140)** and `sub` **(page 140)** elements to be used in the name of the LaTeX document preparation system. In general, authors should not use these elements if the *absence* of those elements would not change the meaning of the content.

When the `sub` **(page 140)** element is used inside a `var` **(page 137)** 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

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
When used in an element whose content model is only strictly inline-level content **(page 67)**: only strictly inline-level content **(page 67)**.
Otherwise: any inline-level content **(page 67)**.

**Element-specific attributes:**
None, but the `title` **(page 72)** attribute has special semantics on this element when used with the `dfn` **(page 125)** element.

**Predefined classes that apply to this element:**
`copyright` **(page 75)**, `error` **(page 75)**, `example` **(page 75)**, `issue` **(page 75)**, `note` **(page 75)**, `search` **(page 76)**, `warning` **(page 76)**

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `span` **(page 141)** element doesn't mean anything on its own, but can be useful when used together with other attributes, e.g. `class` **(page 73)**, `lang` **(page 72)**, or `dir` **(page 73)**, or when used in conjunction with the `dfn` **(page 125)** element.

### 3.12.19. The `i` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None, but the `title` **(page 72)** attribute has special semantics on this element when used with the `dfn` **(page 125)** element.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `i` **(page 141)** 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` **(page 72)** attributes (`xml:lang` **(page 72)** in XML).

The examples below show uses of the `i` **(page 141)** element:

```
<p>The <i>felis silvestris catus</i> is cute.</p>
<p>The <i>block-level elements</i> are defined above.</p>
<p>There is a certain <i lang="fr">je ne sais quoi</i> in the air.</p>
```

In the following example, a dream sequence is marked up using `i` **(page 141)** 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` **(page 141)** element should be used as a last resort when no other element is more appropriate. In particular, citations should use the `cite` **(page 121)** element, defining instances of terms should use the `dfn` **(page 125)** element, stress emphasis should use the `em` **(page 122)** element, importance should be denoted with the `strong` **(page 123)** element, quotes should be marked up with the `q` **(page 120)** element, and small print should use the `small` **(page 124)** element.

***Note: Style sheets can be used to format `i`* (page 141) *elements, just like any other element can be restyled. Thus, it is not the case that content in `i`* (page 141) *elements will necessarily be italicised.***

### 3.12.20. The `b` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `b` **(page 142)** 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` **(page 142)** 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` **(page 123)**, not `b` **(page 142)**.

In the following example, objects in a text adventure are highlighted as being special by use of the `b` **(page 142)** element.

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

The `b` **(page 142)** element should be used as a last resort when no other element is more appropriate. In particular, headers should use the `h1` **(page 98)** to `h6` **(page 98)** elements, stress emphasis should use the `em` **(page 122)** element, importance should be denoted with the `strong` **(page 123)** element, and text marked or highlighted should use the `m` **(page 125)** element.

***Note: Style sheets can be used to format `b`* (page 142) *elements, just like any other element can be restyled. Thus, it is not the case that content in `b`* (page 142) *elements will necessarily be boldened.***

### 3.12.21. The `bdo` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Strictly inline-level content **(page 67)**.

**Element-specific attributes:**
None, but the `dir` **(page 73)** global attribute is required on this element.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `bdo` **(page 143)** element allows authors to override the Unicode bidi algorithm by explicitly specifying a direction override. [BIDI]

Authors must specify the `dir` **(page 73)** 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` **(page 73)** 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` **(page 73)** 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` **(page 143)** 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` **(page 144)** and `del` **(page 145)** elements represent edits to the document.

### 3.13.1. The `ins` element

Transparent **(page 68)** block-level element **(page 67)**, and transparent **(page 68)** strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** is expected.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
Transparent **(page 68)**.

**Element-specific attributes:**
`cite` **(page 146)**
`datetime` **(page 146)**

**Predefined classes that apply to this element:**
None.

**DOM interface:**
Uses the `HTMLModElement` **(page 146)** interface.

The `ins` **(page 144)** element represents an addition to the document.

The `ins` **(page 144)** element must be used only where block-level elements **(page 67)** or strictly inline-level content **(page 67)** can be used.

An `ins` **(page 144)** element can only contain content that would still be conformant if all elements with transparent **(page 68)** content models were replaced by their contents.

The following would be syntactically legal:

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

As would this:

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

However, this last example would be illegal, as `em` **(page 122)** and `p` **(page 108)** cannot both be used inside an `aside` **(page 98)** element at the same time:

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

### 3.13.2. The `del` element

Block-level element **(page 67)**, and strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

Where block-level elements **(page 67)** is expected.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

When the element has a parent: same content model as the parent element (without taking into account the other children of the parent element).
Otherwise: zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**

`cite` **(page 146)**
`datetime` **(page 146)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

Uses the `HTMLModElement` **(page 146)** interface.

The `del` **(page 145)** element represents a removal from the document.

The `del` **(page 145)** element must only contain content that would be allowed inside the parent element (regardless of what the parent element actually contains).

The following would be syntactically legal:

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

...even though the `p` **(page 108)** and `em` **(page 122)** elements would never be allowed side by side in the `aside` **(page 98)** element. This is allowed because the `del` **(page 145)** element represents content that was removed, and it is quite possible that an edit could cause an element to go from being an inline-level container to a block-level container, or vice-versa.

### 3.13.3. Attributes common to `ins` (page 144) and `del` (page 145) 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` **(page 146)** 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` **(page 146)** attribute must be a valid datetime **(page 55)** value.

User agents must parse the `datetime` **(page 146)** attribute according to the parse a string as a datetime value **(page 56)** 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 **(page 55)**). 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` **(page 144)** and `del` **(page 145)** elements must implement the `HTMLModElement` **(page 146)** interface:

```
interface HTMLModElement : HTMLElement (page 27) {
           attribute DOMString cite (page 146);
           attribute DOMString datetime (page 146);
};
```

The **`cite`** and **`datetime`** DOM attributes must reflect the elements' content attributes of the same name.

## 3.14. Embedded content

### 3.14.1. The `figure` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**

Where block-level elements **(page 67)** are expected.

**Content model:**

In any order, exactly one `legend` **(page 254)** element, and exactly one embedded content **(page 68)** element.

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

`example` **(page 75)**, `warning` **(page 76)**

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `figure` **(page 146)** element represents a paragraph **(page 70)** consisting of embedded content and a caption.

The first embedded content **(page 68)** element child of the `figure` **(page 146)** element, if any, is the paragraph's content.

The first `legend` **(page 254)** element child of the element, if any, represents the caption of the embedded content. If there is no child `legend` **(page 254)** element, then there is no caption.

If the embedded content cannot be used, then, for the purposes of establishing what the `figure` **(page 146)** element represents:

↪ **If the embedded content's fallback content (page 68) is a single embedded content (page 68) element**

The `figure` **(page 146)** element must be treated as if that embedded content **(page 68)** element was the `figure` **(page 146)** element's embedded content. (If that embedded content can't be used either, then this processing must be done again, with the new embedded content's fallback content **(page 68)**.)

↪ **If the embedded content's fallback is nothing**

The entire `figure` **(page 146)** element (including the caption, if any) must be ignored.

↪ **If the embedded content's fallback is inline-level content (page 67)**

The entire `figure` **(page 146)** element (including the caption, if any) must be treated as being a single paragraph **(page 70)** with that inline-level content **(page 67)** as its content.

↪ **Otherwise**

The entire `figure` **(page 146)** element (including the caption, if any) must be treated as being replaced by that fallback content.

### 3.14.2. The `img` element

Strictly inline-level **(page 67)** embedded content **(page 68)**.

**Contexts in which this element may be used:**

As the only embedded content **(page 68)** child of a `figure` **(page 146)** element.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

Empty.

**Element-specific attributes:**

`alt` **(page 148)** (required)
`src` **(page 148)** (required)
`usemap` **(page 189)**
`ismap` **(page 149)** (but only if one of the ancestor elements is an `a` **(page 118)** element)
`height` **(page 149)**
`width` **(page 149)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLImageElement : HTMLElement (page 27) {
           attribute DOMString alt (page 150);
           attribute DOMString src (page 150);
           attribute DOMString useMap (page 150);
           attribute boolean isMap (page 150);
           attribute long height (page 150);
           attribute long width (page 150);
  readonly attribute boolean complete (page 150);
};
```

***Note: An instance of `HTMLImageElement`* (page 148) *can be obtained using the `Image`* (page 288) *constructor.***

The `img` **(page 148)** element represents a piece of text with an alternate graphical representation. The text is given by the **`alt`** attribute, which must be present, and the URI to the graphical representation of that text is given in the **`src`** attribute, which must also be present.

The image given by the `src` **(page 148)** attribute is the embedded content, and the value of the `alt` **(page 148)** attribute is the `img` **(page 148)** element's fallback content **(page 68)**.

When the `alt` **(page 148)** attribute's value is the empty string, the image supplements the surrounding content. In such cases, the image could be omitted without affecting the meaning of the document.

If the `alt` **(page 148)** attribute is omitted, user agents must treat the element as if it had an `alt` **(page 148)** attribute set to the empty string.

The `alt` **(page 148)** attribute does not represent advisory information. User agents must not present the contents of the `alt` **(page 148)** attribute in the same way as content of the `title` **(page 72)** attribute.

Guidelines on writing "alt" text here.

The `src` **(page 148)** attribute must contain a URI (or IRI). If the `src` **(page 148)** attribute is omitted, there is no alternative image representation.

When the `src` **(page 148)** 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 **(page 434)**.

Once the download has completed, if the image is a valid image, the user agent must fire a `load` event **(page 273)** on the `img` **(page 148)** 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 **(page 273)** on the `img` **(page 148)** element.

The remote server's response metadata (e.g. an HTTP 404 status code, or associated Content-Type headers **(page 265)**) 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` **(page 148)** element.

The `usemap` **(page 189)** attribute, if present, can indicate that the image has an associated image map **(page 189)**.

The **`ismap`** attribute, when used on an element that is a descendant of an `a` **(page 118)** element with an `href` **(page 273)** 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` **(page 118)** element.

The `ismap` **(page 149)** attribute is a boolean attribute **(page 48)**. The attribute must not be specified on an element that does not have an ancestor `a` **(page 118)** element.

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

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

The values of the `height` **(page 149)** and `width` **(page 149)** attributes must be either valid non-negative integers **(page 48)** or valid non-negative percentages **(page 52)**.

To parse the attributes, user agents must use the rules for parsing dimension values **(page 52)**. This will return either an integer length, a percentage value, or nothing. When one of these attributes has no value, it must be ignored.

The user agent requirements for processing the values obtained from parsing these attributes are described in the rendering section **(page 439)**.

The `img` **(page 148)** element must be empty.

The DOM attributes **`alt`**, **`src`**, **`useMap`**, and **`isMap`** each must reflect **(page 29)** 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` **(page 148)** attribute, and it is a valid image.

### 3.14.3. The `iframe` element

Strictly inline-level **(page 67)** embedded content **(page 68)**.

**Contexts in which this element may be used:**
: As the only embedded content **(page 68)** child of a `figure` **(page 146)** element.
: Where strictly inline-level content **(page 67)** is allowed.

**Content model:**
: Text (for details, see prose).

**Element-specific attributes:**
: `src` **(page 150)**

**Predefined classes that apply to this element:**
: None.

**DOM interface:**

```
interface HTMLIFrameElement : HTMLElement (page 27) {
           attribute DOMString src (page 151);
};
```

Objects implementing the `HTMLIFrameElement` **(page 150)** interface must also implement the `EmbeddingElement` interface defined in the Window Object specification. [WINDOW]

The `iframe` **(page 150)** element introduces a new nested browsing context **(page 19)**.

The **`src`** attribute, if present, must be a URI (or IRI) to a page that the nested browsing context **(page 19)** is to contain. If the user navigates **(page 257)** away from this page, the `iframe` **(page 150)**'s corresponding `Window` object will reference new `Document` objects, but the `src` **(page 150)** attribute will not change.

Whenever the `src` **(page 150)** attribute is set, the nested browsing context **(page 19)** must be navigated **(page 257)** to the given URI.

The default value, which must be used if the `src` **(page 150)** attribute is not set when the element is created, or if the attribute is ever removed, is `about:blank`.

The download of the resource must delay the `load` event **(page 434)**.

When content loads in an `iframe` **(page 150)**, the user agent must fire a `load` event **(page 273)** at the `iframe` **(page 150)** element. When content fails to load (e.g. due to a network error), then the user agent must fire an `error` event **(page 273)** at the element instead.

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

An `iframe` **(page 150)** element never has fallback content **(page 68)**, as it will always create a nested browsing context **(page 19)**, regardless of whether the specified initial contents are successfully used.

`iframe` **(page 150)** elements may contain any text. `iframe` **(page 150)** elements must not contain element nodes. Descendants of `iframe` **(page 150)** elements represent nothing. (In legacy user agents that do not support `iframe` **(page 150)** elements, the contents would be parsed as markup that could act as fallback content.)

restrictions for what that text must be?

> ***Note: The HTML parser* (page 373) *treats markup inside `iframe`* (page 150) *elements as text.***

The DOM attribute **`src`** must reflect **(page 29)** the content attribute of the same name.

### 3.14.4. The `embed` element

Strictly inline-level **(page 67)** embedded content **(page 68)**.

**Contexts in which this element may be used:**

As the only embedded content **(page 68)** child of a `figure` **(page 146)** element.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

Empty.

**Element-specific attributes:**

`src` **(page 152)** (required)
`type` **(page 152)**
`height`
`width`
Any other attribute that has no namespace (see prose).

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLEmbedElement : HTMLElement (page 27) {
           attribute DOMString src (page 153);
           attribute DOMString type (page 153);
           attribute long height (page 153);
           attribute long width (page 153);
};
```

Depending on the type of content instantiated by the `embed` **(page 151)** element, the node may also support other interfaces.

The `embed` **(page 151)** 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` **(page 152)** attribute is missing, then the `embed` **(page 151)** element must be ignored.

When the `src` **(page 152)** attribute is set, user agents are expected to find an appropriate handler for the specified resource, based on the content's type **(page 153)**, and hand that handler the content of the resource. If the handler supports a scriptable interface, the `HTMLEmbedElement` **(page 152)** object representing the element should expose that interfaces.

The download of the resource must delay the `load` event **(page 434)**.

The user agent should pass the names and values of all the attributes of the `embed` **(page 151)** element that have no namespace to the handler used. Any (namespace-less) attribute may be specified on the `embed` **(page 151)** 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` **(page 151)** element has no fallback content **(page 68)**. 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` **(page 152)** attribute, then the value of the `type` **(page 152)** attribute is the content's type.
2. Otherwise, if the specified resource has explicit Content-Type metadata **(page 265)**, then that is the content's type.
3. Otherwise, the content has no type and there can be no appropriate handler for it.

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

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

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

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

height/width

The DOM attributes **`src`** and **`type`** each must reflect **(page 29)** the respective content attributes of the same name.

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

### 3.14.5. The `object` element

Strictly inline-level **(page 67)** embedded content **(page 68)**.

**Contexts in which this element may be used:**

As the only embedded content **(page 68)** child of a `figure` **(page 146)** element.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

When used as the child of a `figure` **(page 146)** element, or, when used as a *`figure`* **(page 146)** *fallback `object`* **(page 153)**: Zero or more `param` **(page 157)** elements, followed by either zero or more block-level elements **(page 67)** or a single `object` **(page 153)** element, which is then considered to be a *`figure`* **(page 146)** *fallback `object`* **(page 153)**.
Otherwise: Zero or more `param` **(page 157)** elements, followed by inline-level content **(page 67)**.

**Element-specific attributes:**

`data` **(page 154)** (required if `type` **(page 154)** is not given)
`type` **(page 154)** (required if `data` **(page 154)** is not given)
`usemap` **(page 189)**
`height`
`width`

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLObjectElement : HTMLElement (page 27) {
          attribute DOMString data (page 157);
          attribute DOMString type (page 157);
          attribute DOMString useMap (page 157);
          attribute long height (page 157);
          attribute long width (page 157);
};
```

Objects implementing the `HTMLObjectElement` **(page 154)** interface must also implement the `EmbeddingElement` interface defined in the Window Object specification. [WINDOW]

Depending on the type of content instantiated by the `object` **(page 153)** element, the node may also support other interfaces.

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

The `object` **(page 153)** 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 **(page 19)**, 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` **(page 154)** and `type` **(page 154)** attributes must be present.

Whenever the `data` **(page 154)** attribute changes, or, if the `data` **(page 154)** attribute is not present, whenever the `type` **(page 154)** attribute changes, the user agent must follow the following steps to determine what the `object` **(page 153)** element represents:

1. If the `data` **(page 154)** attribute is present, then:

1. Begin a load for the resource.

   The download of the resource must delay the `load` event **(page 434)**.

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 **(page 273)** 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 (page 265)**

   The type is the type specified in the resource's Content-Type metadata **(page 265)**.

   ↪ **Otherwise, if the `type` (page 154) attribute is present**

   The type is the type specified in the `type` **(page 154)** 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` **(page 154)** object representing the element should expose that interface. The handler is not a nested browsing context **(page 19)**. 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` **(page 157)** elements that are children of the `object` **(page 153)** 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` **(page 153)** element must be associated with a nested browsing context **(page 19)**, if it does not already have one. The element's nested browsing context **(page 19)** must then be navigated **(page 257)** to the given resource. (The `data` **(page 154)** attribute of the `object` **(page 153)** 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` **(page 153)** element represents the specified image. The image is not a nested browsing context **(page 19)**.

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

↪ **Otherwise**

The `object` **(page 153)** 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` **(page 153)** element represents.
7. Once the resource is completely loaded, fire a `load` event **(page 273)** at the element.

2. If the `data` **(page 154)** attribute is absent but the `type` **(page 154)** attribute is present, and if the user agent can find a handler suitable according to the value of the `type` **(page 154)** attribute, then that handler should be used. If the handler supports a scriptable interface, the `HTMLObjectElement` **(page 154)** object representing the element should expose that interface. The handler is not a nested browsing context **(page 19)**. If no suitable handler can be found, jump to the next step (fallback).
3. (Fallback.) The `object` **(page 153)** element doesn't represent anything except what the element's contents represent, ignoring any leading `param` **(page 157)** element children. This is the element's fallback content **(page 68)**.

In the absence of other factors (such as style sheets), user agents must show the user what the `object` **(page 153)** element represents. Thus, the contents of `object` **(page 153)** elements act as fallback content **(page 68)**, to be used only when referenced resources can't be shown (e.g. because it returned a 404 error). This allows multiple `object` **(page 153)** 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` **(page 189)** attribute, if present while the `object` **(page 153)** element represents an image, can indicate that the object has an associated image map **(page 189)**. The attribute must be ignored if the `object` **(page 153)** element doesn't represent an image.

height/width

The DOM attributes **`data`**, **`type`**, **`useMap`**, **`height`**, and **`width`** each must reflect **(page 29)** the respective content attributes of the same name.

### 3.14.6. The **`param`** element

**Contexts in which this element may be used:**

As a child of an `object` **(page 153)** element, before any content other than `param` **(page 157)** elements.

**Content model:**

Empty.

**Element-specific attributes:**

`name` **(page 157)** (required)
`value` **(page 157)** (required)

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLParamElement : HTMLElement (page 27) {
           attribute DOMString name (page 158);
           attribute DOMString value (page 158);
};
```

The `param` **(page 157)** element defines parameters for handlers invoked by `object` **(page 153)** 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` **(page 157)** is an `object` **(page 153)** element, then the element defines a **parameter** with the given name/value pair.

The DOM attributes **`name`** and **`value`** must both reflect **(page 29)** the respective content attributes of the same name.

### 3.14.7. The `video` element

Semi-transparent **(page 68)** strictly inline-level **(page 67)** embedded content **(page 68)**.

**Contexts in which this element may be used:**

As the only embedded content **(page 68)** child of a `figure` **(page 146)** element.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

If the element has a `src` **(page 158)** attribute: transparent **(page 68)**.
If the element does not have a `src` **(page 158)** attribute: one or more `source` **(page 168)** elements, then, transparent **(page 68)**.

**Element-specific attributes:**

`src` **(page 158)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLVideoElement : HTMLMediaElement (page 161) {
  readonly attribute unsigned long videoWidth (page 159);
  readonly attribute unsigned long videoHeight (page 159);
};
```

A `video` **(page 158)** element represents a video or movie.

Content may be provided inside the `video` **(page 158)** element so that older Web browsers, which do not support `video` **(page 158)**, can display text to the user informing them of how to access the video contents. User agents should not show this fallback content to the user.

The `video` **(page 158)** element is a media element **(page 161)** whose media data **(page 161)** is video data, possibly with associated audio data.

> ***Note: In this version, the `video`* (page 158) *element is aimed primarily at authors wanting to write their own video player user interfaces, rather than those authors wanting to expose video clips as part of other content, without concern to the details of the UI. To this end, the element in this version merely exposes a JavaScript API, and does not have declarative features or the ability to enable a default user interface. A future version of this specification will probably introduce such features.***

The **`src`** attribute gives the address of the video to show. The attribute, if present, must contain a URI (or IRI). This is the media source attribute **(page 162)**.

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, 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, `video` **(page 158)** elements represent pure black.

In the PLAYING **(page 162)** state, `video` **(page 158)** elements represent the frame of video at the continuously increasing "current" position **(page 161)**. When the current playback position **(page 161)** changes such that the last frame rendered is no longer the frame corresponding to the current playback position **(page 161)** 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 **(page 161)**. If there is associated audio data, the audio must play synchronised with the video, at the specified volume **(page 167)** with the specified mute state **(page 167)**.

In the PAUSED **(page 162)** state, the element represents the frame of video at the current playback position **(page 161)**, or, if that is not available yet (e.g. because the video is seeking or buffering), the last rendered frame of video.

In the AUTO-PAUSED **(page 162)** state, the element represents either the last frame of the video to have been rendered, or pure black if no frame from the video has been rendered yet.

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 should be filled with pure black.

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. In such an independent context, however, user agents may make full user interfaces visible, with, e.g., play, pause, seeking, and volume controls.

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.

#### *3.14.7.1. Video and audio codecs for* `video` **(page 158)** *elements*

User agents may support any video and audio codecs and container formats.

User agents should support Ogg Theora video and Ogg Vorbis audio, as well as the Ogg container format. [THEORA] [VORBIS] [OGG]

### 3.14.8. The `audio` element

Semi-transparent **(page 68)** strictly inline-level **(page 67)** embedded content **(page 68)**.

**Contexts in which this element may be used:**

As the only embedded content **(page 68)** child of a `figure` **(page 146)** element.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

If the element has a `src` **(page 160)** attribute: transparent **(page 68)**.
If the element does not have a `src` **(page 160)** attribute: one or more `source` **(page 168)** elements, then, transparent **(page 68)**.

**Element-specific attributes:**

`src` **(page 160)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLAudioElement : HTMLMediaElement (page 161) {
  // no members
};
```

An `audio` **(page 363)** element represents a sound or audio stream.

Content may be provided inside the `audio` **(page 363)** element so that older Web browsers, which do not support `audio` **(page 363)**, can display text to the user informing them of how to access the audio contents. User agents should not show this fallback content to the user.

The `audio` **(page 363)** element is a media element **(page 161)** whose media data **(page 161)** is audio data.

> ***Note: In this version, the `audio`*** **(page 363)** ***element is aimed primarily at authors wanting to add sound effects to games. To this end, the element in this version merely exposes a JavaScript API, and does not have declarative features or the ability to enable a default user interface. A future version of this specification will probably introduce such features.***

The **`src`** attribute gives the address of the audio to show. The attribute, if present, must contain a URI (or IRI). This is the media source attribute **(page 162)**.

In the PLAYING **(page 162)** state, `audio` **(page 363)** elements must have their audio data played synchronised with the current playback position **(page 161)**, at the specified volume **(page 167)** with the specified mute state **(page 167)**.

In the PAUSED **(page 162)** and AUTO-PAUSED **(page 162)** states, audio must not play.

*3.14.8.1. Video and audio codecs for `audio` **(page 363)** 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 (page 27) {
  // media
           attribute DOMString src (page 163);
  readonly attribute DOMString currentSrc (page 163);
  readonly attribute float duration (page 166);

  // playback state
  const unsigned short STOPPED (page 162) = 0;
  const unsigned short PLAYING (page 162) = 1;
  const unsigned short PAUSED (page 162) = 2;
  const unsigned short AUTO_PAUSED (page 162) = 3;
  readonly attribute unsigned short state (page 161);
  readonly attribute boolean buffering (page 166);
  readonly attribute boolean seeking (page 167);
  readonly attribute boolean ended (page 167);
  readonly attribute float bufferingRate (page 167);
  readonly attribute TimeRanges (page 168) buffered (page 167);
  readonly attribute float position (page 167);
  readonly attribute TimeRanges (page 168) played (page 167);

  // controls
           attribute float volume (page 167);
           attribute boolean muted (page 167);
  void load (page 163)();
  void play (page 164)();
  void pause (page 165)();
  void stop (page 166)();
  void seek (page 165)(in float offset);
};
```

Media elements **(page 161)** 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 **(page 161)** 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.

Media elements **(page 161)** have a **current playback position**, which must initially be zero. The current position is a time.

Media elements **(page 161)** can be in one of four states: STOPPED **(page 162)**, PLAYING **(page 162)**, PAUSED **(page 162)**, and AUTO-PAUSED **(page 162)**. The current state is exposed to scripts through the **`state`** DOM attribute, whose value must be the value of the constant corresponding to the element's state.

When created, media elements **(page 161)** must start in the STOPPED state. This state is represented by the **`STOPPED`** constant, whose value is zero. In the STOPPED state, no media data **(page 161)** is available.

The PLAYING state is the state when the element has media data **(page 161)** and is rendering it in real time. This state is represented by the **`PLAYING`** constant, whose value is 1.

The PAUSED state is the state when the element has media data **(page 161)** but its rendering has been intentionally stopped. This state is represented by the **`PAUSED`** constant, whose value is 2.

The AUTO-PAUSED state is the state when the element would be playing if there was media data **(page 161)** available, but, for one reason or another, the media data **(page 161)** is not yet available and therefore rendering has been automatically (and typically temporarily) stopped by the user agent. This state is represented by the **`AUTO_PAUSED`** constant, whose value is 3.

Media elements **(page 161)** can have a **media source attribute** (their `src` attribute). If it is specified, the resource it specified is the media resource **(page 161)** that will be used. Otherwise, the resource specified by the first suitable `source` **(page 168)** element child of the media element **(page 161)** is the one used, as described below.

To **pick a media resource** for a media element **(page 161)**, a user agent must follow the following steps:

1. If the media element **(page 161)** has a media source attribute **(page 162)**, then the address given in that attribute, resolved to an absolute URI relative to the media element **(page 161)**, is the address of the media resource **(page 161)**; jump to the last step.
2. Otherwise, let *candidate* be the first `source` **(page 168)** element child in the media element **(page 161)**, or null if there is no such child.
3. If *candidate* is not null and the *candidate* element has a `src` **(page 169)** attribute, and either:

   - the *candidate* element has a `type` **(page 169)** attribute and that attribute's value, when parsed as a MIME type, represents a type that the user agent can render (including any codecs described by the `codec` parameter), or [RFC2046] [RFC4281]
   - the *candidate* element has no `type` **(page 169)** attribute,

   ...then the address given in that *candidate* element's `src` **(page 169)** attribute, resolved to an absolute URI relative to the *candidate* element, is the address of the media resource **(page 161)**; jump to the last step.
4. Let *candidate* be the next `source` **(page 168)** element child in the media element **(page 161)**, 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 **(page 161)**. Abort these steps.

7. Let the **chosen media resource** be the one with the address that was resolved before jumping to this step.

The **`src`** DOM attribute on media elements **(page 161)** must reflect **(page 29)** the content attribute of the same name.

The **`currentSrc`** DOM attribute must return the empty string if the media element **(page 161)** is in the STOPPED **(page 162)** state, and the absolute URL of the chosen media resource **(page 163)** otherwise.

All media elements **(page 161)** have a **begun flag**, which must begin in the false state.

When the **`load()`** method on a media element **(page 161)** is invoked, the user agent must run the following steps. Note that this algorithm might get aborted, e.g. if the `stop()` **(page 166)** method is invoked, or if the 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. The media element **(page 161)**'s `buffering` **(page 166)** attribute must be set to false.
3. If the media element **(page 161)** is not in the STOPPED **(page 162)** state, then the element must be switched to the STOPPED **(page 162)** state and the user agent must synchronously fire a `stopped` event at the element.
4. The user agent must pick a media resource **(page 162)** for the media element **(page 161)**. If that fails, the method must raise an `INVALID_STATE_ERR` exception, and abort these steps.
5. The element must be switched to the PAUSED **(page 162)** state and the user agent must synchronously fire a `paused` event at the element.
6. The method must return, but these steps must continue.
7. Playback of any previously playing media resource **(page 161)** for this element stops.
8. If the element's begun flag **(page 163)** is true, then the begun flag **(page 163)** must be set to false and the user agent must fire an `abort` event at the media element **(page 161)**.
9. If a download is in progress for the media element **(page 161)**, the user agent should stop the download.
10. The element's `buffering` **(page 166)** attribute must be set to true.
11. The user agent must then set the begun flag **(page 163)** to true and fire a `begin` event at the media element **(page 161)**.
12. The user agent must then begin to download the chosen media resource **(page 163)**. 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.

13. While the download is progressing, the user agent must fire a `progress` event 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 `stalled` event at the element.

    User agents may allow users to selectively block or slow media data **(page 161)** downloads. When a media element **(page 161)**'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, or using HTTP partial range requests. The user agent must only consider a resource erroneous if it has given up trying to download it.

    When the user agent has completed the download of the entire media resource **(page 161)**, or when it has determined that the resource is erroneous, it must move on to the next step.

14. Once the download stops, for whatever reason, the element's `buffering` **(page 166)** attribute must be set to false.

15. If the resource is erroneous and no media data **(page 161)** has been obtained (e.g. because it is an HTTP 404 response, or the server returned a file that turns out to not be pure audio when the media element **(page 161)** is a `video` **(page 158)** element, or the file uses an unsupported codec), then:

    1. The user agent should cancel the download.

    2. The element must be switched to the STOPPED **(page 162)** state and the user agent must synchronously fire a `stopped` event at the element.

    3. The begun flag **(page 163)** must be set to false and the user agent must fire an `error` event **(page 273)** at the media element **(page 161)**.

16. In the case of a download being interrupted after some media data **(page 161)** has been received, then the begun flag **(page 163)** must be set to false and the user agent must fire an `error` event **(page 273)** at the media element **(page 161)**, but the element must not be switched to the STOPPED **(page 162)** state.

17. If the download completes without errors, the begun flag **(page 163)** must be set to false and the user agent must fire a `load` event **(page 273)** at the element.

If a media element **(page 161)** in the STOPPED **(page 162)** state is inserted into a document, user agents must implicitly invoke the `load()` **(page 163)** method on the media element **(page 161)**. Any exceptions raised must be ignored.

When the **`play()`** method on a media element **(page 161)** is invoked, the user agent must run the following steps.

1. If the media element **(page 161)** is in the STOPPED **(page 162)** state, then the user agent must invoke the `load()` **(page 163)** method. If that raises an exception, that exception must be reraised by the `play()` **(page 164)** method.
2. If the media element **(page 161)** is in either the PLAYING **(page 162)** state or the AUTO-PAUSED **(page 162)** state, then the method must return and the user agent must abort these steps.
3. If the user agent has enough media data **(page 161)** to begin playback at the current playback position **(page 161)**, then the media element **(page 161)** must be switched to the PLAYING **(page 162)** state, the user agent must synchronously fire a `playing` event at the element, and then the method call must return. These steps must be aborted at this point. As described below, playback will then start.
4. Otherwise, the user agent does not have enough data to begin playback at the current playback position **(page 161)**.

   If the media element **(page 161)** is in the PAUSED **(page 162)** state, then it must be switched to the AUTO-PAUSED **(page 162)** state; the user agent must then synchronously fire an `autopaused` event at the element, and the method call must return. These steps must then be aborted. As described below, playback will resume when the data is available.

When the **`pause()`** method is invoked, the user agent must run the following steps:

1. If the media element **(page 161)** is in the STOPPED **(page 162)** state, then the user agent must raise an `INVALID_STATE_ERR` exception, and abort these steps.
2. If the media element **(page 161)** is in the PAUSED **(page 162)** state, then the method must return and the user agent must abort these steps.
3. The media element **(page 161)** must be switched to the PAUSED **(page 162)** state and the user agent must synchronously fire a `paused` event at the element.
4. The method must then return. As described below, playback of will stop.

When the **`seek(`*`offset`*`)`** method is invoked, the user agent must run the following steps:

1. If the media element **(page 161)** is in the STOPPED **(page 162)** state, then the user agent must raise an `INVALID_STATE_ERR` exception, and abort these steps.
2. If the given *offset* is below zero, then the user agent must raise an `INDEX_SIZE_ERR` exception, and abort these steps.
3. The current playback position **(page 161)** must be set to the given *offset*, treating this argument as a time in seconds from the start of the media resource **(page 161)**. If the given *offset* is beyond the end of the media resource, then the current playback position **(page 161)** must be set to the end of the media resource **(page 161)**.

4. If the media element **(page 161)** is in the PLAYING **(page 162)** state, then the element must be switched to the AUTO-PAUSED **(page 162)** state, and the user agent must synchronously fire an `autopaused` event at the element.
5. The method must then return. As described below, the user agent will then seek to the given *offset*, and, if appropriate, start playing.

When the **`stop()`** method is invoked, the user agent must run the following steps:

1. Any already-running instance of the algorithm for the `load()` **(page 163)** method 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. The media element **(page 161)**'s `buffering` **(page 166)** attribute must be set to false.
3. If the media element **(page 161)** is not in the STOPPED **(page 162)** state, then the element must be switched to the STOPPED **(page 162)** state and the user agent must synchronously fire a `stopped` event at the element.
4. The method must return, but these steps must continue.
5. Playback stops.
6. If the element's begun flag **(page 163)** is true, then the begun flag **(page 163)** must be set to false and the user agent must fire an `abort` event at the media element **(page 161)**.
7. If a download is in progress for the media element **(page 161)**, the user agent should stop the download.

While in the PLAYING **(page 162)** state, the current playback position **(page 161)** must increase monotonically at one unit time of media time per unit time of wall clock time.

If the media data **(page 161)** for the current playback position **(page 161)** is not yet available, the user agent must switch the media element **(page 161)** to the AUTO-PAUSED **(page 162)** state and stop playback.

While in the AUTO-PAUSED **(page 162)** state, once media data **(page 161)** becomes available for playing back at the current playback position **(page 161)**, and if the user agent estimates that data is being downloaded at a rate where the current playback position **(page 161)**, if it were to advance in real time, would not soon overtake the available data, and once the user agent is able to start playing at the current playback position **(page 161)** (e.g. after seeking to that position), the user agent must switch the media element **(page 161)** to the PLAYING **(page 162)** state and resume playback.

The **`duration`** attribute must return the length of the media resource **(page 161)**, in seconds. If no media data **(page 161)** is available, then the attributes must return 0. If media data **(page 161)** is available but the length is not known, the attribute must return the length of the currently available data, in seconds.

The **`buffering`** attribute returns true if data is still being buffered, and false otherwise. The exact rules for this are described above as part of the algorithm for the `play()` **(page 164)** method. When the media element **(page 161)** is created, the `buffering` **(page 166)** attribute must initially be false.

The **`seeking`** attribute must return false if the media element **(page 161)** is in the STOPPED **(page 162)** state or if the user agent would be able to immediately start playing at the current playback position **(page 161)**. Otherwise it must return true.

The **`ended`** attribute must return true if the media element **(page 161)** is in the PAUSED **(page 162)** or AUTO-PAUSED **(page 162)** state and the current playback position **(page 161)** is at the end of the available media data **(page 161)**. Otherwise it must return false.

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 **`position`** attribute must return the current playback position **(page 161)**, expressed in seconds.

The **`buffered`** attribute must return a static normalised `TimeRanges` object **(page 168)** that represents the ranges of the media resource **(page 161)**, 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.***

The **`played`** attribute must return a static normalised `TimeRanges` object **(page 168)** that represents the ranges of the media resource **(page 161)**, if any, that the user agent has so far rendered, at the time the attribute is evaluated.

The timeline of the objects returned by the `buffered` **(page 167)** and `played` **(page 167)** attributes must be the same as the media resource **(page 161)**'s timeline.

The **`volume`** attribute must return the playback volume of any audio portions of the media element **(page 161)**, 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 **(page 161)** must then be muted, and if false, audio playback must then be enabled.

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 **(page 161)**'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.

*3.14.9.1. Time range*

Objects implementing the `TimeRanges` **(page 168)** interface represent a list of ranges (periods) of time.

```
interface TimeRanges {
  readonly attribute unsigned long length (page 168);
  float start (page 168)(in unsigned long index);
  float end (page 168)(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.

When a `TimeRanges` **(page 168)** 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).

### 3.14.10. The `source` element

**Contexts in which this element may be used:**

As a child of a media element **(page 161)**, before any content other than `source` **(page 168)** elements.

**Content model:**

Empty.

**Element-specific attributes:**

`src` **(page 169)** (required)
`type` **(page 169)** (required)

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLSourceElement : HTMLElement (page 27) {
           attribute DOMString name (page 169);
           attribute DOMString value (page 169);
};
```

The `source` **(page 168)** element allows authors to specify multiple media resources **(page 161)** for media elements **(page 161)**.

The **`src`** attribute gives the address of the media resource **(page 161)**. The value must be a URI (or IRI).

The **`type`** attribute gives the type of the media resource **(page 161)**, to help the user agent determine if it can play this media resource **(page 161)** before downloading it. Its value must be a MIME type. The `codec` parameter may be specified and might be necessary to specify exactly how the resource is encoded. [RFC2046] [RFC4281]

Both attributes must be present.

If a `source` **(page 168)** element is inserted into a media element **(page 161)** that is in the STOPPED **(page 162)** state and that is already in a document, the user agent must implicitly invoke the `load()` **(page 163)** method on the media element **(page 161)**. Any exceptions raised must be ignored.

The DOM attributes **`src`** and **`type`** must both reflect **(page 29)** the respective content attributes of the same name.

### 3.14.11. The `canvas` element

Strictly inline-level **(page 67)** embedded content **(page 68)**.

**Contexts in which this element may be used:**

As the only embedded content **(page 68)** child of a `figure` **(page 146)** element.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

Inline-level content **(page 67)**.

**Element-specific attributes:**

`height` **(page 170)**
`width` **(page 170)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLCanvasElement : HTMLElement (page 27) {
         attribute long width (page 171);
         attribute long height (page 171);

  DOMString toDataURL() (page 171);
  DOMString toDataURL (page 171)(in DOMString type);

  DOMObject getContext (page 171)(in DOMString contextID);
};
```

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

The `canvas` **(page 169)** 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` **(page 169)** element in a document when a more suitable element is available. For example, it is inappropriate to use a `canvas` **(page 169)** 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` **(page 98)**) and then styled using CSS and supporting technologies such as XBL.

When authors use the `canvas` **(page 169)** 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` **(page 169)** element. The contents of the `canvas` **(page 169)** element, if any, are the element's fallback content **(page 68)**.

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` **(page 169)** 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` **(page 169)** 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` **(page 169)** element's fallback content must be used instead.

The `canvas` **(page 169)** element has two attributes to control the size of the coordinate space: **`height`** and **`width`**. These attributes, when specified, must have values that are valid non-negative integers **(page 48)**. The rules for parsing non-negative integers **(page 48)** 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` **(page 169)** element equal the size of the coordinate space, with the numbers interpreted in CSS pixels. However, the element can be sized arbitrarily by a style sheet. During rendering, the image is scaled to fit this layout size.

The size of the coordinate space does not necessarily represent the size of the actual bitmap that the user agent will use internally or during rendering. On high-definition displays, for instance, the user agent may internally use a bitmap with two device pixels per unit in the coordinate space, so that the rendering remains at high quality throughout.

The canvas must initially be fully transparent black.

If the `width` **(page 170)** and `height` **(page 170)** attributes are dynamically modified, the bitmap and any associated contexts must be cleared back to their initial state and reinitialised with the newly specified coordinate space dimensions.

The **`width`** and **`height`** DOM attributes must reflect **(page 29)** the content attributes of the same name.

To draw on the canvas, authors must first obtain a reference to a **context** using the **`getContext`** method of the `canvas` **(page 169)** element.

This specification only defines one context, with the name "`2d` **(page 172)**". If `getContext()` **(page 171)** is called with that exact string, then the UA must return a reference to an object implementing `CanvasRenderingContext2D` **(page 172)**. Other specifications may define their own contexts, which would return different objects.

Vendors may also define experimental contexts using the syntax *`vendorname-context`*, for example, `moz-3d`.

When the UA is passed an empty string or a string specifying a context that it does not support, then it must return null. String comparisons should be literal and case-sensitive.

> ***Note: A future version of this specification will probably define a `3d` context (probably based on the OpenGL ES API).***

The **`toDataURL()`** method must, when called with no arguments, return a `data:` URI containing a representation of the image as a PNG file. [PNG].

The **`toDataURL(`*`type`*`)`** method (when called with one *or more* arguments) must return a `data:` URI containing a representation of the image in the format given by *type*. The possible values are MIME types with no parameters, for example `image/png`, `image/jpeg`, or even maybe `image/svg+xml` if the implementation actually keeps enough information to reliably render an SVG image from the canvas.

Only support for `image/png` is required. User agents may support other types. If the user agent does not support the requested type, it must return the image using the PNG format.

User agents must convert the provided type to lower case before establishing if they support that type and before creating the `data:` URL.

> ***Note: When trying to use types other than `image/png`, authors can check if the image was really returned in the requested format by checking to see if the returned string starts with one the exact strings "`data:image/png,`" or "`data:image/png;`". If it does, the image is PNG, and thus the requested type was not supported.***

Arguments other than the *type* must be ignored, and must not cause the user agent to raise an exception (as would normally occur if a method was called with the wrong number of arguments). A future version of this specification will probably allow extra parameters to be passed to `toDataURL()` **(page 171)** to allow authors to more carefully control compression settings, image metadata, etc.

**Security:** To prevent *information leakage*, the `toDataURL()` **(page 171)** and `getImageData()` **(page 184)** methods should raise a security exception **(page 267)** if the canvas ever had images painted on it that originate from a domain other than the domain of the script **(page 267)** that painted the images onto the canvas.

### *3.14.11.1. The 2D context*

When the `getContext()` **(page 171)** method of a `canvas` **(page 169)** element is invoked with **`2d`** as the argument, a `CanvasRenderingContext2D` **(page 172)** object is returned.

There is only one `CanvasRenderingContext2D` **(page 172)** object per canvas, so calling the `getContext()` **(page 171)** method with the `2d` **(page 172)** 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 (page 170) canvas (page 174);

  // state
  void save (page 175)(); // push state on state stack
  void restore (page 175)(); // pop state stack and restore state

  // transformations (default transform is the identity matrix)
  void scale (page 175)(in float x, in float y);
  void rotate (page 175)(in float angle);
  void translate (page 175)(in float x, in float y);
  void transform (page 175)(in float m11, in float m12, in float m21, in
float m22, in float dx, in float dy);
  void setTransform (page 175)(in float m11, in float m12, in float m21, in
float m22, in float dx, in float dy);

  // compositing
           attribute float globalAlpha (page 176); // (default 1.0)
           attribute DOMString globalCompositeOperation (page 176); //
(default over)

  // colors and styles
           attribute DOMObject strokeStyle (page 177); // (default black)
           attribute DOMObject fillStyle (page 177); // (default black)
  CanvasGradient (page 178) createLinearGradient(in float x0, in float y0,
in float x1, in float y1);
  CanvasGradient (page 178) createRadialGradient(in float x0, in float y0,
in float r0, in float x1, in float y1, in float r1);
  CanvasPattern (page 179) createPattern(in HTMLImageElement (page 148) image,
DOMString repetition);
  CanvasPattern (page 179) createPattern(in HTMLCanvasElement (page 170)
image, DOMString repetition);

  // line caps/joins
           attribute float lineWidth (page 179); // (default 1)
           attribute DOMString lineCap (page 179); // "butt", "round",
"square" (default "butt")
           attribute DOMString lineJoin (page 180); // "round", "bevel",
"miter" (default "miter")
           attribute float miterLimit (page 180); // (default 10)

  // shadows
           attribute float shadowOffsetX (page 180); // (default 0)
           attribute float shadowOffsetY (page 180); // (default 0)
           attribute float shadowBlur (page 180); // (default 0)
           attribute DOMString shadowColor (page 180); // (default black)

  // rects
  void clearRect (page 181)(in float x, in float y, in float w, in float h);
  void fillRect (page 181)(in float x, in float y, in float w, in float h);
  void strokeRect (page 181)(in float x, in float y, in float w, in float h);

  // path API
  void beginPath (page 181)();
  void closePath (page 181)();
  void moveTo (page 181)(in float x, in float y);
  void lineTo (page 182)(in float x, in float y);
  void quadraticCurveTo (page 182)(in float cpx, in float cpy, in float x,
in float y);
  void bezierCurveTo (page 182)(in float cp1x, in float cp1y, in float cp2x,
in float cp2y, in float x, in float y);
  void arcTo (page 182)(in float x1, in float y1, in float x2, in float y2,
in float radius);
  void rect (page 183)(in float x, in float y, in float w, in float h);
  void arc (page 182)(in float x, in float y, in float radius, in float
startAngle, in float endAngle, in boolean anticlockwise);
  void fill (page 183)();
  void stroke (page 183)();
  void clip (page 183)();
  boolean isPointInPath (page 183)(in float x, in float y);

  // drawing images
  void drawImage (page 183)(in HTMLImageElement (page 148) image, in float dx,
in float dy);
  void drawImage (page 183)(in HTMLImageElement (page 148) image, in float dx,
in float dy, in float dw, in float dh);
  void drawImage (page 183)(in HTMLImageElement (page 148) 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 (page 183)(in HTMLCanvasElement (page 170) image, in float
dx, in float dy);
  void drawImage (page 183)(in HTMLCanvasElement (page 170) image, in float
dx, in float dy, in float dw, in float dh);
  void drawImage (page 183)(in HTMLCanvasElement (page 170) 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 (page 174) getImageData (page 184)(in float sx, in float sy, in
float sw, in float sh);
  void putImageData (page 185)(in ImageData (page 174) 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 (page 178)(in float offset, in DOMString color);
};

interface CanvasPattern {
  // opaque object
};

interface ImageData {
  readonly attribute long int width (page 184);
  readonly attribute long int height (page 184);
  readonly attribute int[] data (page 184);
};
```

The **`canvas`** attribute must return the `canvas` **(page 169)** 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` **(page 177)**, `fillStyle` **(page 177)**, `globalAlpha` **(page 176)**, `lineWidth` **(page 179)**, `lineCap` **(page 179)**, `lineJoin` **(page 180)**, `miterLimit` **(page 180)**, `shadowOffsetX` **(page 180)**, `shadowOffsetY` **(page 180)**, `shadowBlur` **(page 180)**, `shadowColor` **(page 180)**, `globalCompositeOperation` **(page 176)**.

  ***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()` (page 181) method. The current bitmap is a property of the canvas, not the context.***

The **`save()`** method pushes a copy of the current drawing state onto the drawing state stack.

The **`restore()`** method pops the top entry in the drawing state stack, and resets the drawing state it describes. If there is no saved state, the method does nothing.

#### 3.14.11.1.2. 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 three transformation methods.

The transformations must be performed in reverse order. For instance, if a scale transformation that doubles the width is applied, followed by a rotation transformation that rotates drawing operations by a quarter turn, and a rectangle twice as wide as it is tall is then drawn on the canvas, the actual result will be a square.

The **`scale(x, y)`** method must add the scaling transformation described by the arguments to the transformation matrix. The *x* argument represents the scale factor in the horizontal direction and the *y* argument represents the scale factor in the vertical direction. The factors are multiples.

The **`rotate(angle)`** method must add the rotation transformation described by the argument to the transformation matrix. The *angle* argument represents a clockwise rotation angle expressed in radians.

The **`translate(x, y)`** method must add the translation transformation described by the arguments to the transformation matrix. The *x* argument represents the translation distance in the horizontal direction and the *y* argument represents the translation distance in the vertical direction. The arguments are in coordinate space units.

The **`transform(m11, m12, m21, m22, dx, dy)`** method must multiply the current transformation matrix with the matrix described by:

$$\begin{matrix} m11 & m21 & dx \\ m12 & m22 & dy \\ 0 & 0 & 1 \end{matrix}$$

The **`setTransform(m11, m12, m21, m22, dx, dy)`** method must reset the current transform to the identity matrix, and then invoke the `transform` **(page 175)**`(m11, m12, m21, m22, dx, dy)` method with the same arguments.

#### 3.14.11.1.3. Compositing

All drawing operations are affected by the global compositing attributes, `globalAlpha` **(page 176)** and `globalCompositeOperation` **(page 176)**.

The **`globalAlpha`** attribute gives an alpha value that is applied to shapes and images before they are composited onto the canvas. The value must be in the range from 0.0 (fully transparent) to 1.0 (no additional transparency). If an attempt is made to set the attribute to a value outside this range, the attribute must retain its previous value. When the context is created, the `globalAlpha` **(page 176)** attribute must initially have the value 1.0.

The **`globalCompositeOperation`** attribute sets how shapes and images are drawn onto the existing bitmap, once they have had `globalAlpha` **(page 176)** and the current transformation matrix applied. It must be set to a value from the following list. In the descriptions below, the source image is the shape or image being rendered, and the destination image is the current state of the bitmap.

The source-* descriptions below don't define what should happen with semi-transparent regions.

**`source-atop`**

Display the source image wherever both images are opaque. Display the destination image wherever the destination image is opaque but the source image is transparent. Display transparency elsewhere.

**`source-in`**

Display the source image wherever both the source image and destination image are opaque. Display transparency elsewhere.

**`source-out`**

Display the source image wherever the source image is opaque and the destination image is transparent. Display transparency elsewhere.

**`source-over` (default)**

Display the source image wherever the source image is opaque. Display the destination image elsewhere.

**`destination-atop`**

Same as `source-atop` **(page 176)** but using the destination image instead of the source image and vice versa.

**`destination-in`**

Same as `source-in` **(page 176)** but using the destination image instead of the source image and vice versa.

**`destination-out`**

Same as `source-out` **(page 176)** but using the destination image instead of the source image and vice versa.

**`destination-over`**

Same as `source-over` **(page 176)** but using the destination image instead of the source image and vice versa.

**`darker`**

Display the sum of the source image and destination images, with color values approaching 0 as a limit.

**`lighter`**

Display the sum of the source image and destination image, with color values approaching 1 as a limit.

**`copy`**

Display the source image instead of the destination image.

**`xor`**

Exclusive OR of the source and destination images.

***`vendorName-operationName`***

Vendor-specific extensions to the list of composition operators should use this syntax.

These values are all case-sensitive — they must be used exactly as shown. User agents must only recognise values that exactly match the values given above.

On setting, if the user agent does not recognise the specified value, it must be ignored, leaving the value of `globalCompositeOperation` **(page 176)** unaffected.

When the context is created, the `globalCompositeOperation` **(page 176)** 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, `CanvasGradient` **(page 178)**s, or `CanvasPattern` **(page 179)**s. On setting, strings should be parsed as CSS <color> values and the color assigned, and `CanvasGradient` **(page 178)** and `CanvasPattern` **(page 179)** 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` **(page 178)**, nor a `CanvasPattern` **(page 179)**, then it must be ignored, and the attribute must retain its previous value.

On getting, if the value is a color, then: if it has alpha equal to 1.0, then the color must be returned as an uppercase six-digit hex value, prefixed with a "#" character (U+0023 NUMBER SIGN), with the first two digits representing the red component, the next two digits representing the green component, and the last two digits representing the blue component, the digits being in the range 0-9 A-F (U+0030 to U+0039 and U+0041 to U+0046). If the value has alpha less than 1.0, then the value must instead be returned in the CSS `rgba()` functional-notation format: the literal string `rgba` (U+0072 U+0067 U+0062 U+0061) followed by a U+0028 LEFT PARENTHESIS, a base-ten integer in the range 0-255 representing the red component (using digits 0-9, U+0030 to U+0039, in the shortest form possible), a literal U+002C COMMA and U+0020 SPACE, an integer for the green component, a comma and a space, an integer for the blue component, another comma and space, a U+0030 DIGIT ZERO, a U+002E FULL STOP (representing the decimal point), one or more digits in the range 0-9 (U+0030 to U+0039) representing the fractional part of the alpha value, and finally a U+0029 RIGHT PARENTHESIS.

Otherwise, if it is not a color but a `CanvasGradient` **(page 178)** or `CanvasPattern` **(page 179)**, then an object supporting those interfaces must be returned. Such objects are opaque and therefore only useful for assigning to other attributes or for comparison to other gradients or patterns.

When the context is created, the `strokeStyle` **(page 177)** and `fillStyle` **(page 177)** attributes must initially have the string value `#000000`.

There are two types of gradients, linear gradients and radial gradients, both represented by objects implementing the opaque **`CanvasGradient`** interface.

Once a gradient has been created (see below), stops must be placed along it to define how the colors are distributed along the gradient. Between each such stop, the colors and the alpha component must be interpolated over the RGBA space to find the color to use at that offset. Immediately before the 0 offset and immediately after the 1 offset, transparent black stops are be assumed.

The **`addColorStop(`*`offset, color`*`)`** method on the `CanvasGradient` **(page 178)** interface adds a new stop to a gradient. If the *offset* is less than 0 or greater than 1 then an `INDEX_SIZE_ERR` exception must be raised. If the *color* cannot be parsed as a CSS color, then a `SYNTAX_ERR` exception must be raised. Otherwise, the gradient must be updated with the new stop information.

The **`createLinearGradient(`*`x0, y0, x1, y1`*`)`** method takes four arguments, representing the start point (*x0*, *y0*) and end point (*x1*, *y1*) of the gradient, in coordinate space units, and must return a linear `CanvasGradient` **(page 178)** initialised with that line.

Linear gradients must be rendered such that at the starting point on the canvas the color at offset 0 is used, that at the ending point the color at offset 1 is used, that all points on a line perpendicular to the line between the start and end points have the color at the point where those two lines cross (interpolation happening as described above), and that any points beyond the start or end points are a transparent black.

The **`createRadialGradient(`*`x0, y0, r0, x1, y1, r1`*`)`** method takes six arguments, the first three representing the start circle with origin (*x0*, *y0*) and radius *r0*, and the last three representing the end circle with origin (*x1*, *y1*) and radius *r1*. The values are in coordinate space units. The method must return a radial `CanvasGradient` **(page 178)** initialised with those two circles.

Radial gradients must be rendered such that a cone is created from the two circles, so that at the circumference of the starting circle the color at offset 0 is used, that at the circumference around the ending circle the color at offset 1 is used, that the circumference of a circle drawn a certain fraction of the way along the line between the two origins with a radius the same fraction of the way between the two radii has the color at that offset (interpolation happening as described above), that the end circle appear to be above the start circle when the end circle is not completely enclosed by the start circle, that the end circle be filled by the color at offset 1, and that any points not described by the gradient are a transparent black.

If a gradient has no stops defined, then the gradient must be treated as a solid transparent black. Gradients are, naturally, only painted where the stroking or filling effect requires that they be drawn.

Support for actually painting gradients is optional. Instead of painting the gradients, user agents may instead just paint the first stop's color. However, `createLinearGradient()` and `createRadialGradient()` must always return objects when passed valid arguments.

Patterns are represented by objects implementing the opaque **CanvasPattern** interface.

To create objects of this type, the **createPattern(image, repetition)** method is used. The first argument gives the image to use as the pattern (either an HTMLImageElement **(page 148)** or an HTMLCanvasElement **(page 170)**). 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 **(page 179)** object suitably initialised.

The *image* argument must be an instance of an HTMLImageElement **(page 148)** or HTMLCanvasElement **(page 170)**. If the *image* is of the wrong type, the implementation must raise a TYPE_MISMATCH_ERR exception.

Patterns must be painted so that the top left of the first image is anchored at the origin of the coordinate space, and images are then repeated horizontally to the left and right (if the repeat-x string was specified) or vertically up and down (if the repeat-y string was specified) or in all four directions all over the canvas (if the repeat string was specified). The images are not be scaled by this process; one CSS pixel of the image must be painted on one coordinate space unit. Of course, patterns must only actually painted where the stroking or filling effect requires that they be drawn, and are affected by the current transformation matrix.

Support for patterns is optional. If the user agent doesn't support patterns, then createPattern() must return null.

#### 3.14.11.1.5. 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 **(page 179)** 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 **(page 179)** 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` **(page 180)** 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` **(page 180)** 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` **(page 180)** 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` **(page 180)** 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 **(page 180)**, global alpha, clipping paths **(page 183)**, 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` **(page 177)**. If either height or width are zero, this method has no effect.

The **`strokeRect()`** method must draw a rectangular outline of the specified size using the `strokeStyle` **(page 177)**, `lineWidth` **(page 179)**, `lineJoin` **(page 180)**, and (if appropriate) `miterLimit` **(page 180)** attributes. What should happen with zero heights or widths?

### 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()` **(page 181)** call.)

New points and the lines connecting them are added to subpaths using the methods described below. In all cases, the methods only modify the last subpath in the context's paths.

The **`lineTo(`*`x`*`, `*`y`*`)`** method must do nothing if the context has no subpaths. Otherwise, it must connect the last point in the subpath to the given point (*x*, *y*) using a straight line, and must then add the given point (*x*, *y*) to the subpath.

The **`quadraticCurveTo(`*`cpx`*`, `*`cpy`*`, `*`x`*`, `*`y`*`)`** method must do nothing if the context has no subpaths. Otherwise it must connect the last point in the subpath to the given point (*x*, *y*) by a quadratic curve with control point (*cpx*, *cpy*), and must then add the given point (*x*, *y*) to the subpath.

The **`bezierCurveTo(`*`cp1x`*`, `*`cp1y`*`, `*`cp2x`*`, `*`cp2y`*`, `*`x`*`, `*`y`*`)`** method must do nothing if the context has no subpaths. Otherwise, it must connect the last point in the subpath to the given point (*x*, *y*) using a bezier curve with control points (*cp1x*, *cp1y*) and (*cp2x*, *cp2y*). Then, it must add the point (*x*, *y*) to the subpath.

The **`arcTo(`*`x1`*`, `*`y1`*`, `*`x2`*`, `*`y2`*`, `*`radius`*`)`** method must do nothing if the context has no subpaths. If the context *does* have a subpath, then the behaviour depends on the arguments and the last point in the subpath.

Let the point (*x0*, *y0*) be the last point in the subpath. Let *The Arc* be the shortest arc given by circumference of the circle that has one point tangent to the line defined by the points (*x0*, *y0*) and (*x1*, *y1*), another point tangent to the line defined by the points (*x1*, *y1*) and (*x2*, *y2*), and that has radius *radius*. The points at which this circle touches these two lines are called the start and end tangent points respectively.

If the point (*x2*, *y2*) is on the line defined by the points (*x0*, *y0*) and (*x1*, *y1*) then the method must do nothing, as no arc would satisfy the above constraints.

Otherwise, the method must connect the point (*x0*, *y0*) to the start tangent point by a straight line, then connect the start tangent point to the end tangent point by *The Arc*, and finally add the start and end tangent points to the subpath.

Negative or zero values for *radius* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`arc(`*`x`*`, `*`y`*`, `*`radius`*`, `*`startAngle`*`, `*`endAngle`*`, `*`anticlockwise`*`)`** method draws an arc. If the context has any subpaths, then the method must add a straight line from the last point in the subpath to the start point of the arc. In any case, it must draw the arc between the start point of the arc and the end point of the arc, and add the start and end points of the arc to the subpath. The arc and its start and end points are defined as follows:

Consider a circle that has its origin at (*x*, *y*) and that has radius *radius*. The points at *startAngle* and *endAngle* along the circle's circumference, measured in radians clockwise from the positive x-axis, are the start and end points respectively. The arc is the path along the circumference of this circle from the start point to the end point, going anti-clockwise if the *anticlockwise* argument is true, and clockwise otherwise.

Negative or zero values for *radius* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`rect(x, y, w, h)`** method must create a new subpath containing just the four points (*x*, *y*), (*x*+*w*, *y*), (*x*+*w*, *y*+*h*), (*x*, *y*+*h*), with those four points connected by straight lines, and must then mark the subpath as closed. It must then create a new subpath with the point (*x*, *y*) as the only point in the subpath.

Negative values for *w* and *h* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`fill()`** method must fill each subpath of the current path in turn, using `fillStyle` **(page 177)**, and using the non-zero winding number rule. Open subpaths must be implicitly closed when being filled (without affecting the actual subpaths).

The **`stroke()`** method must stroke each subpath of the current path in turn, using the `strokeStyle` **(page 177)**, `lineWidth` **(page 179)**, `lineJoin` **(page 180)**, and (if appropriate) `miterLimit` **(page 180)** attributes.

Paths, when filled or stroked, must be painted without affecting the current path, and must be subject to transformations **(page 175)**, shadow effects **(page 180)**, global alpha **(page 176)**, clipping paths **(page 183)**, and global composition operators **(page 176)**.

> ***Note: The transformation is applied to the path when it is drawn, not when the path is constructed. Thus, a single path can be constructed and then drawn according to different transformations without recreating the path.***

The **`clip()`** method must create a new **clipping path** by calculating the intersection of the current clipping path and the area described by the current path (after applying the current transformation), using the non-zero winding number rule. Open subpaths must be implicitly closed when computing the clipping path, without affecting the actual subpaths.

When the context is created, the initial clipping path is the rectangle with the top left corner at (0,0) and the width and height of the coordinate space.

The **`isPointInPath(x, y)`** method must return true if the point given by the *x* and *y* coordinates passed to the method, when treated as coordinates in the canvas' coordinate space unaffected by the current transformation, is within the area of the canvas that is inside the current path; and must return false otherwise.

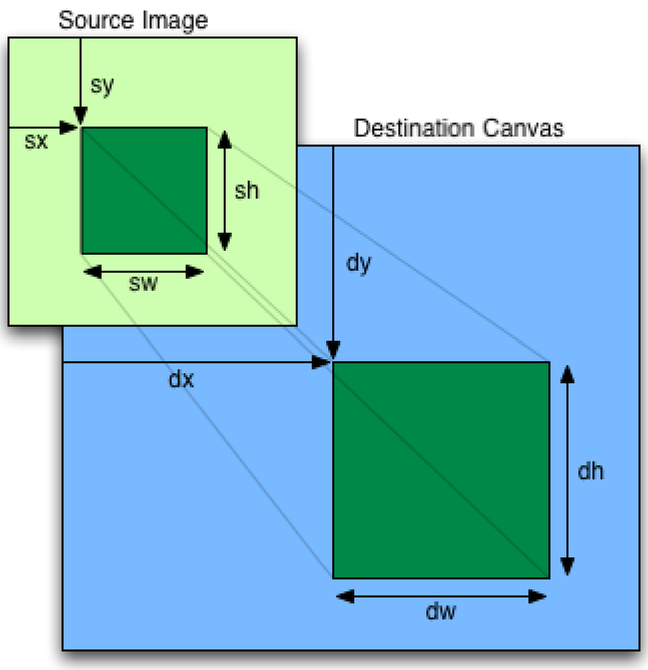
#### 3.14.11.1.9. Images

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` **(page 148)** or an `HTMLCanvasElement` **(page 170)** 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` **(page 148)** or `HTMLCanvasElement` **(page 170)**. If the *image* is of the wrong type, the implementation must raise a `TYPE_MISMATCH_ERR` exception. If one of the *sy*, *sw*, *sw*, and *sh* arguments is outside the size of the image, or if one of the *dw* and *dh* arguments is negative, the implementation must raise an `INDEX_SIZE_ERR` exception.

When `drawImage()` **(page 183)** 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 **(page 175)**, shadow effects **(page 180)**, global alpha **(page 176)**, clipping paths **(page 183)**, and global composition operators **(page 176)**.

### 3.14.11.1.10. Pixel manipulation

The **`getImageData(`*`sx, sy, sw, sh`*`)`** method must return an `ImageData` **(page 174)** object representing the underlying pixel data for the area of the canvas denoted by the rectangle which has one corner at the (*sx*, *sy*) coordinate, and that has width *sw* and height *sh*. Pixels outside the canvas must be returned as transparent black.

`ImageData` **(page 174)** objects must be initialised so that their **`height`** attribute is set to *h*, the number of rows in the image data, their **`width`** attribute is set to *w*, the number of physical device pixels per row in the image data, and the **`data`** attribute is initialised to an array of *h*×*w*×4 integers. The pixels must be represented in this array in left-to-right order, row by row, starting at the top left, with each pixel's red, green, blue, and alpha components being given in that order. Each component of each device pixel represented in this array must be in the range 0..255, representing the 8 bit value for that component.

The **`putImageData(`*`image`*`, `*`dx`*`, `*`dy`*`)`** method must take the given `ImageData` **(page 174)** structure, and draw it at the specified location *dx*,*dy* in the canvas coordinate space, mapping each pixel represented by the `ImageData` **(page 174)** structure into one device pixel.

The handling of pixel rounding when the specified coordinates do not exactly map to the device coordinate space is not defined by this specification, except that the following must result in no visible changes to the rendering:

```
context.putImageData(context.getImageData(x, y, w, h), x, y);
```

...for any value of *x* and *y*. In other words, while user agents may round the arguments of the two methods so that they map to device pixel boundaries, any rounding performed must be performed consistently for both the `getImageData()` **(page 184)** and `putImageData()` **(page 185)** operations.

The current transformation matrix must not affect the `getImageData()` **(page 184)** and `putImageData()` **(page 185)** methods.

#### 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. The coordinates are transformed by the current transformation matrix.
2. The shape or image is rendered, creating image *A*, as described in the previous sections. For shapes, the current fill, stroke, and line styles must be honoured.
3. The shadow is rendered from image *A*, using the current shadow styles, creating image *B*.
4. Image *A* is composited over image *B* creating the source image.
5. The source image has its alpha adjusted by `globalAlpha` **(page 176)**.
6. Within the clip region (as affected by the current transformation matrix), the source image is composited over the current canvas bitmap using the current composition operator.

### 3.14.12. The `map` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**
Zero or more block-level elements **(page 67)**.

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLMapElement : HTMLElement (page 27) {
  readonly attribute HTMLCollection (page 31) areas (page 186);
};
```

The `map` **(page 185)** element, in conjuction with any `area` **(page 186)** element descendants, defines an image map **(page 189)**.

The **`areas`** attribute must return an `HTMLCollection` **(page 31)** rooted at the `map` **(page 185)** element, whose filter matches only `area` **(page 186)** elements.

### 3.14.13. The `area` element

Strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

Where strictly inline-level content **(page 67)** is allowed, but only as a descendant of a `map` **(page 185)** element.

**Content model:**

Empty.

**Element-specific attributes:**

`alt` **(page 187)**
`coords` **(page 187)**
`shape` **(page 187)**
`href` **(page 273)**
`target` **(page 274)**
`ping` **(page 274)**
`rel` **(page 274)**
`media` **(page 274)**
`hreflang` **(page 274)**
`type` **(page 274)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLAreaElement : HTMLElement (page 27) {
           attribute DOMString alt (page 189);
           attribute DOMString coords (page 189);
           attribute DOMString shape (page 189);
           attribute DOMString href (page 189);
           attribute DOMString target (page 189);
           attribute DOMString ping (page 189);
           attribute DOMString rel (page 189);
  readonly attribute DOMTokenList relList (page 189);
           attribute DOMString media (page 189);
           attribute DOMString hreflang (page 189);
           attribute DOMString type (page 189);
};
```

The `area` **(page 186)** element represents either a hyperlink with some text and a corresponding area on an image map **(page 189)**, or a dead area on an image map.

If the `area` **(page 186)** element has an `href` **(page 273)** attribute, then the `area` **(page 186)** element represents a hyperlink **(page 273)**; the **`alt`** attribute, which must then be present, specifies the text.

However, if the `area` **(page 186)** element has no `href` **(page 273)** attribute, then the area represented by the element cannot be selected, and the `alt` **(page 187)** attribute must be omitted.

In both cases, the `shape` **(page 187)** and `coords` **(page 187)** attributes specify the area.

The **`shape`** attribute is an enumerated attribute **(page 63)**. 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 **(page 187)** state.

The **`coords`** attribute must, if specified, contain a valid list of integers **(page 53)**. This attribute gives the coordinates for the shape described by the `shape` **(page 187)** attribute. The processing for this attribute is described as part of the image map **(page 189)** processing model.

In the circle state **(page 187)**, `area` **(page 186)** elements must have a `coords` **(page 187)** 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 **(page 187)** state, `area` **(page 186)** elements must not have a `coords` **(page 187)** attribute.

In the polygon state **(page 187)**, `area` **(page 186)** elements must have a `coords` **(page 187)** 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 **(page 187)**, `area` **(page 186)** elements must have a `coords` **(page 187)** 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 **(page 274)** created using the `area` **(page 186)** element, as described in the next section, the `href` **(page 273)**, `target` **(page 274)** `and ping` **(page 274)** `attributes decide how the link is followed. The rel` **(page 274)**`, media` **(page 274)**`, hreflang` **(page 274)**`, and type` **(page 274)** `attributes may be used to indicate to the user the likely nature of the target resource before the user follows the link.`

The `target` **(page 274)**, `ping` **(page 274)**, `rel` **(page 274)**, `media` **(page 274)**, `hreflang` **(page 274)**, and `type` **(page 274)** attributes must be omitted if the `href` **(page 273)** attribute is not present.

The activation behavior **(page 19)** of `area` **(page 186)** elements is to run the following steps:

1. If the `DOMActivate` event in question is not trusted (i.e. a `click()` **(page 78)** method call was the reason for the event being dispatched), and the `area` **(page 186)** element's `target` attribute is ... then raise an `INVALID_ACCESS_ERR` exception.
2. Otherwise, the user agent must follow the hyperlink **(page 274)** defined by the `area` **(page 186)** element, if any.

   ***Note: One way that a user agent can enable users to follow hyperlinks is by allowing `area`*** **(page 186)** ***elements to be clicked, or focussed and activated by the keyboard. This will cause*** **(page 69)** ***the aforementioned activation behavior*** **(page 19)** ***to be invoked.***

The DOM attributes **`alt`**, **`coords`**, **`shape`**, **`href`**, **`target`**, **`ping`**, **`rel`**, **`media`**, **`hreflang`**, and **`type`**, each must reflect **(page 29)** the respective content attributes of the same name.

The DOM attribute **`relList`** must reflect **(page 29)** the `rel` **(page 274)** content attribute.

### 3.14.14. [TBW] Image maps

An **image map** allows geometric areas on an image to be associated with hyperlinks **(page 273)**.

An image, in the form of an `img` **(page 148)** element or an `object` **(page 153)** element representing an image, may be associated with an image map (in the form of a `map` **(page 185)** element) by specifying a **`usemap`** attribute on the `img` **(page 148)** or `object` **(page 153)** element. The `usemap` attribute, if specified, must be a valid hashed ID reference **(page 64)** to a `map` **(page 185)** element.

If an `img` **(page 148)** element or an `object` **(page 153)** 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 **(page 64)** to a `map` **(page 185)** 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` **(page 186)** elements that are descendants of the *map*. Let those be the *areas*.

Having obtained the list of `area` **(page 186)** 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` **(page 148)** element represents, then it must use the following steps.

> ***Note: In user agents that do not support images, or that have images disabled, `object`* (page 153) *elements cannot represent images, and thus this section never applies (the fallback content is shown instead). The following steps therefore only apply to `img`* (page 148) *elements.***

1. Remove all the `area` **(page 186)** elements in *areas* that have no `href` **(page 273)** attribute.
2. Remove all the `area` **(page 186)** elements in *areas* that have no `alt` **(page 187)** attribute, or whose `alt` **(page 187)** attribute's value is the empty string, *if* there is another `area` **(page 186)** element in *areas* with the same value in the `href` **(page 273)** attribute and with a non-empty `alt` **(page 187)** attribute.
3. Each remaining `area` **(page 186)** element in *areas* represents a hyperlink **(page 273)**. Those hyperlinks should all be made available to the user in a manner associated with the text of the `img` **(page 148)** element.

In this context, user agents may represent `area` **(page 186)** and `img` **(page 148)** 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` **(page 186)** elements in *areas*, in reverse tree order (so the last specified `area` **(page 186)** 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` **(page 186)** 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` **(page 187)** attribute represents.
2. Use the rules for parsing a list of integers **(page 53)** to parse the element's `coords` **(page 187)** 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` **(page 186)** element's current state, as per the following table, then the shape is empty; abort these steps.

| State | Minimum number of items |
|---|---|
| Circle state **(page 187)** | 3 |
| Default state **(page 187)** | 0 |
| Polygon state **(page 187)** | 6 |
| Rectangle state **(page 187)** | 4 |

4. Check for excess items in the *coords* list as per the entry in the following list corresponding to the `shape` **(page 187)** attribute's state:

   ↪ **Circle state (page 187)**
   Drop any items in the list beyond the third.

   ↪ **Default state (page 187)**
   Drop all items in the list.

   ↪ **Polygon state (page 187)**
   Drop the last item if there's an odd number of items.

   ↪ **Rectangle state (page 187)**
   Drop any items in the list beyond the fourth.

5. If the `shape` **(page 187)** attribute represents the rectangle state **(page 187)**, 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` **(page 187)** attribute represents the rectangle state **(page 187)**, 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` **(page 187)** attribute represents the circle state **(page 187)**, 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` **(page 187)** attribute:

   ↪ **Circle state (page 187)**

   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 (page 187)**

   The shape is a rectangle that exactly covers the entire image.

   ↪ **Polygon state (page 187)**

   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 (page 187)**

   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.

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` **(page 186)** elements representing hyperlinks **(page 273)** to be selected and activated (e.g. using a keyboard); events from this are not also propagated to the image.

> ***Note: Because a `map`*** **(page 185)** ***element (and its `area`*** **(page 186)** ***elements) can be associated with multiple `img`*** **(page 148)** ***elements, it is possible for an `area`*** **(page 186)** ***element to correspond to multiple focusable areas of the document.***

Image maps are *live* **(page 22)**; if the DOM is mutated, then the user agent must act as if it had rerun the algorithms for image maps.

## 3.15. Tabular data

### 3.15.1. The `table` element

Block-level element **(page 67)**, and structured inline-level element **(page 68)**.

**Contexts in which this element may be used:**

Where block-level elements **(page 67)** are expected.
Where structured inline-level elements **(page 68)** are allowed.

**Content model:**

In this order: optionally a `caption` **(page 194)** element, followed by either zero or more `colgroup` **(page 195)** elements, followed optionally by a `thead` **(page 198)** element, followed optionally by a `tfoot` **(page 198)** element, followed by either zero or more `tbody` **(page 196)** elements *or* one or more `tr` **(page 199)** elements, followed optionally by a `tfoot` **(page 198)** element (but there can only be one `tfoot` **(page 198)** element child in total).

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLTableElement : HTMLElement (page 27) {
           attribute HTMLTableCaptionElement caption (page 193);
  HTMLElement createCaption (page 193)();
  void deleteCaption (page 193)();
           attribute HTMLTableSectionElement (page 197) tHead (page 193);
  HTMLElement createTHead (page 193)();
  void deleteTHead (page 193)();
           attribute HTMLTableSectionElement (page 197) tFoot (page 193);
  HTMLElement createTFoot (page 193)();
  void deleteTFoot (page 193)();
  readonly attribute HTMLCollection (page 31) tBodies (page 193);
  readonly attribute HTMLCollection (page 31) rows (page 193);
  HTMLElement insertRow (page 194)(in long index);
  void deleteRow (page 194)(in long index);
};
```

The `table` **(page 192)** element represents data with more than one dimension (a table **(page 202)**).

The children of a `table` **(page 192)** element must be, in order:

1. Zero or one `caption` **(page 194)** elements.
2. Zero or more `colgroup` **(page 195)** elements.
3. Zero or one `thead` **(page 198)** elements.
4. Zero or one `tfoot` **(page 198)** elements, if the last element in the table is not a `tfoot` **(page 198)** element.

5. Either:
    - Zero or more `tbody` **(page 196)** elements, or
    - One or more `tr` **(page 199)** elements.
6. Zero or one `tfoot` **(page 198)** element, if there are no other `tfoot` **(page 198)** elements in the table.

The `table` **(page 192)** element takes part in the table model **(page 202)**.

The **`caption`** DOM attribute must return, on getting, the first `caption` **(page 194)** element child of the `table` **(page 192)** element. On setting, the first `caption` **(page 194)** element child of the `table` **(page 192)** element, if any, must be removed, and the new value must be inserted as the first node of the `table` **(page 192)** element.

The **`createCaption()`** method must return a new `caption` **(page 194)** element.

The **`deleteCaption()`** method must remove the first `caption` **(page 194)** element child of the `table` **(page 192)** element, if any.

The **`tHead`** DOM attribute must return, on getting, the first `thead` **(page 198)** element child of the `table` **(page 192)** element. On setting, the first `thead` **(page 198)** element child of the `table` **(page 192)** element, if any, must be removed, and the new value must be inserted immediately before the first element in the `table` **(page 192)** element that is neither a `caption` **(page 194)** element nor a `colgroup` **(page 195)** element.

The **`createTHead()`** method must return a new `thead` **(page 198)** element.

The **`deleteTHead()`** method must remove the first `thead` **(page 198)** element child of the `table` **(page 192)** element, if any.

The **`tFoot`** DOM attribute must return, on getting, the first `tfoot` **(page 198)** element child of the `table` **(page 192)** element. On setting, the first `tfoot` **(page 198)** element child of the `table` **(page 192)** element, if any, must be removed, and the new value must be inserted immediately before the first element in the `table` **(page 192)** element that is neither a `caption` **(page 194)** element, a `colgroup` **(page 195)** element, nor a `thead` **(page 198)** element.

The **`createTFoot()`** method must return a new `tfoot` **(page 198)** element.

The **`deleteTFoot()`** method must remove the first `tfoot` **(page 198)** element child of the `table` **(page 192)** element, if any.

The **`tBodies`** attribute must return an `HTMLCollection` **(page 31)** rooted at the `table` **(page 192)** node, whose filter matches only `tbody` **(page 196)** elements that are children.

The **`rows`** attribute must return an `HTMLCollection` **(page 31)** rooted at the `table` **(page 192)** node, whose filter matches only `tr` **(page 199)** elements that are either children of the `table` **(page 192)** element, or children of `thead` **(page 198)**, `tbody` **(page 196)**, or `tfoot` **(page 198)** elements that are themselves children of the `table` **(page 192)** element. The elements in the collection must be ordered such that those elements whose parent is a `thead` **(page 198)** are included first, in tree order, followed by those elements whose parent is either a `table` **(page 192)** or `tbody` **(page 196)** element, again in tree order, followed finally by those elements whose parent is a `tfoot` **(page 198)** 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` (page 193) collection:**

  The method must raise an `INDEX_SIZE_ERR` exception.

↪ **If the `rows` (page 193) collection has zero elements in it, and the `table` (page 192) has no `tbody` (page 196) elements in it:**

  The method must create a `tbody` **(page 196)** element, then create a `tr` **(page 199)** element, then append the `tr` **(page 199)** element to the `tbody` **(page 196)** element, then append the `tbody` **(page 196)** element to the `table` **(page 192)** element, and finally return the `tr` **(page 199)** element.

↪ **If the `rows` (page 193) collection has zero elements in it:**

  The method must create a `tr` **(page 199)** element, append it to the last `tbody` **(page 196)** element in the table, and return the `tr` **(page 199)** element.

↪ **If *index* is equal to -1 or equal to the number of items in `rows` (page 193) collection:**

  The method must create a `tr` **(page 199)** element, and append it to the parent of the last `tr` **(page 199)** element in the `rows` **(page 193)** collection. Then, the newly created `tr` **(page 199)** element must be returned.

↪ **Otherwise:**

  The method must create a `tr` **(page 199)** element, insert it immediately before the *index*th `tr` **(page 199)** element in the `rows` **(page 193)** collection, in the same parent, and finally must return the newly created `tr` **(page 199)** element.

The **`deleteRow(`*`index`*`)`** method must remove the *index*th element in the `rows` **(page 193)** collection from its parent. If *index* is less than zero or greater than or equal to the number of elements in the `rows` **(page 193)** collection, the method must instead raise an `INDEX_SIZE_ERR` exception.

### 3.15.2. The `caption` element

**Contexts in which this element may be used:**

  As the first element child of a `table` **(page 192)** element.

**Content model:**

  Significant **(page 68)** strictly inline-level content **(page 67)**.

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `caption` **(page 194)** element represents the title of the `table` **(page 192)** that is its parent, if it has a parent and that is a `table` **(page 192)** element.

The `caption` **(page 194)** element takes part in the table model **(page 202)**.

### 3.15.3. The `colgroup` element

**Contexts in which this element may be used:**

As a child of a `table` **(page 192)** element, after any `caption` **(page 194)** elements and before any `thead` **(page 198)**, `tbody` **(page 196)**, `tfoot` **(page 198)**, and `tr` **(page 199)** elements.

**Content model:**

Zero or more `col` **(page 196)** elements.

**Element-specific attributes:**

`span` **(page 195)**, but only if the element contains no `col` **(page 196)** elements

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLTableColElement : HTMLElement (page 27) {
           attribute unsigned long span (page 196);
};
```

The `colgroup` **(page 195)** element represents a group **(page 203)** of one or more columns **(page 203)** in the `table` **(page 192)** that is its parent, if it has a parent and that is a `table` **(page 192)** element.

If the `colgroup` **(page 195)** element contains no `col` **(page 196)** elements, then the element may have a **`span`** content attribute specified, whose value must be a valid non-negative integer **(page 48)** greater than zero. Its default value, which must be used if parsing the attribute as a non-negative integer **(page 48)** returns either an error or zero, is 1.

The `colgroup` **(page 195)** element and its `span` **(page 195)** attribute take part in the table model **(page 202)**.

The **span** DOM attribute must reflect **(page 29)** the content attribute of the same name, with the exception that on setting, if the new value is 0, then an `INDEX_SIZE_ERR` exception must be raised.

### 3.15.4. The `col` element

**Contexts in which this element may be used:**

As a child of a `colgroup` **(page 195)** element that doesn't have a `span` **(page 196)** attribute.

**Content model:**

Empty.

**Element-specific attributes:**

`span` **(page 196)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

`HTMLTableColElement` **(page 195)**, same as for `colgroup` **(page 195)** elements. This interface defines one member, `span` **(page 196)**.

If a `col` **(page 196)** element has a parent and that is a `colgroup` **(page 195)** element that itself has a parent that is a `table` **(page 192)** element, then the `col` **(page 196)** element represents one or more columns **(page 203)** in the column group **(page 203)** represented by that `colgroup` **(page 195)**.

The element may have a **span** content attribute specified, whose value must be a valid non-negative integer **(page 48)** greater than zero. Its default value, which must be used if parsing the attribute as a non-negative integer **(page 48)** returns either an error or zero, is 1.

The `col` **(page 196)** element and its `span` **(page 196)** attribute take part in the table model **(page 202)**.

The **span** DOM attribute must reflect **(page 29)** the content attribute of the same name, with the exception that on setting, if the new value is 0, then an `INDEX_SIZE_ERR` exception must be raised.

### 3.15.5. The `tbody` element

**Contexts in which this element may be used:**

As a child of a `table` **(page 192)** element, after any `caption` **(page 194)**, `colgroup` **(page 195)**, and `thead` **(page 198)** elements, but only if there are no `tr` **(page 199)** elements that are children of the `table` **(page 192)** element.

**Content model:**

One or more `tr` **(page 199)** elements

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLTableSectionElement : HTMLElement (page 27) {
  readonly attribute HTMLCollection (page 31) rows (page 197);
  HTMLElement (page 27) insertRow (page 197)(in long index);
  void deleteRow (page 197)(in long index);
};
```

The `HTMLTableSectionElement` **(page 197)** interface is also used for `thead` **(page 198)** and `tfoot` **(page 198)** elements.

The `tbody` **(page 196)** element represents a block **(page 203)** of rows **(page 203)** that consist of a body of data for the parent `table` **(page 192)** element, if the `tbody` **(page 196)** element has a parent and it is a `table` **(page 192)**.

The `tbody` **(page 196)** element takes part in the table model **(page 202)**.

The **`rows`** attribute must return an `HTMLCollection` **(page 31)** rooted at the element, whose filter matches only `tr` **(page 199)** 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` **(page 197)** 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` **(page 197)** collection, the method must create a `tr` **(page 199)** element, append it to the element *table section*, and return the newly created `tr` **(page 199)** element.

Otherwise, the method must create a `tr` **(page 199)** element, insert it as a child of the *table section* element, immediately before the *index*th `tr` **(page 199)** element in the `rows` **(page 197)** collection, and finally must return the newly created `tr` **(page 199)** element.

The **`deleteRow(`*`index`*`)`** method must remove the *index*th element in the `rows` **(page 197)** collection from its parent. If *index* is less than zero or greater than or equal to the number of elements in the `rows` **(page 197)** collection, the method must instead raise an `INDEX_SIZE_ERR` exception.

### 3.15.6. The `thead` element

**Contexts in which this element may be used:**

As a child of a `table` **(page 192)** element, after any `caption` **(page 194)**, and `colgroup` **(page 195)** elements and before any `tbody` **(page 196)**, `tfoot` **(page 198)**, and `tr` **(page 199)** elements, but only if there are no other `thead` **(page 198)** elements that are children of the `table` **(page 192)** element.

**Content model:**

One or more `tr` **(page 199)** elements

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

`HTMLTableSectionElement` **(page 197)**, as defined for `tbody` **(page 196)** elements.

The `thead` **(page 198)** element represents the block **(page 203)** of rows **(page 203)** that consist of the column labels (headers) for the parent `table` **(page 192)** element, if the `thead` **(page 198)** element has a parent and it is a `table` **(page 192)**.

The `thead` **(page 198)** element takes part in the table model **(page 202)**.

### 3.15.7. The `tfoot` element

**Contexts in which this element may be used:**

As a child of a `table` **(page 192)** element, after any `caption` **(page 194)**, `colgroup` **(page 195)**, and `thead` **(page 198)** elements and before any `tbody` **(page 196)** and `tr` **(page 199)** elements, but only if there are no other `tfoot` **(page 198)** elements that are children of the `table` **(page 192)** element.
As a child of a `table` **(page 192)** element, after any `caption` **(page 194)**, `colgroup` **(page 195)**, `thead` **(page 198)**, `tbody` **(page 196)**, and `tr` **(page 199)** elements, but only if there are no other `tfoot` **(page 198)** elements that are children of the `table` **(page 192)** element.

**Content model:**

One or more `tr` **(page 199)** elements

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

`HTMLTableSectionElement` **(page 197)**, as defined for `tbody` **(page 196)** elements.

The `tfoot` **(page 198)** element represents the block **(page 203)** of rows **(page 203)** that consist of the column summaries (footers) for the parent `table` **(page 192)** element, if the `tfoot` **(page 198)** element has a parent and it is a `table` **(page 192)**.

The `tfoot` **(page 198)** element takes part in the table model **(page 202)**.

### 3.15.8. The `tr` element

**Contexts in which this element may be used:**

As a child of a `thead` **(page 198)** element.
As a child of a `tbody` **(page 196)** element.
As a child of a `tfoot` **(page 198)** element.
As a child of a `table` **(page 192)** element, after any `caption` **(page 194)**, `colgroup` **(page 195)**, and `thead` **(page 198)** elements, but only if there are no `tbody` **(page 196)** elements that are children of the `table` **(page 192)** element.

**Content model:**

One or more `td` **(page 200)** or `th` **(page 201)** elements

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLTableRowElement : HTMLElement (page 27) {
  readonly attribute long rowIndex (page 199);
  readonly attribute long sectionRowIndex (page 199);
  readonly attribute HTMLCollection (page 31) cells (page 200);
  HTMLElement (page 27) insertCell (page 200)(in long index);
  void deleteCell(in long index);
};
```

The `tr` **(page 199)** element represents a row **(page 203)** of cells **(page 202)** in a table **(page 202)**.

The `tr` **(page 199)** element takes part in the table model **(page 202)**.

The **`rowIndex`** element must, if the element has a parent `table` **(page 192)** element, or a parent `tbody` **(page 196)**, `thead` **(page 198)**, or `tfoot` **(page 198)** element and a *grandparent* `table` **(page 192)** element, return the index of the `tr` **(page 199)** element in that `table` **(page 192)** element's `rows` **(page 193)** collection. If there is no such `table` **(page 192)** element, then the attribute must return 0.

The **`rowIndex`** DOM attribute must, if the element has a parent `table` **(page 192)**, `tbody` **(page 196)**, `thead` **(page 198)**, or `tfoot` **(page 198)** element, return the index of the `tr` **(page 199)** element in the parent element's `rows` collection (for tables, that's the `rows` **(page 193)** collection; for table sections, that's the `rows` **(page 197)** collection). If there is no such parent element, then the attribute must return 0.

The **`cells`** attribute must return an `HTMLCollection` **(page 31)** rooted at the `tr` **(page 199)** element, whose filter matches only `td` **(page 200)** and `th` **(page 201)** elements that are children of the `tr` **(page 199)** element.

The **`insertCell(`*`index`*`)`** method must act as follows:

If *index* is less than -1 or greater than the number of elements in the `cells` **(page 200)** collection, the method must raise an `INDEX_SIZE_ERR` exception.

If *index* is equal to -1 or equal to the number of items in `cells` **(page 200)** collection, the method must create a `td` **(page 200)** element, append it to the `tr` **(page 199)** element, and return the newly created `td` **(page 200)** element.

Otherwise, the method must create a `td` **(page 200)** element, insert it as a child of the `tr` **(page 199)** element, immediately before the *index*th `td` **(page 200)** or `th` **(page 201)** element in the `cells` **(page 200)** collection, and finally must return the newly created `td` **(page 200)** element.

The **`deleteCell(`*`index`*`)`** method must remove the *index*th element in the `cells` **(page 200)** collection from its parent. If *index* is less than zero or greater than or equal to the number of elements in the `cells` **(page 200)** collection, the method must instead raise an `INDEX_SIZE_ERR` exception.

### 3.15.9. The `td` element

**Contexts in which this element may be used:**
As a child of a `tr` **(page 199)** element.

**Content model:**
Zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**
`colspan` **(page 200)**
`rowspan` **(page 201)**

**Predefined classes that apply to this element:**
None.

**DOM interface:**

```
interface HTMLTableCellElement : HTMLElement (page 27) {
           attribute long colSpan (page 201);
           attribute long rowSpan (page 201);
  readonly attribute long cellIndex (page 201);
};
```

The `td` **(page 200)** element represents a data cell **(page 202)** in a table.

The `td` **(page 200)** element may have a **`colspan`** content attribute specified, whose value must be a valid non-negative integer **(page 48)** greater than zero. Its default value, which must be used if parsing the attribute as a non-negative integer **(page 48)** returns either an error or zero, is 1.

The td **(page 200)** element may also have a **rowspan** content attribute specified, whose value must be a valid non-negative integer **(page 48)**. Its default value, which must be used if parsing the attribute as a non-negative integer **(page 48)** returns an error, is also 1.

The td **(page 200)** element and its colspan **(page 200)** and rowspan **(page 201)** attributes take part in the table model **(page 202)**.

The **colspan** DOM attribute must reflect **(page 29)** 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 **(page 29)** the content attribute of the same name.

The **cellIndex** DOM attribute must, if the element has a parent tr **(page 199)** element, return the index of the cell's element in the parent element's cells **(page 200)** collection. If there is no such parent element, then the attribute must return 0.

### 3.15.10. The th element

**Contexts in which this element may be used:**
As a child of a tr **(page 199)** element.

**Content model:**
Zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**
colspan **(page 201)**
rowspan **(page 201)**
scope **(page 202)**

**Predefined classes that apply to this element:**
None.

**DOM interface:**

```
interface HTMLTableHeaderCellElement : HTMLTableCellElement (page 200)
{
           attribute DOMString scope (page 202);
};
```

The th **(page 201)** element represents a header cell **(page 202)** in a table.

The th **(page 201)** element may have a **colspan** content attribute specified, whose value must be a valid non-negative integer **(page 48)** greater than zero. Its default value, which must be used if parsing the attribute as a non-negative integer **(page 48)** returns either an error or zero, is 1.

The th **(page 201)** element may also have a **rowspan** content attribute specified, whose value must be a valid non-negative integer **(page 48)**. Its default value, which must be used if parsing the attribute as a non-negative integer **(page 48)** returns an error, is also 1.

The `th` **(page 201)** element may have a **`scope`** content attribute specified. The `scope` **(page 202)** attribute is an enumerated attribute **(page 63)** 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` **(page 202)** 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 **(page 208)**, which user agents must apply to determine the relationships between data cells and header cells.

The `th` **(page 201)** element and its `colspan` **(page 201)**, `rowspan` **(page 201)**, and `scope` **(page 202)** attributes take part in the table model **(page 202)**.

The **`scope`** DOM attribute must reflect **(page 29)** the content attribute of the same name.

The `HTMLTableHeaderCellElement` **(page 201)** interface inherits from the `HTMLTableCellElement` **(page 200)** interface and therefore also has the DOM attributes defined above in the `td` **(page 200)** 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` **(page 192)** 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` **(page 200)** elements, and have zero or more associated header cells. Header cells correspond to `th` **(page 201)** 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` **(page 199)** 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` **(page 196)** elements, but in the absense of `col` **(page 196)** elements are implied.

A **row group** is a set of rows **(page 203)** 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 $1 \leq x < x_{max}$ and $group_y \leq y < group_y+height$. Row groups correspond to `tbody` **(page 196)**, `thead` **(page 198)**, and `tfoot` **(page 198)** elements. Not every row is necessarily in a row group.

A **column group** is a set of columns **(page 203)** 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 $group_x \leq x < group_x+width$ and $1 \leq y < y_{max}$. Column groups correspond to `colgroup` **(page 195)** elements. Not every column is necessarily in a column group.

Row groups **(page 203)** cannot overlap each other. Similarly, column groups **(page 203)** cannot overlap each other.

A cell **(page 202)** cannot cover slots that are from two or more row groups **(page 203)**. It is, however, possible for a cell to be in multiple column groups **(page 203)**. All the slots that form part of one cell are part of zero or one row groups **(page 203)** and zero or more column groups **(page 203)**.

In addition to cells **(page 202)**, columns **(page 203)**, rows **(page 203)**, row groups **(page 203)**, and column groups **(page 203)**, tables **(page 202)** can have a `caption` **(page 194)** 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` **(page 192)** 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 **(page 202)** associated with a `table` **(page 192)** element, to determine the dimensions of the table ($x_{max}$ and $y_{max}$), and to determine if there are any table model errors **(page 203)**, 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 **(page 202)** represented by the `table` **(page 192)** element. The $x_{max}$ and $y_{max}$ variables give *the table*'s extent. *The table* is initially empty.
4. If the `table` **(page 192)** 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` **(page 192)** element.

   If a step in this algorithm ever requires the *current element* to be advanced to the next child of the `table` **(page 192)** 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` **(page 192)**:

   - `caption` **(page 194)**
   - `colgroup` **(page 195)**
   - `thead` **(page 198)**
   - `tbody` **(page 196)**
   - `tfoot` **(page 198)**
   - `tr` **(page 199)**

7. If the *current element* is a `caption` **(page 194)**, then that is the `caption` **(page 194)** element associated with *the table*. Otherwise, it has no associated `caption` **(page 194)** element.

8. If the *current element* is a `caption` **(page 194)**, then while the *current element* is not one of the following elements, advance the *current element* to the next child of the `table` **(page 192)**:

   - `colgroup` **(page 195)**
   - `thead` **(page 198)**
   - `tbody` **(page 196)**
   - `tfoot` **(page 198)**
   - `tr` **(page 199)**

   (Otherwise, the *current element* will already be one of those elements.)

9. If the *current element* is a `colgroup` **(page 195)**, follow these substeps:

   1. Let *next column* be 1.

   2. *Column groups.* Process the *current element* according to the appropriate one of the following two cases:

      ↪ **If the *current element* has any `col` (page 196) element children**

      Follow these steps:

      1. Let $x_{start}$ have the value $x_{max}$+1.

      2. Let the *current column* be the first `col` **(page 196)** element child of the `colgroup` **(page 195)** element.

      3. *Columns.* If the *current column* `col` **(page 196)** element has a `span` **(page 196)** attribute, then parse its value using the rules for parsing non-negative integers **(page 48)**.

         If the result of parsing the value is not an error or zero, then let *span* be that value.

Otherwise, if the `col` **(page 196)** element has no `span` **(page 196)** attribute, or if trying to parse the attribute's value resulted in an error, then let *span* be 1.

4. Increase $x_{max}$ by *span*.
5. Let the last *span* columns **(page 203)** in *the table* correspond to the *current column* `col` **(page 196)** element.
6. If *current column* is not the last `col` **(page 196)** element child of the `colgroup` **(page 195)** element, then let the *current column* be the next `col` **(page 196)** element child of the `colgroup` **(page 195)** element, and return to the third step of this innermost group of steps (columns).
7. Let all the last columns **(page 203)** in *the table* from x=$x_{start}$ to x=$x_{max}$ form a new column group **(page 203)**, anchored at the slot ($x_{start}$, 1), with width $x_{max}$-$x_{start}$-1, corresponding to the `colgroup` **(page 195)** element.

↪ **If the *current element* has no `col` (page 196) element children**

1. If the `colgroup` **(page 195)** element has a `span` **(page 195)** attribute, then parse its value using the rules for parsing non-negative integers **(page 48)**.

   If the result of parsing the value is not an error or zero, then let *span* be that value.

   Otherwise, if the `colgroup` **(page 195)** element has no `span` **(page 196)** attribute, or if trying to parse the attribute's value resulted in an error, then let *span* be 1.
2. Increase $x_{max}$ by *span*.
3. Let the last *span* columns **(page 203)** in *the table* form a new column group **(page 203)**, anchored at the slot ($x_{max}$-*span*+1, 1), with width *span*, corresponding to the `colgroup` **(page 195)** element.

3. Advance the *current element* to the next child of the `table` **(page 192)**.
4. While the *current element* is not one of the following elements, advance the *current element* to the next child of the `table` **(page 192)**:

   - `colgroup` **(page 195)**
   - `thead` **(page 198)**
   - `tbody` **(page 196)**
   - `tfoot` **(page 198)**
   - `tr` **(page 199)**
5. If the *current element* is a `colgroup` **(page 195)** element, jump to step 2 in these substeps (column groups).

10. Let $y_{current}$ be zero. When the algorithm is aborted, if $y_{current}$ does not equal $y_{max}$, then that is a table model error **(page 203)**.

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` **(page 192)**:

    - `thead` **(page 198)**
    - `tbody` **(page 196)**
    - `tfoot` **(page 198)**
    - `tr` **(page 199)**

13. If the *current element* is a `tr` **(page 199)**, then run the algorithm for processing rows **(page 206)** (defined below), then return to the previous step (rows).

14. Otherwise, run the algorithm for ending a row group **(page 206)**.

15. Let $y_{start}$ have the value $y_{max}$+1.

16. For each `tr` **(page 199)** element that is a child of the *current element*, in tree order, run the algorithm for processing rows **(page 206)** (defined below).

17. If $y_{max} \geq y_{start}$, then let all the last rows **(page 203)** in *the table* from y=$y_{start}$ to y=$y_{max}$ form a new row group **(page 203)**, 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 **(page 206)** again.

19. Return to step 12 (rows).

The **algorithm for ending a row group**, which is invoked by the set of steps above when starting and eding a block of rows, is:

1. If $y_{current}$ is less than $y_{max}$, then this is a table model error **(page 203)**.

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 **(page 207)**.

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` **(page 199)** elements, is:

1. Increase $y_{current}$ by 1.

2. Run the algorithm for growing downward-growing cells **(page 207)**.

3. Let $x_{current}$ be 1.

4. If the `tr` **(page 199)** element being processed contains no `td` **(page 200)** or `th` **(page 201)** elements, then abort this set of steps and return to the algorithm above.

5. Let *current cell* be the first `td` **(page 200)** or `th` **(page 201)** element in the `tr` **(page 199)** 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 $x_{current} \leq x < x_{current}+colspan$ and $y_{current} \leq y < y_{current}+rowspan$ be covered by a new cell **(page 202)** *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` **(page 201)** element, let this new cell *c* be a header cell; otherwise, let it be a data cell. To establish what header cells apply to a data cell, use the algorithm for assigning header cells to data cells **(page 208)** described in the next section.

    If any of the slots involved already had a cell **(page 202)** covering them, then this is a table model error **(page 203)**. 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` **(page 200)** or `th` **(page 201)** element in the `tr` **(page 199)** element being processed, then abort this set of steps and return to the algorithm above.
17. Let *current cell* be the next `td` **(page 200)** or `th` **(page 201)** element in the `tr` **(page 199)** 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 **(page 202)** *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 **(page 203)** in the table **(page 202)** containing only slots that do not have a cell **(page 202)** anchored to them, then this is a table model error **(page 203)**.

*3.15.11.2. [TBW] 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 **(page 21)**:

1. Let ($header_x$, $header_y$) be the coordinate of the slot to which the header cell is anchored.

2. Examine the `scope` **(page 202)** attribute of the `th` **(page 201)** element corresponding to the header cell, and, based on its state, apply the appropriate substep:

   ↪ **If it is in the *row* (page 202) 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* (page 202) 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* (page 202) state**

   If the header cell is not in a row group **(page 203)**, 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* (page 202) state**

   If the header cell is not in a column group **(page 203)**, 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. [TBW] Forms

This section will contain definitions of the `form` element and so forth.

This section will be a rewrite of the HTML4 Forms and Web Forms 2.0 specifications, with hopefully no normative changes.

### 3.16.1. The `form` element

### 3.16.2. The `fieldset` element

### 3.16.3. The `input` element

### 3.16.4. The `button` element

### 3.16.5. The `label` element

### 3.16.6. The `select` element

### 3.16.7. The `datalist` element

### 3.16.8. The `optgroup` element

### 3.16.9. The `option` element

### 3.16.10. The `textarea` element

### 3.16.11. The `output` element

### 3.16.12. Processing model

See WF2 for now

#### *3.16.12.1. Form submission*

See WF2 for now

## 3.17. Scripting

### 3.17.1. The `script` element

Block-level element **(page 67)**, strictly inline-level content **(page 67)**, and metadata element **(page 80)**.

**Contexts in which this element may be used:**

In a `head` **(page 80)** element.
Where block-level elements **(page 67)** are expected.
Where inline-level content **(page 67)** is expected.

**Content model:**

If there is no `src` **(page 211)** attribute, depends on the value of the `type` **(page 211)** attribute.
If there *is* a `src` **(page 211)** attribute, the element must be empty.

**Element-specific attributes:**

`src` **(page 211)**
`defer` **(page 211)** (if the `src` **(page 211)** attribute is present)
`async` **(page 211)** (if the `src` **(page 211)** attribute is present)
`type` **(page 211)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLScriptElement : HTMLElement (page 27) {
           attribute DOMString src (page 214);
           attribute boolean defer (page 214);
           attribute boolean async (page 214);
           attribute DOMString type (page 214);
           attribute DOMString text (page 215);
};
```

The `script` **(page 210)** element allows authors to include dynamic script in their documents.

When the **`src`** attribute is set, the `script` **(page 210)** element refers to an external file. The value of the attribute must be a URI (or IRI).

If the `src` **(page 211)** attribute is not set, then the script is given by the contents of the element.

The language of the script is given by the **`type`** attribute. The value must be a valid MIME type, optionally with parameters. [RFC2046]

The **`defer`** and **`async`** attributes are boolean attributes **(page 48)** that indicate how the script should be executed.

There are three possible modes that can be selected using these attributes. If the `defer` **(page 211)** attribute is present, then the script is executed when the page has finished parsing. If the `defer` **(page 211)** attribute is not present but the `async` **(page 211)** attribute is present, then the script will be executed asynchronously, as soon as it is available. 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.

Changing the `src` **(page 211)**, `type` **(page 211)**, `defer` **(page 211)** and `async` **(page 211)** 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` **(page 210)** elements have two associated pieces of metadata. The first is a flag indicating whether or not the script block has been **"already executed"**. Initially, `script` **(page 210)** elements must have this flag unset (script blocks, when created, are not "already executed"). When a `script` **(page 210)** 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 **(page 373)** and is used to handle `document.write()` **(page 39)** calls.

**Running a script**: when a script block is inserted into a document, the user agent must act as follows:

1. How to handle the `type` **(page 211)** and `language` attributes should be defined here, probably with reference to the next section.

2. If scripting is disabled **(page 266)**, or if the user agent does not support the scripting language in question, or if the `Document` has `designMode` **(page 318)** enabled, or if the `script` **(page 210)** element has its "already executed" **(page 211)** 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" **(page 211)** flag.

4. If the element has a `src` **(page 211)** 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*** **(page 213).**

   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` **(page 211)** 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` (page 211) attribute**

   The element must be added to the end of the list of scripts that will execute when the document has finished parsing **(page 213)**. The user agent ~~must begin the next set of steps **(page 213)** when the script~~ is ~~ready.~~ ~~This isn't compatible with IE for inline deferred scripts, but then what IE does is pretty hard to pin down~~ exactly. Do we want to ~~keep this like it is? Be more compatible?~~

   ↪ **If the element has an `async` (page 211) attribute and a `src` (page 211) attribute**

   The element must be added to the end of the list of scripts that will execute asynchronously **(page 213)**. The user agent must jump to the next set of steps **(page 213)** once the script is ready.

↪ **If the element has an `async` (page 211) attribute but no `src` (page 211) attribute, and the list of scripts that will execute asynchronously (page 213) is not empty**

The element must be added to the end of the list of scripts that will execute asynchronously **(page 213)**.

↪ **If the element has a `src` (page 211) attribute and has been flagged as "parser-inserted" (page 212)**

The element is the script that will execute as soon as the parser resumes **(page 214)**. (There can only be one such script at a time.)

↪ **If the element has a `src` (page 211) attribute**

The element must be added to the end of the list of scripts that will execute as soon as possible **(page 214)**. The user agent must jump to the next set of steps **(page 213)** when the script is ready.

↪ **Otherwise**

The user agent must immediately execute the script **(page 214)**, even if other scripts are already executing.

**When a script completes loading**: If a script whose element was added to one of the lists mentioned above completes loading while the document is still being parsed, then the parser handles it. Otherwise, when a script completes loading, the UA must follow the following steps as soon as as any other scripts that may be executing have finished executing:

↪ **If the script's element was added to the list of scripts that will execute when the document has finished parsing:**

1. If the script's element is not the first element in the list, then do nothing yet. Stop going through these steps.
2. Otherwise, execute the script **(page 214)** (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 **(page 214)** (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` **(page 211)** 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 **(page 214)**.
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 **(page 407)** (amazingly enough).

The download of an external script must delay the `load` event **(page 434)**.

**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 **(page 273)** at the element.

If the load was successful, then first the user agent must fire a `load` event **(page 273)** at the element, and then, if scripting is enabled **(page 266)** and the `Document` does not have `designMode` **(page 318)** enabled, the user agent must run the script according to the semantics of the relevant scripting language defines.

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` **(page 215)** 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` **(page 210)** element as the script to execute.

In any case, the user agent must execute the script according to the semantics of the relevant language specification, as determined by the `type` **(page 211)** and `language` attributes on the `script` **(page 210)** element when the element was inserted into the document, as described above.

Have to define that the script executes in context of global scope **(page 287)**, scripting context **(page 266)**, browsing context **(page 19)**, etc.

***Note: The element's attributes' values might have changed between when the element was inserted into the document and when the script has finished loading, as may its other attributes; similarly, the element itself might have been taken back out of the DOM, or had other changes made. These changes do not in any way affect the above steps; only the values of the attributes at the time the `script` (page 210) element is first inserted into the document matter.***

The DOM attributes **`src`**, **`type`**, **`defer`**, **`async`**, each must reflect **(page 29)** the respective content attributes of the same name.

The DOM attribute **`text`** must return a concatenation of the contents of all the text nodes **(page 22)** that are direct children of the `script` **(page 210)** element (ignoring any other nodes such as comments or elements), in tree order. On setting, it must act the same way as the `textContent` **(page 19)** DOM attribute.

#### *3.17.1.1. Script languages*

The following lists some MIME types and the languages to which they refer:

**`text/javascript`**
: ECMAScript. [ECMA262]

**`text/javascript;e4x=1`**
: ECMAScript with ECMAScript for XML. [ECMA357]

User agents may support other MIME types and other languages.

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

### 3.17.2. The `noscript` element

When scripting is disabled **(page 266)**: transparent **(page 68)** block-level element **(page 67)**, and transparent **(page 68)** strictly inline-level content **(page 67)**.

When scripting is enabled **(page 266)**: block-level element **(page 67)**, and strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**
In a `head` **(page 80)** element of an HTML document **(page 25)**, if there are no ancestor `noscript` **(page 215)** elements.
Where block-level elements **(page 67)** are expected in HTML documents **(page 25)**, if there are no ancestor `noscript` **(page 215)** elements.
Where inline-level content **(page 67)** is expected in HTML documents **(page 25)**, if there are no ancestor `noscript` **(page 215)** elements.

**Content model:**
When scripting is disabled **(page 266)**: transparent **(page 68)**, but there must be no `noscript` **(page 215)** element descendants.
When scripting is enabled **(page 266)**: Text that conforms to the requirements given in the prose.

**Element-specific attributes:**
None.

**Predefined classes that apply to this element:**
None.

**DOM interface:**
No difference from `HTMLElement` **(page 27)**.

The `noscript` **(page 215)** 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` **(page 215)** element must not be used in XML documents **(page 25)**.

When used in HTML documents **(page 25)**, the allowed content model depends on whether scripting is enabled or not.

If scripting is disabled **(page 266)**, then the content model of a `noscript` **(page 215)** element is transparent **(page 68)**, with the additional restriction that a `noscript` **(page 215)** element must not have a `noscript` **(page 215)** element as an ancestor (that is, `noscript` **(page 215)** can't be nested).

If scripting is enabled **(page 266)**, then the content model of a `noscript` **(page 215)** element is text, except that the text must be such that running the following algorithm results in a conforming document with no `noscript` **(page 215)** elements and no `script` **(page 210)** elements, and such that no step in the algorithm causes an HTML parser **(page 373)** to flag a parse error **(page 374)**:

1. Remove every `script` **(page 210)** element from the document.
2. Make a list of every `noscript` **(page 215)** element in the document. For every `noscript` **(page 215)** element in that list, perform the following steps:
   1. Let the *parent element* be the parent element of the `noscript` **(page 215)** element.
   2. Take all the children of the *parent element* that come before the `noscript` **(page 215)** element, and call these elements *the before children*.
   3. Take all the children of the *parent element* that come *after* the `noscript` **(page 215)** element, and call these elements *the after children*.
   4. Let *s* be the concatenation of all the text node **(page 22)** children of the `noscript` **(page 215)** element.
   5. Set the `innerHTML` **(page 39)** attribute of the *parent element* to the value of *s*. (This, as a side-effect, causes the `noscript` **(page 215)** 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` **(page 215)** element has no other requirements. In particular, children of the `noscript` **(page 215)** 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`*** **(page 215)** ***element causes the HTML parser*** **(page 373)** ***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

Block-level element **(page 67)**, strictly inline-level content **(page 67)**, and metadata element **(page 80)**.

**Contexts in which this element may be used:**

In a `head` **(page 80)** element.
Where block-level elements **(page 67)** are expected.
Where inline-level content **(page 67)** is expected.

**Content model:**

Empty.

**Element-specific attributes:**

`src` **(page 217)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLEventSourceElement : HTMLElement (page 27) {
           attribute DOMString src (page 218);
};
```

The `event-source` **(page 217)** 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` **(page 217)** attribute specified, the user agent must act as if the `addEventSource()` **(page 342)** method on the `event-source` **(page 217)** element had been invoked with the URI resulting from resolving the `src` **(page 217)** attribute's value to an absolute URI.

While the element is in a document, if its `src` **(page 217)** attribute is mutated, the user agent must act as if first the `removeEventSource()` **(page 342)** method on the `event-source` **(page 217)** 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()` **(page 342)** method on the element had been invoked with the URI resulting from resolving the *new* value of the `src` **(page 217)** attribute to an absolute URI.

When the element is removed from the document, if it has the `src` **(page 217)** attribute specified, or, when the `src` **(page 217)** attribute is about to be removed, the user agent must act as if the `removeEventSource()` **(page 342)** method on the `event-source` **(page 217)** element had been invoked with the URI resulting from resolving the `src` **(page 217)** attribute's value to an absolute URI.

There can be more than one `event-source` **(page 217)** 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

Interactive **(page 69)**, block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected.

**Content model:**
One `legend` **(page 254)** element followed by either one or more block-level elements **(page 67)** or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**
`open` **(page 218)**

**Predefined classes that apply to this element:**
None.

**DOM interface:**

```
interface HTMLDetailsElement : HTMLElement (page 27) {
  attribute boolean open (page 218);
};
```

The `details` **(page 218)** element represents additional information or controls which the user can obtain on demand.

The first element child of a `details` **(page 218)** element, if it is a `legend` **(page 254)** element, represents the summary of the details.

If the first element is not a `legend` **(page 254)** element, the UA should provide its own legend (e.g. "Details").

The **`open`** content attribute is a boolean attribute **(page 48)**. 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 **(page 29)** the `open` **(page 218)** content attribute.

Rendering will be described in the Rendering section in due course. Basically CSS :open and :closed match the element, it's a block-level element by default, and when it matches :closed it renders as if it had an XBL binding attached to it whose template was just `<template>▶<content includes="legend:first-child">Details</content></template>`, and when it's :open it acts as if it had an XBL binding attached to it whose template was just `<template>▼<content includes="legend:first-child">Details</content><content/></template>` or some such.

Clicking the legend would make it open/close (and would change the content attribute). Question: Do we want the content attribute to reflect the actual state like this? I think we do, the DOM not reflecting state has been a pain in the neck before. But is it semantically ok?

### 3.18.2. [SCS] The `datagrid` element

Interactive **(page 69)**, block-level element **(page 67)**.

**Contexts in which this element may be used:**
Where block-level elements **(page 67)** are expected, if there are no ancestor interactive elements **(page 69)**.

**Content model:**
Zero or more block-level elements **(page 67)**.

**Element-specific attributes:**
`multiple` **(page 241)**
`disabled` **(page 239)**

**Predefined classes that apply to this element:**
None.

**DOM interface:**

```
interface HTMLDataGridElement : HTMLElement (page 27) {
           attribute DataGridDataProvider (page 221) data (page 222);
  readonly attribute DataGridSelection (page 240) selection (page 240);
           attribute boolean multiple (page 220);
           attribute boolean disabled (page 220);
  void updateEverything (page 238)();
  void updateRowsChanged (page 239)(in RowSpecification (page 221) row,
in unsigned long count);
  void updateRowsInserted (page 239)(in RowSpecification (page 221) row,
in unsigned long count);
  void updateRowsRemoved (page 239)(in RowSpecification (page 221) row,
in unsigned long count);
  void updateRowChanged (page 239)(in RowSpecification (page 221) row);
  void updateColumnChanged (page 239)(in unsigned long column);
  void updateCellChanged (page 239)(in RowSpecification (page 221) 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` **(page 219)** 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` **(page 219)** element, or from a scripted data provider given by the `data` **(page 222)** DOM attribute.

The `multiple` **(page 241)** and `disabled` **(page 239)** attributes are boolean attributes **(page 48)**. Their effects are described in the processing model sections below.

The **`multiple`** and **`disabled`** DOM attributes must reflect **(page 29)** the `multiple` **(page 241)** and `disabled` **(page 239)** content attributes respectively.

### *3.18.2.1. The `datagrid`* **(page 219)** *data model*

*This section is non-normative.*

In the `datagrid` **(page 219)** 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` **(page 219)** operates at the row level. If the `multiple` **(page 241)** attribute is present, multiple rows can be selected at once, otherwise the user can only select one row at a time.

The `datagrid` **(page 219)** element can be disabled entirely by setting the `disabled` **(page 239)** 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 **(page 224)** of the `datagrid` **(page 219)** element, and are also passed to the style system **(page 439)**. They are similar in concept to the `class` **(page 73)** 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 **(page 221)** 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 **(page 221)**, as is `1` on its own. However, `[1,0.2,9]` is not a RowSpecification **(page 221)** object, since its second value is not an integer.

User agents must always represent `RowSpecification` **(page 221)**s in ECMAScript by using arrays, even if the path only has one number.

The root of the tree is represented by the empty path; in ECMAScript, this is the empty array (`[]`). Only the `getRowCount()` **(page 222)** and `GetChildAtPosition()` **(page 222)** 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` **(page 221)**, *including (and most commonly) the content author's implementation(s).*

```
// To be implemented by Web authors as a JS object
interface DataGridDataProvider {
  void initialize (page 222)(in HTMLDataGridElement datagrid);
  unsigned long getRowCount (page 222)(in RowSpecification (page 221) row);
  unsigned long getChildAtPosition (page 222)(in RowSpecification (page 221)
parentRow, in unsigned long position);
  unsigned long getColumnCount (page 222)();
  DOMString getCaptionText (page 222)(in unsigned long column);
  void getCaptionClasses (page 223)(in unsigned long column, in DOMTokenList
classes);
  DOMString getRowImage (page 223)(in RowSpecification (page 221) row);
  HTMLMenuElement getRowMenu (page 223)(in RowSpecification (page 221) row);
  void getRowClasses (page 223)(in RowSpecification (page 221) row, in
DOMTokenList classes);
  DOMString getCellData (page 223)(in RowSpecification (page 221) row, in
unsigned long column);
  void getCellClasses (page 223)(in RowSpecification (page 221) row, in
unsigned long column, in DOMTokenList classes);
  void toggleColumnSortState (page 223)(in unsigned long column);
  void setCellCheckedState (page 224)(in RowSpecification (page 221) row, in
unsigned long column, in long state);
  void cycleCell (page 224)(in RowSpecification (page 221) row, in unsigned
long column);
  void editCell (page 224)(in RowSpecification (page 221) row, in unsigned
long column, in DOMString data);
};
```

The `DataGridDataProvider` **(page 221)** interface represents the interface that objects must implement to be used as custom data views for `datagrid` **(page 219)** elements.

Not all the methods are required. The minimum number of methods that must be implemented in a useful view is two: the `getRowCount()` **(page 222)** and `getCellData()` **(page 223)** methods.

Once the object is written, it must be hooked up to the `datagrid` **(page 219)** using the **`data`** DOM attribute.

The following methods may be usefully implemented:

**`initialize(`*`datagrid`*`)`**

Called by the `datagrid` **(page 219)** 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` **(page 219)** element when it is first loaded. The data provider could also use this method call to register a `select` **(page 241)** event handler on the `datagrid` **(page 219)** 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` **(page 219)** 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`*`)` **(page 222)**.

**`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` **(page 219)**'s `updateEverything()` **(page 238)** 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` **(page 219)**'s `updateColumnChanged()` **(page 239)** 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` **(page 219)**'s `updateColumnChanged()` **(page 239)** method must be called with the appropriate column index. Some classes have predefined meanings **(page 224)**.

**`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` **(page 219)**'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` **(page 219)**'s update methods must be called to update the row in question. Some classes have predefined meanings **(page 224)**.

**`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 **(page 225)**, 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` **(page 219)**'s update methods must be called to update the rows that changed. If only one cell changed, the `updateCellChanged()` **(page 239)** 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` **(page 219)**'s update methods must be called to update the rows or cells in question. Some classes have predefined meanings **(page 224)**.

**`toggleColumnSortState(`*`column`*`)`**

Called by the `datagrid` **(page 219)** when the user tries to sort the data using a particular column *column*. The data provider must update its state so that the `GetChildAtPosition()` **(page 222)** method returns the new order, and the classes of the columns returned by `getCaptionClasses()` **(page 223)** represent the new sort status. There is no need to tell the `datagrid` **(page 219)** that it the data has changed, as the `datagrid` **(page 219)** automatically assumes that the entire data model will need updating.

**`setCellCheckedState(`*`row, column, state`*`)`**

Called by the `datagrid` **(page 219)** 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` **(page 219)** that the cell has changed, as the `datagrid` **(page 219)** automatically assumes that the given cell will need updating.

**`cycleCell(`*`row, column`*`)`**

Called by the `datagrid` **(page 219)** 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` **(page 219)** that the cell has changed, as the `datagrid` **(page 219)** automatically assumes that the given cell will need updating.

**`editCell(`*`row, column, data`*`)`**

Called by the `datagrid` **(page 219)** 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` **(page 219)** that the cell has changed, as the `datagrid` **(page 219)** 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` **(page 224)** and `progress` **(page 225)** classes override this, though.) |
| **`cyclable`** | Cells | The cell can be cycled through multiple values. (The `progress` **(page 225)** class overrides this, though.) |
| **`editable`** | Cells | The cell can be edited. (The `cyclable` **(page 224)**, `progress` **(page 225)**, `checked` **(page 224)**, `unchecked` **(page 224)** and `indeterminate` **(page 224)** 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` **(page 224)** nor `unchecked` **(page 224)** classes are present, then the checkbox is in that state, too. (The `cyclable` **(page 224)** and `progress` **(page 225)** classes override this, though.) |
| **`initially-hidden`** | Columns | The column will not be shown when the `datagrid` **(page 219)** is initially rendered. If |

| | | |
|---|---|---|
| | | this class is not present on the column when the `datagrid` **(page 219)** is initially rendered, the column will be visible if space allows. |
| **`initially-closed`** | Rows | The row will be closed when the `datagrid` **(page 219)** is initially rendered. If neither this class nor the `initially-open` **(page 225)** class is present on the row when the `datagrid` **(page 219)** is initially rendered, the initial state will depend on platform conventions. |
| **`initially-open`** | Rows | The row will be opened when the `datagrid` **(page 219)** is initially rendered. If neither this class nor the `initially-closed` **(page 225)** class is present on the row when the `datagrid` **(page 219)** 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` **(page 224)** and `separator` **(page 225)** classes override this, though.) |
| **`separator`** | Rows | The row is a separator row, not a data row. (The `header` **(page 224)** 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` **(page 225)** class is also present, the sort direction is ascending. |
| **`unchecked`** | Cells | The cell has a checkbox and, unless the `checked` **(page 224)** class is present as well, it is unchecked. (The `cyclable` **(page 224)** and `progress` **(page 225)** 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` **(page 219)**'s `data` **(page 222)** 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` **(page 219)**.

↪ **While the first element child is a `table` (page 192)**

**`getRowCount(row)` (page 222)**: 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` **(page 199)** elements that are children of `tbody` **(page 196)** elements that are children of the `table` **(page 192)**, if there are any such child `tbody` **(page 196)** elements. If there are no such `tbody` **(page 196)** elements then the number of rows returned for the root must be the number of `tr` **(page 199)** elements that are children of the `table` **(page 192)**.

When *row* is not empty, the number of rows returned must be zero.

> ***Note: The `table`* (page 192)*-based default data provider cannot represent a tree.***

> ***Note: Rows in `thead`* (page 198) *elements do not contribute to the number of rows returned, although they do affect the columns and column captions. Rows in `tfoot`* (page 198) *elements are ignored* (page 21) *completely by this algorithm.***

**`getChildAtPosition(row, i)` (page 222)**: The default data provider must return the mapping appropriate to the current sort order **(page 227)**.

**`getColumnCount()` (page 222)**: The number of columns returned must be the number of `td` **(page 200)** element children in the first `tr` **(page 199)** element child of the first `tbody` **(page 196)** element child of the `table` **(page 192)**, if there are any such `tbody` **(page 196)** elements. If there are no such `tbody` **(page 196)** elements, then it must be the number of `td` **(page 200)** element children in the first `tr` **(page 199)** element child of the `table` **(page 192)**, 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)` (page 222)**: If the `table` **(page 192)** has no `thead` **(page 198)** element child, or if its first `thead` **(page 198)** element child has no `tr` **(page 199)** element child, the default data provider must return the empty string for all captions. Otherwise, the value of the `textContent` **(page 19)** attribute of the *i*th `th` **(page 201)** element child of the first `tr` **(page 199)** element child of the first `thead` **(page 198)** element child of the `table` **(page 192)** element must be returned. If there is no such `th` **(page 201)** element, the empty string must be returned.

**`getCaptionClasses(i, classes)` (page 223)**: If the `table` **(page 192)** has no `thead` **(page 198)** element child, or if its first `thead` **(page 198)** element child has no `tr` **(page 199)** element child, the default data provider must not add any classes for any of the captions. Otherwise, each class in the `class` **(page 73)** attribute of the *i*th `th` **(page 201)** element child of the first `tr` **(page 199)** element child of the first `thead` **(page 198)** element child of the `table` **(page 192)** element must be added to the *classes*. If there is no such `th` **(page 201)** element, no classes must be added. The user agent must then:

1. Remove the `sorted` **(page 225)** and `reversed` **(page 225)** classes.

2. If the `table` **(page 192)** element has a `class` **(page 73)** attribute that includes the `sortable` class, add the `sortable` **(page 225)** class.

3. If the column is the one currently being used to sort the data, add the `sorted` **(page 225)** 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` **(page 225)** 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` **(page 192)** 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` **(page 192)** has `tbody` **(page 196)** element children, the element for the *i*th row is the *i*th `tr` **(page 199)** element that is a child of a `tbody` **(page 196)** element that is a child of the `table` **(page 192)** element. If the `table` **(page 192)** does not have `tbody` **(page 196)** element children, then the element for the *i*th real row is the *i*th `tr` **(page 199)** element that is a child of the `table` **(page 192)** element.

**For cells**: Given a row and its element, the row's *i*th cell's element is the *i*th `td` **(page 200)** element child of the row element.

> ***Note: The `colspan` and `rowspan` attributes are ignored* (page 21) *by this algorithm.***

**`getRowImage(i)`** **(page 223)**: If the row's first cell's element has an `img` **(page 148)** element child, then the URI of the row's image is the URI of the first `img` **(page 148)** element child of the row's first cell's element. Otherwise, the URI of the row's image is the empty string.

**`getRowMenu(i)`** **(page 223)**: If the row's first cell's element has a `menu` **(page 245)** element child, then the row's menu is the first `menu` **(page 245)** element child of the row's first cell's element. Otherwise, the row has no menu.

**`getRowClasses(i, classes)`** **(page 223)**: The default data provider must never add a class to the row's classes.

**`toggleColumnSortState(i)`** **(page 223)**: 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()` **(page 222)** method for the default data provider will change appropriately **(page 226)**.

**`getCellData(`*`i`*`, `*`j`*`)` (page 223), `getCellClasses(`*`i`*`, `*`j`*`, `*`classes`*`)` (page 223), `getCellCheckedState(`*`i`*`, `*`j`*`, `*`state`*`)` (page 224), `cycleCell(`*`i`*`, `*`j`*`)` (page 224), and `editCell(`*`i`*`, `*`j`*`, `*`data`*`)` (page 224)**: See the common definitions below **(page 231)**.

The data provider must call the `datagrid` **(page 219)**'s update methods appropriately whenever the descendants of the `datagrid` **(page 219)** mutate. For example, if a `tr` **(page 199)** is removed, then the `updateRowsRemoved()` **(page 239)** methods would probably need to be invoked, and any change to a cell or its descendants must cause the cell to be updated. If the `table` **(page 192)** element stops being the first child of the `datagrid` **(page 219)**, then the data provider must call the `updateEverything()` **(page 238)** method on the `datagrid` **(page 219)**. Any change to a cell that is in the column that the data provider is currently using as its sort column must also cause the sort to be reperformed, with a call to `updateEverything()` **(page 238)** if the change did affect the sort order.

↪ **While the first element child is a `select`**

The default data provider must return 1 for the column count, the empty string for the column's caption, and must not add any classes to the column's classes.

For the rows, assume the existence of a node filter view of the descendants of the first `select` element child of the `datagrid` **(page 219)** element, that skips all nodes other than `optgroup` and `option` elements, as well as any descendents of any `option` elements.

Given a path *row*, the corresponding element is the one obtained by drilling into the view, taking the child given by the path each time.

> Given the following XML markup:
>
> ```
> <datagrid>
>  <select>
>   <!-- the options and optgroups have had their labels and
> values removed
>        to make the underlying structure clearer -->
>   <optgroup>
>    <option/>
>    <option/>
>   </optgroup>
>   <optgroup>
>    <option/>
>    <optgroup id="a">
>     <option/>
>     <option/>
>     <bogus/>
>     <option id="b"/>
> ```

```
    </optgroup>
    <option/>
   </optgroup>
  </select>
 </datagrid>
```

The path "1,1,2" would select the element with ID "b". In the filtered view, the text nodes, comment nodes, and bogus elements are ignored; so for instance, the element with ID "a" (path "1,1") has only 3 child nodes in the view.

`getRowCount(row)` **(page 222)** must drill through the view to find the element corresponding to the method's argument, and return the number of child nodes in the filtered view that the corresponding element has. (If the *row* is empty, the corresponding element is the `select` element at the root of the filtered view.)

`getChildAtPosition(row, position)` **(page 222)** must return *position*. (The `select` default data provider does not support sorting the data grid.)

`getRowImage(i)` **(page 223)** must return the empty string, `getRowMenu(i)` **(page 223)** must return null.

`getRowClasses(row, classes)` **(page 223)** 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` **(page 224)**
- If the *row*'s corresponding element contains other elements that are also in the view, and the element's `class` **(page 73)** attribute contains the `closed` class: `initially-closed` **(page 225)**
- If the *row*'s corresponding element contains other elements that are also in the view, and the element's `class` **(page 73)** attribute contains the `open` class: `initially-open` **(page 225)**

The `getCellData(row, cell)` **(page 223)** 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` **(page 19)** DOM attribute.

The `getCellClasses(row, cell, classes)` **(page 223)** 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` **(page 219)**'s update methods appropriately whenever the descendants of the `datagrid` **(page 219)** 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` **(page 219)** that skips all nodes other than `li` **(page 114)**, `h1` **(page 98)**-`h6` **(page 98)**, and `hr` **(page 109)** elements, and skips any descendants of `menu` **(page 245)** 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`*`)` **(page 222)** 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` **(page 114)** elements actually appear in the data grid; the `ol` **(page 112)** element does not affect the data grid's processing model.

`getChildAtPosition(`*`row, position`*`)` **(page 222)** must return *position*. (The generic default data provider does not support sorting the data grid.)

`getRowImage(`*`i`*`)` **(page 223)** must return the URI of the image given by the first `img` **(page 148)** 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)` **(page 223)** must return the first `menu` **(page 245)** 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)` **(page 223)** 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` **(page 73)** attribute contains the `closed` class: `initially-closed` **(page 225)**
- If the row's element contains other elements that are also in the view, and the element's `class` **(page 73)** attribute contains the `open` class: `initially-open` **(page 225)**
- If the row's element is an `h1` **(page 98)**-`h6` **(page 98)** element: `header` **(page 224)**
- If the row's element is an `hr` **(page 109)** element: `separator` **(page 225)**

The `getCellData(i, j)` **(page 223)**, `getCellClasses(i, j, classes)` **(page 223)**, `getCellCheckedState(i, j, state)` **(page 224)**, `cycleCell(i, j)` **(page 224)**, and `editCell(i, j, data)` **(page 224)** methods must act as described in the common definitions below **(page 231)**, treating the row's element as being the cell's element.

selection handling?

The data provider must call the `datagrid` **(page 219)**'s update methods appropriately whenever the descendants of the `datagrid` **(page 219)** 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` **(page 219)**'s child list changes such that there is a first element child, then the data provider must call the `updateEverything()` **(page 238)** method on the `datagrid` **(page 219)**.

#### 3.18.2.4.1. Common default data provider method definitions for cells

These definitions are used for the cell-specific methods of the default data providers (other than in the `select` case). How they behave is based on the contents of an element that represents the cell given by their first two arguments. Which element that is is defined in the previous section.

**Cyclable cells**

If the first element child of a cell's element is a `select` element that has a no `multiple` attribute and has at least one `option` element descendent, then the cell acts as a cyclable cell.

The "current" `option` element is the selected `option` element, or the first `option` element if none is selected.

The `getCellData()` **(page 223)** method must return the `textContent` **(page 19)** of the current `option` element (the `label` attribute is ignored **(page 21)** in this context as the `optgroup`s are not displayed).

The `getCellClasses()` **(page 223)** method must add the `cyclable` **(page 224)** class and then all the classes of the current `option` element.

The `cycleCell()` **(page 224)** 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 **(page 21)**). 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()` **(page 224)** and `editCell()` **(page 224)** methods must do nothing.

**Progress bar cells**

If the first element child of a cell's element is a `progress` **(page 134)** element, then the cell acts as a progress bar cell.

The `getCellData()` **(page 223)** method must return the value returned by the `progress` **(page 134)** element's `position` **(page 137)** DOM attribute.

The `getCellClasses()` **(page 223)** method must add the `progress` **(page 225)** class.

The `setCellCheckedState()` **(page 224)**, `cycleCell()` **(page 224)**, and `editCell()` **(page 224)** 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()` **(page 223)** method must return the `textContent` **(page 19)** of the cell element.

The `getCellClasses()` **(page 223)** method must add the `checked` **(page 224)** class if the `input` element is checked, and the `unchecked` **(page 225)** class otherwise.

The `setCellCheckedState()` **(page 224)** 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()` **(page 224)** and `editCell()` **(page 224)** 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()` **(page 223)** method must return the `value` of the `input` element.

The `getCellClasses()` **(page 223)** method must add the `editable` **(page 224)** class.

The `editCell()` **(page 224)** method must set the `input` element's `value` DOM attribute to the value of the third argument to the method.

The `setCellCheckedState()` **(page 224)** and `cycleCell()` **(page 224)** methods must do nothing.

### *3.18.2.5. Populating the `datagrid` **(page 219)** element*

A `datagrid` **(page 219)** must be disabled until its end tag has been parsed (in the case of a `datagrid` **(page 219)** 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` **(page 219)** must then populate itself using the data provided by the data provider assigned to the `data` **(page 222)** DOM attribute. After the view is populated (using the methods described below), the `datagrid` **(page 219)** must invoke the `initialize()` **(page 222)** method on the data provider specified by the `data` **(page 222)** attribute, passing itself (the `HTMLDataGridElement` **(page 219)** object) as the only argument.

When the `data` **(page 222)** attribute is null, the `datagrid` **(page 219)** 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()` **(page 222)** 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()` **(page 222)** method with the index of the column in question. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of columns. The return value is the string to use when referring to that column. If the method returns null or the empty string, the column has no caption. If the method is not defined, then none of the columns have any captions.

**To establish what classes apply to a column**

Invoke the `getCaptionClasses()` **(page 223)** method with the index of the column in question, and an object implementing the `DOMTokenList` **(page 33)** 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` **(page 33)** 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` **(page 224)** 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` **(page 225)** 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` **(page 225)** 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` **(page 225)** 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` **(page 225)** 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()` **(page 222)** method with a `RowSpecification` **(page 221)** 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()` **(page 222)** method with a `RowSpecification` **(page 221)** 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()` **(page 222)** method with a `RowSpecification` **(page 221)** 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()` **(page 222)** 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()` **(page 222)** 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()` **(page 222)** 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()` **(page 223)** method with a `RowSpecification` **(page 221)** object representing the row in question, and a `DOMTokenList` **(page 33)** associated with an empty string. The tokens contained in the `DOMTokenList` **(page 33)** 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` **(page 224)** 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` **(page 225)** class applies to the row, then in the place of the row, a separator should be shown. Otherwise, if the `selectable-separator` **(page 225)** class applies to the row, then the row should be a data row, but represented as a separator. (The difference between a `separator` **(page 225)** and a `selectable-separator` **(page 225)** 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 **(page 236)**, examine the classes that apply to the row. If the `initially-open` **(page 225)** class applies to the row, then it should be initially open. Otherwise, if the `initially-closed` **(page 225)** 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()` **(page 223)** method with a `RowSpecification` **(page 221)** 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` **(page 219)**'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()` **(page 223)** method with a `RowSpecification` **(page 221)** object representing the row in question. The return value is a reference to an object implementing the `HTMLMenuElement` interface, i.e. a `menu` **(page 245)** 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` **(page 219)** element.

**To establish the value of a particular cell**

Invoke the `getCellData()` **(page 223)** method with the first argument being a `RowSpecification` **(page 221)** 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()` **(page 223)** method with the first argument being a `RowSpecification` **(page 221)** 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` **(page 33)** 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` **(page 33)** 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` **(page 225)** class applies to the cell, it is a progress bar. Otherwise, if the `cyclable` **(page 224)** 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 **(page 49)** in the range $0.0 \le x \le 1.0$, 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` **(page 224)**, `unchecked` **(page 225)**, or `indeterminate` **(page 224)** 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` **(page 224)** 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` **(page 224)** class applies to the cell. If it does, the cell is checked. Otherwise, check whether the `unchecked` **(page 225)** class applies to the cell. If it does, the cell is unchecked. Otherwise, the `indeterminate` **(page 224)** class appplies to the cell and the cell's checkbox is in an indeterminate state. When the `indeterminate` **(page 224)** 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` **(page 224)** and/or `unchecked` **(page 225)** 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` **(page 219)** is invoking one of its methods, the `datagrid` **(page 219)** must act, for the purposes of that particular method call, as if the relevant method had not been defined.

A `RowSpecification` **(page 221)** 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()` **(page 222)** method when it was passed the `RowSpecification` **(page 221)** 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` **(page 219)** element. If a user agent is returned inconsistent data, for example if the number of rows returned by `getRowCount()` **(page 222)** varies in ways that do not match the calls made to the update methods, the user agent may disable the `datagrid` **(page 219)**. User agents that do not disable the `datagrid` **(page 219)** 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` **(page 223)** 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`* **(page 219)**

Whenever the `data` **(page 222)** attribute is set to a new value, the `datagrid` **(page 219)** 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` **(page 219)** 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` **(page 219)**. 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)` **(page 239)**.

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` **(page 219)**'s selection is not considered a change to the selection, and must therefore not fire the `select` **(page 241)** 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()` **(page 239)** and `updateRowsRemoved()` **(page 239)** methods without actually inserting or removing rows from the data provider is likely to result in inconsistent renderings **(page 238)**, 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` **(page 219)** element has a **`disabled`** attribute, then the user agent must disable the `datagrid` **(page 219)**, preventing the user from interacting with it. The `datagrid` **(page 219)** element should still continue to update itself when the data provider signals changes to the data, though. Obviously, conformance requirements stating that `datagrid` **(page 219)** elements must react to users in particular ways do not apply when one is disabled.

If a row is openable **(page 236)**, 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 **(page 237)**, 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` **(page 219)** must invoke the data provider's `cycleCell()` **(page 224)** method, with a `RowSpecification`

**(page 221)** object representing the cell's row as the first argument and the cell's column index as the second. The `datagrid` **(page 219)** must act as if the `datagrid` **(page 219)**'s `updateCellChanged()` **(page 239)** method had been invoked with those same arguments immediately before the provider's method was invoked.

When a cell has a checkbox **(page 237)**, 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` **(page 219)** must invoke the data provider's `setCellCheckedState()` **(page 224)** method, with a `RowSpecification` **(page 221)** 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` **(page 224)** class set). The `datagrid` **(page 219)** must act as if the `datagrid` **(page 219)**'s `updateCellChanged()` **(page 239)** method had been invoked, specifying the same cell, immediately before the provider's method was invoked.

If a cell is editable **(page 237)**, the user must be able to edit the data for that cell, and doing so must cause the user agent to invoke the `editCell()` **(page 224)** method of the data provider with three arguments: a `RowSpecification` **(page 221)** 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()` **(page 239)** 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` **(page 240)** attribute must return null.*

```
interface DataGridSelection {
  readonly attribute unsigned long length;
  RowSpecification (page 221) item(in unsigned long index);
  boolean isSelected (page 241)(in RowSpecification (page 221) row);
  void setSelected (page 241)(in RowSpecification (page 221) row, in boolean
selected);

  void selectAll (page 241)();
  void invert (page 241)();
  void clear (page 241)();
};
```

Each `datagrid` **(page 219)** 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` **(page 219)** is represented by its **`selection`** DOM attribute, which must be a `DataGridSelection` **(page 240)** object.

`DataGridSelection` **(page 240)** 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()` **(page 241)**, the `length` **(page 241)** 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()` **(page 241)**, the `length` **(page 241)** attribute will return zero.

If the `datagrid` **(page 219)** 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`*** **(page 241)** ***attribute is absent.***

Whenever the selection of a `datagrid` **(page 219)** changes, whether due to the user interacting with the element, or as a result of calls to methods of the `selection` **(page 240)** object, a **`select`** event that bubbles but is not cancelable must be fired on the `datagrid` **(page 219)** element. If changes are made to the selection via calls to the object's methods during the execution of a script, then the `select` **(page 241)** events must be coalesced into one, which must then be fired when the script execution has completed.

> ***Note: The `DataGridSelection` (page 240) interface has no relation to the `Selection` (page 337) interface.***

### *3.18.2.9. Columns and captions*

*This section only applies to interactive user agents.*

Each `datagrid` **(page 219)** 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` **(page 224)** 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()` **(page 222)** 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()` (page 222) method of the data provider.***

If a column is sortable **(page 234)**, then the user must be able to invoke it to sort the data. When the user does so, then the `datagrid` **(page 219)** must invoke the data provider's `toggleColumnSortState()` **(page 223)** method, with the column's index as the only argument. The `datagrid` **(page 219)** must *then* act as if the `datagrid` **(page 219)**'s `updateEverything()` **(page 238)** method had been invoked.

## 3.18.3. The `command` element

Metadata element **(page 80)**, and strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

In a `head` **(page 80)** element.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

Empty.

**Element-specific attributes:**

`type` **(page 243)**
`label` **(page 243)**
`icon` **(page 243)**

`hidden` **(page 243)**
`disabled` **(page 243)**
`checked` **(page 244)**
`radiogroup` **(page 244)**
`default` **(page 244)**
Also, the `title` **(page 243)** attribute has special semantics on this element.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLCommandElement : HTMLElement (page 27) {
           attribute DOMString type (page 244);
           attribute DOMString label (page 244);
           attribute DOMString icon (page 244);
           attribute boolean hidden (page 244);
           attribute boolean disabled (page 244);
           attribute boolean checked (page 244);
           attribute DOMString radiogroup (page 244);
           attribute boolean default (page 244);
 void click (page 244)(); // shadows HTMLElement (page 27).click() (page
78)
};
```

The `Command` **(page 250)** interface must also be implemented by this element.

The `command` **(page 242)** 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 **(page 48)** that, if present, indicates that the command is not relevant and is to be hidden.

The **`disabled`** attribute is a boolean attribute **(page 48)** that, if present, indicates that the command is not available in the current state.

***Note: The distinction between Disabled State* (page 249) *and Hidden State* (page 249) *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 **(page 48)** 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` **(page 243)** attribute has the value "`radio`". The scope of the name is the child list of the parent element.

If the `command` **(page 242)** 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` **(page 244)** attribute is a boolean attribute **(page 48)**.

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` **(page 242)** attributes off, and that has a default command.

The **`type`**, **`label`**, **`icon`**, **`hidden`**, **`disabled`**, **`checked`**, **`radiogroup`**, and **`default`** DOM attributes must reflect **(page 29)** their respective namesake content attributes.

The **`click()`** method's behaviour depends on the value of the `type` **(page 243)** attribute of the element, as follows:

↪ **If the `type` (page 243) attribute has the value `checkbox`**

If the element has a `checked` **(page 244)** attribute, the UA must remove that attribute. Otherwise, the UA must add a `checked` **(page 244)** attribute, with the literal value `checked`. The UA must then fire a `click` event **(page 273)** at the element.

↪ **If the `type` (page 243) attribute has the value `radio`**

If the element has a parent, then the UA must walk the list of child nodes of that parent element, and for each node that is a `command` **(page 242)** element, if that element has a `radiogroup` **(page 244)** attribute whose value exactly matches the current element's (treating missing `radiogroup` **(page 244)** attributes as if they were the empty string), and has a `checked` **(page 244)** attribute, must remove that attribute and fire a `click` event **(page 273)** at the element.

Then, the element's `checked` **(page 244)** 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 **(page 273)** at the element.

***Note: Firing a synthetic `click` event at the element does not cause any of the actions described above to happen.***

should change all the above so it actually is just trigged by a click event, then we could remove the shadowing click() method and rely on actual events.

Need to define the command="" attribute

***Note: `command`*** **(page 242)** ***elements are not rendered unless they form part of a menu*** **(page 245)*****.***

## 3.18.4. The `menu` element

Block-level element **(page 67)**, and structured inline-level element **(page 68)**.

**Contexts in which this element may be used:**

Where block-level elements **(page 67)** are expected.
Where structured inline-level elements **(page 68)** are allowed.

**Content model:**

Zero or more `li` **(page 114)** elements, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**

`type` **(page 245)**
`label` **(page 246)**
`autosubmit` **(page 246)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLCommandElement : HTMLElement (page 27) {
           attribute DOMString type;
           attribute DOMString label;
           attribute boolean autosubmit;
};
```

The `menu` **(page 245)** element represents a list of commands.

The **`type`** attribute indicates the kind of menu. It must have either the value `popup` (to declare a context menu) or the value `toolbar` (to define a tool bar). The attribute may also be omitted, to indicate that the element is merely a list of commands that is neither declaring a context menu nor defining a tool bar.

If a menu **(page 245)** element has a type **(page 245)** attribute with the value popup, then it represents the commands of a context menu, and the user can only interact with the commands if that context menu is activated.

If a menu **(page 245)** element has a type **(page 245)** attribute with the value toolbar, then it represents a list of active commands that the user can immediately interact with.

Otherwise, if a menu **(page 245)** element has no type **(page 245)** attribute, or if has a type **(page 245)** attribute with a value other than popup or toolbar, then it either represents an unordered list of items (each represented by an li **(page 114)** element), each of which represents a command that the user may perform or activate, or, if the element has no li **(page 114)** element children, a paragraph **(page 70)** 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 **(page 246)** attribute for the submenu's menu label.

The **autosubmit** attribute is a boolean attribute **(page 48)** 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 **(page 245)** 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 **(page 245)** element has an autosubmit **(page 246)** 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. [TBW] Introduction*

*This section is non-normative.*

...

### *3.18.4.2. Building menus*

A menu consists of a list of zero or more of the following components:

- Commands **(page 249)**, which can be marked as default commands
- Separators
- Other menus (which allows the list to be nested)

The list corresponding to a particular menu **(page 245)** element is built by iterating over its child nodes. For each child node in tree order **(page 21)**, the required behaviour depends on what the node is, as follows:

↪ **An element that defines a command (page 249)**

Append the command to the menu. If the element is a `command` **(page 242)** element with a `default` **(page 244)** attribute, mark the command as being a default command.

↪ **An `hr` (page 109) element**

↪ **An `option` element that has a `value` attribute set to the empty string, and has a `disabled` attribute, and whose `textContent` (page 19) consists of a string of one or more hyphens (U+002D HYPHEN-MINUS)**

Append a separator to the menu.

↪ **An `li` (page 114) element**

Iterate over the children of the `li` **(page 114)** element.

↪ **A `menu` (page 245) element with no `label` (page 246) attribute**

↪ **A `select` element**

Append a separator to the menu, then iterate over the children of the `menu` **(page 245)** or `select` element, then append another separator.

↪ **A `menu` (page 245) element with a `label` (page 246) 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 **(page 21)** 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 **(page 247)**. The value must be the ID of a `menu` **(page 245)** 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` **(page 245)** 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 **(page 273)** 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` **(page 247)** 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 **(page 273)** on the relevant `menu` **(page 245)** element.

The default action of *this* event is that the user agent must show a context menu built **(page 246)** from the `menu` **(page 245)** 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 **(page 249)**.

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 **(page 29)** the `contextmenu` **(page 247)** content attribute.

#### *3.18.4.4. Toolbars*

Toolbars are a kind of menu that is always visible.

When a `menu` **(page 245)** element has a `type` **(page 245)** attribute with the value `toolbar`, then the user agent must build **(page 246)** the menu for that `menu` **(page 245)** element and render it in the document in a position appropriate for that `menu` **(page 245)** element.

The user agent must reflect changes made to the `menu` **(page 245)**'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 **(page 249)** 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 **(page 249)**.

**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 **(page 249)** is true (hidden) then the Disabled State **(page 249)** will be true (disabled) regardless.

We could make this into a string value that acts as a Hint for why the command is disabled.

**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` **(page 250)** interface:

```
interface Command {
  readonly attribute DOMString commandType (page 250);
  readonly attribute DOMString id (page 250);
  readonly attribute DOMString label (page 250);
  readonly attribute DOMString title (page 250);
  readonly attribute DOMString icon (page 250);
  readonly attribute boolean hidden (page 250);
  readonly attribute boolean disabled (page 251);
  readonly attribute boolean checked (page 251);
  void click (page 251)();
  readonly attribute HTMLCollection (page 31) triggers (page 251);
  readonly attribute Command (page 242) command;
};
```

The `Command` **(page 250)** 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` **(page 250)** 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` **(page 250)** interface.

The **`commandType`** attribute must return a string whose value is either "`command`", "`radio`", or "`checked`", depending on whether the Type **(page 249)** 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 **(page 249)**, or null if the element does not define a command or defines an anonymous command **(page 249)**. This attribute will be shadowed by the `id` **(page 72)** DOM attribute on the `HTMLElement` **(page 27)** interface.

The **`label`** attribute must return the command's Label **(page 249)**, or null if the element does not define a command or does not specify a Label **(page 249)**. This attribute will be shadowed by the `label` DOM attribute on `option` and `command` **(page 242)** elements.

The **`title`** attribute must return the command's Hint **(page 249)**, or null if the element does not define a command or does not specify a Hint **(page 249)**. This attribute will be shadowed by the `title` **(page 72)** DOM attribute on the `HTMLElement` **(page 27)** interface.

The **`icon`** attribute must return an absolute URI to the command's Icon **(page 249)**. 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` **(page 244)** DOM attribute on `command` **(page 242)** elements.

The **`hidden`** attribute must return true if the command's Hidden State **(page 249)** 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` **(page 244)** DOM attribute on `command` **(page 242)** elements.

The **`disabled`** attribute must return true if the command's Disabled State **(page 249)** 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 **(page 249)**. 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` **(page 242)** elements.

The **`checked`** attribute must return true if the command's Checked State **(page 249)** 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` **(page 242)** elements.

The **`click()`** method must trigger the Action **(page 249)** for the command. If the element does not define a command, this method must do nothing. This method will be shadowed by the `click()` **(page 78)** 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 **(page 249)**). The list must be live **(page 22)**. While the element does not define a command, the list must be empty.

The **`commands`** attribute of the document's `HTMLDocument` **(page 25)** interface must return an `HTMLCollection` **(page 31)** rooted at the `Document` node, whose filter matches only elements that define commands and have IDs.

The following elements can define commands: `a` **(page 251)**, `button` **(page 252)**, `input` **(page 252)**, `option` **(page 253)**, `command` **(page 254)**.

#### *3.18.5.1. Using the `a` element to define a command*

An `a` **(page 118)** element with an `href` **(page 273)** attribute defines a command **(page 249)**.

The Type **(page 249)** of the command is "command".

The ID **(page 249)** of the command is the value of the `id` **(page 71)** attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command **(page 249)**.

The Label **(page 249)** of the command is the string given by the element's `textContent` **(page 19)** DOM attribute.

The Hint **(page 249)** of the command is the value of the `title` **(page 72)** attribute of the `a` **(page 118)** element. If the attribute is not present, the Hint **(page 249)** is the empty string.

The Icon **(page 249)** 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` **(page 148)** 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 **(page 249)** and Disabled State **(page 249)** facets of the command are always false. (The command is always enabled.)

The Checked State **(page 249)** of the command is always false. (The command is never checked.)

The Action **(page 249)** of the command is to fire a `click` event **(page 273)** at the element.

### *3.18.5.2. Using the `button` element to define a command*

A `button` element always defines a command **(page 249)**.

The Type **(page 249)**, ID **(page 249)**, Label **(page 249)**, Hint **(page 249)**, Icon **(page 249)**, Hidden State **(page 249)**, Checked State **(page 249)**, and Action **(page 249)** facets of the command are determined as for `a` elements **(page 251)** (see the previous section).

The Disabled State **(page 249)** 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 **(page 249)**.

The Type **(page 249)** 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 **(page 249)** of the command is the value of the `id` **(page 71)** attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command **(page 249)**.

The Label **(page 249)** of the command depends on the Type of the command:

If the Type **(page 249)** 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 **(page 249)** is "radio" or "checkbox". If the element has a `label` element associated with it, the `textContent` **(page 19)** of the first such element is the Label **(page 249)** (in DOM terms, this the string given by `*element*.labels[0].textContent`). Otherwise, the value of the `value` attribute, if present, is the Label **(page 249)**. Otherwise, the Label **(page 249)** is the empty string.

The Hint **(page 249)** of the command is the value of the `title` **(page 72)** attribute of the `input` element. If the attribute is not present, the Hint **(page 249)** is the empty string.

There is no Icon **(page 249)** for the command.

The Hidden State **(page 249)** of the command is always false. (The command is never hidden.)

The Disabled State **(page 249)** 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 **(page 249)** of the command is true if the command is of Type **(page 249)** "radio" or "checkbox" and the element has a `checked` attribute, and false otherwise.

The Action **(page 249)** of the command is to fire a `click` event **(page 273)** 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 **(page 249)**.

The Type **(page 249)** of the command is "radio" if the `option`'s nearest ancestor `select` element has no `multiple` attribute, and "checkbox" if it does.

The ID **(page 249)** of the command is the value of the `id` **(page 71)** attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command **(page 249)**.

The Label **(page 249)** 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` **(page 19)** DOM attribute if it doesn't.

The Hint **(page 249)** of the command is the string given by the element's `title` **(page 72)** attribute, if any, and the empty string if the attribute is absent.

There is no Icon **(page 249)** for the command.

The Hidden State **(page 249)** of the command is always false. (The command is never hidden.)

The Disabled State **(page 249)** of the command is true (disabled) if the element has a `disabled` attribute, and false otherwise.

The Checked State **(page 249)** of the command is true (checked) if the element's `selected` DOM attribute is true, and false otherwise.

The Action **(page 249)** of the command depends on its Type **(page 249)**. If the command is of Type **(page 249)** "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 **(page 273)** 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` **(page 242)** element defines a command **(page 249)**.

The Type **(page 249)** of the command is "radio" if the `command` **(page 242)**'s `type` **(page 243)** attribute is "`radio`", "checkbox" if the attribute's value is "`checkbox`", and "command" otherwise.

The ID **(page 249)** of the command is the value of the `id` **(page 71)** attribute of the element, if the attribute is present and not empty. Otherwise the command is an anonymous command **(page 249)**.

The Label **(page 249)** of the command is the value of the element's `label` **(page 243)** attribute, if there is one, or the empty string if it doesn't.

The Hint **(page 249)** of the command is the string given by the element's `title` **(page 243)** attribute, if any, and the empty string if the attribute is absent.

The Icon **(page 249)** for the command is the absolute URI resulting from resolving the value of the element's `icon` **(page 243)** attribute as a URI relative to the element's base URI. If the element has no `icon` **(page 243)** attribute then the command has no Icon **(page 249)**.

The Hidden State **(page 249)** of the command is true (hidden) if the element has a `hidden` **(page 243)** attribute, and false otherwise.

The Disabled State **(page 249)** of the command is true (disabled) if the element has either a `disabled` **(page 243)** attribute or a `hidden` **(page 243)** attribute (or both), and false otherwise.

The Checked State **(page 249)** of the command is true (checked) if the element has a `checked` **(page 244)** attribute, and false otherwise.

The Action **(page 249)** of the command is to invoke the behaviour described in the definition of the `click()` **(page 244)** method of the `HTMLCommandElement` **(page 245)** interface.

## 3.19. Miscellaneous elements

### 3.19.1. The `legend` element

**Contexts in which this element may be used:**

As the first child of a `fieldset` element.
As the first child of a `details` **(page 218)** element.
As a child of a `figure` **(page 146)** element, if there are no other `legend` **(page 254)** element children of that element.

**Content model:**

If used as a child of a `fieldset` or `details` **(page 218)** element: significant **(page 68)** strictly inline-level content **(page 67)**
If used as a child of a `figure` **(page 146)** element: inline-level content **(page 67)**.

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

None.

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `legend` **(page 254)** element represents a title or explanatory caption for the rest of the contents of the `legend` **(page 254)** element's parent element.

### 3.19.2. The `div` element

Block-level element **(page 67)**.

**Contexts in which this element may be used:**

Where block-level elements **(page 67)** are expected.

**Content model:**

Zero or more `style` **(page 91)** elements, followed by either zero or more block-level elements **(page 67)**, or inline-level content **(page 67)** (but not both).

**Element-specific attributes:**

None.

**Predefined classes that apply to this element:**

`error` **(page 75)**, `example` **(page 75)**, `issue` **(page 75)**, `note` **(page 75)**, `search` **(page 76)**, `warning` **(page 76)**

**DOM interface:**

No difference from `HTMLElement` **(page 27)**.

The `div` **(page 255)** element represents nothing at all. It can be used with the `class` **(page 73)**, `lang` **(page 72)**/`xml:lang` **(page 72)**, and `title` **(page 72)** attributes to mark up semantics common to a group of consecutive elements.

# 4. Browsing contexts

This part of the spec is under heavy development right now. It needs massive editorial work. The actual normative rules won't change (at least not as part of the reorg), they'll just be made clearer.

## 4.1. Navigating across documents

*This section only applies to Web browsers.*

Certain actions cause the browsing context **(page 19)** to **navigate**. For example, following a hyperlink **(page 274)**, form submission, and the `window.open()` and `location.assign()` **(page 293)** 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 follow the following steps:

1. Cancel any existing attempt to navigate the browsing context to a new URI.
2. If the new resource is the same as the current resource, but a fragment identifier has been specified, then scroll the document to the specified element **(page 265)** 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 **(page 294)** for the given scheme, then display the inline content **(page 265)** 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. Start fetching the specified resource, as appropriate (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 **(page 267)**). If this results in a redirect, return to step 2 with the new resource.
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 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***

8. Let *type* be the sniffed type of the resource **(page 259)**.

9. If the user agent has been configured to process resources of the given *type* using some mechanism other than native rendering, 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 **(page 263)** section, and abort these steps.

   ↪ **Any type ending in "+xml"**

   ↪ **"application/xml"**

   ↪ **"text/xml"**

   Follow the steps given in the XML document **(page 264)** 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 **(page 264)** section, and abort these steps.

10. 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, a full-page plugin, an error message because the specified type is not supported, or an inline prompt to allow the user to select a registered handler **(page 294)** for the given type, then display the inline content **(page 265)** and abort these steps.

11. 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 these steps. From the point of view of any script, these steps must occur atomically.

1. If appropriate, update the current entry **(page 289)** in the browsing context **(page 19)**'s `Document` object's `History` **(page 289)** 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.

2. Remove all the entries after the current entry **(page 289)** in the browsing context **(page 19)**'s `Document` object's `History` **(page 289)** object.

***Note: This doesn't necessarily have to affect* (page 293) *the user agent's user interface.***

3. Append a new entry at the end of the `History` **(page 289)** object representing the new resource and its `Document` object and related state.
4. Traverse the history **(page 290)** to the new entry.
5. If the navigation was initiated with **replacement enabled**, remove the entry that was the current entry **(page 289)** before the method call (thus simply causing the previous page to be replaced by the new one). Navigation is initiated with replacement disabled unless it is stated otherwise.

### 4.1.1. 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* **(page 260)** 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* **(page 260)** step below.
3. 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.

4. 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* **(page 261)** section below.
5. If *official type* is "text/html", then jump to the *feed or HTML* **(page 262)** section below.
6. Otherwise, the sniffed type of the resource is *official type*.

#### *4.1.1.1. 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".
4. 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".
5. Otherwise, the sniffed type of the resource is "text/plain".

#### *4.1.1.2. 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:
   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.

4. As a last-ditch effort, jump to the text or binary **(page 260)** section.

| Bytes in Hexadecimal | | Sniffed type | Comment |
|---|---|---|---|
| Mask | Pattern | | |
| FF FF DF DF DF DF DF DF DF FF DF DF DF DF | 3C 31 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 DF DF DF DF | 3C 48 54 4D 4C | text/html | The string "`<HTML`" in US-ASCII or compatible encodings, case-insensitively. |
| 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. |
| FF FF FF | FF D8 FF | image/jpeg | A JPEG SOI marker followed by the first byte of another marker. |

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).

#### *4.1.1.3. 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. |

| Bytes in Hexadecimal | Sniffed type | Comment |
|---|---|---|
| FF D8 FF | image/jpeg | A JPEG SOI marker followed by the first byte of another marker. |

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*.

*4.1.1.4. 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 6) 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 6 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 6 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.***

### 4.1.2. [TBW] Page load processing model for HTML files

When an HTML document is to be loaded in a browsing context **(page 19)**, the user agent must create a `Document` object, mark it as being an HTML document, create an HTML parser **(page 373)**, associate it with the document, and begin to use the bytes provided for the document as the input stream **(page 375)** for that parser.

***Note: The input stream* (page 375) *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 **(page 258)**.

### 4.1.3. [TBW] 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.

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 **(page 257)** steps above.

Otherwise, then, with the newly created `Document`, the user agents must update the session history with the new page **(page 258)**. 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`.

### 4.1.4. [TBW] Page load processing model for text files

When a plain text document is to be loaded in a browsing context **(page 19)**, the user agent should create a `Document` object, mark it as being an HTML document, create an HTML parser **(page 373)**, associate it with the document, act as if the tokeniser had emitted a start tag token with the tag name "pre", set the tokenisation **(page 382)** stage's content model flag **(page 382)** 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]

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 **(page 258)**.

### 4.1.5. Scrolling to a fragment identifier

When a user agent is supposed to scroll to a particular element, it may change the scrolling position of the document as desired, or perform any other relevant action.

how to get a "particular element" from a frag id -- id="", name="", XPointer, etc; missing IDs (e.g. the infamous "#top")

Then, the user agent must update the session history with the new page **(page 258)**, where "the new page" has the same `Document` as before, but potentially has a different scroll position.

### 4.1.6. Displaying inline content that doesn't have a DOM

**display a user agent page inline**, update history object

### 4.1.7. [TBW] Content-Type metadata

This is a placeholder section for some text that will define exactly how to handle Content-Type headers for top-level browsing contexts, for <img>, <embed>, <object>, etc; and will cover things like the fact that on some operating systems the extension of a URI determines the Content-Type for file:// content, or something.

## 4.2. [WIP] Scripting

Here are some of the things I think we should list:

- onload/onunload events
- alert() blocks scripts, even those on timeouts or readystatechange events

### 4.2.1. Running executable code

Various mechanisms can cause author-provided executable code to run in the context of a document. These mechanisms include, but are probably not limited to:

- Processing of `script` **(page 210)** elements.
- Processing of inline `javascript:` URIs (e.g. the `src` **(page 148)** attribute of `img` **(page 148)** elements, or an `@import` rule in a CSS `style` **(page 91)** element block).

- Event handlers, whether registered through the DOM using `addEventListener()`, by explicit event handler content attributes **(page 269)**, by event handler DOM attributes **(page 270)**, or otherwise.
- Processing of technologies like XBL or SVG that have their own scripting features.

User agents may provide a mechanism to enable or disable the execution of author-provided code. When the user agent is configured such that author-provided code does not execute, or if the user agent is implemented so as to never execute author-provided code, it is said that **scripting is disabled**. When author-provided code *does* execute, **scripting is enabled**. A user agent with scripting disabled is a user agent with no scripting support **(page 16)** for the purposes of conformance.

### 4.2.2. Scripting contexts

Each `Document` in a browsing context **(page 19)** has an associated **scripting context**.

clean up this document regarding where exactly scripts run -- some parts say global scope, some say scripting context, some refer to browsing contexts...

### 4.2.3. Threads

Each scripting context **(page 266)** is defined as having a list of zero or more **reachable scripting contexts**. These are:

- All the scripting contexts **(page 266)** of all the `Document`s in each nested browsing context **(page 19)** inside the scripting context's `Document`.
- The scripting context **(page 266)** of the `Document` that contains the `Document`'s browsing context **(page 19)**.
- All the scripting contexts **(page 266)** of all the `Document`s associated with any top-level browsing contexts **(page 19)** that were opened from a `Document` associated with the scripting context **(page 266)**'s `Document`'s browsing context **(page 19)**.
- All the scripting contexts **(page 266)** of all the `Document`s associated with the top-level browsing contexts **(page 19)** of the browsing context **(page 19)** of the `WindowHTML` **(page 287)** object returned by the scripting context **(page 266)**'s `Document`'s `WindowHTML` **(page 287)**'s `opener` **(page 313)** attribute.

The transitive closure of all the scripting contexts **(page 266)** that are reachable scripting contexts **(page 266)** consists of a **unit of related scripting contexts**.

All the executable code in a unit of related scripting contexts **(page 266)** 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 scripting contexts **(page 266)** there is only one script being executed at a time.

### 4.2.4. [WIP] The security model

*Web browser vendors should implement this security model, to provide Web authors with a consistent development environment that is interoperable across different implementations. However, implementors may use any other model if desired.*

The security model for Web browsers has grown organically with the development of the Web, and as such is somewhat unique.

Access to resources and APIs is granted or denied to Web content (scripts, elements, etc) based on the content's **origin**. For historical reasons, the mechanism for determining the *origin* **(page 267)** of a particular piece of content depends on the nature of the content.

...

The **domain of a `Document` object** is the domain given by the `hostname` attribute of the `Location` **(page 292)** object returned by the `Document` object's `location` **(page 292)** attribute, *if* that `hostname` attribute is not the empty string. If it is, the domain of the document is UA-defined. For now. Need to be more correct about where .location is defined. It's not actually on Document.

The **domain of a script** is the domain of the `Document` object **(page 267)** that is returned by the `document` attribute of the script's primary `Window` object (in UAs that implement ECMAScript, that is the global scope **(page 287)** object).

**The string representing the script's domain in IDNA format** is obtained as follows: take the script's domain **(page 267)** and apply the IDNA ToASCII algorithm and then the IDNA ToUnicode algorithm to each component of the domain name (with both the AllowUnassigned and UseSTD3ASCIIRules flags set both times). [RFC3490] If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, then the string representing the script's domain in IDNA format cannot be obtained. (ToUnicode is defined to never fail.)

This section is *so* not complete.

Define **security exception**.

### 4.2.5. 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 navigated **(page 257)** to a `javascript:` URI, the dereference context must be the global scope **(page 287)** of the browsing context **(page 19)** being navigated. Otherwise, the dereference context must be global scope

**(page 287)** of the browsing context **(page 19)** to which belongs the element for which the URI is being dereferenced.

URIs using the `javascript:` protocol should be evaluated when the resource for that URI is needed, unless scripting is disabled **(page 266)** or the `Document` corresponding to the dereference context (as defined above), if any, has `designMode` **(page 318)** 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 Content-Type metadata is `text/html` and whose response body is the dereference by-product, converted to a string value.

***Note: Certain contexts, in particular `img`* (page 148) *elements, ignore the Content-Type metadata.***

So for example a `javascript:` URI for a `src` **(page 148)** attribute of an `img` **(page 148)** 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 `a` **(page 118)** element would only be evaluated when the link was followed **(page 274)**.

The `src` **(page 150)** attribute of an `iframe` **(page 150)** element would be evaluated in the context of the `iframe` **(page 150)**'s own browsing context **(page 19)**; once evaluated, its return value (if it was not void) would replace that browsing context **(page 19)**'s document, thus changing the variables visible in the global scope **(page 287)** of that browsing context **(page 19)**.

### 4.2.6. [SCS] Runtime script errors

Whenever a runtime script error occurs in one of the scripts associated with the document, the value of the **`onerror`** attribute of the `window` object (defined on the `WindowHTML` **(page 287)** interface of that object), must be processed, as follows:

↪ **If the value is a function**

The function referenced by the `onerror` **(page 268)** 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` **(page 268)** must be `undefined`.

## 4.2.7. Events

### *4.2.7.1. Event handler attributes*

HTML elements **(page 20)** can have **event handler attributes** specified. These act as bubbling event listeners for the element on which they are specified.

Each event handler attribute has two parts, an event handler content attribute **(page 269)** and an event handler DOM attribute **(page 270)**. Event handler attributes must initially be set to null. When their value changes (through the changing of their event handler content attribute or their event handler DOM attribute), they will either be null, or have an `EventListener` object assigned to them.

In the ECMAScript DOM binding, the ECMAScript native `Function` type must implement the `EventListener` interface such that invoking the `handleEvent()` method of that interface on the object from another language binding invokes the function itself, with the `event` argument as its only argument. In the ECMAScript binding itself, however, the `handleEvent()` method of the interface is not directly accessible on `Function` objects. Such functions, when invoked, must be called in the global scope **(page 287)**. If the function returns false, the event's `preventDefault()` method must then invoked. Exception: for historical reasons, for the HTML `mouseover` event, the `preventDefault()` method must be called when the function returns true instead.

**Event handler content attributes**, when specified, must contain valid ECMAScript code matching the ECMAScript `FunctionBody` production. [ECMA262]

When an event handler content attribute is set, its new value must be interpreted as the body of an anonymous function with a single argument called `event`, with the new function's scope chain being linked from the activation object of the handler, to the element, to the element's `form` element if it is a form control, to the `Document` object, to the global scope **(page 287)**. 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]***

How do we allow non-JS

**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 **(page 20)**, as both content attributes and DOM attributes:

**`onabort`**
: Must be invoked whenever an `abort` event is targeted at or bubbles through the element.

**`onbeforeunload`**
: Must be invoked whenever a `beforeunload` event is targeted at or bubbles through the element.

**`onblur`**
: Must be invoked whenever a `blur` event is targeted at or bubbles through the element.

**`onchange`**
: Must be invoked whenever a `change` event is targeted at or bubbles through the element.

**`onclick`**
: Must be invoked whenever a `click` event is targeted at or bubbles through the element.

**`oncontextmenu`**
: Must be invoked whenever a `contextmenu` event is targeted at or bubbles through the element.

**`ondblclick`**
: Must be invoked whenever a `dblclick` event is targeted at or bubbles through the element.

**`ondrag`**
: Must be invoked whenever a `drag` **(page 321)** event is targeted at or bubbles through the element.

**`ondragend`**
: Must be invoked whenever a `dragend` **(page 321)** event is targeted at or bubbles through the element.

**`ondragenter`**
: Must be invoked whenever a `dragenter` **(page 321)** event is targeted at or bubbles through the element.

**`ondragleave`**
: Must be invoked whenever a `dragleave` **(page 321)** event is targeted at or bubbles through the element.

**ondragover**

Must be invoked whenever a `dragover` **(page 321)** event is targeted at or bubbles through the element.

**ondragstart**

Must be invoked whenever a `dragstart` **(page 321)** event is targeted at or bubbles through the element.

**ondrop**

Must be invoked whenever a `drop` **(page 321)** event is targeted at or bubbles through the element.

**onerror**

Must be invoked whenever an `error` event is targeted at or bubbles through the element.

**onfocus**

Must be invoked whenever a `focus` event is targeted at or bubbles through the element.

**onkeydown**

Must be invoked whenever a `keydown` event is targeted at or bubbles through the element.

**onkeypress**

Must be invoked whenever a `keypress` event is targeted at or bubbles through the element.

**onkeyup**

Must be invoked whenever a `keyup` event is targeted at or bubbles through the element.

**onload**

Must be invoked whenever a `load` event is targeted at or bubbles through the element.

**onmessage**

Must be invoked whenever a `message` **(page 341)** 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` **(page 241)** 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 scripting is disabled **(page 266)**, event handler attributes must do nothing.

When scripting is enabled **(page 266)**, 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.

#### *4.2.7.2. Event firing*

maybe _this_ should be moved higher up (terminology? conformance? DOM?)

Certain operations and methods are defined as firing events on elements. For example, the `click()` **(page 78)** method on the `HTMLElement` **(page 27)** 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` **(page 273)**. **Firing a `load` event** means firing a simple event called `load` **(page 273)**. **Firing an `error` event** means firing a simple event called `error` **(page 273)**.

The default action of these event is to do nothing unless otherwise stated.

> If you dispatch a custom "click" event at an element that would normally have default actions, should they get triggered? If so, we need to go through the entire spec and make sure that any default actions are defined in terms of *any* event of the right type on that element, not those that are dispatched in expected ways.

## 4.3. Links

### 4.3.1. Hyperlink elements

The `a` **(page 118)**, `area` **(page 186)**, and `link` **(page 82)** 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` (page 273) attribute on `a` (page 118) and `area` (page 186) elements is not required; when those elements do not have `href` (page 273) attributes they do not represent hyperlinks.***

***The `href` (page 83) attribute on the `link` (page 82) element is required, but whether a `link` (page 82) element represents a hyperlink or not depends on the value of the `rel` (page 83) attribute of that element.***

The **`target`** attribute, if present, must be a valid browsing context name **(page 313)**. User agents use this name when following hyperlinks **(page 274)**.

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 **(page 274)**.

For `a` **(page 118)** and `area` **(page 186)** 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 an unordered set of space-separated tokens **(page 62)**. The allowed values and their meanings **(page 276)** are defined below. The `rel` **(page 274)** 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` **(page 274)** attribute is omitted or has an invalid value, is `all`.

The **`hreflang`** attribute on hyperlink elements, if present, gives the language of the linked resource. It is purely advisory. The value must be a valid RFC 3066 language code. [RFC3066] User agents must not consider this attribute authoritative — upon fetching the resource, user agents must only use language information associated with the resource to determine its language, not metadata included in the link to the resource.

The **`type`** attribute, if present, gives the MIME type of the linked resource. It is purely advisory. The value must be a valid MIME type, optionally with parameters. [RFC2046] User agents must not consider the `type` **(page 274)** attribute authoritative — upon fetching the resource, user agents must only use the Content-Type information associated with the resource **(page 265)** to determine its type, not metadata included in the link to the resource.

### 4.3.2. Following hyperlinks

When a user *follows a hyperlink*, the user agent must navigate **(page 257)** a browsing context **(page 19)** to the URI specified of the hyperlink.

The URI of the hyperlink is URI given by resolving the the `href` **(page 273)** 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* **(page 120)** 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 is the browsing context that must be navigated.

Otherwise, if the hyperlink element is an `a` **(page 118)** or `area` **(page 186)** element that has a `target` **(page 274)** 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 **(page 313)**, using the value of the `target` **(page 274)** attribute as the browsing context name.

Otherwise, if the hyperlink element is a sidebar hyperlink **(page 284)** 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` **(page 118)** or `area` **(page 186)** element with no `target` **(page 274)** attribute, but there is a `base` **(page 81)** element with a `target` **(page 82)** attribute in the document, then the browsing context that is navigated must be chosen by applying the rules for chosing a browsing context given a browsing context name **(page 313)**, using the value of the `target` **(page 82)** attribute of the first such `base` **(page 81)** element as the browsing context name.

Otherwise, the browsing context that must be navigated is the same browsing context as the one which the hyperlink element itself is in.

#### *4.3.2.1. Hyperlink auditing*

If an `a` **(page 118)** or `area` **(page 186)** hyperlink element has a `ping` **(page 274)** attribute and the user follows the hyperlink, the user agent must take the `ping` **(page 274)** attribute's value, strip leading and trailing spaces **(page 47)**, split the value on sequences of spaces, treat each resulting part as a URI (resolving relative URIs according to element's base URI) and then should send a request to each of the resulting URIs. This may be done in parallel with the primary request, and is independent of the result of that request.

User agents should allow the user to adjust this behaviour, for example in conjunction with a setting that disables the sending of HTTP Referrer headers. Based on the user's preferences, UAs may either ignore **(page 21)** the `ping` **(page 274)** 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` **(page 274)** 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`*** **(page 274)** ***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`*** **(page 274)** ***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`*** **(page 274)** ***attribute so that the user agent can improve the user experience.***

### 4.3.3. Link types

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

In this section, the term *referenced document* refers to the resource identified by the element representing the link, and the term *current document* refers to the resource within which the element representing the link finds itself.

To determine which link types apply to a `link` **(page 82)**, `a` **(page 118)**, or `area` **(page 186)** element, the element's `rel` attribute must be split on spaces **(page 62)**. The resulting tokens are the link types that apply to that element.

| **Link type** | **Effect on...** | | **Brief description** |
|---|---|---|---|
| | **`link` (page 82)** | **`a` (page 118) and `area` (page 186)** | |
| `alternate` **(page 278)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Gives alternate representations of the current document. |
| `archives` **(page 279)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Provides a link to a collection of records, documents, or other materials of historical interest. |

| Link type | Effect on... | | Brief description |
|---|---|---|---|
| | `link` **(page 82)** | `a` **(page 118) and** `area` **(page 186)** | |
| `author` **(page 279)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Gives a link to the current document's author. |
| `bookmark` **(page 280)** | *not allowed* | Hyperlink **(page 273)** | Gives the permalink for the nearest ancestor section. |
| `contact` **(page 280)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Gives a link to contact information for the current document. |
| `external` **(page 281)** | *not allowed* | Hyperlink **(page 273)** | Indicates that the referenced document is not part of the same site as the current document. |
| `feed` **(page 281)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Gives the address of a syndication feed for the current document. |
| `first` **(page 284)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Indicates that the current document is a part of a series, and that the first document in the series is the referenced document. |
| `help` **(page 282)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Provides a link to context-sensitive help. |
| `icon` **(page 282)** | External Resource **(page 83)** | *not allowed* | Imports an icon to represent the current document. |
| `index` **(page 284)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Gives a link to the document that provides a table of contents or index listing the current document. |
| `last` **(page 284)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Indicates that the current document is a part of a series, and that the last document in the series is the referenced document. |
| `license` **(page 282)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Indicates that the current document is covered by the copyright license described by the referenced document. |
| `next` **(page 285)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Indicates that the current document is a part of a series, and that the next document in the series is the referenced document. |
| `nofollow` **(page 282)** | *not allowed* | Hyperlink **(page 273)** | Indicates that the current document's original author or publisher does not endorse the referenced document. |
| `pingback` **(page 283)** | External Resource **(page 83)** | *not allowed* | Gives the address of the pingback server that handles pingbacks to the current document. |

| Link type | Effect on... | | Brief description |
|---|---|---|---|
| | `link` **(page 82)** | `a` **(page 118) and** `area` **(page 186)** | |
| `prefetch` **(page 283)** | External Resource **(page 83)** | *not allowed* | Specifies that the target resource should be pre-emptively cached. |
| `prev` **(page 285)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Indicates that the current document is a part of a series, and that the previous document in the series is the referenced document. |
| `search` **(page 283)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Gives a link to a resource that can be used to search through the current document and its related pages. |
| `stylesheet` **(page 283)** | External Resource **(page 83)** | *not allowed* | Imports a stylesheet. |
| `sidebar` **(page 283)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Specifies that the referenced document, if retrieved, is intended to be shown in the browser's sidebar (if it has one). |
| `tag` **(page 284)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Gives a tag (identified by the given address) that applies to the current document. |
| `up` **(page 285)** | Hyperlink **(page 83)** | Hyperlink **(page 273)** | Provides a link to a document giving the context for the current document. |

Some of the types described below list synonyms for these values. These are to be handled as specified by user agents, but must not be used in documents.

#### *4.3.3.1. Link type "`alternate`"*

The `alternate` **(page 278)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, if the `rel` **(page 83)** attribute does not also contain the keyword `stylesheet` **(page 283)**, it creates a hyperlink **(page 83)**; but if it *does* also contains the keyword `stylesheet` **(page 283)**, the `alternate` **(page 278)** keyword instead modifies the meaning of the `stylesheet` **(page 283)** keyword in the way described for that keyword, and the rest of this subsection doesn't apply.

The `alternate` **(page 278)** 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` **(page 274)**, `hreflang` **(page 274)**, and `type` **(page 274)** attributes.

If the `alternate` **(page 278)** keyword is used with the `media` **(page 274)** attribute, it indicates that the referenced document is intended for use with the media specified.

If the `alternate` **(page 278)** keyword is used with the `hreflang` **(page 274)** attribute, and that attribute's value differs from the root element **(page 21)**'s language **(page 72)**, it indicates that the referenced document is a translation.

If the `alternate` **(page 278)** keyword is used with the `type` **(page 274)** attribute, it indicates that the referenced document is a reformulation of the current document in the specified format.

The `media` **(page 274)**, `hreflang` **(page 274)**, and `type` **(page 274)** attributes can be combined when specified with the `alternate` **(page 278)** 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` **(page 278)** keyword is used with the `type` **(page 274)** 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` **(page 281)** keyword specified as well.

The `alternate` **(page 278)** link relationship is transitive — that is, if a document links to two other documents with the link type "`alternate` **(page 278)**", then, in addition to implying that those documents are alternative representations of the first document, it is also implying that those two documents are alternative representations of each other.

### *4.3.3.2. Link type "`archives`"*

The `archives` **(page 279)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `archives` **(page 279)** 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` **(page 279)** keyword.

### *4.3.3.3. Link type "`author`"*

The `author` **(page 279)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

For `a` **(page 118)** and `area` **(page 186)** elements, the `author` **(page 279)** keyword indicates that the referenced document provides further information about the author of the section that the element defining the hyperlink applies **(page 94)** to.

For `link` **(page 82)** elements, the `author` **(page 279)** 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` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements that have a `rev` attribute with the value "`made`" as having the `author` **(page 279)** keyword specified as a link relationship.

### *4.3.3.4. Link type "`bookmark`"*

The `bookmark` **(page 280)** keyword may be used with `a` **(page 118)** and `area` **(page 186)** elements.

The `bookmark` **(page 280)** keyword gives a permalink for the nearest ancestor `article` **(page 96)** element of the linking element in question, or of the section the linking element is most closely associated with **(page 105)**, if there are no ancestor `article` **(page 96)** 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>
 ...
```

### *4.3.3.5. Link type "`contact`"*

The `contact` **(page 280)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

For `a` **(page 118)** and `area` **(page 186)** elements, the `contact` **(page 280)** keyword indicates that the referenced document provides further contact information for the section that the element defining the hyperlink applies **(page 94)** to.

User agents must treat any hyperlink in an `address` **(page 101)** element as having the `contact` **(page 280)** link type specified.

For `link` **(page 82)** elements, the `contact` **(page 280)** keyword indicates that the referenced document provides further contact information for the page as a whole.

### *4.3.3.6. Link type "`external`"*

The `external` **(page 281)** keyword may be used with `a` **(page 118)** and `area` **(page 186)** elements.

The `external` **(page 281)** keyword indicates that the link is leading to a document that is not part of the site that the current document forms a part of.

### *4.3.3.7. Link type "`feed`"*

The `feed` **(page 281)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `feed` **(page 281)** keyword indicates that the referenced document is a syndication feed. If the `alternate` **(page 278)** 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` **(page 82)**, `a` **(page 118)**, or `area` **(page 186)** element in the document (in tree order) that creates a hyperlink with the link type `feed` **(page 281)** must be treated as the default syndication feed for the purposes of feed autodiscovery.

> ***Note: The `feed`* (page 281) *keyword is implied by the `alternate`* (page 278) *link type in certain cases (q.v.).***

The following two `link` **(page 82)** 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>
```

#### *4.3.3.8. Link type "`help`"*

The `help` **(page 282)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

For `a` **(page 118)** and `area` **(page 186)** elements, the `help` **(page 282)** 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` **(page 82)** elements, the `help` **(page 282)** keyword indicates that the referenced document provides help for the page as a whole.

#### *4.3.3.9. Link type "`icon`"*

The `icon` **(page 282)** keyword may be used with `link` **(page 82)** elements, for which it creates an external resource link **(page 83)**.

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` **(page 84)** attribute.

#### *4.3.3.10. Link type "`license`"*

The `license` **(page 282)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `license` **(page 282)** 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` **(page 282)** keyword.

#### *4.3.3.11. Link type "`nofollow`"*

The `nofollow` **(page 282)** keyword may be used with `a` **(page 118)** and `area` **(page 186)** elements.

The `nofollow` **(page 282)** keyword indicates that the link is not endorsed by the original author or publisher of the page.

#### *4.3.3.12. Link type "`pingback`"*

The `pingback` **(page 283)** keyword may be used with `link` **(page 82)** elements, for which it creates an external resource link **(page 83)**.

For the semantics of the `pingback` **(page 283)** keyword, see the Pingback 1.0 specification. [PINGBACK]

#### *4.3.3.13. Link type "`prefetch`"*

The `prefetch` **(page 283)** keyword may be used with `link` **(page 82)** elements, for which it creates an external resource link **(page 83)**.

The `prefetch` **(page 283)** keyword indicates that preemptively fetching and caching the specified resource is likely to be beneficial, as it is highly likely that the user will require this resource.

#### *4.3.3.14. Link type "`search`"*

The `search` **(page 283)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `search` **(page 283)** 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`*** **(page 82)** ***elements and the `search`*** **(page 283)** ***link type to enable user agents to autodiscover search interfaces.***

#### *4.3.3.15. Link type "`stylesheet`"*

The `stylesheet` **(page 283)** keyword may be used with `link` **(page 82)** elements, for which it creates an external resource link **(page 83)** that contributes to the styling processing model **(page 93)**.

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` **(page 278)** keyword is also specified on the `link` **(page 82)** element, then the link is an alternative stylesheet.

#### *4.3.3.16. Link type "`sidebar`"*

The `sidebar` **(page 283)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `sidebar` **(page 283)** 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 **(page 273)** with with the `sidebar` **(page 283)** keyword specified is a **sidebar hyperlink**.

##### *4.3.3.17. Link type "`tag`"*

The `tag` **(page 284)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `tag` **(page 284)** keyword indicates that the *tag* that the referenced document represents applies to the current document.

##### *4.3.3.18. Hierarchical link types*

Some documents form part of a hierarchical structure of documents.

A hierarchical structure of documents is one where each document can have various subdocuments. A subdocument is said to be a *child* of the document it is a subdocument of. The document of which it is a subdocument is said to be its *parent*. The children of a document have a relative order; the subdocument that precedes another is its *previous sibling*, and the one that follows it is its *next sibling*. A document with no parent forms the top of the hierarchy.

###### 4.3.3.18.1. LINK TYPE "`FIRST`"

The `first` **(page 284)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `first` **(page 284)** keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the first child of the current document's parent document.

**Synonyms**: For historical reasons, user agents must also treat the keywords "`begin`" and "`start`" like the `first` **(page 284)** keyword.

###### 4.3.3.18.2. LINK TYPE "`INDEX`"

The `index` **(page 284)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `index` **(page 284)** keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the top of the hierarchy.

**Synonyms**: For historical reasons, user agents must also treat the keywords "`top`", "`contents`", and "`toc`" like the `index` **(page 284)** keyword.

###### 4.3.3.18.3. LINK TYPE "`LAST`"

The `last` **(page 284)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `last` **(page 284)** keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the last child of the current document's parent document.

**Synonyms**: For historical reasons, user agents must also treat the keyword "`end`" like the `last` **(page 284)** keyword.

##### 4.3.3.18.4. Link type "next"

The `next` **(page 285)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `next` **(page 285)** keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the next sibling of the current document.

##### 4.3.3.18.5. Link type "prev"

The `prev` **(page 285)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `prev` **(page 285)** keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the previous sibling of the current document.

**Synonyms**: For historical reasons, user agents must also treat the keyword "`previous`" like the `prev` **(page 285)** keyword.

##### 4.3.3.18.6. Link type "up"

The `up` **(page 285)** keyword may be used with `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** elements. For `link` **(page 82)** elements, it creates a hyperlink **(page 83)**.

The `up` **(page 285)** keyword indicates that the document is part of a hierarchical structure, and that the link is leading to the document that is the parent of the current document.

#### *4.3.3.19. Other link types*

Other than the types defined above, only types defined as extensions in the WHATWG Wiki RelExtensions page may be used with the `rel` attribute on `link` **(page 82)**, `a` **(page 118)**, and `area` **(page 186)** 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` (page 82)**

One of the following:

**not allowed**
The keyword is not allowed to be specified on `link` **(page 82)** elements.

**Hyperlink**
The keyword may be specified on a `link` **(page 82)** element; it creates a hyperlink link **(page 83)**.

**External Resource**
The keyword may be specified on a `link` **(page 82)** element; it creates a external resource link **(page 83)**.

**Effect on... `a` (page 118) and `area` (page 186)**

One of the following:

**not allowed**
The keyword is not allowed to be specified on `a` **(page 118)** and `area` **(page 186)** elements.

**Hyperlink**
The keyword may be specified on `a` **(page 118)** and `area` **(page 186)** elements; it creates a hyperlink **(page 273)**.

**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` **(page 82)**" and "Effect on... `a` **(page 118)** and `area` **(page 186)**" 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.

## 4.4. The global scope

Many of the APIs are part of the `WindowHTML` **(page 287)** interface. The `WindowHTML` **(page 287)** interface must be obtainable from the `Window` object using binding-specific casting methods. [WINDOW]

```
interface WindowHTML {

  // defined in this section
  readonly attribute History (page 289) history (page 288);
  readonly attribute ClientInformation (page 294) navigator (page 294);
  readonly attribute UndoManager (page 330) undoManager (page 331);
  Selection (page 337) getSelection (page 337)();
  readonly attribute Storage (page 300) sessionStorage (page 303);
  readonly attribute StorageList (page 304) globalStorage (page 304);

  // defined in other sections
           attribute Object onerror (page 268);

  // 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);

  // browsing contexts
  readonly attribute Window window (page 313);
  readonly attribute Window frames (page 313);
  readonly attribute Window self (page 313);
  readonly attribute unsigned long length (page 313);
  readonly attribute Window opener (page 313);
  Window open (page 312)();
  Window open (page 312)(in DOMString url);
  Window open (page 312)(in DOMString url, in DOMString target);
  Window open (page 312)(in DOMString url, in DOMString target, in DOMString
features);
  Window open (page 312)(in DOMString url, in DOMString target, in DOMString
features, in DOMString replace);
};
```

The `WindowHTML` **(page 287)** object must provide the following constructors:

**`Audio()`**

Constructs an `Audio` **(page 363)** object.

**`Image()`**

**`Image(in unsigned long` *`w`*`)`**

**`Image(in unsigned long` *`w`*`, in unsigned long` *`h`*`)`**

Constructs an `HTMLImageElement` **(page 148)** object (a new `img` **(page 148)** element). If the *h* argument is present, the new object's `height` **(page 149)** content attribute must be set to *h*. If the *w* argument is present, the new object's `width` **(page 149)** content attribute must be set to *w*.

**`Option()`**

**`Option(in DOMString` *`name`*`)`**

**`Option(in DOMString` *`name`*`, in DOMString` *`value`*`)`**

Constructs an `HTMLOptionElement` object (a new `option` element). need to define argument processing

***Note: `Window` objects also have an implicit [[Get]] method* (page 313).**

## 4.5. [SCS] Session history and navigation

### 4.5.1. The session history of browsing contexts

`History` **(page 289)** objects provide a representation of the pages in the session history of their `Window` object's browsing context **(page 19)**. Each browsing context (`frame`, `iframe` **(page 150)**, etc) has a distinct session history.

Each `Document` object in a browsing context's session history is associated with a unique instance of the `History` **(page 289)** object, although they all must model the same underlying session history.

The **`history`** attribute of the `WindowHTML` **(page 287)** interface must return the object implementing the `History` **(page 289)** interface for that `WindowHTML` **(page 287)** object's associated `Document` object.

`History` **(page 289)** objects represent their browsing context **(page 19)**'s session history as a flat list of URIs and state objects **(page 288)**. (This does not imply that the UI need be linear. See the notes below **(page 293)**.)

Typically, the history list will consist of only URIs. However, a page can add **(page 291) state objects** between its entry in the session history and the next ("forward") entry. These are then returned to the script **(page 292)** when the user (or script) goes back in the history, thus enabling authors to use the "navigation" metaphor even in one-page applications.

Entries that consist of state objects **(page 288)** share the same `Document` as the entry for the URI itself. Contiguous entries that differ just by fragment identifier must 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.***

At any point, one of the entries in the session history is the **current entry**. This is the entry representing the page in this browsing context **(page 19)** that is considered the "current" page by the UA. The current entry **(page 289)** is usually an entry for the location **(page 293)** of the `Document`. However, it can also be one of the entries for state objects **(page 288)** added to the history by that document.

When the browser's navigation model differs significantly from the sequential model represented by the `History` **(page 289)** interface, for example if separate `Document` objects in the session history are all simulatenously displayed and active, then the current entry **(page 289)** could even be an entry unrelated to the `History` **(page 289)** object's own `Document` object. If, when a method is invoked on a `History` **(page 289)** object, the current entry **(page 289)** for that browsing context **(page 19)**'s session history has a different `Document` object than the `History` **(page 289)** object's own `Document` object, then the user agent must raise a `NO_MODIFICATION_ALLOWED_ERR` DOM exception. (This can only happen if scripts are allowed to run in documents that are not the current document. Typically, however, user agents only allow scripts from the current entry **(page 289)** to execute.)

User agents may **discard** the DOMs of entries other than the current entry **(page 289)**, 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 a user agent discards the DOM from an entry in the session history, it must also discard all the entries from the first state object entry for that `Document` object up to and including the last entry for that `Document` object (including any non-state-object entries in that range, such as entries where the user navigated using fragment identifiers). These entries are not recreated if the user or script navigates back to the page. If there are no state object entries for that `Document` object then no entries are removed.

### 4.5.2. The `History` (page 289) interface

```
interface History {
  readonly attribute long length (page 290);
  void go (page 290)(in long delta);
  void go (page 291)();
  void back (page 291)();
  void forward (page 291)();
  void pushState (page 291)(in DOMObject data);
  void clearState (page 291)();
};
```

The **`length`** attribute of the `History` **(page 289)** 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 **(page 289)** plus *delta* is less than zero or greater than or equal to the number of items in the session history **(page 290)**, 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 **(page 19)** to traverse the history **(page 290)** to the specified entry, as described below. The specified entry is the one whose index equals the index of the current entry **(page 289)** plus *delta*.

When a user agent is required to **traverse the history** to an entry *target*, the user agent must act as follows:

1. If there are any entries with state objects between the current entry **(page 289)** and the *target* entry (not inclusive), then the user agent must iterate through every entry between the current entry and the *target* entry, starting with the entry closest to the current entry, and ending with the one closest to the *target* entry. For each entry, if the entry is a state object, the user agent must activate the state object **(page 292)**.
2. If the specified entry has a different `Document` object than the current entry **(page 289)** then the user agent must make that `Document` object the user's "current" one for that browsing context **(page 19)**, and must update the browsing context's `Window` object so that any properties that were added while the current entry's `Document` was active are removed, and any properties that were added while the *target* entry's `Document` was active are added back. (If the *target* entry is new, as it is if the user agent should navigated **(page 257)** to it, then there are no such new properties.) The user agent must also update the current location **(page 292)** object to the new location.
3. If the specified entry is a state object, the user agent must activate that state object **(page 292)**.
4. 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.

how does the changing of the global attributes affect .watch() when seen from other Windows?

When the user navigates through a browsing context **(page 19)**, 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`*`)` **(page 290)** method on the various affected `window` **(page 313)** objects.

Some of the other members of the `History` **(page 289)** interface are defined in terms of the `go()` **(page 290)** method, as follows:

| Member | Definition |
|---|---|
| **`go()`** | Must do the same as `go(0)` **(page 290)** |
| **`back()`** | Must do the same as `go(-1)` **(page 290)** |
| **`forward()`** | Must do the same as `go(1)` **(page 290)** |

The **`pushState(`*`data`*`)`** method adds a state object to the history.

When this method is invoked, the user agent must first remove from the session history any entries for that `Document` from the entry after the current entry **(page 289)** up to the last entry in the session history that references the same `Document` object, if any. If the current entry **(page 289)** is the last entry in the session history, or if there are no entries after the current entry **(page 289)** 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 **(page 289)**, with the specified *data* as the state object.

Finally, the user agent must update the current entry **(page 289)** to be the this newly added entry.

There has been a suggestion that pushState() should take a URI and a string; the URI to allow for the page to be bookmarked, and the string to allow the UA to give the page a meaningful title in the history state, if it shows history state.

User agents may limit the number of state objects added to the session history per page. If a page hits the UA-defined limit, user agents must remove the entry immediately after the first entry for that `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 **(page 289)** was removed in the previous step, the current entry **(page 289)** must be set to the last entry for that `Document` object in the session history.

### 4.5.3. 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 **(page 35)** using the `PopStateEvent` **(page 292)** interface, with the state object in the `state` **(page 292)** attribute. This event bubbles but is not cancelable and has no default action.

```
interface PopStateEvent : Event {
  readonly attribute DOMObject state (page 292);
  void initPopStateEvent (page 292)(in DOMString typeArg, in boolean
canBubbleArg, in boolean cancelableArg, in DOMObject statetArg);
  void initPopStateEventNS (page 292)(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)

### 4.5.4. The `Location` (page 292) interface

The **`location`** attribute of the `WindowHTML` **(page 287)** interface must return an object implementing the `Location` **(page 292)** interface.

For historical reasons, the **`location`** attribute of the `HTMLDocument` **(page 25)** interface must return the same object as the `location` **(page 292)** attribute on its associated `WindowHTML` **(page 287)** object.

`Location` **(page 292)** objects provide a representation of the URI of their document, and allow the current entry **(page 289)** of the browsing context **(page 19)**'s session history to be changed, by adding or replacing entries in the `history` **(page 288)** object.

```
interface Location {
  readonly attribute DOMString hash;
  readonly attribute DOMString host;
  readonly attribute DOMString hostname;
  readonly attribute DOMString href (page 293);
  readonly attribute DOMString pathname;
  readonly attribute DOMString port;
  readonly attribute DOMString protocol;
  readonly attribute DOMString search;
  void assign (page 293)(in DOMString url);
  void replace (page 293)(in DOMString url);
  void reload();
};
```

In the ECMAScript DOM binding, objects implementing this interface must stringify to the same value as the `href` **(page 293)** attribute.

In the ECMAScript DOM binding, the `location` members of the `HTMLDocument` **(page 25)** and `WindowHTML` **(page 287)** interfaces behave as if they had a setter: user agents must treats attempts to set these `location` attribute as attempts at setting the `href` **(page 293)** attribute of the relevant `Location` **(page 292)** 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()` **(page 293)** method had been called with the new value as its argument.

When the **`assign(`*`url`*`)`** method is invoked, the UA must navigate **(page 257)** to the specified *url*.

When the **`replace(`*`url`*`)`** method is invoked, the UA must navigate **(page 257)** to the specified *url* with replacement enabled **(page 259)**.

Relative *url* arguments for `assign()` **(page 293)** and `replace()` **(page 293)** must be resolved relative to the base URI of the script that made the method call.

The component parts and .reload() are yet to be defined. If anyone can come up with a decent definition, let me know.

### 4.5.5. Implementation notes for session history

*This section is non-normative.*

The `History` **(page 289)** 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` **(page 288)** object are derived from the actual session history as seen from the user's perspective.

Similarly, a page containing two `iframe` **(page 150)**s has a `history` **(page 288)** object distinct from the `iframe` **(page 150)**s' `history` **(page 288)** 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()` **(page 291)**, 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()` **(page 291)** 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.

## 4.6. Browser state

The **`navigator`** attribute of the `WindowHTML` **(page 287)** interface must return an instance of the `ClientInformation` **(page 294)** 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 (page 294);
  void registerProtocolHandler (page 294)(in DOMString protocol, in
DOMString uri, in DOMString title);
  void registerContentHandler (page 294)(in DOMString mimeType, in DOMString
uri, in DOMString title);
};
```

### 4.6.1. [SCS] Offline Web applications

The **`navigator.onLine`** attribute must return false if the user agent will not contact the network when the user follows links or when a script requests a remote page (or knows that such an attempt would fail), and must return true otherwise.

The **`offline`** event must be fired when the value of the `navigator.onLine` **(page 294)** attribute of the `WindowHTML` **(page 287)** changes from true to false.

The **`online`** event must be fired when the value of the `navigator.onLine` **(page 294)** attribute of the `WindowHTML` **(page 287)** changes from false to true.

These events are in no namespace, do bubble, are not cancelable, have no default action, and use the normal `Event` interface. They must be fired on the body element **(page 35)**. (As the events bubble, they will reach the `WindowHTML` **(page 287)** object.)

### 4.6.2. [SCS] Custom protocol and content handlers

The **`registerProtocolHandler()`** method allows Web sites to register themselves as possible handlers for particular protocols. For example, an online fax service could register itself as a handler of the `fax:` protocol ([RFC2806]), so that if the user clicks on such a link, he is given the opportunity to use that Web site. Analogously, the **`registerContentHandler()`** method allows Web sites to register themselves as possible handlers for content in a particular MIME type. For example, the same online fax service could register itself as a handler for `image/g3fax` files ([RFC1494]), so that if the user has no native application capable of handling G3 Facsimile byte streams, his Web browser can instead suggest he use that site to view the image.

User agents may, within the constraints described in this section, do whatever they like when the methods are called. A UA could, for instance, prompt the user and offer the user the opportunity to add the site to a shortlist of handlers, or make the handlers his default, or cancel the request. UAs could provide such a UI through modal UI or through a non-modal transient notification interface. UAs could also simply silently collect the information, providing it only when relevant to the user.

There is an example of how these methods could be presented to the user **(page 297)** below.

The arguments to the methods have the following meanings:

***protocol*** **(`registerProtocolHandler()` (page 294) only)**

A scheme, such as `ftp` or `fax`. The scheme must be treated case-insensitively by user agents for the purposes of comparing with the scheme part of URIs that they consider against the list of registered handlers.

The *protocol* value, if it contains a colon (as in "`ftp:`"), will never match anything, since schemes don't contain colons.

***mimeType*** **(`registerContentHandler()` (page 294) 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 `chickenkiwi.soup` file was served with the MIME type `application/x-soup`, the UA might instead navigate to the following URI:

```
http://example.com/
soup?url=http%3A%2F%2Fwww.example.net%2Fchickenk%C3%AFwi.soup
```

This site could then fetch the `chickenkiwi.soup` file and do whatever it is that it does with soup (synthesise it and ship it to the user, or whatever).

***title***

A descriptive title of the handler, which the UA might use to remind the user what the site in question is.

User agents should raise security exceptions **(page 267)** 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.

#### *4.6.2.1. Security and privacy*

These mechanisms can introduce a number of concerns, in particular privacy concerns.

**Hijacking all Web usage.** User agents should not allow protocols that are key to its normal operation, such as `http` or `https`, to be rerouted through third-party sites. This would allow a user's activities to be trivially tracked, and would allow user information, even in secure connections, to be collected.

**Hijacking defaults.** It is strongly recommended that user agents do not automatically change any defaults, as this could lead the user to send data to remote hosts that the user is not expecting. New handlers registering themselves should never automatically cause those sites to be used.

**Registration spamming.** User agents should consider the possibility that a site will attempt to register a large number of handlers, possibly from multiple domains (e.g. by redirecting through a series of pages each on a different domain, and each registering a handler for `video/mpeg` — analogous practices abusing other Web browser features have been used by pornography Web sites for many years). User agents should gracefully handle such hostile attempts, protecting the user.

**Misleading titles.** User agents should not rely wholy on the *title* argument to the methods when presenting the registered handlers to the user, since sites could easily lie. For example, a site `hostile.example.net` could claim that it was registering the "Cuddly Bear Happy Content Handler". User agents should therefore use the handler's domain in any UI along with any title.

**Hostile handler metadata.** User agents should protect against typical attacks against strings embedded in their interface, for example ensuring that markup or escape characters in such strings are not executed, that null bytes are properly handled, that over-long strings do not cause crashes or buffer overruns, and so forth.

**Leaking Intranet URIs.** The mechanism described in this section can result in secret Intranet URIs being leaked, in the following manner:

1. The user registers a third-party content handler as the default handler for a content type.
2. The user then browses his corporate Intranet site and accesses a document that uses that content type.
3. The user agent contacts the third party and hands the third party the URI to the Intranet content.

No actual confidential file data is leaked in this manner, but the URIs themselves could contain confidential information. For example, the URI could be `https://www.corp.example.com/upcoming-aquisitions/samples.egf`, which might tell the third party that Example Corporation is intending to merge with Samples LLC. Implementors might wish to consider allowing administrators to disable this feature for certain subdomains, content types, or protocols.

**Leaking secure URIs.** User agents should not send HTTPS URIs to third-party sites registered as content handlers, in the same way that user agents do not send `Referer` headers from secure sites to third-party sites.

**Leaking credentials.** User agents must never send username or password information in the URIs that are escaped and included sent to the handler sites. User agents may even avoid attempting to pass to Web-based handlers the URIs of resources that are known to require authentication to access, as such sites would be unable to access the resources in question without prompting the user for credentials themselves (a practice that would require the user to know whether to trust the third-party handler, a decision many users are unable to make or even understand).

#### *4.6.2.2. Sample user interface*

*This section is non-normative.*

A simple implementation of this feature for a desktop Web browser might work as follows.

The `registerProtocolHandler()` **(page 294)** 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()` **(page 294)** 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%3AS2I0dGVucyBhcmUgdGhlIGN1dGVzdCE%253D".

The `registerContentHandler()` **(page 294)** method would work equivalently, but for unknown MIME types instead of unknown protocols.

## 4.7. [SCS] Client-side session and persistent storage

### 4.7.1. 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` **(page 303)** DOM attribute. Sites can add data to the session storage, and it will be accessible to any page from that domain opened in that window.

> For example, a page could have a checkbox that the user ticks to indicate that he wants insurance:
>
> ```
> <label>
>  <input type="checkbox" onchange="sessionStorage.insurance = checked">
>  I want insurance on this trip.
> </label>
> ```
>
> A later page could then check, from script, whether the user had checked the checkbox or not:
>
> ```
> if (sessionStorage.insurance) { ... }
> ```
>
> If the user had multiple windows opened on the site, each one would have its own individual copy of the session storage object.

The second storage mechanism is designed for storage that spans multiple windows, and lasts beyond the current session. In particular, Web applications may wish to store megabytes of user data, such as entire user-authored documents or a user's mailbox, on the clientside for performance reasons.

Again, cookies do not handle this case well, because they are transmitted with every request.

The `globalStorage` **(page 304)** DOM attribute is used to access the global storage areas.

> The site at example.com can display a count of how many times the user has loaded its page by putting the following at the bottom of its page:
>
> ```
> <p>
>   You have viewed this page
>   <span id="count">an untold number of</span>
>   time(s).
> </p>
> <script>
>   var storage = globalStorage['example.com'];
>   if (!storage.pageLoadCount)
>     storage.pageLoadCount = 0;
>   storage.pageLoadCount = parseInt(storage.pageLoadCount, 10) + 1;
>   document.getElementById('count').textContent = storage.pageLoadCount;
> </script>
> ```

Each domain and each subdomain has its own separate storage area. Subdomains can access the storage areas of parent domains, and domains can access the storage areas of subdomains.

- `globalStorage['']` is accessible to all domains.
- `globalStorage['com']` is accessible to all .com domains
- `globalStorage['example.com']` is accessible to example.com and any of its subdomains
- `globalStorage['www.example.com']` is accessible to www.example.com and example.com, but not www2.example.com.

Storage areas (both session storage and global storage) store strings. To store structured data in a storage area, you must first convert it to a string.

### 4.7.2. The `Storage` (page 300) interface

```
interface Storage {
  readonly attribute unsigned long length (page 301);
  DOMString key (page 301)(in unsigned long index);
  StorageItem (page 302) getItem (page 301)(in DOMString key);
  void setItem (page 301)(in DOMString key, in DOMString data);
  void removeItem (page 301)(in DOMString key);
};
```

Each `Storage` **(page 300)** object provides access to a list of key/value pairs, which are sometimes called items. Keys are strings, and any string (including the empty string) is a valid key. Values are strings with associated metadata, represented by `StorageItem` **(page 302)** objects.

Each `Storage` **(page 300)** object is associated with a list of key/value pairs when it is created, as defined in the sections on the `sessionStorage` **(page 303)** and `globalStorage` **(page 304)** attributes. Multiple separate objects implementing the `Storage` **(page 300)** interface can all be associated with the same list of key/value pairs simultaneously.

Key/value pairs have associated metadata. In particular, a key/value pair can be marked as either "safe only for secure content", or as "safe for both secure and insecure content".

A key/value pair is **accessible** if either it is marked as "safe for both secure and insecure content", or it is marked as "safe only for secure content" and the script in question is running in a secure scripting context.

The **`length`** attribute must return the number of key/value pairs currently present and accessible **(page 301)** in the list associated with the object.

The **`key(`*`n`*`)`** method must return the name of the *n*th accessible key in the list. The order of keys is user-agent defined, but must be consistent within an object between changes to the number of keys. (Thus, adding **(page 301)** or removing **(page 301)** a key may change the order of the keys, but merely changing the value of an existing key must not.) If *n* is less than zero or greater than or equal to the number of key/value pairs in the object, then this method must raise an `INDEX_SIZE_ERR` exception.

The **`getItem(`*`key`*`)`** method must return the `StorageItem` **(page 302)** object representing the key/value pair with the given *key*. If the given *key* does not exist in the list associated with the object, or is not accessible, then this method must return null. Subsequent calls to this method with the same key from scripts running in the same security context must return the same instance of the `StorageItem` **(page 302)** interface. (Such instances must not be shared across security contexts, though.)

The **`setItem(`*`key, value`*`)`** method must first check if a key/value pair with the given *key* already exists in the list associated with the object.

If it does not, then a new key/value pair must be added to the list, with the given *key* and *value*, such that any current or future `StorageItem` **(page 302)** objects referring to this key/value pair will return the value given in the *value* argument. If the script setting the value is running in a secure scripting context, then the key/value pair must be marked as "safe only for secure content", otherwise it must be marked as "safe for both secure and insecure content".

If the given *key does* exist in the list, then, if the key/value pair with the given *key* is accessible, it must have its value updated so that any current or future `StorageItem` **(page 302)** objects referring to this key/value pair will return the value given in the *value* argument. If it is *not* accessible, the method must raise a security exception **(page 267)**.

When the `setItem()` **(page 301)** method is successfully invoked (i.e. when it doesn't raise an exception), events are fired on other `HTMLDocument` **(page 25)** objects that can access the newly stored data, as defined in the sections on the `sessionStorage` **(page 303)** and `globalStorage` **(page 304)** attributes.

The **`removeItem(`*`key`*`)`** method must cause the key/value pair with the given *key* to be removed from the list associated with the object, if it exists and is accessible. If no item with that key exists, the method must do nothing. If an item with that key exists but is not accessible, the method must raise a security exception **(page 267)**.

The `setItem()` **(page 301)** and `removeItem()` **(page 301)** 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` **(page 300)** object must enumerate through the currently stored and accessible keys in the list the object is associated with. (It must not enumerate the values or the actual members of the interface). In the ECMAScript DOM binding, `Storage` **(page 300)** 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` **(page 300)** interface nor of `Object`) must invoke the `getItem()` **(page 301)** method with the property's name as the argument, and setting such a property must invoke the `setItem()` **(page 301)** method with the property's name as the first argument and the given value as the second argument.

### 4.7.3. The `StorageItem` (page 302) interface

Items in `Storage` **(page 300)** objects are represented by objects implementing the `StorageItem` **(page 302)** interface.

```
interface StorageItem {
          attribute boolean secure (page 302);
          attribute DOMString value (page 302);
};
```

In the ECMAScript DOM binding, `StorageItem` **(page 302)** objects must stringify to their `value` **(page 302)** attribute's value.

The **`value`** attribute must return the current value of the key/value pair represented by the object. When the attribute is set, the user agent must invoke the `setItem()` **(page 301)** method of the `Storage` **(page 300)** object that the `StorageItem` **(page 302)** object is associated with, with the key that the `StorageItem` **(page 302)** object is associated with as the first argument, and the new given value of the attribute as the second argument.

`StorageItem` **(page 302)** objects must be *live* **(page 22)**, meaning that as the underlying `Storage` **(page 300)** object has its key/value pairs updated, the `StorageItem` **(page 302)** objects must always return the actual value of the key/value pair they represent.

If the key/value pair has been deleted, the `StorageItem` **(page 302)** object must act as if its value was the empty string. On setting, the key/value pair will be recreated.

The **`secure`** attribute must raise an `INVALID_ACCESS_ERR` exception when accessed or set from a script whose script context is not considered secure. (Basically, if the page is not an SSL page.)

If the scripting context *is* secure, then the `secure` **(page 302)** attribute must return true if the key/value pair is considered "safe only for secure content", and false if it is considered "safe for both secure and insecure content". If it is set to true, then the key/value pair must be flagged as "safe only for secure content". If it is set to false, then the key/value pair must be flagged as "safe for both secure and insecure content".

If a `StorageItem` **(page 302)** object is obtained by a script that is not running in a secure scripting context, and the item is then marked with the "safe only for secure content" flag by a script that *is* running in a secure context, the `StorageItem` **(page 302)** object must continue to be available to the first script, who will be able to read the value of the object. However, any attempt to *set* the value would then start raising exceptions as described in the previous section, and the key/value pair would no longer appear in the appropriate `Storage` **(page 300)** object.

### 4.7.4. The `sessionStorage` (page 303) attribute

The **`sessionStorage`** attribute represents the storage area specific to the current top-level browsing context **(page 19)**.

Each top-level browsing context **(page 19)** has a unique set of session storage areas, one for each domain.

User agents should not expire data from a browsing context's session storage areas, but may do so when the user requests that such data be deleted, or when the UA detects that it has limited storage space, or for security reasons. User agents should always avoid deleting data while a script that could access that data is running. When a top-level browsing context is destroyed (and therefore permanently inaccessible to the user) the data stored in its session storage areas can be discarded with it, as the API described in this specification provides no way for that data to ever be subsequently retrieved.

> ***Note: The lifetime of a browsing context can be unrelated to the lifetime of the actual user agent process itself, as the user agent may support resuming sessions after a restart.***

When a new `HTMLDocument` **(page 25)** is created, the user agent must check to see if the document's top-level browsing context **(page 19)** has allocated a session storage area for that document's domain **(page 267)**. If it has not, a new storage area for that document's domain must be created.

The `Storage` **(page 300)** object for the document's associated `WindowHTML` **(page 287)** object's `sessionStorage` **(page 303)** attribute must then be associated with the domain's session storage area.

When a new top-level browsing context **(page 19)** is created by cloning an existing browsing context **(page 19)**, 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 **(page 19)** is created by a script in an existing browsing context **(page 19)**, or by the user following a link in an existing browsing context, or in some other way related to a specific `HTMLDocument` **(page 25)**, then, if the new context's first `HTMLDocument` **(page 25)** has the same domain **(page 267)** as the `HTMLDocument` **(page 25)** from which the new context was created, the new browsing context must start with a single session storage area. That storage area must be a copy of that domain's session storage area in the original browsing context, which from that point on must be considered separate, with the two storage areas not affecting each other in any way.

When the `setItem()` **(page 301)** method is called on a `Storage` **(page 300)** object *x* that is associated with a session storage area, then, if the method does not raise a security exception **(page 267)**, in every `HTMLDocument` **(page 25)** object whose `WindowHTML` **(page 287)** object's `sessionStorage` **(page 303)** attribute's `Storage`

**(page 300)** object is associated with the same storage area, other than *x*, a `storage` **(page 306)** event must be fired, as described below **(page 306)**.

### 4.7.5. The `globalStorage` (page 304) attribute

```
interface StorageList {
  Storage (page 300) namedItem (page 304)(in DOMString domain);
};
```

The **`globalStorage`** object provides a `Storage` **(page 300)** object for each domain.

In the ECMAScript DOM binding, `StorageList` **(page 304)** objects must support dereferencing such that getting a property that is not a member of the object (i.e. is neither a member of the `StorageList` **(page 304)** interface nor of `Object`) must invoke the `namedItem()` **(page 304)** method with the property's name as the argument.

User agents must have a set of global storage areas, one for each domain.

User agents should only expire data from the global storage areas for security reasons or when requested to do so by the user. User agents should always avoid deleting data while a script that could access that data is running. Data stored in global storage areas should be considered potentially user-critical. It is expected that Web applications will use the global storage areas for storing user-written documents.

The **`namedItem(`*`domain`*`)`** method tries to returns a `Storage` **(page 300)** object associated with the given domain, according to the rules that follow.

The *domain* must first be split into an array of strings, by splitting the string at "." characters (U+002E FULL STOP). If the *domain* argument is the empty string, then the array is empty as well. If the *domain* argument is not empty but has no dots, then the array has one item, which is equal to the *domain* argument. If the *domain* argument contains consecutive dots, there will be empty strings in the array (e.g. the string "hello..world" becomes split into the three strings "hello", "", and "world", with the middle one being the empty string).

Each component of the array must then have the IDNA ToASCII algorithm applied to it, with both the AllowUnassigned and UseSTD3ASCIIRules flags set. [RFC3490] If ToASCII fails to convert one of the components of the string, e.g. because it is too long or because it contains invalid characters, then the user agent must raise a `SYNTAX_ERR` exception. [DOM3CORE] The components after this step consist of only US-ASCII characters.

The components of the array must then be converted to lowercase. Since only US-ASCII is involved at this step, this only requires converting characters in the range A-Z to the corresponding characters in the range a-z.

The resulting array is used in a comparison with another array, as described below. In addition, its components are concatenated together, each part separated by a dot (U+002E), to form the **normalised requested domain**.

If the original *domain* was "Åsgård.Example.Com", then the resulting array would have the three items "xn--sgrd-poac", "example", and "com", and the normalised requested domain would be "xn--sgrd-poac.example.com".

Next, the script's own domain **(page 267)** is processed to find if it is allowed to access the requested domain.

If the script's domain name in not known, e.g. if only the server's IP address is known, and the normalised requested domain **(page 304)** is not the empty string, then the user agent must raise a security exception **(page 267)**.

***Note: If the normalised requested domain* (page 304) *is the empty string, then the rest of this algorithm can be skipped. This is because in that situation, the comparison of the two arrays below will always find them to be the same — the first array in such a situation is also empty and so permission to access that storage area will always be given.***

If the script's domain contains no dots (U+002E) then the string "`.localdomain`" must be appended to the script's domain.

Then, the script's domain must be turned into an array, being split, converted to ASCII, and lowercased as described for the *domain* argument above **(page 304)**.

Of the two arrays, the longest one must then be shortened to the length of the shorter one, by dropping items from the start of the array.

If the *domain* argument is "www.example.com" and the script's domain is "example.com" then the first array will be a three item array ("www", "example", "com"), and the second will be a two item array ("example", "com"). The first array is therefore shortened, dropping the leading parts, making both into the same array ("example", "com").

If the two arrays are not component-for-component identical in literal string comparisons, then the user agent must then raise a security exception **(page 267)**.

Otherwise, the user agent must check to see if it has allocated global storage area for the normalised requested domain **(page 304)**. If it has not, a new storage area for that domain must be created.

The user agent must then create a `Storage` **(page 300)** object associated with that domain's global storage area, and return it.

When the requested *domain* is a top level domain, or the empty string, or a country-specific sub-domain like "co.uk" or "ca.us", the associated global storage area is known as **public storage area**

The `setItem()` **(page 301)** method might be called on a `Storage` **(page 300)** object that is associated with a global storage area for a domain *d*, created by a `StorageList` **(page 304)** object associated with a `WindowHTML` **(page 287)** object *x*. Whenever this occurs, if the method didn't raise an exception, a `storage` **(page 306)** event must be fired, as described below, in every `HTMLDocument` **(page 25)** object that matches the following conditions:

- Its `WindowHTML` **(page 287)** object is not *x*, and
- Its `WindowHTML` **(page 287)** object's `globalStorage` **(page 303)** attribute's `StorageList` **(page 304)** object's `namedItem()` **(page 304)** method would not raise a security exception **(page 267)** according to the rules above if it was invoked with the domain *d*.

In other words, every other document that has access to that domain's global storage area is notified of the change.

### 4.7.6. The **`storage`** (page 306) event

The **`storage`** event is fired in an `HTMLDocument` **(page 25)** when a storage area changes, as described in the previous two sections (for session storage **(page 303)**, for global storage **(page 305)**).

When this happens, a `storage` **(page 300)** event in no namespace, which bubbles, is not cancelable, has no default action, and which uses the `StorageEvent` **(page 306)** interface described below, must be fired on the body element **(page 35)**.

However, it is possible (indeed, for session storage areas, likely) that the target `HTMLDocument` **(page 25)** object is not active at that time. For example, it might not be the current entry **(page 289)** in the session history; user agents typically stop scripts from running in pages that are in the history. In such cases, the user agent must instead delay the firing of the event until such time as the `HTMLDocument` **(page 25)** object in question becomes active again.

When there are multiple delayed `storage` **(page 300)** events for the same `HTMLDocument` **(page 25)** object, user agents should coalesce events with the same `domain` **(page 306)** value (dropping duplicates).

If the DOM of a page that has delayed `storage` **(page 300)** events queued up is discarded **(page 289)**, then the delayed events are dropped as well.

```
interface StorageEvent : Event {
  readonly attribute DOMString domain (page 306);
  void initStorageEvent (page 306)(in DOMString typeArg, in boolean
canBubbleArg, in boolean cancelableArg, in DOMString domainArg);
  void initStorageEventNS (page 306)(in DOMString namespaceURIArg, in
DOMString typeArg, in boolean canBubbleArg, in boolean cancelableArg, in
DOMString domainArg);
};
```

The **`initStorageEvent()`** and **`initStorageEventNS()`** methods must initialise the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS] **(page 0)**

The **`domain`** attribute of the `StorageEvent` **(page 306)** event object must be set to the name of the domain associated with the storage area that changed if that storage area is a global storage area, or the string "`#session`" if it was a session storage area.

### 4.7.7. Miscellaneous implementation requirements for storage areas

#### *4.7.7.1. Disk space*

User agents should limit the total amount of space allowed for a domain based on the domain of the page setting the value.

User agents should not limit the total amount of space allowed on a per-storage-area basis, otherwise a site could just store data in any number of subdomains, e.g. storing up to the limit in a1.example.com, a2.example.com, a3.example.com, etc, circumventing per-domain limits.

User agents should consider additional quota mechanisms (for example limiting the amount of space provided to a domain's subdomains as a group) so that hostile authors can't run scripts from multiple subdomains all adding data to the global storage area in an attempted denial-of-service attack.

User agents may prompt the user when per-domain space quotas are reached, allowing the user to grant a site more space. This enables sites to store many user-created documents on the user's computer, for instance.

User agents should allow users to see how much space each domain is using.

If the storage area space limit is reached during a `setItem()` **(page 301)** call, the user agent should raise an exception.

A mostly arbitrary limit of five megabytes per domain is recommended. Implementation feedback is welcome and will be used to update this suggestion in future.

#### *4.7.7.2. Threads*

Multiple browsing contexts must be able to access the global storage areas simultaneously in a predictable manner. Scripts must not be able to detect any concurrent script execution.

This is required to guarentee that the `length` **(page 301)** attribute of a `Storage` **(page 300)** object never changes while a script is executing, other than in a way that is predictable by the script itself.

There are various ways of implementing this requirement. One is that if a script running in one browsing context accesses a global storage area, the UA blocks scripts in other browsing contexts when they try to access *any* global storage area until the first script has executed to completion. (Similarly, when a script in one browsing context accesses its session storage area, any scripts that have the same top level browsing context and the same domain would block when accessing their session storage area until the first script has executed to completion.) Another (potentially more efficient but probably more complex) implementation strategy is to use optimistic transactional script execution. This specification does not require any particular implementation strategy, so long as the requirement above is met.

### 4.7.8. Security and privacy

#### *4.7.8.1. User tracking*

A third-party advertiser (or any entity capable of getting content distributed to multiple sites) could use a unique identifier stored in its domain's global storage area to track a user across multiple sessions, building a profile of the user's interests to allow for highly 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.

The `globalStorage` **(page 304)** object also introduces a way for sites to cooperate to track users over multiple domains, by storing identifying data in "public **(page 305)**" top-level domain storage area, accessible by any domain.

There are a number of techniques that can be used to mitigate the risk of user tracking:

- Blocking third-party storage: user agents may restrict access to the `globalStorage` **(page 304)** object to scripts originating at the domain of the top-level document of the browsing context **(page 19)**.

  This blocks a third-party site from using its private storage area for tracking a user, but top-level sites could still cooperate with third parties to perferm user tracking by using the "public **(page 305)**" storage area.

- Expiring stored data: user agents may automatically delete stored data after a period of time.

  For example, a user agent could treat the global storage area as session-only storage, deleting the data once the user had closed all the browsing contexts that could access it.

  This can restrict the ability of a site to track a user, as the site would then only be able to track the user across multiple sessions when he authenticates with the site itself (e.g. by making a purchase or logging in to a service).

- Blocking access to the top-level domain ("public **(page 305)**") storage areas: user agents may prevent domains from storing data in and reading data from the top-level domain entries in the `globalStorage` **(page 304)** object.

  In practice this requires a detailed list of all the "public" second-level (and third-level) domains. For example, content at the domain `www.example.com` would be allowed to access `example.com` data but not `com` data; content at the domain `example.co.uk` would be allowed access to `example.co.uk` but not `co.uk` or `uk`; and content at `example.chiyoda.tokyo.jp` would be allowed access to `example.chiyoda.tokyo.jp` but not `chiyoda.tokyo.jp`, `tokyo.jp`, or `jp`, while content at `example.metro.tokyo.jp` would be allowed access to both `example.metro.tokyo.jp` and `metro.tokyo.jp` but not `tokyo.jp` or `jp`. The problem is even more convoluted when one considers private domains with third-party subdomains such as `dyndns.org` or `uk.com`.

Blocking access to the "public **(page 305)**" storage areas can also prevent innocent sites from cooperating to provide services beneficial to the user.

- Treating persistent storage as cookies: user agents may present the persistent storage feature to the user in a way that does not distinguish it from HTTP session cookies. [RFC2965]

  This might encourage users to view persistent storage with healthy suspicion.

- Site-specific white-listing of access to "public **(page 305)**" storage area: user agents may allow sites to access persistent storage for their own domain and subdomains in an unrestricted manner, but require the user to authorise access to the storage area of higher-level domains.

  For example, code at `example.com` would be always allowed to read and write data for `www.example.com` and `example.com`, but if it tried to access `com`, the user agent could display a non-modal message informing the user that the page requested access to `com` and offering to allow it.

- Origin-tracking of persistent storage data: user agents may record the domain of the script that caused data to be stored.

  If this information is then used to present the view of data currently in persistent storage, it would allow the user to make informed decisions about which parts of the persistent storage to prune. Combined with a blacklist ("delete this data and prevent this domain from ever storing data again"), the user can restrict the use of persistent storage to sites that he trusts.

- Shared blacklists: user agents may allow users to share their persistent storage domain blacklists.

  This would allow communities to act together to protect their privacy.

While these suggestions prevent trivial use of this API for user tracking, they do not block it altogether. Within a single domain, a site can continue to track the user across multiple sessions, and can then pass all this information to the third party along with any identifying information (names, credit card numbers, addresses) obtained by the site. If a third party cooperates with multiple sites to obtain such information, a profile can still be created.

However, user tracking is to some extent possible even with no cooperation from the user agent whatsoever, for instance by using session identifiers in URIs, a technique already commonly used for innocuous purposes but easily repurposed for user tracking (even retroactively). This information can then be shared with other sites, using using visitors' IP addresses and other user-specific data (e.g. user-agent headers and configuration settings) to combine separate sessions into coherent user profiles.

#### *4.7.8.2. Cookie resurrection*

If the user interface for persistent storage presents data in the persistent storage feature separately from data in HTTP session cookies, then users are likely to delete data in one and not the other. This would allow sites to use the two features as redundant backup for each other, defeating a user's attempts to protect his privacy.

#### *4.7.8.3. Integrity of "public" storage areas*

Since the "public **(page 305)**" global storage areas are accessible by content from many different parties, it is possible for third-party sites to delete or change information stored in those areas in ways that the originating sites may not expect.

Authors must not use the "public **(page 305)**" global storage areas for storing sensitive data. Authors must not trust information stored in "public **(page 305)**" global storage areas.

#### *4.7.8.4. Cross-protocol and cross-port attacks*

This API makes no distinction between content served over HTTP, FTP, or other host-based protocols, and does not distinguish between content served from different ports at the same host.

Thus, for example, data stored in the global persistent storage for domain "www.example.com" by a page served from HTTP port 80 will be available to a page served in `http://example.com:18080/`, even if the latter is an experimental server under the control of a different user.

Since the data is not sent over the wire by the user agent, this is not a security risk in its own right. However, authors must take proper steps to ensure that all hosts that have fully qualified host names that are subsets of hosts dealing with sensitive information are as secure as the originating hosts themselves.

Similarly, authors must ensure that all Web servers on a host, regardless of the port, are equally trusted if any of them are to use persistent storage. For instance, if a Web server runs a production service that makes use of the persistent storage feature, then other users that have access to that machine and that can run a Web server on another port will be able to access the persistent storage added by the production service (assuming they can trick a user into visiting their page).

However, if one is able to trick users into visiting a Web server with the same host name but on a different port as a production service used by these users, then one could just as easily fake the look of the site and thus trick users into authenticating with the fake site directly, forwarding the request to the real site and stealing the credentials in the process. Thus, the persistent storage feature is considered to only minimally increase the risk involved.

What about if someone is able to get a server up on a port, and can then send people to that URI? They could steal all the data with no further interaction. How about putting the port number at the end of the string being compared? (Implicitly.)

#### *4.7.8.5. DNS spoofing attacks*

Because of the potential for DNS spoofing attacks, one cannot guarentee that a host claiming to be in a certain domain really is from that domain. The `secure` **(page 302)** attribute is provided to mark certain key/value pairs as only being accessible to pages that have been authenticated using secure certificates (or similar mechanisms).

Authors must ensure that they do not mark sensitive items as "safe for both secure and insecure content". (To prevent the risk of a race condition, data stored by scripts in secure contexts default to being marked as "safe only for secure content".)

#### *4.7.8.6. Cross-directory attacks*

Different authors sharing one host name, for example users hosting content on `geocities.com`, all share one persistent storage object. There is no feature to restrict the access by pathname. Authors on shared hosts are therefore recommended to avoid using the persistent storage feature, as it would be trivial for other authors to read from and write to the same storage area.

> ***Note: Even if a path-restriction feature was made available, the usual DOM scripting security model would make it trivial to bypass this protection and access the data from any path.***

#### *4.7.8.7. Public storage areas corresponding to hosts*

If a "public **(page 305)**" global storage area corresponds to a host, as it typically does if for private domains with third-party subdomains such as dyndns.org or uk.com, the host corresponding to the "public" domain has access to all the storage areas of its third-party subdomains. In general, authors are discouraged from using the `globalStorage` **(page 304)** API for sensitive data unless the operators of all the domains involved are trusted.

User agents may mitigate this problem by preventing hosts corresponding to "public **(page 305)**" global storage areas from accessing any storage areas other than their own.

#### *4.7.8.8. Storage areas in the face of untrusted higher-level domains that do not correspond to public storage areas*

Authors should not store sensitive data using the global storage APIs if there are hosts with fully-qualified domain names that are subsets of their own which they do not trust. For example, an author at `finance.members.example.net` should not store sensitive financial user data in the `finance.members.example.net` storage area if he does not trust the host that runs `example.net`.

#### *4.7.8.9. Storage areas in the face of untrusted subdomains*

If an author publishing content on one host, e.g. `example.com`, wishes to use the `globalStorage` **(page 304)** API but does not wish any content on the host's subdomains to access the data, the author should use an otherwise non-existent subdomain name, e.g., `private.example.com`, to store the data. This will be accessible only to that host (and its parent domains), and not to any of the real subdomains (e.g. `upload.example.com`).

#### *4.7.8.10. Implementation risks*

The two primary risks when implementing this persistent storage feature are letting hostile sites read information from other domains, and letting hostile sites write information that is then read from other domains.

Letting third-party sites read data that is not supposed to be read from their domain causes *information leakage*, For example, a user's shopping wishlist on one domain could be used by another domain for 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.

A risk is also presented by servers on local domains having host names matching top-level domain names, for instance having a host called "com" or "net". Such hosts might, if implementations fail to correctly implement the `.localdomain` suffixing, have full access to all the data stored in a UA's persistent storage for that top level domain.

Thus, strictly following the model described in this specification is important for user security.

In addition, a number of optional restrictions related to the "public **(page 305)**" global storage areas are suggested in the previous sections. The design of this API is intended to be such that not supporting these restrictions, or supporting them less than perfectly, does not result in critical security problems. However, implementations are still encouraged to create and maintain a list of "public **(page 305)**" domains, and apply the restrictions described above.

## 4.8. User prompts

window.alert(), .confirm(), .prompt()

## 4.9. Auxillary browsing contexts

The **`window.open()`** method provides a mechanism for navigating an existing browsing context or opening and navigating an auxillary browsing context.

The method has four (optional) arguments.

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 name of the browsing context that is to be navigated. Some of the values have special meanings, namely "`_blank`", "`_self`", and "`_top`". If fewer than two arguments are provided, then the *name* argument defaults to the value "`_blank`".

The third argument, *features*, has no effect and is supported for historical reasons only.

The fourth argument, *replace*, specifies whether or not the new page will replace the page currently loaded in the browsing context, when *target* identifies an existing browsing context. 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 **(page 313)** using the *target* argument as the name, 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` **(page 270)** handler uses the `window.open()` **(page 312)** 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 **(page 257)** the selected browsing context to the URI given in *url*. If the *replace* is true, then replacement must be enabled **(page 259)**.

### 4.9.1. Accessing other browsing contexts

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, if there is one and it is still available.

In ECMAScript implementations, objects that implement the `WindowHTML` **(page 287)** interface must also 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 current `Document`.

The **`length`** DOM attribute on the `WindowHTML` **(page 287)** interface must return the number of child browsing contexts of the current `Document`.

The **`window`**, **`frames`**, **`self`** DOM attributes must all return the `Window` object itself.

### 4.9.2. Browsing context names

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: `_blank`, `_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:

1. If the given browsing context name is the empty string or `_self`, then the chosen browsing context must be the current one.
2. If the given browsing context name is `_parent`, then the chosen browsing context must be the *parent* browsing context of the current one.

3. If the given browsing context name is `_top`, then the chosen browsing context must be the most removed ancestor browsing context of the current one.
4. If the given browsing context name is not `_blank` and there exists a browsing context whose name is the same as the given browsing context name, and that browsing context's document's domain **(page 267)** is the same as the current browsing context's document's domain **(page 267)**, then that browsing context should 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, or most recently focused.
5. Otherwise, the chosen browsing context must be a new auxillary browsing context. If the given browsing context name is not `_blank`, then the new auxillary browsing context's name must be the given browsing context name. Otherwise, it has no name.

this section should move to somewhere where we define browsing contexts.

# 5. Editing

This section describes various features that allow authors to enable users to edit documents and parts of documents interactively.

## 5.1. [TBW] Introduction

*This section is non-normative.*

Would be nice to explain how these features work together.

## 5.2. [SCS] The `contenteditable` (page 315) attribute

The **`contenteditable`** attribute is a common attribute. User agents must support this attribute on all HTML elements **(page 20)**.

The `contenteditable` **(page 315)** attribute is an enumerated attribute **(page 63)** 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` **(page 315)** attribute set to the true state, or if its nearest ancestor with the `contenteditable` **(page 315)** attribute set has its attribute set to the true state, or if it has no ancestors with the `contenteditable` **(page 315)** attribute set but the `Document` has `designMode` **(page 318)** enabled, then the UA must treat the element as **editable** (as described below).

Otherwise, either the HTML element has a `contenteditable` **(page 315)** attribute set to the false state, or its nearest ancestor with the `contenteditable` **(page 315)** attribute set is not *editable* **(page 315)**, or it has no ancestor with the `contenteditable` **(page 315)** attribute set and the `Document` itself has `designMode` **(page 318)** 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 **(page 315)** and its parent element is not, or if an element is editable **(page 315)** 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 typicially 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 **(page 336)**.

> ***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 `mouseydown` events.

**Change the selection**

User agents must allow users to change the selection **(page 336)** within an editing host, even into nested editable elements. This could be triggered as the default action of `keydown` events with various key identifiers and as the default action of `mouseydown` events.

**Insert text**

This action must be triggered as the default action of a `textInput` event, and may be triggered by other commands as well. It must cause the user agent to insert the specified text (given by the event object's `data` attribute in the case of the `textInput` event) at the caret.

If the caret is positioned somewhere where inline-level content **(page 67)** is not allowed (e.g. because the element accepts "both block-level and inline-level content but not both", and the element already contains block-level content), then the user agent must not insert the text directly at the caret position. In such cases the behaviour is UA-dependent, but user agents must not, in response to a request to insert text, generate a DOM that is less conformant than the DOM prior to the request.

User agents should allow users to insert new paragraphs into elements that only contain block-level content.

> For example, given the markup:
>
> ```
> <section>
>  <dl>
>   <dt> Ben </dt>
>   <dd> Goat </dd>
>  </dl>
> </section>
> ```
> ...the user agent should allow the user to insert `p` **(page 108)** elements before and after the `dl` **(page 115)** element, as children of the `section` **(page 94)** element.

**Break block**

UAs should offer a way for the user to request that the current block be broken at the caret, e.g. as the default action of a `keydown` event whose identifier is the "Enter" key and that has no modifiers set.

The exact behaviour is UA-dependent, but user agents must not, in response to a request to break a block, generate a DOM that is less conformant than the DOM prior to the request.

**Insert a line separator**

UAs should offer a way for the user to request an explicit line break at the caret position without breaking the block, for example as in a poem verse or an address. To insert a line break, the user agent must insert a `br` **(page 110)** element.

If the caret is positioned somewhere where inline-level content **(page 67)** is not allowed (e.g. because the element accepts "both block-level and inline-level content but not both", and the element already contains block-level content), then the user agent must not insert the `br` **(page 110)** 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 **(page 336)** 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 **(page 122)** and as being important **(page 123)**, and may offer the user the ability to mark text and blocks with other semantics.

UAs should similarly offer a way for the user to insert empty semantic elements (such as, again, `em` **(page 122)**, `strong` **(page 123)**, 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 **(page 319)** 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 **(page 315)** form control is edited, the changes must be reflected in both its current value *and* its default value. For `input` elements this means updating the `defaultValue` DOM attribute as well as the `value` DOM attribute; for `select` elements it means updating the `option` elements' `defaultSelected` DOM attribute as well as the `selected` DOM attribute; for `textarea` elements this means updating the `defaultValue` DOM attribute as well as the `value` DOM attribute. (Updating the `default*` DOM attributes causes content attributs to be updated as well.)

User agents may perform several commands per user request; for example if the user selects a block of text and hits `Enter`, the UA might interpret that as a request to delete the content of the selection **(page 336)** followed by a request to break the block at that position.

### 5.2.2. Making entire documents editable

Documents have a

**`designMode`**, which can be either enabled or disabled.

The `designMode` **(page 318)** 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` **(page 318)** must be enabled, and if it matches the "`off`" value, then `designMode` **(page 318)** must be disabled. Other values must be ignored.

When `designMode` **(page 318)** 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` **(page 318)** disabled.

Enabling `designMode` **(page 318)** causes scripts in general to be disabled and the document to become editable.

When the `Document` has `designMode` **(page 318)** enabled, event listeners registered on the document or any elements owned by the document must do nothing.

## 5.3. [SCS] 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 **(page 336)** 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`** (page 319) and **`DataTransfer`** (page 319) interfaces

The drag-and-drop processing model involves several events. They all use the `DragEvent` **(page 319)** interface.

```
interface DragEvent : Event {
  readonly attribute DataTransfer (page 319) dataTransfer (page 319);
  void initDragEvent (page 319)(in DOMString typeArg, in boolean
canBubbleArg, in boolean cancelableArg);
  void initDragEventNS (page 319)(in DOMString namespaceURIArg, in DOMString
typeArg, in boolean canBubbleArg, in boolean cancelableArg);
};
```

The **`initDragEvent()`** and **`initDragEventNS()`** methods must initialise the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS] **(page 0)**

The **`dataTransfer`** attribute of the `DragEvent` **(page 319)** interface represents the context information for the event.

When a `DragEvent` **(page 319)** object is created, a new `DataTransfer` **(page 319)** object must be created and assigned to the `dataTransfer` **(page 319)** context information field of the event object.

```
interface DataTransfer {
           attribute DOMString dropEffect (page 320);
           attribute DOMString effectAllowed (page 320);
  void clearData (page 320)(in DOMString format);
  void setData (page 320)(in DOMString format, in DOMString data);
  DOMString getData (page 320)(in DOMString format);
  void setDragImage (page 321)(in Element image, in long x, in long y);
  void addElement (page 321)(in Element element);
};
```

DataTransfer **(page 319)** objects can conceptually contain various kinds of data.

When a DragEvent **(page 319)** event object is initialised, the DataTransfer **(page 319)** object created for the event's dataTransfer **(page 319)** member must be initialised as follows:

- The DataTransfer **(page 319)** object must initially contain no data, no elements, and have no associated image.
- The DataTransfer **(page 319)** object's effectAllowed **(page 320)** attribute must be set to "uninitialized".
- The dropEffect **(page 320)** 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 **(page 320)** attribute during the dragenter **(page 321)** and dragover **(page 321)** 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 **(page 319)** 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 **(page 319)** 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 **(page 319)** 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 **(page 323)**. The *element* argument can be any `Element`; if it is an `img` **(page 148)** 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 **(page 323)**. It adds an element to the `DataTransfer` **(page 319)** 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` **(page 319)** interface defined above, must have the bubbling and cancelable behaviours given in the table below, and must have the context information set up as described after the table.

| Event Name | Target | Bubbles? | Cancelable? | `dataTransfer` (page 321) | `effectAllowed` (page 320) | `dropEffect` (page 320) | Default Action |
|---|---|---|---|---|---|---|---|
| **`dragstart`** | Source node **(page 323)** | ✓ Bubbles | ✓ Cancelable | Contains source node **(page 323)** unless a selection is being dragged, in which case it is empty | `uninitialized` | `none` | Initiate the drag-and-drop operation |
| **`drag`** | Source node **(page 323)** | ✓ Bubbles | ✓ Cancelable | Empty | Same as last event **(page 321)** | `none` | Continue the drag-and-drop operation |
| **`dragenter`** | Immediate user selection **(page 324)** or the body element **(page 35)** | ✓ Bubbles | ✓ Cancelable | Empty | Same as last event **(page 321)** | Based on `effectAllowed` value **(page 322)** | Reject immediate user selection **(page 324)** as potential target element **(page 324)** |
| **`dragleave`** | Previous target element **(page 324)** | ✓ Bubbles | — | Empty | Same as last event **(page 321)** | `none` | None |
| **`dragover`** | Current target element **(page 324)** | ✓ Bubbles | ✓ Cancelable | Empty | Same as last event **(page 321)** | Based on `effectAllowed` value **(page 322)** | Reset the current drag operation **(page 324)** to "none" |
| **`drop`** | Current target element **(page 324)** | ✓ Bubbles | ✓ Cancelable | `getData()` returns data set in `dragstart` event | Same as last event **(page 321)** | Current drag operation **(page 324)** | Varies |
| **`dragend`** | Source node **(page 323)** | ✓ Bubbles | — | Empty | Same as last event **(page 321)** | Current drag operation **(page 324)** | Varies |

The `dataTransfer` **(page 319)** object's contents are empty except for `dragstart` **(page 321)** events and `drop` **(page 321)** events, for which the contents are set as described in the processing model, below.

The `effectAllowed` **(page 320)** attribute must be set to "`uninitialized`" for `dragstart` **(page 321)** 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` **(page 320)** attribute must be set to "`none`" for `dragstart` **(page 321)**, `drag` **(page 321)**, `dragleave` **(page 321)**, and `dragend` **(page 321)** events (except when stated otherwise in the algorithms given in the sections below), to the value corresponding to the current drag operation **(page 324)** for `drop` **(page 321)** events, and to a value based on the `effectAllowed` **(page 320)** attribute's value and to the drag-and-drop source, as given by the following table, for the remaining events (`dragenter` **(page 321)** and `dragover` **(page 321)**):

| **`effectAllowed` (page 320)** | **`dropEffect` (page 320)** |
|---|---|
| `none` | `none` |
| `copy`, `copyLink`, `copyMove`, `all` | `copy` |
| `link`, `linkMove` | `link` |
| `move` | `move` |
| `uninitialized` when what is being dragged is a selection from a text field | `move` |
| `uninitialized` when what is being dragged is a selection | `copy` |
| `uninitialized` when what is being dragged is an `a` **(page 118)** 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` **(page 328)** 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`* (page 148) *elements and `a`* (page 118) *elements with an `href`* (page 273) *attribute have their `draggable`* (page 328) *attribute set to true by default.***

If the user agent determines that something can be dragged, a `dragstart` **(page 321)** event must then be fired.

If it is a selection that is being dragged, then this event must be fired on the node that the user started the drag on (typically the text node that the user originally clicked). If the user did not specify a particular node, for example if the user just told the user agent to begin a drag of "the selection", then the event must be fired on the deepest node that is a common ancestor of all parts of the selection.

If it is not a selection that is being dragged, then the event must be fired on the element that is being dragged.

The node on which the event is fired is the **source node**. Multiple events are fired on this node during the course of the drag-and-drop operation.

If it is a selection that is being dragged, the `dataTransfer` **(page 319)** member of the event must be created with no nodes. Otherwise, it must be created containing just the source node **(page 323)**. Script can use the `addElement()` **(page 321)** method to add further elements to the list of what is being dragged.

If it is a selection that is being dragged, the `dataTransfer` **(page 319)** 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` **(page 148)** element being dragged, then the value of the element's `src` **(page 150)** DOM attribute must be added, associated with the `text/uri-list` format. Otherwise, if it is an `a` **(page 118)** element being dragged, then the value of the element's `href` **(page 120)** 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()` **(page 321)** method of the `dataTransfer` **(page 319)** object of the `dragstart` **(page 321)** 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` **(page 319)** object, both before the event was fired, and during the handling of the event using the `addElement()` **(page 321)** 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 **(page 320)** in the `dataTransfer` **(page 319)** object. This data will be made available again when the `drop` **(page 321)** 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 **(page 329)** 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 **(page 324)** 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 **(page 324)** 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 **(page 324)** changes when the immediate user selection **(page 324)** changes, based on the results of event handlers in the document, as described below.

Both the current target element **(page 324)** and the immediate user selection **(page 324)** 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 **(page 324)** 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` **(page 321)** event at the source node **(page 323)**. If this event is canceled, the user agent must set the current drag operation **(page 324)** to none (no drag operation).
2. Next, if the `drag` **(page 321)** 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 **(page 324)** than during the last iteration (or if this is the first iteration), and if this immediate user selection **(page 324)** is not the same as the current target element **(page 324)**, then the current target element **(page 324)** must be updated, as follows:
        1. If the new immediate user selection **(page 324)** is null, or is in a non-DOM document or application, then set the current target element **(page 324)** to the same value.
        2. Otherwise, the user agent must fire a `dragenter` **(page 321)** event at the immediate user selection **(page 324)**.
        3. If the event is canceled, then the current target element **(page 324)** must be set to the immediate user selection **(page 324)**.

   4. Otherwise, if the current target element **(page 324)** is not the body element **(page 35)**, the user agent must fire a `dragenter` **(page 321)** event at the body element **(page 35)**, and the current target element **(page 324)** must be set to the body element **(page 35)**, regardless of whether that event was canceled or not. (If the body element **(page 35)** is null, then the current target element **(page 324)** 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 **(page 324)** 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` **(page 321)** event at the previous target element.

3. If the current target element **(page 324)** is a DOM element, the user agent must fire a `dragover` **(page 321)** event at this current target element **(page 324)**.

   If the `dragover` **(page 321)** event is canceled, the current drag operation **(page 324)** must be reset to "none".

   Otherwise, the current drag operation **(page 324)** must be set based on the values the `effectAllowed` **(page 320)** and `dropEffect` **(page 320)** attributes of the `dataTransfer` **(page 319)** object had after the event was handled, as per the following table:

| **`effectAllowed` (page 320)** | **`dropEffect` (page 320)** | **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` **(page 321)** event was canceled or not, the drag feedback (e.g. the mouse cursor) must be updated to match the current drag operation **(page 324)**, 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 **(page 324)** 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* **(page 324)**.

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` **(page 321)** event was canceled, then this will be the last iteration. The user agent must follow the following steps, then stop looping.

   1. If the current drag operation **(page 324)** 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 **(page 324)** is null, then the drag operation failed. If the current target element **(page 324)** is a DOM element, the user agent must fire a `dragleave` **(page 321)** 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 **(page 324)** is a DOM element, the user agent must fire a `drop` **(page 321)** event at it; otherwise, it must use platform-specific conventions for indicating a drop.

      When the target is a DOM element, the `dropEffect` **(page 320)** attribute of the event's `dataTransfer` **(page 319)** object must be given the value representing the current drag operation **(page 324)** (`copy`, `link`, or `move`), and the object must be set up so that the `getData()` **(page 320)** method will return the data that was added during the `dragstart` **(page 321)** event.

      If the event is canceled, the current drag operation **(page 324)** must be set to the value of the `dropEffect` **(page 320)** attribute of the event's `dataTransfer` **(page 319)** 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 (page 324) 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 **(page 324)** to "none".

   3. Finally, the user agent must fire a `dragend` **(page 321)** event at the source node **(page 323)**, with the `dropEffect` **(page 320)** attribute of the event's `dataTransfer` **(page 319)** object being set to the value corresponding to the current drag operation **(page 324)**.

      ***Note: The current drag operation* (page 324) *can change during the processing of the `drop`* (page 321) *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 (page 324) is a text field (e.g. `textarea`, or an `input` element with `type="text"`), and a `drop` (page 321) event was fired in the previous step, and the current drag operation (page 324) 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 (page 324) is a text field (e.g. `textarea`, or an `input` element with `type="text"`), and a `drop` (page 321) event was fired in the previous step, and the current drag operation (page 324) 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 **(page 323)** 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` **(page 319)** object when the drag started, even though no `dragstart` **(page 321)** 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` **(page 327)** content attribute set. The `draggable` **(page 327)** attribute is an enumerated attribute **(page 63)**. 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` **(page 328)** DOM attribute is true become draggable as well.

If an element's `draggable` **(page 327)** content attribute has the state *true*, the `draggable` **(page 328)** DOM attribute must return true.

Otherwise, if the element's `draggable` **(page 327)** content attribute has the state *false*, the `draggable` **(page 328)** DOM attribute must return false.

Otherwise, the element's `draggable` **(page 327)** content attribute has the state *auto*. If the element is an `img` **(page 148)** element, or, if the element is an `a` **(page 118)** element with an `href` **(page 273)** content attribute, the `draggable` **(page 328)** DOM attribute must return true.

Otherwise, the `draggable` **(page 328)** DOM must return false.

If the `draggable` **(page 328)** DOM attribute is set to the value false, the `draggable` **(page 327)** content attribute must be set to the literal value `false`. If the `draggable` **(page 328)** DOM attribute is set to the value true, the `draggable` **(page 327)** 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 **(page 324)**) 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 **(page 317)**.

#### *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 **(page 324)**) 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 **(page 324)**) 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` **(page 319)** object during the `dragstart` **(page 321)** event available to scripts until the `drop` **(page 321)** 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` **(page 321)** 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. [SCS] 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` **(page 25)** object.

The undo transaction history **(page 329)** is a list of entries. The entries are of two type: DOM changes **(page 330)** and undo objects **(page 330)**.

Each **DOM changes** entry in the undo transaction history **(page 329)** consists of batches of one or more of the following:

- Changes to the content attributes **(page 20)** of an `Element` node.
- Changes to the DOM attributes **(page 20)** of a `Node`.
- Changes to the DOM hierarchy of nodes that are descendants of the `HTMLDocument` **(page 25)** 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 **(page 330)** 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 **(page 330)** entries are handled by the UA in response to user edits of form controls and editing hosts on the page, and undo object **(page 330)** 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` (page 330) interface

This API sucks. Seriously. It's a terrible API. Really bad. I hate it. Here are the requirements:

- Has to cope with cases where the server has undo state already when the page is loaded, that can be stuffed into the undo buffer onload.
- Has to support undo/redo.
- Has to cope with the "undo" action being "contact the server and tell it to undo", rather than it being the opposite of the "redo" action.
- Has to cope with some undo states expiring from the undo history (e.g. server can only remember one undelete action) but other states not expiring (e.g. client can undo arbitrary amounts of local edits).

To manage undo object **(page 330)** entries in the undo transaction history **(page 329)**, the `UndoManager` **(page 330)** interface can be used:

```
interface UndoManager {
  unsigned long add (page 331)(in DOMObject data, in DOMStrong title);
  void remove (page 332)(in unsigned long index);
  void clearUndo (page 332)();
  void clearRedo (page 332)();
  DOMObject item (page 331)(in unsigned long index);
  readonly attribute unsigned long length (page 331);
  readonly attribute unsigned long position (page 331);
};
```

The **`undoManager`** attribute of the `WindowHTML` **(page 287)** interface must return the object implementing the `UndoManager` **(page 330)** interface for that `WindowHTML` **(page 287)** object's associated `HTMLDocument` **(page 25)** object.

In the ECMAScript DOM binding, objects implementing this interface must also support being dereferenced using the square bracket notation, such that dereferencing with an integer index is equivalent to invoking the `item()` **(page 331)** method with that index (e.g. `undoManager[1]` returns the same as `undoManager.item(1)`).

`UndoManager` **(page 330)** objects represent their document's undo transaction history **(page 329)**. Only undo object **(page 330)** entries are visible with this API, but this does not mean that DOM changes **(page 330)** entries are absent from the undo transaction history **(page 329)**.

The **`length`** attribute must return the number of undo object **(page 330)** entries in the undo transaction history **(page 329)**.

The **`item(`*`n`*`)`** method must return the *n*th undo object **(page 330)** entry in the undo transaction history **(page 329)**.

The undo transaction history **(page 329)** has a **current position**. This is the position between two entries in the undo transaction history **(page 329)**'s list where the previous entry represents what needs to happen if the user invokes the "undo" command (the "undo" side, lower numbers), and the next entry represents what needs to happen if the user invokes the "redo" command (the "redo" side, higher numbers).

The **`position`** attribute must return the index of the undo object **(page 330)** entry nearest to the undo position **(page 331)**, on the "redo" side. If there are no undo object **(page 330)** entries on the "redo" side, then the attribute must return the same as the `length` **(page 331)** attribute. If there are no undo object **(page 330)** entries on the "undo" side of the undo position **(page 331)**, the `position` **(page 331)** attribute returns zero.

> ***Note: Since the undo transaction history*** **(page 329)** ***contains both undo object*** **(page 330)** ***entries and DOM changes*** **(page 330)** ***entries, but the `position`*** **(page 331)** ***attribute only returns indices relative to undo object*** **(page 330)** ***entries, it is possible for several "undo" or "redo" actions to be performed without the value of the `position`*** **(page 331)** ***attribute changing.***

The **`add(`*`data, title`*`)`** method's behaviour depends on the current state. Normally, it must insert the *data* object passed as an argument into the undo transaction history **(page 329)** immediately before the undo position **(page 331)**, optionally remembering the given *title* to use in the UI. If the method is called during an undo operation **(page 332)**, however, the object must instead be added immediately *after* the undo position **(page 331)**.

If the method is called and there is neither an undo operation in progress **(page 332)** nor a redo operation in progress **(page 333)** then any entries in the undo transaction history **(page 329)** after the undo position **(page 331)** must be removed (as if `clearRedo()` **(page 332)** had been called).

> We could fire events when someone adds something to the undo history -- one event per undo object entry before the position (or after, during redo addition), allowing the script to decide if that entry should remain or not. Or something. Would make it potentially easier to expire server-held state when the server limitations come into play.

The **`remove(`*`index`*`)`** method must remove the undo object **(page 330)** entry with the specified *index*. If the index is less than zero or greater than or equal to `length` **(page 331)** then the method must raise an `INDEX_SIZE_ERR` exception. DOM changes **(page 330)** entries are unaffected by this method.

The **`clearUndo()`** method must remove all entries in the undo transaction history **(page 329)** before the undo position **(page 331)**, be they DOM changes **(page 330)** entries or undo object **(page 330)** entries.

The **`clearRedo()`** method must remove all entries in the undo transaction history **(page 329)** after the undo position **(page 331)**, be they DOM changes **(page 330)** entries or undo object **(page 330)** 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()` **(page 334)** method is called with the `undo` **(page 334)** command, the user agent must perform an undo operation.

If the undo position **(page 331)** is at the start of the undo transaction history **(page 329)**, then the user agent must do nothing.

If the entry immediately before the undo position **(page 331)** is a DOM changes **(page 330)** entry, then the user agent must remove that DOM changes **(page 330)** 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 **(page 330)** entry (consisting of the opposite of those DOM changes) to the undo transaction history **(page 329)** on the other side of the undo position **(page 331)**.

If the DOM changes cannot be undone (e.g. because the DOM state is no longer consistent with the changes represented in the entry), then the user agent must simply remove the DOM changes **(page 330)** entry, without doing anything else.

If the entry immediately before the undo position **(page 331)** is an undo object **(page 330)** entry, then the user agent must first remove that undo object **(page 330)** entry from the undo transaction history **(page 329)**, and then must fire an `undo` **(page 333)** event on the `Document` object, using the undo object **(page 330)** entry's associated undo object as the event's data.

Any calls to `add()` **(page 331)** 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()` **(page 334)** method is called with the `redo` **(page 334)** command, the user agent must perform a redo operation.

This is mostly the opposite of an undo operation **(page 332)**, but the full definition is included here for completeness.

If the undo position **(page 331)** is at the end of the undo transaction history **(page 329)**, then the user agent must do nothing.

If the entry immediately after the undo position **(page 331)** is a DOM changes **(page 330)** entry, then the user agent must remove that DOM changes **(page 330)** 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 **(page 330)** entry (consisting of the opposite of those DOM changes) to the undo transaction history **(page 329)** on the other side of the undo position **(page 331)**.

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 **(page 330)** entry, without doing anything else.

If the entry immediately after the undo position **(page 331)** is an undo object **(page 330)** entry, then the user agent must first remove that undo object **(page 330)** entry from the undo transaction history **(page 329)**, and then must fire a `redo` **(page 333)** event on the `Document` object, using the undo object **(page 330)** entry's associated undo object as the event's data.

### 5.4.4. The `UndoManagerEvent` (page 333) interface and the `undo` (page 333) and `redo` (page 333) events

```
interface UndoManagerEvent : Event {
  readonly attribute DOMObject data (page 333);
  void initUndoManagerEvent (page 333)(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] **(page 0)**

The **`data`** attribute represents the undo object **(page 330)** 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` **(page 333)** interface, with the `data` **(page 333)** field containing the relevant undo object **(page 330)**.

#### 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 **(page 329)**, it could manipulate the undo transaction history **(page 329)** 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 **(page 329)** 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 **(page 329)** (such as providing a tree-based approach to document state). Such UI models should be based upon the single undo transaction history **(page 329)** 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` **(page 25)** interface allows scripts to perform actions on the current selection **(page 336)** 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 **(page 315)**, of if there is no selection and the caret is not inside an editing host **(page 315)**, then the user agent must do nothing.

**If the *commandID* is `undo`**
: The user agent must move back one step **(page 332)** in its undo transaction history **(page 329)**, restoring the associated state. If there is no further undo information the user agent must do nothing. See the undo history **(page 329)**.

**If the *commandID* is `redo`**
: The user agent must move forward one step **(page 333)** in its undo transaction history **(page 329)**, restoring the associated state. If there is no further undo (well, "redo") information the user agent must do nothing. See the undo history **(page 329)**.

**If the *commandID* is `selectAll`**

The user agent must change the selection so that all the content in the currently focused editing host **(page 315)** is selected. If no editing host **(page 315)** 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.* **(page 334)** The user agent must act as if the user had requested that the selection be wrapped in the semantics **(page 317)** of the `sup` **(page 140)** 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.* **(page 334)** The user agent must act as if the user had requested that the selection be wrapped in the semantics **(page 317)** of the *`sub`* **(page 140)** element (or, again, unwrapped, or have that semantic inserted or removed, as defined by the UA).

**If the *commandID* is `formatBlock`**

*Editing hosts only.* **(page 334)** This command changes the semantics of the blocks containing the selection.

If there is no selection, then, where in the description below refers to the selection, the user agent must act as if the selection was an empty range at the caret position.

If the *value* parameter is not specified or has a value other than one of the following literal strings:

- `<address>`
- `<aside>`
- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`
- `<nav>`
- `<p>`
- `<pre>`

...then the user agent must do nothing.

Otherwise, the user agent must, for every position in the selection, take the furthest block-level element **(page 67)** ancestor of that position that contains only inline-level content **(page 67)** and is not being used as a structured inline-level element **(page 68)**, and, if that element is a descendant of the editing host, rename it according to the *value*, by stripping the leading < character and the trailing > character and using the rest as the new tag name.

**If the *commandID* is `delete`**

*Editing hosts only.* **(page 334)** The user agent must act as if the user had performed a backspace operation **(page 317)**.

**If the *commandID* is `forwardDelete`**

*Editing hosts only.* **(page 334)** The user agent must act as if the user had performed a forward delete operation **(page 317)**.

**If the *commandID* is `insertLineBreak`**

*Editing hosts only.* **(page 334)** The user agent must act as if the user had requested a line separator **(page 317)**.

**If the *commandID* is `insertParagraph`**

*Editing hosts only.* **(page 334)** The user agent must act as if the user had performed a break block **(page 317)** editing action.

**If the *commandID* is `insertText`**

*Editing hosts only.* **(page 334)** The user agent must act as if the user had inserted text **(page 316)** 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. [SCS] The text selection APIs

Every browsing context **(page 19)** has **a selection**. The selection may be empty, and the selection may have more than one range (a disjointed selection). The user should be able to change the selection. User agents are not required to let the user select more than one range, and may collapse multiple ranges in the selection to a single range when the user interacts with the selection. (But, of course, the user agent may let the user create selections with multiple ranges.)

This one selection must be shared by all the content of the browsing context (though not by nested browsing contexts **(page 19)**), 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 **(page 19)**'s selection **(page 336)**, each `textarea` and `input` element has an independent selection. These are the **text field selections**.

The `datagrid` **(page 219)** 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`* (page 337) *pseudo-element. [SELECTORS] [CSS21]***

### 5.6.1. APIs for the browsing context selection

The **`getSelection()`** method on the `WindowHTML` **(page 287)** interface must return the `Selection` **(page 337)** object representing the selection **(page 336)** of that `WindowHTML` **(page 287)** object's browsing context **(page 19)**.

For historical reasons, the **`getSelection()`** method on the `HTMLDocument` **(page 25)** interface must return the same `Selection` **(page 337)** object.

```
interface Selection {
  readonly attribute Node anchorNode (page 338);
  readonly attribute long anchorOffset (page 338);
  readonly attribute Node focusNode (page 338);
  readonly attribute long focusOffset (page 338);
  readonly attribute boolean isCollapsed (page 338);
  void collapse (page 338)(in Node parentNode, in long offset);
  void collapseToStart (page 338)();
  void collapseToEnd (page 338)();
  void selectAllChildren (page 338)(in Node parentNode);
  void deleteFromDocument (page 338)();
  readonly attribute long rangeCount (page 338);
  Range getRangeAt (page 338)(in long index);
  void addRange (page 338)(in Range range);
  void removeRange (page 339)(in Range range);
  void removeAllRanges (page 339)();
  DOMString toString (page 339)();
};
```

The `Selection` **(page 337)** 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` **(page 337)** 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` **(page 337)** 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` **(page 25)** object with which the `Selection` **(page 337)** object is associated. Otherwise it is, and the method must remove all the ranges in the `Selection` **(page 337)** 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()` **(page 338)** 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()` **(page 338)** 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()` **(page 338)** 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` **(page 338)** 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()` **(page 339)** 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` **(page 338)** attribute returns 0 after the `removeAllRanges()` **(page 339)** 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` **(page 337)** interface must stringify to the value returned by the object's `toString()` **(page 339)** 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`*** **(page 337)** ***interface has no relation to the `DataGridSelection`*** **(page 240)** ***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 (page 339)();
           attribute unsigned long selectionStart (page 339);
           attribute unsigned long selectionEnd (page 340);
  void setSelectionRange (page 340)(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()` **(page 340)** method had been called, with the new value as the first argument, and the current value of the selectionEnd **(page 340)** attribute as the second argument, unless the current value of the selectionEnd **(page 340)** 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() **(page 340)** method had been called, with the current value of the selectionStart **(page 339)** 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 **(page 360)** and, by default, in server-sent DOM events **(page 341)**, use the **`message`** event.

The following interface is defined for this event:

```
interface MessageEvent : Event {
  readonly attribute DOMString data (page 341);
  readonly attribute DOMString domain (page 341);
  readonly attribute DOMString uri (page 341);
  readonly attribute Document source (page 341);
  void initMessageEvent (page 341)(in DOMString typeArg, in boolean
canBubbleArg, in boolean cancelableArg, in DOMString dataArg, in DOMString
domainArg, in DOMString uriArg, in Document documentArg);
  void initMessageEventNS (page 341)(in DOMString namespaceURI, in DOMString
typeArg, in boolean canBubbleArg, in boolean cancelableArg, in DOMString
dataArg, in DOMString domainArg, in DOMString uriArg, in Document
documentArg);
};
```

The **`initMessageEvent()`** and **`initMessageEventNS()`** methods must initialise the event in a manner analogous to the similarly-named methods in the DOM3 Events interfaces. [DOM3EVENTS] **(page 0)**

The **`data`** attribute represents the message being sent.

The **`domain`** attribute represents, in cross-document messaging **(page 360)**, the domain of the document from which the message came.

The **`uri`** attribute represents, in cross-document messaging **(page 360)**, the address of the document from which the message came.

The **`source`** attribute represents, in cross-document messaging **(page 360)**, the `Document` from which the message came.

## 6.2. [SCS] Server-sent DOM events

This section describes a mechanism for allowing servers to dispatch DOM events into documents that expect it. The `event-source` **(page 217)** 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` **(page 341)** interface.

```
interface RemoteEventTarget {
  void addEventSource (page 342)(in DOMString src);
  void removeEventSource (page 342)(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 **(page 342)** 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 **(page 342)** for that object. If the same URI has been registered multiple times, removing it must only remove one instance of that URI for each invokation of the method.

Relative URIs must be resolved relative to $\boxed{...}$.

### 6.2.2. Connecting to an event stream

Each object implementing the `EventTarget` and `RemoteEventTarget` **(page 341)** 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 **(page 267)**, the semantics of the Access-Control HTTP header must be followed. [ACCESSCONTROL]

HTTP 200 OK responses with a `Content-Type` **(page 265)** header specifying the type `application/x-dom-event-stream` that are either from the document's domain **(page 267)** or explicitly allowed by the Access-Control HTTP headers must be processed line by line as described below **(page 345)**.

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 that are not valid UTF-8 sequences must be interpreted as the U+FFFD REPLACEMENT CHARACTER.

A leading U+FEFF BYTE ORDER MARK character must be ignored if 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 `message` **(page 341)** 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 section 1.4.2 "Complete list of event types", then the interface used must default to the interface relevant for that event type. [DOM3EVENTS]

> For example:

> ```
> Event: click
> ```
> ...would cause `Class` to be treated as `MouseEvent`.

If the `Namespace` is null and the `Event` field is `message` **(page 341)** (including if it was not specified explicitly), then the `MessageEvent` **(page 341)** 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 section 1.4.2 "Complete list of event types", 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. [DOM3EVENTS]

> 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` **(page 341)** with the interface `MessageEvent` **(page 341)** to be dispatched on the `event-source` **(page 217)** element, which would then bubble up the DOM, and whose `data` **(page 341)** 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` **(page 341)** 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` **(page 341)** object that field is ignored, leaving the event as a second `message` **(page 341)** 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` **(page 341)** 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` **(page 217)** attribute of the `event-source` **(page 217)** 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` **(page 217)** 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` **(page 217)** functionality on a per-page basis.

## 6.3. [SCS] 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` **(page 349)** interface.

The `WindowHTML` **(page 287)** interface provides three constructors for creating `Connection` **(page 349)** objects: `TCPConnection()` **(page 351)**, for creating a direct (possibly encrypted) link to another node on the Internet using TCP/IP; `LocalBroadcastConnection()` **(page 353)**, 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()` **(page 355)**, 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. [TBW] 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` (page 349) interface

```
interface Connection {
  readonly attribute DOMString network (page 350);
  readonly attribute DOMString peer (page 350);
  readonly attribute int readyState (page 350);
           attribute EventListener onopen (page 350);
           attribute EventListener onread (page 350);
           attribute EventListener onclose (page 350);
  void send (page 350)(in DOMString data);
  void disconnect (page 351)();
};
```

`Connection` **(page 349)** objects must also implement the `EventTarget` interface. [DOM3EVENTS]

When a `Connection` **(page 349)** 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` **(page 350)** attribute must be set as soon as the `Connection` **(page 349)** object is created, and keeps the same value for the lifetime of the object. The `peer` **(page 350)** 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` **(page 350)** attribute's value must be changed to 1, and the `open` **(page 351)** event must be fired on the `Connection` **(page 349)** object.

When data is received, the `read` **(page 351)** event will be fired on the `Connection` **(page 349)** object.

When the connection is closed, the `readyState` **(page 350)** attribute's value must be changed to 2, and the `close` **(page 351)** event must be fired on the `Connection` **(page 349)** object.

The **`onopen`**, **`onread`**, and **`onclose`** attributes must, when set, register their new value as an event listener for their respective events (namely `open` **(page 351)**, `read` **(page 351)**, and `close` **(page 351)**), and unregister their previous value if any.

The **`send()`** method transmits data using the connection. If the connection is not yet established, it must raise an `INVALID_STATE_ERR` exception. If the connection *is* established, then the behaviour depends on the connection type, as described below.

The **disconnect()** method must close the connection, if it is open. If the connection is already closed, it must do nothing. Closing the connection causes a close **(page 351)** event to be fired and the readyState **(page 350)** attribute's value to change, as described above **(page 350)**.

### 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 disconnect() **(page 351)** 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 **(page 351)** interface for this event.

```
interface ConnectionReadEvent : Event {
  readonly attribute DOMString data (page 351);
  readonly attribute DOMString source (page 351);
  void initConnectionReadEvent (page 351)(in DOMString typeArg, in boolean
canBubbleArg, in boolean cancelableArg, in DOMString dataArg);
  void initConnectionReadEventNS (page 351)(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] **(page 0)**

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 peer **(page 350)** attribute on the Connection **(page 349)** 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 read **(page 351)** 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 WindowHTML **(page 287)** interface returns a new object implementing the

`Connection` **(page 349)** 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.

We currently don't allow connections to be set up back to an originating IP address, but we could, if the subdomain is the empty string.

First, if the script's domain **(page 267)** is not a host name (e.g. it is an IP address) then the UA must raise a security exception **(page 267)**.

Then, if the *subdomain* argument is null or the empty string, the target host is the script's domain **(page 267)**. Otherwise, the *subdomain* argument is prepended to the script's domain **(page 267)** with a dot separating the two strings, and that is the target host.

If either:

- the target host is not a valid host name, or
- the *port* argument is neither equal to 80, nor equal to 443, nor greater than or equal to 1024 and less than or equal to 65535,

...then the UA must raise a security exception **(page 267)**.

Otherwise, the user agent must verify that the the string representing the script's domain in IDNA format **(page 267)** can be obtained without errors. If it cannot, then the user agent must raise a security exception **(page 267)**.

The user agent may also raise a security exception **(page 267)** at this time if, for some reason, permission to create a direct TCP connection to the relevant host is denied. Reasons could include the UA being instructed by the user to not allow direct connections, or the UA establishing (for instance using UPnP) that the network topology will cause connections on the specified port to be directed at the wrong host.

If no exceptions are raised by the previous steps, then a new `Connection` **(page 349)** object must be created, its `peer` **(page 350)** attribute must be set to a string consisting of the name of the target host, a colon (U+003A COLON), and the port number as decimal digits, and its `network` **(page 350)** attribute must be set to the same value as the `peer` **(page 350)** attribute.

This object must then be returned.

The user agent must then begin trying to establish a connection with the target host and specified port. (This typically would begin in the backgound, while the script continues to execute.)

If the *secure* boolean argument is set to true, then the user agent must establish a secure connection with the target host and specified port using TLS or another protocol, negotiated with the server. [RFC2246] If this fails the user agent must act as if it had closed the connection **(page 350)**.

Once a secure connection is established, or if the *secure* boolean argument is not set to true, then the user agent must continue to connect to the server using the protocol described in the section entitled clients connecting over TCP **(page 357)**. All data on connections made using TLS must be sent as "application data".

Once the connection is established, the UA must act as described in the section entitled sending and receiving data over TCP **(page 359)**.

User agents should allow multiple TCP connections to be established per host. In particular, user agents should not apply per-host HTTP connection limits to connections established with the `TCPConnection` **(page 351)** constructor.

### 6.3.5. Broadcast connections

The **`LocalBroadcastConnection()`** constructor on the `WindowHTML` **(page 287)** interface returns a new object implementing the `Connection` **(page 349)** interface, set up to broadcast on the local network.

When this constructor is invoked, a new `Connection` **(page 349)** object must be created.

The `network` **(page 350)** attribute of the object must be set to the string representing the script's domain in IDNA format **(page 267)**. If this string cannot be obtained, then the user agent must raise a security exception **(page 267)** exception when the constructor is called.

The `peer` **(page 350)** attribute must be set to the empty string.

The object must then be returned, unless, for some reason, permission to broadcast on the local network is to be denied. In the latter case, a security exception **(page 267)** must be raised instead. User agents may deny such permission for any reason, for example a user preference.

If the object is returned (i.e. if no exception is raised), the user agent must the begin broadcasting and listening on the local network, in the background, as described below. The user agent may define "the local network" in any way it considers appropriate and safe; for instance the user agent may ask the user which network (e.g. Bluetooth, IrDA, Ethernet, etc) the user would like to broadcast on before beginning broadcasting.

UAs may broadcast and listen on multiple networks at once. For example, the UA could broadcast on both Bluetooth and Wifi at the same time.

As soon as the object is returned, the connection has been established **(page 350)**, which implies that the `open` **(page 351)** 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 `send(data)` **(page 350)** method is invoked on a `Connection` **(page 349)** object that was created by the `LocalBroadcastConnection()` **(page 353)** constructor, the user agent must follow these steps:

1. Create a string consisting of the value of the `network` **(page 350)** attribute of the `Connection` **(page 349)** 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()` **(page 355)** 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` **(page 349)** object that was created by the `LocalBroadcastConnection()` **(page 353)** constructor and whose `network` **(page 350)** attribute exactly matches *d*, a `read` **(page 351)** event must be fired on the `Connection` **(page 349)** object. The string *s*, with the first character removed, must be used as the `data` **(page 351)**, and the source IP address of the packet as the `source` **(page 351)**.

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. [TBW] Broadcasting over Bluetooth*

Does anyone know enough about Bluetooth to write this section?

*6.3.5.3. [TBW] 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 `WindowHTML` **(page 287)** interface returns a new object implementing the `Connection` **(page 349)** interface, set up for a direct connection to a user-specified host.

When this constructor is invoked, a new `Connection` **(page 349)** object must be created.

The `network` **(page 350)** attribute of the object must be set to the string representing the script's domain in IDNA format **(page 267)**. If this string cannot be obtained, then the user agent must raise a security exception **(page 267)** exception when the constructor is called.

The `peer` **(page 350)** 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 **(page 267)** 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` **(page 349)** 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` **(page 349)** 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 **(page 350)**, 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` **(page 350)** 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` **(page 350)** 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 **(page 358)**.

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 **(page 357)**.

Once the connection is established, the UA must act as described in the section entitled sending and receiving data over TCP **(page 359)**.

***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. [TBW] Peer-to-peer connections over Bluetooth*

Does anyone know enough about Bluetooth to write this section?

*6.3.6.3. [TBW] 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` **(page 351)** and `PeerToPeerConnection` **(page 355)** 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` **(page 351)** and `PeerToPeerConnection` **(page 355)** 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 **(page 350)**.

***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 **(page 350)** and give up trying to connect.

Otherwise, the user agent must then take the string representing the script's domain in IDNA format **(page 267)**, 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 **(page 350)** and give up trying to connect.

Otherwise, the connection has been established **(page 350)** (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 **(page 350)** 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` **(page 355)**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 **(page 350)** and give up trying to connect.

Otherwise, the user agent must then take the string representing the script's domain in IDNA format **(page 267)**, 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 **(page 350)** and give up trying to connect.

Otherwise, the connection has been established **(page 350)** (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 **(page 350)** and give up trying to connect.

### *6.3.7.3. Sending and receiving data over TCP*

When the `send(`*`data`*`)` **(page 350)** method is invoked on the connection's corresponding `Connection` **(page 349)** 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` **(page 351)** event must be fired on the `Connection` **(page 349)** object with that string as the `data` **(page 351)**. 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*** **(page 161)*****) 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. [TBW] 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. [TBW] Relationship to other standards

Should have a section talking about the fact that we blithely ignoring IANA's port assignments here.

Should explain why we are not reusing HTTP for this. (HTTP is too heavy-weight for such a simple need; requiring authors to implement an HTTP server just to have a party line is too much of a barrier to entry; cannot rely on prebuilt components; having a simple protocol makes it much easier to do RAD; HTTP doesn't fit the needs and doesn't have the security model needed; etc)

## 6.4. [SCS] 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 `Document` object, the user agent must create an event that uses the `MessageEvent` **(page 341)** interface, with the event name `message` **(page 341)**, which bubbles, is cancelable, and has no default action. The `data` **(page 341)** attribute must be set to the value passed as the *message* argument to the `postMessage()` **(page 360)** method, the `domain` **(page 341)** attribute must be set to the domain of the document that the script that invoked the methods is associated with, the `uri` **(page 341)** attribute must be set to the URI of that document, and the `source` **(page 341)** attribute must be set to the `Document` object representing that document.

The event must then be dispatched at the `Document` object itself.

⚠**Warning!** ***Authors should check the `domain`* (page 341) *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` **(page 153)** element that contains document B, and script in document A calls `postMessage()` **(page 360)** on document B, then a message event will be fired on that element, marked as originating from document A. The script in document A might look like:

```
var o = document.getElementsByTagName('object')[0];
o.contentDocument.postMessage (page 360)('Hello world');
```

To register an event handler for incoming events, the script would use `addEventListener()` (or similar mechanisms). For example, the script in document B might look like:

```
document.addEventListener('message', receiver, false);
function receiver(e) {
  if (e.domain == 'example.com') {
    if (e.data == 'Hello world') {
      e.source.postMessage('Hello');
    } else {
      alert(e.data);
    }
  }
}
```

This script first checks the domain is the expected domain, and then looks at the message, which it either displays to the user, or responds to by sending a message back to the document which sent the message in the first place.

***Note: Implementors are urged to take extra care in the implementation of this feature. It allows authors to transmit information from one domain to another domain, which is normally disallowed for security reasons. It also requires that UAs be careful to allow access to certain properties but not others.***

# 7. Miscellaneous APIs

## 7.1. [TBW] Repetition templates

See WF2 for now

## 7.2. [SCS] Sound

This section is about to be collapsed into the new `audio` **(page 363)** element section.

We need to add an API for object to support pausing, etc, of streaming APIs.

The `Audio` **(page 363)** interface allows scripts to play sound clips. This interface is intended for sound effects, not for streaming audio or multimedia; for the latter, the `object` **(page 153)** element is more appropriate.

There is no markup element that corresponds to `Audio` **(page 363)** objects, they are only accessible from script.

User agents should allow users to dynamically enable and disable sound output, but doing so must not affect how `Audio` **(page 363)** objects act in any way other than whether sounds are physically played back or not. For instance, sound files must still be downloaded, `load` and `error` events must still fire, and if two identical clips are started with a two second interval then when the sound is reenabled they must still be two seconds out of sync.

When multiple sounds are played simultaneously, the user agent must mix the sounds together.

```
interface Audio {
           attribute EventListener onload;
           attribute EventListener onerror;
  void play();
  void loop();
  void loop(in unsigned long playCount);
  void stop();
};
```

`Audio` **(page 363)** objects must also implement the `EventTarget` interface. [DOM3EVENTS]

In ECMAScript, an instance of `Audio` **(page 363)** can be created using the `Audio(`*`uri`*`)` constructor:

```
var a = new Audio("test.wav");
```

The **`Audio()` constructor** takes a single argument, a URI (or IRI), which is resolved using the script context's `window.location.href` value as the base, and which returns an `Audio` **(page 363)** object that will, at the completion of the current script, start loading that URI.

Once the URI is loaded, a `load` event must be fired on the `Audio` **(page 363)** object.

`Audio` **(page 363)** objects have a current position and a play count. Both are initially zero.

The `Audio` **(page 363)** interface has the following members:

**onload**
: An event listener that is invoked along with any other appropriate event listeners that are registered on this object when a `load` event is fired on it.

**play()**
: Begins playing the sound at the current position, setting the play count to 1.

**loop()**
: Begins playing the sound at the current position, setting the play count to infinity.

**loop(*playCount*)**
: Begins playing the sound at the current position, setting the play count to *playCount*.

**stop()**
: Stops playing the clip and resets the current position and play count to zero.

When playback of the sound reaches the end of the available data, its current position is reset to the start of the clip, and the play count is decreased by one (unless it is infinite). If the play count is greater than zero, then the sound is played again.

# 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 **(page 373)** and space characters **(page 47)**.
3. A DOCTYPE **(page 365)**.
4. Any number of comments **(page 373)** and space characters **(page 47)**.
5. The root element, in the form of an `html` **(page 79)** element **(page 366)**.
6. Any number of comments **(page 373)** and space characters **(page 47)**.

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 **(page 90)** are to be serialised, as discussed in the section on that topic.

### 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 **(page 47)**.
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 **(page 47)**.
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` **(page 81)**, `link` **(page 82)**, `meta` **(page 86)**, `hr` **(page 109)**, `br` **(page 110)**, `img` **(page 148)**, `embed` **(page 151)**, `param` **(page 157)**, `area` **(page 186)**, `col` **(page 196)**, `input`

**CDATA elements**

`style` **(page 91)**, `script` **(page 210)**

**RCDATA elements**

`title` **(page 80)**, `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 **(page 367)** to indicate where they begin, and an end tag **(page 368)** to indicate where they end. The start and end tags of certain normal elements can be omitted **(page 369)**, 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 **(page 369)**) and just before the end tag (which again, might be implied in certain cases **(page 369)**). 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 **(page 372)**, but the text must not contain the two character sequence "`</`" (U+003C LESS-THAN SIGN, U+002F SOLIDUS).

RCDATA elements can have text **(page 372)** and character entity references **(page 372)**, but the text must not contain the character U+003C LESS-THAN SIGN (`<`) or the character U+0026 AMPERSAND (`&`).

Normal elements can have text **(page 372)**, character entity references **(page 372)**, other elements **(page 366)**, and comments **(page 373)**, but the text must not contain the character U+003C LESS-THAN SIGN (`<`) or the character U+0026 AMPERSAND (`&`). Some normal elements also have yet more restrictions **(page 371)** 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 **(page 367)**.
3. If there are to be any attributes in the next step, there must first be one or more space characters **(page 47)**.
4. Then, the start tag may have a number of attributes, the syntax for which **(page 368)** is described below. Attributes may be separated from each other by one or more space characters **(page 47)**.
5. After the attributes, there may be one or more space characters **(page 47)**. (Some attributes are required to be followed by a space. See the attributes section **(page 368)** 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 **(page 367)**.

4. After the tag name, there may be one or more space characters **(page 47)**.

5. Finally, end tags must be closed by a U+003E GREATER-THAN SIGN (`>`) character.

*8.1.2.3. Attributes*

**Attributes** for an element are expressed inside the element's start tag.

Attributes have a name and a value. **Attribute names** use characters in the range U+0061 LATIN SMALL LETTER A .. U+007A LATIN SMALL LETTER Z, or, in uppercase, U+0041 LATIN CAPITAL LETTER A .. U+005A LATIN CAPITAL LETTER Z, and U+002D HYPHEN-MINUS (`-`). In the HTML syntax, attribute names may be written with any mix of lower- and uppercase letters that, when converted to all-lowercase, matches the attribute's name; attribute names are case-insensitive.

**Attribute values** are a mixture of text **(page 372)** and character entity references **(page 372)**, except with the additional restriction that the text cannot contain a U+0026 AMPERSAND (`&`) character.

Attributes can be specified in four different ways:

**Empty attribute syntax**

Just the attribute name **(page 368)**.

> 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 **(page 47)** separating the two.

**Unquoted attribute value syntax**

The attribute name **(page 368)**, followed by zero or more space characters **(page 47)**, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters **(page 47)**, followed by the attribute value **(page 368)**, which, in addition to the requirements given above for attribute values, must not contain any literal space characters **(page 47)**, U+003E GREATER-THAN SIGN (>) characters, or U+003C LESS-THAN SIGN (<) characters, and must not, furthermore, start with either a literal U+0022 QUOTATION MARK (`"`) character or a literal U+0027 APOSTROPHE (`'`) character.

> In the following example, the `value` attribute is given with the unquoted attribute value syntax:
>
> ```
> <input value=yes>
> ```

If an attribute using the unquoted attribute syntax is to be followed by another attribute or by one of the optional U+002F SOLIDUS (/) characters allowed in step 6 of the start tag syntax above, then there must be a space character **(page 47)** separating the two.

**Single-quoted attribute value syntax**

The attribute name **(page 368)**, followed by zero or more space characters **(page 47)**, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters **(page 47)**, followed by a single U+0027 APOSTROPHE (`'`) character, followed by the attribute value **(page 368)**, which, in addition to the requirements given above for attribute values, must not contain any literal U+0027 APOSTROPHE (`'`) characters, and finally followed by a second single U+0027 APOSTROPHE (`'`) character.

> In the following example, the `type` attribute is given with the single-quoted attribute value syntax:
>
> ```
> <input type='checkbox'>
> ```

**Double-quoted attribute value syntax**

The attribute name **(page 368)**, followed by zero or more space characters **(page 47)**, followed by a single U+003D EQUALS SIGN character, followed by zero or more space characters **(page 47)**, followed by a single U+0022 QUOTATION MARK (`"`) character, followed by the attribute value **(page 368)**, 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` **(page 79)** element's start tag may be omitted if the first thing inside the `html` **(page 79)** element is not a space character **(page 47)** or a comment **(page 373)**.

An `html` **(page 79)** element's end tag may be omitted if the `html` **(page 79)** element is not immediately followed by a space character **(page 47)** or a comment **(page 373)**.

A `head` **(page 80)** element's start tag may be omitted if the first thing inside the `head` **(page 80)** element is an element.

A `head` **(page 80)** element's end tag may be omitted if the `head` **(page 80)** element is not immediately followed by a space character **(page 47)** or a comment **(page 373)**.

A `body` **(page 94)** element's start tag may be omitted if the first thing inside the `body` **(page 94)** element is not a space character **(page 47)** or a comment **(page 373)**, except if the first thing inside the `body` **(page 94)** element is a `script` **(page 210)** or `style` **(page 91)** element and the node immediately preceding the `body` **(page 94)** element is a `head` **(page 80)** element whose end tag has been omitted.

A `body` **(page 94)** element's end tag may be omitted if the `body` **(page 94)** element is not immediately followed by a space character **(page 47)** or a comment **(page 373)**.

A `li` **(page 114)** element's end tag may be omitted if the `li` **(page 114)** element is immediately followed by another `li` **(page 114)** element or if there is no more content in the parent element.

A `dt` **(page 117)** element's end tag may be omitted if the `dt` **(page 117)** element is immediately followed by another `dt` **(page 117)** element or a `dd` **(page 117)** element.

A `dd` **(page 117)** element's end tag may be omitted if the `dd` **(page 117)** element is immediately followed by another `dd` **(page 117)** element or a `dt` **(page 117)** element, or if there is no more content in the parent element.

A `p` **(page 108)** element's end tag may be omitted if the `p` **(page 108)** element is immediately followed by an `address` **(page 101)**, `blockquote` **(page 96)**, `dl` **(page 115)**, `fieldset`, `form`, `h1` **(page 98)**, `h2` **(page 98)**, `h3` **(page 98)**, `h4` **(page 98)**, `h5` **(page 98)**, `h6` **(page 98)**, `hr` **(page 109)**, `menu` **(page 245)**, `ol` **(page 112)**, `p` **(page 108)**, `pre` **(page 112)**, `table` **(page 192)**, or `ul` **(page 113)** 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` **(page 195)** element's start tag may be omitted if the first thing inside the `colgroup` **(page 195)** element is a `col` **(page 196)** element, and if the element is not immediately preceeded by another `colgroup` **(page 195)** element whose end tag has been omitted.

A `colgroup` **(page 195)** element's end tag may be omitted if the `colgroup` **(page 195)** element is not immediately followed by a space character **(page 47)** or a comment **(page 373)**.

A `thead` **(page 198)** element's end tag may be omitted if the `thead` **(page 198)** element is immediately followed by a `tbody` **(page 196)** or `tfoot` **(page 198)** element.

A `tbody` **(page 196)** element's start tag may be omitted if the first thing inside the `tbody` **(page 196)** element is a `tr` **(page 199)** element, and if the element is not immediately preceeded by a `tbody` **(page 196)**, `thead` **(page 198)**, or `tfoot` **(page 198)** element whose end tag has been omitted.

A `tbody` **(page 196)** element's end tag may be omitted if the `tbody` **(page 196)** element is immediately followed by a `tbody` **(page 196)** or `tfoot` **(page 198)** element, or if there is no more content in the parent element.

A `tfoot` **(page 198)** element's end tag may be omitted if the `tfoot` **(page 198)** element is immediately followed by a `tbody` **(page 196)** element, or if there is no more content in the parent element.

A `tr` **(page 199)** element's end tag may be omitted if the `tr` **(page 199)** element is immediately followed by another `tr` **(page 199)** element, or if there is no more content in the parent element.

A `td` **(page 200)** element's end tag may be omitted if the `td` **(page 200)** element is immediately followed by a `td` **(page 200)** or `th` **(page 201)** element, or if there is no more content in the parent element.

A `th` **(page 201)** element's end tag may be omitted if the `th` **(page 201)** element is immediately followed by a `td` **(page 200)** or `th` **(page 201)** 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` **(page 108)** element must not contain `blockquote` **(page 96)**, `dl` **(page 115)**, `menu` **(page 245)**, `ol` **(page 112)**, `pre` **(page 112)**, `table` **(page 192)**, or `ul` **(page 113)** elements, even though these elements are technically allowed inside `p` **(page 108)** elements according to the content models described in this specification. (In fact, if one of those elements is put inside a `p` **(page 108)** element in the markup, it will instead imply a `p` **(page 108)** 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` **(page 192)** element must not contain `tr` **(page 199)** elements, even though these elements are technically allowed inside `table` **(page 192)** elements according to the content models described in this specification. (If a `tr` **(page 199)** element is put inside a `table` **(page 192)** in the markup, it will in fact imply a `tbody` **(page 196)** start tag before it.)

A single U+000A LINE FEED (LF) character may be placed immediately after a `pre` **(page 112)** element's start tag. 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` **(page 112)** blocks are equivalent:
>
> ```
> <pre>Hello</pre>
> <pre>
> Hello</pre>
> ```

### 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 **(page 47)**. 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 **(page 372)** may be mixed with **character entity references**. These can be used to escape characters that couldn't otherwise legally be included in text **(page 372)**.

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 **(page 434)** section, using the same case. Finally, after the name, the entity must be terminated by a U+003B SEMICOLON character (`;`).

**Decimal numeric entities**
: The ampersand must be followed by a U+0023 NUMBER SIGN (`#`) character, followed by one or more digits in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, representing a base-ten integer that itself is a valid Unicode code point that isn't U+0000. The digits must then be followed by a U+003B SEMICOLON character (`;`).

**Hexadecimal numeric entities**

The ampersand must be followed by a U+0023 NUMBER SIGN (`#`) character, which must be followed by either a U+0078 LATIN SMALL LETTER X or a U+0058 LATIN CAPITAL LETTER X character, which must then be followed by one or more digits in the range U+0030 DIGIT ZERO .. U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A .. U+0066 LATIN SMALL LETTER F, and U+0041 LATIN CAPITAL LETTER A .. U+0046 LATIN CAPITAL LETTER F, representing a base-sixteen integer that itself is a valid Unicode code point that isn't U+0000. The digits must then be followed by a U+003B SEMICOLON character (`;`).

### 8.1.5. Comments

**Comments** must start with the four character sequence U+003C LESS-THAN SIGN, U+0021 EXCLAMATION MARK, U+002D HYPHEN-MINUS, U+002D HYPHEN-MINUS (`<!--`). Following this sequence, the comment may have text **(page 372)**, 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. [SCS] Parsing HTML documents

*This section only applies to user agents, data mining tools, and conformance checkers.*

The rules for parsing XML documents **(page 25)** (and thus XHTML **(page 18)** 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 **(page 25)**, user agents must use the parsing rules described in this section to generate the DOM trees. Together, these rules define what is referred to as the **HTML parser**.

***While the HTML form of HTML5 bears a close resemblance to SGML and XML, it is a separate language with its own parsing rules.***

***Some earlier versions of HTML (in particular from HTML2 to HTML4) were based on SGML and used SGML parsing rules. However, few (if any) web browsers ever implemented true SGML parsing for HTML documents; the only user agents to strictly handle HTML as an SGML application have historically been validators. The resulting confusion — with validators claiming documents to have one representation while widely deployed Web browsers interoperably implemented a different representation — has resulted in this version of HTML returning to a non-SGML basis.***

***Authors interested in using SGML tools in their authoring pipeline are encouraged to use the XML serialisation of HTML5 instead of the HTML serialisation.***

This specification defines the parsing rules for HTML documents, whether they are syntactically valid or not. Certain points in the parsing algorithm are said to be **parse errors**. The error handling for parse errors is well-defined: user agents must either act as described below when encountering such problems, or must abort processing at the first error that they encounter for which they do not wish to apply the rules described below.

Conformance checkers must report at least one parse error condition to the user if one or more parse error conditions exist in the document and must not report parse error conditions if none exist in the document. Conformance checkers may report more than one parse error condition if more than one parse error conditions exist in the document. Conformance checkers are not required to recover from parse errors.

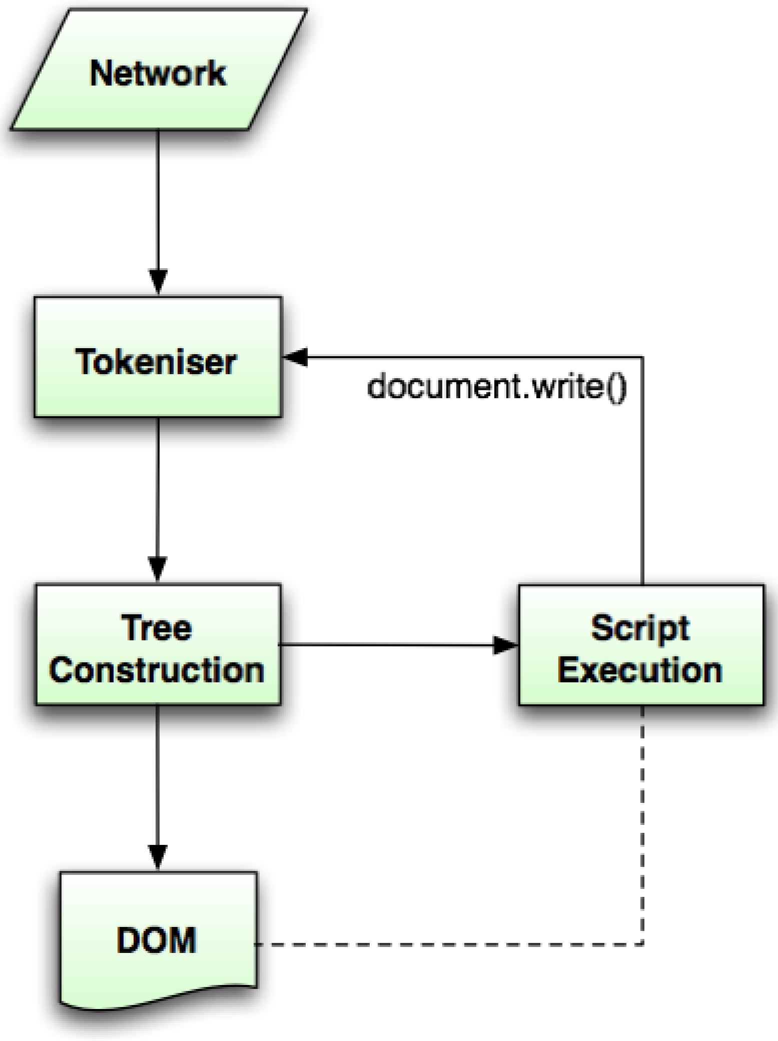
> ***Note: Parse errors are only errors with the syntax of HTML. In addition to checking for parse errors, conformance checkers will also verify that the document obeys all the other conformance requirements described in this specification.***

### 8.2.1. Overview of the parsing model

The input to the HTML parsing process consists of a stream of Unicode characters, which is passed through a tokenisation **(page 382)** stage (lexical analysis) followed by a tree construction **(page 394)** stage (semantic analysis). The output is a `Document` object.

> ***Note: Implementations that do not support scripting*** **(page 16)** ***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 **(page 37)**, e.g. using the `document.write()` **(page 39)** API.



There is only one set of state for the tokeniser stage and the tree construction stage, but the tree construction stage is reentrant, meaning that while the tree construction stage is handling one token, the tokeniser might be resumed, causing further tokens to be emitted and processed before the first token's processing is complete.

> In the following example, the tree construction stage will be called upon to handle a "p" start tag token while handling the "script" start tag token:
>
> ```
> ...
> <script>
>  document.write('<p>');
> </script>
> ...
> ```

### 8.2.2. The input stream

The stream of Unicode characters that consists the input to the tokenisation stage will be initially seen by the user agent as a stream of bytes (typically coming over the network or from the local file system). The bytes encode the actual characters according to a particular *character encoding*, which the user agent must use to decode the bytes into characters.

For HTML, user agents must use the following algorithm in determining the character encoding of a document:

1. If the transport layer specifies an encoding, use that, and abort these steps.
2. The user agent may wait for 512 or more bytes of the resource to be available.
3. Let *n* be the smaller of either 512 or the number of bytes already available.
4. For each of the rows in the following table, starting with the first one and going down, if *n* is equal to or greater 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 use the encoding given in the cell in the second column of that row, and abort these steps:

| Bytes in Hexadecimal | Description |
|---|---|
| 00 00 FE FF | UTF-32BE BOM |
| FF FE 00 00 | UTF-32LE BOM |
| FE FF | UTF-16BE BOM |
| FF FE | UTF-16LE BOM |
| EF BB BF | UTF-8 BOM |

5. Otherwise, the user agent will have to search for explicit character encoding information in the file itself. This must 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 steps the *position* pointer points beyond the *n*th byte of the input stream, then skip to the next step of the overall character encoding detection algorithm (the step which mentions frequency analysis below).

   Now, repeat the following "two" steps until the algorithm aborts (either because *position* reaches beyond the *n*th byte, 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 preceeded by two 0x2D bytes (i.e. at the end of an ASCII '-->' sequence) and comes after the second 0x2D byte that was found. (The two 0x2D bytes cannot be the same as the those in the '<!--' sequence.) If no such byte is found before the *n*th byte, abort this "two step" algorithm.

      ↪ **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), if there is one before the *n*th byte. If there isn't, abort the "two step" algorithm.

2. Get an attribute **(page 379)** 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.

   ***Note: As required above, if the position pointer points beyond the nth byte after the "get an attribute" step, the "two step" algorithm will abort.***

3. Examine the attribute's name:

   ↪ **If it is 'charset'**
   If the attribute's value is a supported character encoding, then use the given encoding, and abort all these steps. Otherwise, do nothing with this attribute.

   ↪ **If it is 'content'**
   The attribute's value is now parsed.

   1. Skip characters in the attribute's value 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', abort this very inner set of steps (parsing the attribute's value), and continue looking for other attributes.

   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 ('='), abort this very inner set of steps (parsing the attribute's value), and continue looking for other attributes.

   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 the attribute's value**
: Let *tentative encoding* be the string between the two quotation marks.

↪ **If it is a U+0027 APOSTROPHE (''') and there is a later U+0027 APOSTROPHE (''') in the attribute's value**
: Let *tentative encoding* be the string between the two apostrophes.

↪ **If it is an unmatched U+0022 QUOTATION MARK ("")**
↪ **If it is an unmatched U+0027 APOSTROPHE (''')**
: There is no *tentative encoding*.

↪ **Otherwise**
: Let *tentative encoding* be 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 the attribute's value, whichever comes first.

8. If there is a *tentative encoding* and it is the name of a supported character encoding, then use that encoding; abort all these steps.
9. 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, if there is one before the *n*th byte. If there isn't, abort the "two step" algorithm.
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 **(page 379)** 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 0x2D (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. If no such byte is found before the *n*th byte, abort this "two step" algorithm.

- ↪ **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.

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 use that; abort these steps. [UNIVCHARDET]

7. Otherwise, use an implementation-defined or user-specified default character encoding. 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.

***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]***

When a user agent would otherwise use the ISO-8859-1 encoding, it must instead use the Windows-1252 encoding. User agents must not support the CESU-8, UTF-7, BOCU-1 and SCSU encodings. [CESU8] [UTF7] [BOCU1] [SCSU]

Bytes or sequences of bytes in the original byte stream that could not be converted to Unicode characters must be converted to U+FFFD REPLACEMENT CHARACTER code points.

A leading U+FEFF BYTE ORDER MARK (BOM) must be dropped if present.

All U+0000 NULL characters in the input must be replaced by U+FFFD REPLACEMENT CHARACTERs.

U+000D CARRIAGE RETURN (CR) characters, and U+000A LINE FEED (LF) characters, are treated specially. Any CR characters that are followed by LF characters must be removed, and any CR characters not followed by LF characters must be converted to LF characters. Thus, newlines in HTML DOMs are represented by LF characters, and there are never any CR characters in the input to the tokenisation **(page 382)** stage.

The **next input character** is the first character in the input stream that has not yet been **consumed**. Initially, the *next input character* **(page 382)** 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()` **(page 39)** 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 **(page 375)**. If the parser is a script-created parser **(page 38)**, then the end of the input stream **(page 375)** is reached when an **explicit "EOF" character** (inserted by the `document.close()` **(page 39)** method) is consumed. Otherwise, the "EOF" character is not a real character in the stream, but rather the lack of any further characters.

### 8.2.3. Tokenisation

Implementations must act as if they used the following state machine to tokenise HTML. The state machine must start in the data state **(page 382)**. Most states consume a single character, which may have various side-effects, and either switches the state machine to a new state to *reconsume* the same character, or switches it to a new state (to consume the next character), or repeats the same state (to consume the next character). Some states have more complicated behaviour and can consume several characters before switching to another state.

The exact behaviour of certain states depends on a **content model flag** that is set after certain tokens are emitted. The flag has several states: *PCDATA*, *RCDATA*, *CDATA*, and *PLAINTEXT*. Initially it is in the PCDATA state.

The output of the tokenisation step is a series of zero or more of the following tokens: DOCTYPE, start tag, end tag, comment, character, end-of-file. DOCTYPE tokens have names and can be either correct or in error. Start and end tag tokens have a tagname and a list of attributes, each of which has a name and a value. Comment and character tokens have data.

When a token is emitted, it must immediately be handled by the tree construction **(page 394)** stage. The tree construction stage can affect the state of the content model flag **(page 382)**, and can insert additional characters into the stream. (For example, the `script` **(page 210)** element can result in scripts executing and using the dynamic markup insertion **(page 37)** APIs to insert characters into the stream being tokenised.)

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 **(page 214)** or the first script in the list of scripts that will execute asynchronously **(page 213)**, has completed loading. If one has, then it must be executed **(page 214)** and removed from its list.

The tokeniser state machine is as follows:

**Data state**

Consume the next input character **(page 382)**:

↪ **U+0026 AMPERSAND (&)**
  When the content model flag **(page 382)** is set to one of the PCDATA or RCDATA states: switch to the entity data state **(page 383)**.
  Otherwise: treat it as per the "anything else" entry below.

↪ **U+003C LESS-THAN SIGN (<)**
  When the content model flag **(page 382)** is set to a state other than the PLAINTEXT state: switch to the tag open state **(page 383)**.
  Otherwise: treat it as per the "anything else" entry below.

↪ **EOF**
  Emit an end-of-file token.

↪ **Anything else**
  Emit the input character as a character token. Stay in the data state **(page 382)**.

**Entity data state**

*(This cannot happen if the content model flag* **(page 382)** *is set to the CDATA state.)*

Attempt to consume an entity **(page 392)**.

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 **(page 382)**.

**Tag open state**

The behaviour of this state depends on the content model flag **(page 382)**.

**If the content model flag (page 382) is set to the RCDATA or CDATA states**
If the next input character **(page 382)** is a U+002F SOLIDUS (/) character, consume it and switch to the close tag open state **(page 384)**. If the next input character **(page 382)** is not a U+002F SOLIDUS (/) character, emit a U+003C LESS-THAN SIGN character token and reconsume the current input character in the data state **(page 382)**.

**If the content model flag (page 382) is set to the PCDATA state**
Consume the next input character **(page 382)**:

↪ **U+0021 EXCLAMATION MARK (!)**
  Switch to the markup declaration open state **(page 389)**.

↪ **U+002F SOLIDUS (/)**
  Switch to the close tag open state **(page 384)**.

↪ **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 **(page 385)**. (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 **(page 385)**. (Don't

emit the token yet; further details will be filled in before it is emitted.)

↪ **U+003E GREATER-THAN SIGN (>)**
Parse error **(page 374)**. Emit a U+003C LESS-THAN SIGN character token and a U+003E GREATER-THAN SIGN character token. Switch to the data state **(page 382)**.

↪ **U+003F QUESTION MARK (?)**
Parse error **(page 374)**. Switch to the bogus comment state **(page 389)**.

↪ **Anything else**
Parse error **(page 374)**. Emit a U+003C LESS-THAN SIGN character token and reconsume the current input character in the data state **(page 382)**.

**Close tag open state**

If the content model flag **(page 382)** is set to the RCDATA or CDATA states then examine the next few characters. If they do not match the tag name of the last start tag token emitted (case insensitively), or if they do but they are not immediately followed by one of the following characters:

- U+0009 CHARACTER TABULATION
- U+000A LINE FEED (LF)
- U+000B LINE TABULATION
- U+000C FORM FEED (FF)
- U+0020 SPACE
- U+003E GREATER-THAN SIGN (>)
- U+002F SOLIDUS (/)
- U+003C LESS-THAN SIGN (<)
- EOF

...then there is a parse error **(page 374)**. Emit a U+003C LESS-THAN SIGN character token, a U+002F SOLIDUS character token, and reconsume the current input character in the data state **(page 382)**.

Otherwise, if the content model flag **(page 382)** is set to the PCDATA state, or if the next few characters *do* match that tag name, consume the next input character **(page 382)**:

↪ **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 **(page 385)**. (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 **(page 385)**. (Don't emit the token yet; further details will be filled in before it is emitted.)

↪ **U+003E GREATER-THAN SIGN (>)**
Parse error **(page 374)**. Switch to the data state **(page 382)**.

↪ **EOF**

Parse error **(page 374)**. Emit a U+003C LESS-THAN SIGN character token and a U+002F SOLIDUS character token. Reconsume the EOF character in the data state **(page 382)**.

↪ **Anything else**

Parse error **(page 374)**. Switch to the bogus comment state **(page 389)**.

**Tag name state**

Consume the next input character **(page 382)**:

↪ **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 **(page 385)**.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state **(page 382)**.

↪ **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 **(page 385)**.

↪ **U+003C LESS-THAN SIGN (<)**

↪ **EOF**

Parse error **(page 374)**. Emit the current tag token. Reconsume the character in the data state **(page 382)**.

↪ **U+002F SOLIDUS (/)**

Parse error **(page 374)** unless this is a permitted slash **(page 392)**. Switch to the before attribute name state **(page 385)**.

↪ **Anything else**

Append the current input character to the current tag token's tag name. Stay in the tag name state **(page 385)**.

**Before attribute name state**

Consume the next input character **(page 382)**:

↪ **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 **(page 385)**.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state **(page 382)**.

↪ **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 **(page 386)**.

↪ **U+002F SOLIDUS (/)**
: Parse error **(page 374)** unless this is a permitted slash **(page 392)**. Stay in the before attribute name state **(page 385)**.

↪ **U+003C LESS-THAN SIGN (<)**

↪ **EOF**
: Parse error **(page 374)**. Emit the current tag token. Reconsume the character in the data state **(page 382)**.

↪ **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 **(page 386)**.

**Attribute name state**

Consume the next input character **(page 382)**:

↪ **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 **(page 387)**.

↪ **U+003D EQUALS SIGN (=)**
: Switch to the before attribute value state **(page 387)**.

↪ **U+003E GREATER-THAN SIGN (>)**
: Emit the current tag token. Switch to the data state **(page 382)**.

↪ **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 **(page 386)**.

↪ **U+002F SOLIDUS (/)**
: Parse error **(page 374)** unless this is a permitted slash **(page 392)**. Switch to the before attribute name state **(page 385)**.

↪ **U+003C LESS-THAN SIGN (<)**

↪ **EOF**
: Parse error **(page 374)**. Emit the current tag token. Reconsume the character in the data state **(page 382)**.

↪ **Anything else**
: Append the current input character to the current attribute's name. Stay in the attribute name state **(page 386)**.

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 **(page 374)** 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 **(page 382)**:

↪ **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 **(page 387)**.

↪ **U+003D EQUALS SIGN (=)**

Switch to the before attribute value state **(page 387)**.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state **(page 382)**.

↪ **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 **(page 386)**.

↪ **U+002F SOLIDUS (/)**

Parse error **(page 374)** unless this is a permitted slash **(page 392)**. Switch to the before attribute name state **(page 385)**.

↪ **U+003C LESS-THAN SIGN (<)**

↪ **EOF**

Parse error **(page 374)**. Emit the current tag token. Reconsume the character in the data state **(page 382)**.

↪ **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 **(page 386)**.

**Before attribute value state**

Consume the next input character **(page 382)**:

↪ **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 **(page 387)**.

↪ **U+0022 QUOTATION MARK (")**

Switch to the attribute value (double-quoted) state **(page 388)**.

↪ **U+0026 AMPERSAND (&)**

Switch to the attribute value (unquoted) state **(page 388)** and reconsume this input character.

↪ **U+0027 APOSTROPHE (')**

Switch to the attribute value (single-quoted) state **(page 388)**.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state **(page 382)**.

↪ **U+003C LESS-THAN SIGN (<)**

↪ **EOF**

Parse error **(page 374)**. Emit the current tag token. Reconsume the character in the data state **(page 382)**.

↪ **Anything else**

Append the current input character to the current attribute's value. Switch to the attribute value (unquoted) state **(page 388)**.

**Attribute value (double-quoted) state**

Consume the next input character **(page 382)**:

↪ **U+0022 QUOTATION MARK (")**

Switch to the before attribute name state **(page 385)**.

↪ **U+0026 AMPERSAND (&)**

Switch to the entity in attribute value state **(page 389)**.

↪ **EOF**

Parse error **(page 374)**. Emit the current tag token. Reconsume the character in the data state **(page 382)**.

↪ **Anything else**

Append the current input character to the current attribute's value. Stay in the attribute value (double-quoted) state **(page 388)**.

**Attribute value (single-quoted) state**

Consume the next input character **(page 382)**:

↪ **U+0027 APOSTROPHE (')**

Switch to the before attribute name state **(page 385)**.

↪ **U+0026 AMPERSAND (&)**

Switch to the entity in attribute value state **(page 389)**.

↪ **EOF**

Parse error **(page 374)**. Emit the current tag token. Reconsume the character in the data state **(page 382)**.

↪ **Anything else**

Append the current input character to the current attribute's value. Stay in the attribute value (single-quoted) state **(page 388)**.

**Attribute value (unquoted) state**

Consume the next input character **(page 382)**:

↪ **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 **(page 385)**.

↪ **U+0026 AMPERSAND (&)**

Switch to the entity in attribute value state **(page 389)**.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current tag token. Switch to the data state **(page 382)**.

↪ **U+003C LESS-THAN SIGN (<)**

↪ **EOF**

Parse error **(page 374)**. Emit the current tag token. Reconsume the character in the data state **(page 382)**.

↪ **Anything else**

Append the current input character to the current attribute's value. Stay in the attribute value (unquoted) state **(page 388)**.

**Entity in attribute value state**

Attempt to consume an entity **(page 392)**.

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* **(page 382)** *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 **(page 382)**.

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* **(page 382)** *is set to the PCDATA state.)*

If the next two characters are both U+002D HYPHEN-MINUS (-) characters, consume those two characters, create a comment token whose data is the empty string, and switch to the comment state **(page 389)**.

Otherwise if the next seven chacacters are a case-insensitive match for the word "DOCTYPE", then consume those characters and switch to the DOCTYPE state **(page 390)**.

Otherwise, is is a parse error **(page 374)**. Switch to the bogus comment state **(page 389)**. The next character that is consumed, if any, is the first character that will be in the comment.

**Comment state**

Consume the next input character **(page 382)**:

↪ **U+002D HYPHEN-MINUS (-)**
Switch to the comment dash state **(page 390)**

↪ **EOF**
Parse error **(page 374)**. Emit the comment token. Reconsume the EOF character in the data state **(page 382)**.

↪ **Anything else**
Append the input character to the comment token's data. Stay in the comment state **(page 389)**.

**Comment dash state**

Consume the next input character **(page 382)**:

↪ **U+002D HYPHEN-MINUS (-)**
Switch to the comment end state **(page 390)**

↪ **EOF**
Parse error **(page 374)**. Emit the comment token. Reconsume the EOF character in the data state **(page 382)**.

↪ **Anything else**
Append a U+002D HYPHEN-MINUS (-) character and the input character to the comment token's data. Switch to the comment state **(page 389)**.

**Comment end state**

Consume the next input character **(page 382)**:

↪ **U+003E GREATER-THAN SIGN (>)**
Emit the comment token. Switch to the data state **(page 382)**.

↪ **U+002D HYPHEN-MINUS (-)**
Parse error **(page 374)**. Append a U+002D HYPHEN-MINUS (-) character to the comment token's data. Stay in the comment end state **(page 390)**.

↪ **EOF**
Parse error **(page 374)**. Emit the comment token. Reconsume the EOF character in the data state **(page 382)**.

↪ **Anything else**
Parse error **(page 374)**. Append two U+002D HYPHEN-MINUS (-) characters and the input character to the comment token's data. Switch to the comment state **(page 389)**.

**DOCTYPE state**

Consume the next input character **(page 382)**:

↪ **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 **(page 391)**.

↪ **Anything else**
Parse error **(page 374)**. Reconsume the current character in the before DOCTYPE name state **(page 391)**.

**Before DOCTYPE name state**

Consume the next input character **(page 382)**:

↪ **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 **(page 391)**.

↪ **U+0061 LATIN SMALL LETTER A through to U+007A LATIN SMALL LETTER Z**

Create a new DOCTYPE token. Set the token's name name to the uppercase version of the current input character (subtract 0x0020 from the character's code point), and mark it as being in error. Switch to the DOCTYPE name state **(page 391)**.

↪ **U+003E GREATER-THAN SIGN (>)**

Parse error **(page 374)**. Emit a DOCTYPE token whose name is the empty string and that is marked as being in error. Switch to the data state **(page 382)**.

↪ **EOF**

Parse error **(page 374)**. Emit a DOCTYPE token whose name is the empty string and that is marked as being in error. Reconsume the EOF character in the data state **(page 382)**.

↪ **Anything else**

Create a new DOCTYPE token. Set the token's name name to the current input character, and mark it as being in error. Switch to the DOCTYPE name state **(page 391)**.

**DOCTYPE name state**

First, consume the next input character **(page 382)**:

↪ **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 **(page 392)**.

↪ **U+003E GREATER-THAN SIGN (>)**

Emit the current DOCTYPE token. Switch to the data state **(page 382)**.

↪ **U+0061 LATIN SMALL LETTER A through to U+007A LATIN SMALL LETTER Z**

Append the uppercase version of the current input character (subtract 0x0020 from the character's code point) to the current DOCTYPE token's name. Stay in the DOCTYPE name state **(page 391)**.

↪ **EOF**

Parse error **(page 374)**. Emit the current DOCTYPE token. Reconsume the EOF character in the data state **(page 382)**.

↪ **Anything else**

Append the current input character to the current DOCTYPE token's name. Stay in the DOCTYPE name state **(page 391)**.

Then, if the name of the DOCTYPE token is exactly the four letters "HTML", then mark the token as being correct. Otherwise, mark it as being in error.

***Note: Because lowercase letters in the name are uppercased by the algorithm above, the "HTML" letters are actually case-insensitive relative to the markup.***

**After DOCTYPE name state**

Consume the next input character **(page 382)**:

↪ **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 **(page 392)**.

↪ **U+003E GREATER-THAN SIGN (>)**
Emit the current DOCTYPE token. Switch to the data state **(page 382)**.

↪ **EOF**
Parse error **(page 374)**. Emit the current DOCTYPE token. Reconsume the EOF character in the data state **(page 382)**.

↪ **Anything else**
Parse error **(page 374)**. Mark the DOCTYPE token as being in error, if it is not already. Switch to the bogus DOCTYPE state **(page 392)**.

**Bogus DOCTYPE state**

Consume the next input character **(page 382)**:

↪ **U+003E GREATER-THAN SIGN (>)**
Emit the current DOCTYPE token. Switch to the data state **(page 382)**.

↪ **EOF**
Parse error **(page 374)**. Emit the current DOCTYPE token. Reconsume the EOF character in the data state **(page 382)**.

↪ **Anything else**
Stay in the bogus DOCTYPE state **(page 392)**.

When an end tag token is emitted, the content model flag **(page 382)** must be switched to the PCDATA state.

When an end tag token is emitted with attributes, that is a parse error **(page 374)**.

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` **(page 81)**, `link` **(page 82)**, `meta` **(page 86)**, `hr` **(page 109)**, `br` **(page 110)**, `img` **(page 148)**, `embed` **(page 151)**, `param` **(page 157)**, `area` **(page 186)**, `col` **(page 196)**, `input`

### *8.2.3.1. Tokenising entities*

This section defines how to **consume an entity**. This definition is used when parsing entities in text **(page 383)** and in attributes **(page 389)**.

The behaviour depends on the identity of the next character (the one immediately after the U+0026 AMPERSAND character):

↪ **U+0023 NUMBER SIGN (#)**

Consume the U+0023 NUMBER SIGN.

The behaviour further depends on the character after the U+0023 NUMBER SIGN:

↪ **U+0078 LATIN SMALL LETTER X**

↪ **U+0058 LATIN CAPITAL LETTER X**

Consume the X.

Follow the steps below, but using the range of characters U+0030 DIGIT ZERO through to U+0039 DIGIT NINE, U+0061 LATIN SMALL LETTER A through to U+0078 LATIN SMALL LETTER F, and U+0041 LATIN CAPITAL LETTER A, through to U+0058 LATIN CAPITAL LETTER F (in other words, 0-9, A-F, a-f).

When it comes to interpreting the number, interpret it as a hexadecimal number.

↪ **Anything else**

Follow the steps below, but using the range of characters U+0030 DIGIT ZERO through to U+0039 DIGIT NINE (i.e. just 0-9).

When it comes to interpreting the number, interpret it as a decimal number.

Consume as many characters as match the range of characters given above.

If no characters match the range, then don't consume any characters (and unconsume the U+0023 NUMBER SIGN character and, if appropriate, the X character). This is a parse error **(page 374)**; 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 **(page 374)**.

If one or more characters match the range, then take them all and interpret the string of characters as a number (either hexadecimal or decimal as appropriate), and return a character token for the Unicode character whose code point is that number. If the number is not a valid Unicode character (e.g. if the number is higher than 1114111), or if the number is zero, then return a character token for the U+FFFD REPLACEMENT CHARACTER character instead.

↪ **Anything else**

Consume the maximum number of characters possible, with the consumed characters case-sensitively matching one of the identifiers in the first column of the entities **(page 434)** table.

If no match can be made, then this is a parse error **(page 374)**. No characters are consumed, and nothing is returned.

Otherwise, if the next character is a U+003B SEMICOLON, consume that too. If it isn't, there is a parse error **(page 374)**.

Return a character token for the character corresponding to the entity name (as given by the second column of the entities **(page 434)** table).

> If the markup contains `I'm &notit without you`, the entity is parsed as "not", as in, `I'm ¬it without you`. But if the markup was `I'm &notin without you`, the entity would be parsed as "notin", resulting in `I'm ∉ without you`.

This isn't quite right. For some entities, UAs require a semicolon, for others they don't. We probably need to do the same for backwards compatibility. If we do that we might be able to add more entities, e.g. for mathematics. Probably the way to mark whether or not an entity requires a semicolon is with an additional column in the entity table lower down **(page 434)**.

### 8.2.4. Tree construction

The input to the tree construction stage is a sequence of tokens from the tokenisation **(page 382)** 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 **(page 395)**.

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()` **(page 39)** and `document.writeln()` **(page 37)** 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* (page 18) *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 **(page 382)** stage as follows:

↪ **A DOCTYPE token that is marked as being in error**
↪ **A comment token**
↪ **A start tag token**
↪ **An end tag token**
↪ **A character token that is not one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**
↪ **An end-of-file token**

This specification does not define how to handle this case. In particular, user agents may ignore the entirety of this specification altogether for such documents, and instead invoke special parse modes with a greater emphasis on backwards compatibility.

***Browsers in particular have generally used DOCTYPE-based sniffing to invoke an "alternative conformance mode" known as quirks mode on certain documents. In this mode, emphasis is put on legacy compatibility rather than on standards compliance. This specification takes no position on this behaviour; documents without DOCTYPEs or with DOCTYPEs that do not conform to the syntax allowed by this specification are considered to be out of scope of this specification.***

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

- Comment parsing is different.
- The following is considered one script block (!): `<script><!-- document.write('</script>'); --></script>`
- `</br>` and `</p>` do magical things.
- `p` **(page 108)** can contain `table` **(page 192)**
- Safari and IE have special parsing rules for <% ... %> (even in standards mode, though clearly this should be quirks-only).

Maybe we should just adopt all those and be done with it. One parsing mode to rule them all. Or legitimise/codify the quirks mode parsing in some way.

Would be interesting to do a search to see how many pages hit each of the above.

↪ **A DOCTYPE token marked as being correct**

Append a `DocumentType` node to the `Document` node, with the `name` attribute set to the name given in the DOCTYPE token (which will be

"HTML"), and the other attributes specific to `DocumentType` objects set to null, empty lists, or the empty string as appropriate.

Then, switch to the root element phase **(page 396)** of the tree construction stage.

↪ **A character token that *is* one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append that character **(page 394)** to the `Document` node.

### *8.2.4.2. The root element phase*

After the initial phase **(page 395)**, as each token is emitted from the tokenisation **(page 382)** stage, it must be processed as described in this section.

↪ **A DOCTYPE token**

Parse error **(page 374)**. Ignore the token.

↪ **A comment token**

Append a `Comment` node to the `Document` object with the `data` attribute set to the data given in the comment token.

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append that character **(page 394)** to the `Document` node.

↪ **A character token that is *not* one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**
↪ **A start tag token**
↪ **An end tag token**
↪ **An end-of-file token**

Create an `HTMLElement` **(page 27)** node with the tag name `html` **(page 79)**, in the HTML namespace **(page 434)**. Append it to the `Document` object. Switch to the main phase **(page 396)** and reprocess the current token.

> Should probably make end tags be ignored, so that "</head><!-- --><html>" puts the comment befor the root node (or should we?)

The root element can end up being removed from the `Document` object, e.g. by scripts; nothing in particular happens in such cases, content continues being appended to the nodes as described in the next section.

### *8.2.4.3. The main phase*

After the root element phase **(page 396)**, each token emitted from the tokenisation **(page 382)** 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 **(page 397)**, a list of active formatting elements **(page 398)**, a `head` element pointer **(page 400)**, a `form` element pointer **(page 400)**, and an insertion mode **(page 401)**.

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` **(page 79)** root element node created in the last phase **(page 396)** before switching to *this* phase (or, in the `innerHTML` case **(page 42)**, the `html` **(page 79)** element created to represent the element whose `innerHTML` **(page 39)** attribute is being set). That's the topmost node of the stack. It never gets popped off the stack. (This stack grows downwards.)

The **current node** is the bottommost node in this stack.

Elements in the stack fall into the following categories:

**Special**

The following HTML elements have varying levels of special parsing rules: `address` **(page 101)**, `area` **(page 186)**, `base` **(page 81)**, `basefont`, `bgsound`, `blockquote` **(page 96)**, `body` **(page 94)**, `br` **(page 110)**, `center`, `col` **(page 196)**, `colgroup` **(page 195)**, `dd` **(page 117)**, `dir`, `div` **(page 255)**, `dl` **(page 115)**, `dt` **(page 117)**, `embed` **(page 151)**, `fieldset`, `form`, `frame`, `frameset`, `h1` **(page 98)**, `h2` **(page 98)**, `h3` **(page 98)**, `h4` **(page 98)**, `h5` **(page 98)**, `h6` **(page 98)**, `head` **(page 80)**, `hr` **(page 109)**, `iframe` **(page 150)**, `image`, `img` **(page 148)**, `input`, `isindex`, `li` **(page 114)**, `link` **(page 82)**, `listing`, `menu` **(page 245)**, `meta` **(page 86)**, `noembed`, `noframes`, `noscript` **(page 215)**, `ol` **(page 112)**, `optgroup`, `option`, `p` **(page 108)**, `param` **(page 157)**, `plaintext`, `pre` **(page 112)**, `script` **(page 210)**, `select`, `spacer`, `style` **(page 91)**, `tbody` **(page 196)**, `textarea`, `tfoot` **(page 198)**, `thead` **(page 198)**, `title` **(page 80)**, `tr` **(page 199)**, `ul` **(page 113)**, and `wbr`.

**Scoping**

The following HTML elements introduce new scopes **(page 398)** for various parts of the parsing: `button`, `caption` **(page 194)**, `html` **(page 79)**, `marquee`, `object` **(page 153)**, `table` **(page 192)**, `td` **(page 200)** and `th` **(page 201)**.

**Formatting**

The following HTML elements are those that end up in the list of active formatting elements **(page 398)**: `a` **(page 118)**, `b` **(page 142)**, `big`, `em` **(page 122)**, `font` **(page 437)**, `i` **(page 141)**, `nobr`, `s`, `small` **(page 124)**, `strike`, `strong` **(page 123)**, `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` **(page 217)**, `section` **(page 94)**, `nav` **(page 95)**, `article` **(page 96)**, `aside` **(page 98)**, `header` **(page 99)**, `footer` **(page 100)**, `datagrid` **(page 219)**, `command` **(page 242)**

The stack of open elements **(page 397)** 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 **(page 397)** (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` **(page 192)** 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` **(page 194)**
   - `td` **(page 200)**
   - `th` **(page 201)**
   - `button`
   - `marquee`
   - `object` **(page 153)**
5. Otherwise, if *node* is an `html` **(page 79)** element, terminate in a failure state. (This can only happen if the *node* is the topmost node of the stack of open elements **(page 397)**, and prevents the next step from being invoked if there are no more elements in the stack.)
6. Otherwise, set *node* to the previous entry in the stack of open elements **(page 397)** and return to step 2. (This will never fail, since the loop will always terminate in the previous step if the top of the stack is reached.)

Nothing happens if at any time any of the elements in the stack of open elements **(page 397)** are moved to a new location in, or removed from, the `Document` tree. In particular, the stack is not changed in this situation. This can cause, amongst other strange effects, content to be appended to nodes that are no longer in the DOM.

***Note: In some cases (namely, when closing misnested formatting elements*** **(page 414)*****), the stack is manipulated in a random-access fashion.***

#### 8.2.4.3.2. The list of active formatting elements

Initially the **list of active formatting elements** is empty. It is used to handle mis-nested formatting element tags **(page 397)**.

The list contains elements in the formatting **(page 397)** category, and scope markers. The scope markers are inserted when entering buttons, `object` **(page 153)** elements, marquees, table cells, and table captions, and are used to prevent formatting from "leaking" into tables, buttons, `object` **(page 153)** elements, and marquees.

When the steps below require the UA to **reconstruct the active formatting elements**, the UA must perform the following steps:

1. If there are no entries in the list of active formatting elements **(page 398)**, then there is nothing to reconstruct; stop this algorithm.
2. If the last (most recently added) entry in the list of active formatting elements **(page 398)** is a marker, or if it is an element that is in the stack of open elements **(page 397)**, then there is nothing to reconstruct; stop this algorithm.
3. Let *entry* be the last (most recently added) element in the list of active formatting elements **(page 398)**.
4. If there are no entries before *entry* in the list of active formatting elements **(page 398)**, then jump to step 8.
5. Let *entry* be the entry one earlier than *entry* in the list of active formatting elements **(page 398)**.
6. If *entry* is neither a marker nor an element that is also in the stack of open elements **(page 397)**, go to step 4.
7. Let *entry* be the element one later than *entry* in the list of active formatting elements **(page 398)**.
8. Perform a shallow clone of the element *entry* to obtain *clone*. [DOM3CORE]
9. Append *clone* to the current node **(page 397)** and push it onto the stack of open elements **(page 397)** so that it is the new current node **(page 397)**.
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 **(page 398)** 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* (page 398) *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 **(page 398)**.
2. Remove *entry* from the list of active formatting elements **(page 398)**.

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` **(page 118)** element it would be the `HTMLAnchorElement` **(page 118)** interface), with the tag name being the name of that element, with the node being in the HTML namespace **(page 434)**, 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 **(page 400)**, and then append this node to the current node **(page 397)**, and push it onto the stack of open elements **(page 397)** so that it is the new current node **(page 397)**.

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 **(page 400)** 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` **(page 27)**.

#### 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 **(page 397)** is a `dd` **(page 117)** element, a `dt` **(page 117)** element, an `li` **(page 114)** element, a `p` **(page 108)** element, a `td` **(page 200)** element, a `th` **(page 201)** element, or a `tr` **(page 199)** element, the UA must act as if an end tag with the respective tag name had been seen and then generate implied end tags **(page 400)** 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` **(page 80)** element has been parsed (whether implicitly or explicitly) the `head` element pointer **(page 400)** gets set to point to this node.

The `form` element pointer **(page 400)** 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 **(page 403)**". It can change to "in head **(page 404)**", "after head **(page 408)**", "in body **(page 409)**", "in table **(page 422)**", "in caption **(page 424)**", "in column group **(page 425)**", "in table body **(page 426)**", "in row **(page 427)**", "in cell **(page 428)**", "in select **(page 430)**", "after body **(page 431)**", "in frameset **(page 431)**", and "after frameset **(page 432)**" 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 **(page 396)** to the trailing end phase **(page 433)** 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 **(page 397)**.
3. If *node* is the first node in the stack of open elements, then set *last* to true. If the element whose `innerHTML` **(page 39)** attribute is being set is neither a `td` **(page 200)** element nor a `th` **(page 201)** element, then set *node* to the element whose `innerHTML` **(page 39)** attribute is being set. (`innerHTML` case **(page 42)**)
4. If *node* is a `select` element, then switch the insertion mode **(page 401)** to "in select **(page 430)**" and abort these steps. (`innerHTML` case **(page 42)**)
5. If *node* is a `td` **(page 200)** or `th` **(page 201)** element, then switch the insertion mode **(page 401)** to "in cell **(page 428)**" and abort these steps.
6. If *node* is a `tr` **(page 199)** element, then switch the insertion mode **(page 401)** to "in row **(page 427)**" and abort these steps.
7. If *node* is a `tbody` **(page 196)**, `thead` **(page 198)**, or `tfoot` **(page 198)** element, then switch the insertion mode **(page 401)** to "in table body **(page 426)**" and abort these steps.
8. If *node* is a `caption` **(page 194)** element, then switch the insertion mode **(page 401)** to "in caption **(page 424)**" and abort these steps.
9. If *node* is a `colgroup` **(page 195)** element, then switch the insertion mode **(page 401)** to "in column group **(page 425)**" and abort these steps. (`innerHTML` case **(page 42)**)
10. If *node* is a `table` **(page 192)** element, then switch the insertion mode **(page 401)** to "in table **(page 422)**" and abort these steps.
11. If *node* is a `head` **(page 80)** element, then switch the insertion mode **(page 401)** to "in body **(page 409)**" ("in body **(page 409)**"! *not "in head* **(page 404)***"*!) and abort these steps. (`innerHTML` case **(page 42)**)

12. If *node* is a `body` **(page 94)** element, then switch the insertion mode **(page 401)** to "in body **(page 409)**" and abort these steps.

13. If *node* is a `frameset` element, then switch the insertion mode **(page 401)** to "in frameset **(page 431)**" and abort these steps. (`innerHTML` case **(page 42)**)

14. If *node* is an `html` **(page 79)** element, then: if the `head` element pointer **(page 400)** is null, switch the insertion mode **(page 401)** to "before head **(page 403)**", otherwise, switch the insertion mode **(page 401)** to "after head **(page 408)**". In either case, abort these steps. (`innerHTML` case **(page 42)**)

15. If *last* is true, then set the insertion mode **(page 401)** to "in body **(page 409)**" and abort these steps. (`innerHTML` case **(page 42)**)

16. Let *node* now be the node before *node* in the stack of open elements **(page 397)**.

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 **(page 374)**. Ignore the token.

↪ **A start tag token with the tag name "html"**

If this start tag token was not the first start tag token, then it is a parse error **(page 374)**.

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 **(page 397)**. If it is not, add the attribute and its corresponding value to that element.

↪ **An end-of-file token**

Generate implied end tags. **(page 400)**

If there are more than two nodes on the stack of open elements **(page 397)**, or if there are two nodes but the second node is not a `body` **(page 94)** node, this is a parse error **(page 374)**.

Otherwise, if the parser was originally created in order to handle the setting of an element's `innerHTML` **(page 39)** attribute, and there's more than one element in the stack of open elements **(page 397)**, and the second node on the stack of open elements **(page 397)** is not a `body` **(page 94)** node, then this is a parse error **(page 374)**. (`innerHTML` case **(page 42)**)

Stop parsing. **(page 433)**

> 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 **(page 401)**:

↪ **If the insertion mode (page 401) 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 **(page 394)** to the current node **(page 397)**.

↪ **A comment token**

Append a `Comment` node to the current node **(page 397)** with the `data` attribute set to the data given in the comment token.

↪ **A start tag token with the tag name "head"**

Create an element for the token **(page 400)**.

Set the `head` element pointer **(page 400)** to this new element node.

Append the new element to the current node **(page 397)** and push it onto the stack of open elements **(page 397)**.

Change the insertion mode **(page 401)** to "in head **(page 404)**".

↪ **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`** **(page 80)** ***element being generated, and with the current token being reprocessed in the "in head*** **(page 404)*" insertion mode*** **(page 401)*.***

↪ **An end tag with the tag name "html"**

Act as if a start tag token with the tag name "head" and no attributes had been seen, then reprocess the current token.

↪ **Any other end tag**

Parse error **(page 374)**. Ignore the token.

↪ **A character token that is *not* one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

↪ **Any other start tag token**

Act as if a start tag token with the tag name "head" and no attributes had been seen, then reprocess the current token.

> ***Note: This will result in an empty `head`* (page 80) *element being generated, with the current token being reprocessed in the "after head* (page 408)*" insertion mode* (page 401)*.***

↪ **If the insertion mode (page 401) is "in head"**

Handle the token as follows.

> ***Note: The rules for handling "title", "style", and "script" start tags are similar, but not identical.***

> ***Note: It is possible for the tree construction* (page 394) *stage's main phase* (page 396) *to be in the "in head* (page 404)*" insertion mode* (page 401) *without the current node* (page 397) *being a `head`* (page 80) *element, e.g. if a `head`* (page 80) *end tag is immediately followed by a `meta`* (page 86) *start tag.***

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character **(page 394)** to the current node **(page 397)**.

↪ **A comment token**

Append a `Comment` node to the current node **(page 397)** with the `data` attribute set to the data given in the comment token.

↪ **A start tag with the tag name "title"**

Create an element for the token **(page 400)**.

Append the new element to the node pointed to by the `head` element pointer **(page 400)**, or, if that is null (`innerHTML` case **(page 42)**), to the current node **(page 397)**.

Switch the tokeniser's content model flag **(page 382)** to the RCDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token.

If this process resulted in a collection of character tokens, append a single `Text` node to the `title` **(page 80)** element node whose contents is the concatenation of all those tokens' characters.

The tokeniser's content model flag **(page 382)** will have switched back to the PCDATA state.

If the next token is an end tag token with the tag name "title", ignore it. Otherwise, this is a parse error **(page 374)**.

↪ **A start tag with the tag name "style"**

Create an element for the token **(page 400)**.

Append the new element to the current node **(page 397)**, unless the insertion mode **(page 401)** is "in head **(page 404)**" and the `head` element pointer **(page 400)** is not null, in which case append it to the node pointed to by the `head` element pointer **(page 400)**. .

Switch the tokeniser's content model flag **(page 382)** to the CDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token, or until it stops tokenising.

If this process resulted in a collection of character tokens, append a single `Text` node to the `style` **(page 91)** element node whose contents is the concatenation of all those tokens' characters.

The tokeniser's content model flag **(page 382)** will have switched back to the PCDATA state.

If the next token is an end tag token with the tag name "style", ignore it. Otherwise, this is a parse error **(page 374)**.

↪ **A start tag with the tag name "script"**

Create an element for the token **(page 400)**.

Mark the element as being "parser-inserted" **(page 212)**. This ensures that, if the script is external, any `document.write()` **(page 39)** 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 **(page 382)** 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` **(page 210)** element node whose contents is the concatenation of all those tokens' characters.

The tokeniser's content model flag **(page 382)** 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 **(page 374)**; mark the `script` **(page 210)** element as "already executed" **(page 211)**. Otherwise, the token is the `script` **(page 210)** element's end tag, so ignore it.

If the parser was originally created in order to handle the setting of a node's `innerHTML` **(page 39)** attribute, then mark the `script` **(page 210)** element as "already executed" **(page 211)**, 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 **(page 214)**" are executed). (`innerHTML` case **(page 42)**)

> ***Note: Marking the `script`* (page 210) *element as "already executed" prevents it from executing when it is inserted into the document a few paragraphs below. Scripts missing their end tags and scripts that were inserted using `innerHTML`* (page 39) *aren't executed.***

Let the *old insertion point* have the same value as the current insertion point **(page 382)**. Let the insertion point **(page 382)** be just before the next input character **(page 382)**.

Append the new element to the current node **(page 397)**, unless the insertion mode **(page 401)** is "in head **(page 404)**" and the `head` element pointer **(page 400)** is not null, in which case append it to the node pointed to by the `head` element pointer **(page 400)**. Special processing occurs when a `script` element is inserted into a document **(page 212)** that might cause some script to execute, which might cause new characters to be inserted into the tokeniser **(page 39)**.

Let the insertion point **(page 382)** have the value of the *old insertion point*. (In other words, restore the insertion point **(page 382)** 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 **(page 214)**, then:

↪ **If the tree construction stage is being called reentrantly (page 375), say from a call to `document.write()` (page 39):**
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 **(page 214)**. There is no longer a script that will execute as soon as the parser resumes **(page 214)**.
2. Pause **(page 22)** until the script has completed loading.
3. Let the insertion point **(page 382)** be just before the next input character **(page 382)**.
4. Execute the script **(page 214)**.
5. Let the insertion point **(page 382)** be undefined again.
6. If there is once again a script that will execute as soon as the parser resumes **(page 214)**, then repeat these steps from step 1.

↪ **A start tag with the tag name "base", "link", or "meta"**
Create an element for the token **(page 400)**.

Append the new element to the node pointed to by the `head` element pointer **(page 400)**, or, if that is null (`innerHTML` case **(page 42)**), to the current node **(page 397)**.

Need to cope with second and subsequent `base` **(page 81)** elements affecting subsequent elements magically.

↪ **An end tag with the tag name "head"**
If the current node **(page 397)** is a `head` **(page 80)** element, pop the current node **(page 397)** off the stack of open elements **(page 397)**. Otherwise, this is a parse error **(page 374)**.

Change the insertion mode **(page 401)** to "after head **(page 408)**".

↪ **An end tag with the tag name "html"**
Act as described in the "anything else" entry below.

↪ **A start tag with the tag name "head"**
↪ **Any other end tag**
Parse error **(page 374)**. Ignore the token.

↪ **Anything else**
If the current node **(page 397)** is a `head` **(page 80)** element, act as if an end tag token with the tag name "head" had been seen.

Otherwise, change the insertion mode **(page 401)** to "after head **(page 408)**".

Then, reprocess the current token.

In certain UAs, some elements don't trigger the "in body" mode straight away, but instead get put into the head. Do we want to copy that?

↪ **If the insertion mode (page 401) 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 **(page 394)** to the current node **(page 397)**.

↪ **A comment token**
Append a `Comment` node to the current node **(page 397)** with the `data` attribute set to the data given in the comment token.

↪ **A start tag token with the tag name "body"**
Insert a `body` element **(page 400)** for the token.

Change the insertion mode **(page 401)** to "in body **(page 409)**".

↪ **A start tag token with the tag name "frameset"**
Insert a `frameset` element **(page 400)** for the token.

Change the insertion mode **(page 401)** to "in frameset **(page 431)**".

↪ **A start tag token whose tag name is one of: "base", "link", "meta", "script", "style", "title"**
Parse error **(page 374)**. Switch the insertion mode **(page 401)** back to "in head **(page 404)**" and reprocess the token.

↪ **Anything else**

Act as if a start tag token with the tag name "body" and no attributes had been seen, and then reprocess the current token.

↪ **If the insertion mode (page 401) is "in body"**

Handle the token as follows:

↪ **A character token**

Reconstruct the active formatting elements **(page 399)**, if any.

Append the token's character **(page 394)** to the current node **(page 397)**.

↪ **A comment token**

Append a `Comment` node to the current node **(page 397)** with the `data` attribute set to the data given in the comment token.

↪ **A start tag token whose tag name is one of: "script", "style"**

Process the token as if the insertion mode **(page 401)** had been "in head **(page 404)**".

↪ **A start tag token whose tag name is one of: "base", "link", "meta", "title"**

Parse error **(page 374)**. Process the token as if the insertion mode **(page 401)** had been "in head **(page 404)**".

↪ **A start tag token with the tag name "body"**

Parse error **(page 374)**.

If the second element on the stack of open elements **(page 397)** is not a `body` **(page 94)** element, or, if the stack of open elements **(page 397)** has only one node on it, then ignore the token. (`innerHTML` case **(page 42)**)

Otherwise, for each attribute on the token, check to see if the attribute is already present on the `body` **(page 94)** element (the second element) on the stack of open elements **(page 397)**. If it is not, add the attribute and its corresponding value to that element.

↪ **An end tag with the tag name "body"**

If the second element in the stack of open elements **(page 397)** is not a `body` **(page 94)** element, this is a parse error **(page 374)**. Ignore the token. (`innerHTML` case **(page 42)**)

Otherwise:

> this needs to handle closing of implied elements, but without closing them

If the current node **(page 397)** is not the `body` **(page 94)** element, then this is a parse error **(page 374)**.

Change the insertion mode **(page 401)** to "after body **(page 431)**".

↪ **An end tag with the tag name "html"**

Act as if an end tag with tag name "body" had been seen, then, if that token wasn't ignored, reprocess the current token.

> ***Note: The fake end tag token here can only be ignored in the `innerHTML` case* (page 42)*.***

↪ **A start tag whose tag name is one of: "address", "blockquote", "center", "dir", "div", "dl", "fieldset", "listing", "menu", "ol", "p", "ul"**

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

Insert an HTML element **(page 400)** for the token.

↪ **A start tag whose tag name is "pre"**

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

Insert an HTML element **(page 400)** 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` **(page 112)** blocks are ignored as an authoring convenience.)

↪ **A start tag whose tag name is "form"**

If the `form` element pointer **(page 400)** is not null, ignore the token with a parse error **(page 374)**.

Otherwise:

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

Insert an HTML element **(page 400)** 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 **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

Run the following algorithm:

1. Initialise *node* to be the current node **(page 397)** (the bottommost node of the stack).
2. If *node* is an `li` **(page 114)** element, then pop all the nodes from the current node **(page 397)** up to *node*, including *node*, then stop this algorithm.
3. If *node* is not in the formatting **(page 397)** category, and is not in the phrasing **(page 397)** category, and is not an `address` **(page 101)** or `div` **(page 255)** element, then stop this algorithm.
4. Otherwise, set *node* to the previous entry in the stack of open elements **(page 397)** and return to step 2.

Finally, insert an `li` element **(page 400)**.

↪ **A start tag whose tag name is "dd" or "dt"**

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

Run the following algorithm:

1. Initialise *node* to be the current node **(page 397)** (the bottommost node of the stack).
2. If *node* is a `dd` **(page 117)** or `dt` **(page 117)** element, then pop all the nodes from the current node **(page 397)** up to *node*, including *node*, then stop this algorithm.
3. If *node* is not in the formatting **(page 397)** category, and is not in the phrasing **(page 397)** category, and is not an `address` **(page 101)** or `div` **(page 255)** element, then stop this algorithm.
4. Otherwise, set *node* to the previous entry in the stack of open elements **(page 397)** and return to step 2.

Finally, insert an HTML element **(page 400)** with the same tag name as the token's.

↪ **A start tag token whose tag name is "plaintext"**

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

Insert an HTML element **(page 400)** for the token.

Switch the content model flag **(page 382)** to the PLAINTEXT state.

> ***Note: Once a start tag with the tag name "plaintext" has been seen, that will be the last token ever seen other than character tokens (and the end-of-file token), because there is no way to switch the content model flag* (page 382) *out of the PLAINTEXT state.***

↪ **An end tag whose tag name is one of: "address", "blockquote", "center", "dir", "div", "dl", "fieldset", "listing", "menu", "ol", "pre", "ul"**

If the stack of open elements **(page 397)** has an element in scope **(page 398)** with the same tag name as that of the token, then generate implied end tags **(page 400)**.

Now, if the current node **(page 397)** is not an element with the same tag name as that of the token, then this is a parse error **(page 374)**.

If the stack of open elements **(page 397)** has an element in scope **(page 398)** with the same tag name as that of the token, then pop elements from this stack until an element with that tag name has been popped from the stack.

↪ **An end tag whose tag name is "form"**

If the stack of open elements **(page 397)** has an element in scope **(page 398)** with the same tag name as that of the token, then generate implied end tags **(page 400)**.

Now, if the current node **(page 397)** is not an element with the same tag name as that of the token, then this is a parse error **(page 374)**.

If the stack of open elements **(page 397)** has an element in scope **(page 398)** with the same tag name as that of the token, then pop elements from this stack until an element with that tag name has been popped from the stack.

In any case, set the `form` element pointer **(page 400)** to null.

↪ **An end tag whose tag name is "p"**

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then generate implied end tags **(page 400)**, except for `p` **(page 108)** elements.

If the current node **(page 397)** is not a `p` **(page 108)** element, then this is a parse error **(page 374)**.

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then pop elements from this stack until the stack no longer has a `p` element in scope **(page 398)**.

↪ **An end tag whose tag name is "dd", "dt", or "li"**

If the stack of open elements **(page 397)** has an element in scope **(page 398)** whose tag name matches the tag name of the token, then generate implied end tags **(page 400)**, except for elements with the same tag name as the token.

If the current node **(page 397)** is not an element with the same tag name as the token, then this is a parse error **(page 374)**.

If the stack of open elements **(page 397)** has an element in scope **(page 398)** 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 **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

If the stack of open elements **(page 397)** has in scope **(page 398)** an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then this is a parse error **(page 374)**; pop elements from the stack until an element with one of those tag names has been popped from the stack.

Insert an HTML element **(page 400)** for the token.

↪ **An end tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

If the stack of open elements **(page 397)** has in scope **(page 398)** an element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then generate implied end tags **(page 400)**.

Now, if the current node **(page 397)** is not an element with the same tag name as that of the token, then this is a parse error **(page 374)**.

If the stack of open elements **(page 397)** has in scope **(page 398)** 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 **(page 398)** 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 **(page 374)**; 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 **(page 398)** and the stack of open elements **(page 397)** if the end tag didn't already remove it (it might not have if the element is not in table scope **(page 398)**).

> In the non-conforming stream `<a href="a">a<table><a href="b">b</table>x`, the first `a` **(page 118)** 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` **(page 118)** 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` **(page 118)** element).

Reconstruct the active formatting elements **(page 399)**, if any.

Insert an HTML element **(page 400)** for the token. Add that element to the list of active formatting elements **(page 398)**.

↪ **A start tag whose tag name is one of: "b", "big", "em", "font", "i", "nobr", "s", "small", "strike", "strong", "tt", "u"**

Reconstruct the active formatting elements **(page 399)**, if any.

Insert an HTML element **(page 400)** for the token. Add that element to the list of active formatting elements **(page 398)**.

↪ **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 **(page 398)** 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 **(page 397)** but the element is not in scope **(page 398)**, then this is a parse error **(page 374)**. 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 **(page 397)**, then this is a parse error **(page 374)**; remove the element from the list, and abort these steps.

Otherwise, there is a *formatting element* and that element is in the stack **(page 397)** and is in scope **(page 398)**. If the element is not the current node **(page 397)**, this is a parse error **(page 374)**. 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 **(page 397)** that is lower in the stack than the *formatting element*, and is not an element in the phrasing **(page 397)** or formatting **(page 397)** 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 **(page 397)**, from the current node **(page 397)** up to the *formatting element*, and remove the *formatting element* from the list of active formatting elements **(page 398)**.
4. Let the *common ancestor* be the element immediately above the *formatting element* in the stack of open elements **(page 397)**.
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 **(page 398)** 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 **(page 397)**.
    2. If *node* is not in the list of active formatting elements **(page 398)**, then remove *node* from the stack of open elements **(page 397)** 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 **(page 398)**.
5. If *node* has any children, perform a shallow clone of *node*, replace the entry for *node* in the list of active formatting elements **(page 398)** with an entry for the clone, replace the entry for *node* in the stack of open elements **(page 397)** 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 **(page 398)**, and insert the clone into the list of active formatting elements **(page 398)** at the position of the aforementioned bookmark.
13. Remove the *formatting element* from the stack of open elements **(page 397)**, and insert the clone into the stack of open elements **(page 397)** 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*** **(page 397)** ***category ever get cloned by this algorithm.***

    ***Note: Because of the way this algorithm causes elements to change parents, it has been dubbed the "adoption agency algorithm" (in contrast with other possibly algorithms for dealing with misnested content, which included the "incest***

***algorithm", the "secret affair algorithm", and the "Heisenberg algorithm").***

↪ **A start tag token whose tag name is "button"**

If the stack of open elements **(page 397)** has a `button` element in scope **(page 398)**, then this is a parse error **(page 374)**; act as if an end tag with the tag name "button" had been seen, then reprocess the token.

Otherwise:

Reconstruct the active formatting elements **(page 399)**, if any.

Insert an HTML element **(page 400)** for the token.

Insert a marker at the end of the list of active formatting elements **(page 398)**.

↪ **A start tag token whose tag name is one of: "marquee", "object"**

Reconstruct the active formatting elements **(page 399)**, if any.

Insert an HTML element **(page 400)** for the token.

Insert a marker at the end of the list of active formatting elements **(page 398)**.

↪ **An end tag token whose tag name is one of: "button", "marquee", "object"**

If the stack of open elements **(page 397)** has in scope **(page 398)** an element whose tag name is the same as the tag name of the token, then generate implied end tags **(page 400)**.

Now, if the current node **(page 397)** is not an element with the same tag name as the token, then this is a parse error **(page 374)**.

Now, if the stack of open elements **(page 397)** has an element in scope **(page 398)** whose tag name matches the tag name of the token, then pop elements from the stack until that element has been popped from the stack, and clear the list of active formatting elements up to the last marker **(page 399)**.

↪ **A start tag token whose tag name is "xmp"**

Reconstruct the active formatting elements **(page 399)**, if any.

Insert an HTML element **(page 400)** for the token.

Switch the content model flag **(page 382)** to the CDATA state.

↪ **A start tag whose tag name is "table"**

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

Insert an HTML element **(page 400)** for the token.

Change the insertion mode **(page 401)** to "in table **(page 422)**".

↪ **A start tag whose tag name is one of: "area", "basefont", "bgsound", "br", "embed", "img", "param", "spacer", "wbr"**

Reconstruct the active formatting elements **(page 399)**, if any.

Insert an HTML element **(page 400)** for the token. Immediately pop the current node **(page 397)** off the stack of open elements **(page 397)**.

↪ **A start tag whose tag name is "hr"**

If the stack of open elements **(page 397)** has a `p` element in scope **(page 398)**, then act as if an end tag with the tag name `p` **(page 108)** had been seen.

Insert an HTML element **(page 400)** for the token. Immediately pop the current node **(page 397)** off the stack of open elements **(page 397)**.

↪ **A start tag whose tag name is "image"**

Parse error **(page 374)**. Change the token's tag name to "img" and reprocess it. (Don't ask.)

↪ **A start tag whose tag name is "input"**

Reconstruct the active formatting elements **(page 399)**, if any.

Insert an `input` element **(page 400)** for the token.

If the `form` element pointer **(page 400)** is not null, then associate the `input` element with the `form` element pointed to by the `form` element pointer **(page 400)**.

Pop that `input` element off the stack of open elements **(page 397)**.

↪ **A start tag whose tag name is "isindex"**

Parse error **(page 374)**.

If the `form` element pointer **(page 400)** is not null, then ignore the token.

Otherwise:

Act as if a start tag token with the tag name "form" had been seen.

Act as if a start tag token with the tag name "hr" had been seen.

Act as if a start tag token with the tag name "p" had been seen.

Act as if a start tag token with the tag name "label" had been seen.

Act as if a stream of character tokens had been seen (see below for what they should say).

Act as if a start tag token with the tag name "input" had been seen, with all the attributes from the "isindex" token, except with the "name" attribute set to the value "isindex" (ignoring any explicit "name" attribute).

Act as if a stream of character tokens had been seen (see below for what they should say).

Act as if an end tag token with the tag name "label" had been seen.

Act as if an end tag token with the tag name "p" had been seen.

Act as if a start tag token with the tag name "hr" had been seen.

Act as if an end tag token with the tag name "form" had been seen.

The two streams of character tokens together should, together with the `input` element, express the equivalent of "This is a searchable index. Insert your search keywords here: (input field)" in the user's preferred language.

Then need to specify that if the form submission causes just a single form control, whose name is "isindex", to be submitted, then we submit just the value part, not the "isindex=" part.

↪ **A start tag whose tag name is "textarea"**
Create an element for the token **(page 400)**.

If the `form` element pointer **(page 400)** is not null, then associate the `textarea` element with the `form` element pointed to by the `form` element pointer **(page 400)**.

Append the new element to the current node **(page 397)**.

Switch the tokeniser's content model flag **(page 382)** to the RCDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token, or until it stops tokenising.

If this process resulted in a collection of character tokens, append a single `Text` node, whose contents is the concatenation of all those tokens' characters, to the new element node.

The tokeniser's content model flag **(page 382)** will have switched back to the PCDATA state.

If the next token is an end tag token with the tag name "textarea", ignore it. Otherwise, this is a parse error **(page 374)**.

↪ **A start tag whose tag name is one of: "iframe", "noembed", "noframes"**

↪ **A start tag whose tag name is "noscript", if scripting is enabled (page 266):**

Create an element for the token **(page 400)**.

For "iframe" tags, the node must be an `HTMLIFrameElement` **(page 150)** object, for the other tags it must be an `HTMLElement` **(page 27)** object.

Append the new element to the current node **(page 397)**.

Switch the tokeniser's content model flag **(page 382)** to the CDATA state.

Then, collect all the character tokens that the tokeniser returns until it returns a token that is not a character token, or until it stops tokenising.

If this process resulted in a collection of character tokens, append a single `Text` node, whose contents is the concatenation of all those tokens' characters, to the new element node.

The tokeniser's content model flag **(page 382)** will have switched back to the PCDATA state.

If the next token is an end tag token with the same tag name as the start tag token, ignore it. Otherwise, this is a parse error **(page 374)**.

↪ **A start tag whose tag name is "select"**

Reconstruct the active formatting elements **(page 399)**, if any.

Insert an HTML element **(page 400)** for the token.

Change the insertion mode **(page 401)** to "in select **(page 430)**".

↪ **A start or end tag whose tag name is one of: "caption", "col", "colgroup", "frame", "frameset", "head", "option", "optgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

↪ **An end tag whose tag name is one of: "area", "basefont", "bgsound", "br", "embed", "hr", "iframe", "image", "img", "input", "isindex", "noembed", "noframes", "param", "select", "spacer", "table", "textarea", "wbr"**

↪ **An end tag whose tag name is "noscript", if scripting is enabled (page 266):**

Parse error **(page 374)**. 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 **(page 399)**, if any.

Insert an HTML element **(page 400)** for the token.

> ***Note: This element will be a phrasing* (page 397) *element.***

↪ **An end tag token not covered by the previous entries**

Run the following algorithm:

1. Initialise *node* to be the current node **(page 397)** (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 **(page 400)**.
   2. If the tag name of the end tag token does not match the tag name of the current node **(page 397)**, this is a parse error **(page 374)**.
   3. Pop all the nodes from the current node **(page 397)** up to *node*, including *node*, then stop this algorithm.
3. Otherwise, if *node* is in neither the formatting **(page 397)** category nor the phrasing **(page 397)** category,

then this is a parse error **(page 374)**. Stop this algorithm. The end tag token is ignored.

4. Set *node* to the previous entry in the stack of open elements **(page 397)**.
5. Return to step 2.

↪ **If the insertion mode (page 401) 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 **(page 394)** to the current node **(page 397)**.

↪ **A comment token**

Append a `Comment` node to the current node **(page 397)** 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 **(page 424)**. (See below.)

Insert a marker at the end of the list of active formatting elements **(page 398)**.

Insert an HTML element **(page 400)** for the token, then switch the insertion mode **(page 401)** to "in caption **(page 424)**".

↪ **A start tag whose tag name is "colgroup"**

Clear the stack back to a table context **(page 424)**. (See below.)

Insert an HTML element **(page 400)** for the token, then switch the insertion mode **(page 401)** to "in column group **(page 425)**".

↪ **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 **(page 424)**. (See below.)

Insert an HTML element **(page 400)** for the token, then switch the insertion mode **(page 401)** to "in table body **(page 426)**".

↪ **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 **(page 374)**. Act as if an end tag token with the tag name "table" had been seen, then, if that token wasn't ignored, reprocess the current token.

> ***Note: The fake end tag token here can only be ignored in the `innerHTML` case* (page 42)*.***

↪ **An end tag whose tag name is "table"**
If the stack of open elements **(page 397)** does not have an element in table scope **(page 398)** with the same tag name as the token, this is a parse error **(page 374)**. Ignore the token. (`innerHTML` case **(page 42)**)

Otherwise:

Generate implied end tags **(page 400)**.

Now, if the current node **(page 397)** is not a `table` **(page 192)** element, then this is a parse error **(page 374)**.

Pop elements from this stack until a `table` **(page 192)** element has been popped from the stack.

Reset the insertion mode appropriately **(page 401)**.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**
Parse error **(page 374)**. Ignore the token.

↪ **Anything else**
Parse error **(page 374)**. Process the token as if the insertion mode **(page 401)** was "in body **(page 409)**", with the following exception:

If the current node **(page 397)** is a `table` **(page 192)**, `tbody` **(page 196)**, `tfoot` **(page 198)**, `thead` **(page 198)**, or `tr` **(page 199)** element, then, whenever a node would be inserted into the current node **(page 397)**, it must instead be inserted into the *foster parent element* **(page 423)**.

The **foster parent element** is the parent element of the last `table` **(page 192)** element in the stack of open elements **(page 397)**, if there is a `table` **(page 192)** element and it has such a parent element. If there is no `table` **(page 192)** element in the stack of open elements **(page 397)** (`innerHTML` case **(page 42)**), then the *foster* *parent element* **(page 423)** is the first element in the stack of open elements **(page 397)** (the `html` **(page 79)** element). Otherwise, if there is a `table` **(page 192)** element in the stack of open elements **(page 397)**, but the last `table` **(page 192)** element in the stack of open elements **(page 397)** has no parent, or its parent node is not an element, then the *foster parent element* **(page 423)** is the element before the last `table` **(page 192)** element in the stack of open elements **(page 397)**.

If the *foster parent element* **(page 423)** is the parent element of the last `table` **(page 192)** element in the stack of open elements **(page 397)**, then the new node must be inserted immediately *before* the last `table` **(page 192)** element in the stack of open elements **(page 397)** in the foster parent element **(page 423)**; otherwise, the new node must be *appended* to the foster parent element **(page 423)**.

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 **(page 397)** is not a `table` **(page 192)** element or an `html` **(page 79)** element, pop elements from the stack of open elements **(page 397)**. If this causes any elements to be popped from the stack, then this is a parse error **(page 374)**.

> ***Note: The current node*** **(page 397)** ***being an*** **`html`** **(page 79)** ***element after this process is an*** **`innerHTML`** ***case*** **(page 42)*.***

↪ **If the insertion mode (page 401) is "in caption"**

↪ **An end tag whose tag name is "caption"**

If the stack of open elements **(page 397)** does not have an element in table scope **(page 398)** with the same tag name as the token, this is a parse error **(page 374)**. Ignore the token. (`innerHTML` case **(page 42)**)

Otherwise:

Generate implied end tags **(page 400)**.

Now, if the current node **(page 397)** is not a `caption` **(page 194)** element, then this is a parse error **(page 374)**.

Pop elements from this stack until a `caption` **(page 194)** element has been popped from the stack.

Clear the list of active formatting elements up to the last marker **(page 399)**.

Switch the insertion mode **(page 401)** to "in table **(page 422)**".

↪ **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 **(page 374)**. Act as if an end tag with the tag name "caption" had been seen, then, if that token wasn't ignored, reprocess the current token.

> ***Note: The fake end tag token here can only be ignored in the `innerHTML` case*** **(page 42)*.***

↪ **An end tag whose tag name is one of: "body", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error **(page 374)**. Ignore the token.

↪ **Anything else**

Process the token as if the insertion mode **(page 401)** was "in body **(page 409)**".

↪ **If the insertion mode (page 401) is "in column group"**

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character **(page 394)** to the current node **(page 397)**.

↪ **A comment token**

Append a `Comment` node to the current node **(page 397)** with the `data` attribute set to the data given in the comment token.

↪ **A start tag whose tag name is "col"**

Insert a `col` element **(page 400)** for the token. Immediately pop the current node **(page 397)** off the stack of open elements **(page 397)**.

↪ **An end tag whose tag name is "colgroup"**

If the current node **(page 397)** is the root `html` **(page 79)** element, then this is a parse error **(page 374)**, ignore the token. (`innerHTML` case **(page 42)**)

Otherwise, pop the current node **(page 397)** (which will be a `colgroup` **(page 195)** element) from the stack of open elements **(page 397)**. Switch the insertion mode **(page 401)** to "in table **(page 422)**".

↪ **An end tag whose tag name is "col"**

Parse error **(page 374)**. Ignore the token.

↪ **Anything else**

Act as if an end tag with the tag name "colgroup" had been seen, and then, if that token wasn't ignored, reprocess the current token.

***Note: The fake end tag token here can only be ignored in the `innerHTML` case* (page 42)*.***

↪ **If the insertion mode (page 401) is "in table body"**

↪ **A start tag whose tag name is "tr"**

Clear the stack back to a table body context **(page 427)**. (See below.)

Insert a `tr` element **(page 400)** for the token, then switch the insertion mode **(page 401)** to "in row **(page 427)**".

↪ **A start tag whose tag name is one of: "th", "td"**

Parse error **(page 374)**. Act as if a start tag with the tag name "tr" had been seen, then reprocess the current token.

↪ **An end tag whose tag name is one of: "tbody", "tfoot", "thead"**

If the stack of open elements **(page 397)** does not have an element in table scope **(page 398)** with the same tag name as the token, this is a parse error **(page 374)**. Ignore the token.

Otherwise:

Clear the stack back to a table body context **(page 427)**. (See below.)

Pop the current node **(page 397)** from the stack of open elements **(page 397)**. Switch the insertion mode **(page 401)** to "in table **(page 422)**".

↪ **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 **(page 397)** does not have a `tbody`, `thead`, or `tfoot` element in table scope **(page 398)**, this is a parse error **(page 374)**. Ignore the token. (`innerHTML` case **(page 42)**)

Otherwise:

Clear the stack back to a table body context **(page 427)**. (See below.)

Act as if an end tag with the same tag name as the current node **(page 397)** ("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 **(page 374)**. Ignore the token.

↪ **Anything else**

Process the token as if the insertion mode **(page 401)** was "in table **(page 422)**".

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 **(page 397)** is not a `tbody` **(page 196)**, `tfoot` **(page 198)**, `thead` **(page 198)**, or `html` **(page 79)** element, pop elements from the stack of open elements **(page 397)**. If this causes any elements to be popped from the stack, then this is a parse error **(page 374)**.

***Note: The current node* (page 397) *being an `html`* (page 79) *element after this process is an `innerHTML` case* (page 42)*.***

↪ **If the insertion mode (page 401) is "in row"**

↪ **A start tag whose tag name is one of: "th", "td"**

Clear the stack back to a table row context **(page 428)**. (See below.)

Insert an HTML element **(page 400)** for the token, then switch the insertion mode **(page 401)** to "in cell **(page 428)**".

Insert a marker at the end of the list of active formatting elements **(page 398)**.

↪ **An end tag whose tag name is "tr"**

If the stack of open elements **(page 397)** does not have an element in table scope **(page 398)** with the same tag name as the token, this is a parse error **(page 374)**. Ignore the token. (`innerHTML` case **(page 42)**)

Otherwise:

Clear the stack back to a table row context **(page 428)**. (See below.)

Pop the current node **(page 397)** (which will be a `tr` **(page 199)** element) from the stack of open elements **(page 397)**. Switch the insertion mode **(page 401)** to "in table body **(page 426)**".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "tfoot", "thead", "tr"**

↪ **An end tag whose tag name is "table"**

Act as if an end tag with the tag name "tr" had been seen, then, if that token wasn't ignored, reprocess the current token.

***Note: The fake end tag token here can only be ignored in the `innerHTML` case* (page 42).**

↪ **An end tag whose tag name is one of: "tbody", "tfoot", "thead"**

If the stack of open elements **(page 397)** does not have an element in table scope **(page 398)** with the same tag name as the token, this is a parse error **(page 374)**. 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 **(page 374)**. Ignore the token.

↪ **Anything else**

Process the token as if the insertion mode **(page 401)** was "in table **(page 422)**".

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 **(page 397)** is not a `tr` **(page 199)** element or an `html` **(page 79)** element, pop elements from the stack of open elements **(page 397)**. If this causes any elements to be popped from the stack, then this is a parse error **(page 374)**.

***Note: The current node* (page 397) *being an `html`* (page 79) *element after this process is an `innerHTML` case* (page 42).**

↪ **If the insertion mode (page 401) is "in cell"**

↪ **An end tag whose tag name is one of: "td", "th"**

If the stack of open elements **(page 397)** does not have an element in table scope **(page 398)** with the same tag name as that of the token, then this is a parse error **(page 374)** and the token must be ignored.

Otherwise:

Generate implied end tags **(page 400)**, except for elements with the same tag name as the token.

Now, if the current node **(page 397)** is not an element with the same tag name as the token, then this is a parse error **(page 374)**.

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 **(page 399)**.

Switch the insertion mode **(page 401)** to "in row **(page 427)**". (The current node **(page 397)** will be a `tr` **(page 199)** 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 **(page 397)** does *not* have a `td` or `th` element in table scope **(page 398)**, then this is a parse error **(page 374)**; ignore the token. (`innerHTML` case **(page 42)**)

Otherwise, close the cell **(page 429)** (see below) and reprocess the current token.

↪ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html"**

Parse error **(page 374)**. Ignore the token.

↪ **An end tag whose tag name is one of: "table", "tbody", "tfoot", "thead", "tr"**

If the stack of open elements **(page 397)** does not have an element in table scope **(page 398)** with the same tag name as that of the token (which can only happen for "tbody", "tfoot" and "thead", or, in the `innerHTML` case **(page 42)**), then this is a parse error **(page 374)** and the token must be ignored.

Otherwise, close the cell **(page 429)** (see below) and reprocess the current token.

↪ **Anything else**

Process the token as if the insertion mode **(page 401)** was "in body **(page 409)**".

Where the steps above say to **close the cell**, they mean to follow the following algorithm:

1. If the stack of open elements **(page 397)** has a `td` element in table scope **(page 398)**, then act as if an end tag token with the tag name "td" had been seen.
2. Otherwise, the stack of open elements **(page 397)** will have a `th` element in table scope **(page 398)**; act as if an end tag token with the tag name "th" had been seen.

   ***Note: The stack of open elements* (page 397) *cannot have both a `td`* (page 200) *and a `th`* (page 201) *element in table scope* (page 398) *at the same time, nor can it have neither when the insertion mode* (page 401) *is "in cell* (page 428)*".***

↪ **If the insertion mode (page 401) is "in select"**

Handle the token as follows:

↪ **A character token**

Append the token's character **(page 394)** to the current node **(page 397)**.

↪ **A comment token**

Append a `Comment` node to the current node **(page 397)** with the `data` attribute set to the data given in the comment token.

↪ **A start tag token whose tag name is "option"**

If the current node **(page 397)** is an `option` element, act as if an end tag with the tag name "option" had been seen.

Insert an HTML element **(page 400)** for the token.

↪ **A start tag token whose tag name is "optgroup"**

If the current node **(page 397)** is an `option` element, act as if an end tag with the tag name "option" had been seen.

If the current node **(page 397)** is an `optgroup` element, act as if an end tag with the tag name "optgroup" had been seen.

Insert an HTML element **(page 400)** for the token.

↪ **An end tag token whose tag name is "optgroup"**

First, if the current node **(page 397)** is an `option` element, and the node immediately before it in the stack of open elements **(page 397)** is an `optgroup` element, then act as if an end tag with the tag name "option" had been seen.

If the current node **(page 397)** is an `optgroup` element, then pop that node from the stack of open elements **(page 397)**. Otherwise, this is a parse error **(page 374)**, ignore the token.

↪ **An end tag token whose tag name is "option"**

If the current node **(page 397)** is an `option` element, then pop that node from the stack of open elements **(page 397)**. Otherwise, this is a parse error **(page 374)**, ignore the token.

↪ **An end tag whose tag name is "select"**

If the stack of open elements **(page 397)** does not have an element in table scope **(page 398)** with the same tag name as the token, this is a parse error **(page 374)**. Ignore the token. (`innerHTML` case **(page 42)**)

Otherwise:

Pop elements from the stack of open elements **(page 397)** until a `select` element has been popped from the stack.

Reset the insertion mode appropriately **(page 401)**.

↪ **A start tag whose tag name is "select"**
Parse error **(page 374)**. 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 **(page 374)**.

If the stack of open elements **(page 397)** has an element in table scope **(page 398)** 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 **(page 374)**. Ignore the token.

↪ **If the insertion mode (page 401) is "after body"**
Handle the token as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**
Process the token as it would be processed if the insertion mode **(page 401)** was "in body **(page 409)**".

↪ **A comment token**
Append a `Comment` node to the first element in the stack of open elements **(page 397)** (the `html` **(page 79)** element), with the `data` attribute set to the data given in the comment token.

↪ **An end tag with the tag name "html"**
If the parser was originally created in order to handle the setting of *an element*'s `innerHTML` **(page 39)** attribute, this is a parse error **(page 374)**; ignore the token. (The element will be an `html` **(page 79)** element in this case.) (`innerHTML` case **(page 42)**)

Otherwise, switch to the trailing end phase **(page 433)**.

↪ **Anything else**
Parse error **(page 374)**. Set the insertion mode **(page 401)** to "in body **(page 409)**" and reprocess the token.

↪ **If the insertion mode (page 401) is "in frameset"**
Handle the token as follows:

↪ **A character token that is one of one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000B LINE TABULATION, U+000C FORM FEED (FF), or U+0020 SPACE**

Append the character **(page 394)** to the current node **(page 397)**.

↪ **A comment token**

Append a `Comment` node to the current node **(page 397)** with the `data` attribute set to the data given in the comment token.

↪ **A start tag with the tag name "frameset"**

Insert a `frameset` element **(page 400)** for the token.

↪ **An end tag with the tag name "frameset"**

If the current node **(page 397)** is the root `html` **(page 79)** element, then this is a parse error **(page 374)**; ignore the token. (`innerHTML` case **(page 42)**)

Otherwise, pop the current node **(page 397)** from the stack of open elements **(page 397)**.

If the parser was *not* originally created in order to handle the setting of an element's `innerHTML` **(page 39)** attribute (`innerHTML` case **(page 42)**), and the current node **(page 397)** is no longer a `frameset` element, then change the insertion mode **(page 401)** to "after frameset **(page 432)**".

↪ **A start tag with the tag name "frame"**

Insert an HTML element **(page 400)** for the token. Immediately pop the current node **(page 397)** off the stack of open elements **(page 397)**.

↪ **A start tag with the tag name "noframes"**

Process the token as if the insertion mode **(page 401)** had been "in body **(page 409)**".

↪ **Anything else**

Parse error **(page 374)**. Ignore the token.

↪ **If the insertion mode (page 401) 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 **(page 394)** to the current node **(page 397)**.

↪ **A comment token**

Append a `Comment` node to the current node **(page 397)** with the `data` attribute set to the data given in the comment token.

↪ **An end tag with the tag name "html"**

Switch to the trailing end phase **(page 433)**.

↪ **A start tag with the tag name "noframes"**

Process the token as if the insertion mode **(page 401)** had been "in body **(page 409)**".

↪ **Anything else**

Parse error **(page 374)**. 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 **(page 396)**, as each token is emitted from the tokenisation **(page 382)** stage, it must be processed as described in this section.

↪ **A DOCTYPE token**

Parse error **(page 374)**. 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 **(page 396)**.

↪ **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 **(page 374)**. Switch back to the main phase **(page 396)** and reprocess the token.

↪ **An end-of-file token**

Stop parsing **(page 433)**.

### 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 **(page 213)** 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 **(page 214)** have completed loading, or if the list of scripts that will execute asynchronously **(page 213)** 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` **(page 210)** element definition.

Then, if the list of scripts that will execute when the document has finished parsing **(page 213)** 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.

Once everything that **delays the load event** has completed, the user agent must fire a `load` event **(page 273)** at the `body` element **(page 35)**.

## 8.3. Namespaces

The **HTML namespace** is: `http://www.w3.org/1999/xhtml`

## 8.4. Entities

This table lists the entity names that are supported by HTML, and the code points to which they refer. It is referenced by the previous sections.

| Entity Name | Character |
|---|---|
| `AElig` | U+00C6 |
| `Aacute` | U+00C1 |
| `Acirc` | U+00C2 |
| `Agrave` | U+00C0 |
| `Alpha` | U+0391 |
| `Aring` | U+00C5 |
| `Atilde` | U+00C3 |
| `Auml` | U+00C4 |
| `Beta` | U+0392 |
| `Ccedil` | U+00C7 |
| `Chi` | U+03A7 |
| `Dagger` | U+2021 |
| `Delta` | U+0394 |
| `ETH` | U+00D0 |
| `Eacute` | U+00C9 |
| `Ecirc` | U+00CA |
| `Egrave` | U+00C8 |
| `Epsilon` | U+0395 |
| `Eta` | U+0397 |
| `Euml` | U+00CB |
| `Gamma` | U+0393 |
| `Iacute` | U+00CD |
| `Icirc` | U+00CE |
| `Igrave` | U+00CC |
| `Iota` | U+0399 |
| `Iuml` | U+00CF |
| `Kappa` | U+039A |
| `Lambda` | U+039B |
| `Mu` | U+039C |
| `Ntilde` | U+00D1 |
| `Nu` | U+039D |
| `OElig` | U+0152 |
| `Oacute` | U+00D3 |
| `Ocirc` | U+00D4 |
| `Ograve` | U+00D2 |
| `Omega` | U+03A9 |
| `Omicron` | U+039F |
| `Oslash` | U+00D8 |
| `Otilde` | U+00D5 |
| `Ouml` | U+00D6 |
| `Phi` | U+03A6 |
| `Pi` | U+03A0 |
| `Prime` | U+2033 |
| `Psi` | U+03A8 |
| `Rho` | U+03A1 |

| Entity Name | Character |
|---|---|
| Scaron | U+0160 |
| Sigma | U+03A3 |
| THORN | U+00DE |
| Tau | U+03A4 |
| Theta | U+0398 |
| Uacute | U+00DA |
| Ucirc | U+00DB |
| Ugrave | U+00D9 |
| Upsilon | U+03A5 |
| Uuml | U+00DC |
| Xi | U+039E |
| Yacute | U+00DD |
| Yuml | U+0178 |
| Zeta | U+0396 |
| aacute | U+00E1 |
| acirc | U+00E2 |
| acute | U+00B4 |
| aelig | U+00E6 |
| agrave | U+00E0 |
| alefsym | U+2135 |
| alpha | U+03B1 |
| amp | U+0026 |
| AMP | U+0026 |
| and | U+2227 |
| ang | U+2220 |
| apos | U+0027 |
| aring | U+00E5 |
| asymp | U+2248 |
| atilde | U+00E3 |
| auml | U+00E4 |
| bdquo | U+201E |
| beta | U+03B2 |
| brvbar | U+00A6 |
| bull | U+2022 |
| cap | U+2229 |
| ccedil | U+00E7 |

| Entity Name | Character |
|---|---|
| cedil | U+00B8 |
| cent | U+00A2 |
| chi | U+03C7 |
| circ | U+02C6 |
| clubs | U+2663 |
| cong | U+2245 |
| copy | U+00A9 |
| COPY | U+00A9 |
| crarr | U+21B5 |
| cup | U+222A |
| curren | U+00A4 |
| dArr | U+21D3 |
| dagger | U+2020 |
| darr | U+2193 |
| deg | U+00B0 |
| delta | U+03B4 |
| diams | U+2666 |
| divide | U+00F7 |
| eacute | U+00E9 |
| ecirc | U+00EA |
| egrave | U+00E8 |
| empty | U+2205 |
| emsp | U+2003 |
| ensp | U+2002 |
| epsilon | U+03B5 |
| equiv | U+2261 |
| eta | U+03B7 |
| eth | U+00F0 |
| euml | U+00EB |
| euro | U+20AC |
| exist | U+2203 |
| fnof | U+0192 |
| forall | U+2200 |
| frac12 | U+00BD |
| frac14 | U+00BC |
| frac34 | U+00BE |

| Entity Name | Character |
|---|---|
| frasl | U+2044 |
| gamma | U+03B3 |
| ge | U+2265 |
| gt | U+003E |
| GT | U+003E |
| hArr | U+21D4 |
| harr | U+2194 |
| hearts | U+2665 |
| hellip | U+2026 |
| iacute | U+00ED |
| icirc | U+00EE |
| iexcl | U+00A1 |
| igrave | U+00EC |
| image | U+2111 |
| infin | U+221E |
| int | U+222B |
| iota | U+03B9 |
| iquest | U+00BF |
| isin | U+2208 |
| iuml | U+00EF |
| kappa | U+03BA |
| lArr | U+21D0 |
| lambda | U+03BB |
| lang | U+2329 |
| laquo | U+00AB |
| larr | U+2190 |
| lceil | U+2308 |
| ldquo | U+201C |
| le | U+2264 |
| lfloor | U+230A |
| lowast | U+2217 |
| loz | U+25CA |
| lrm | U+200E |
| lsaquo | U+2039 |
| lsquo | U+2018 |
| lt | U+003C |

| Entity Name | Character |
|---|---|
| LT | U+003C |
| macr | U+00AF |
| mdash | U+2014 |
| micro | U+00B5 |
| middot | U+00B7 |
| minus | U+2212 |
| mu | U+03BC |
| nabla | U+2207 |
| nbsp | U+00A0 |
| ndash | U+2013 |
| ne | U+2260 |
| ni | U+220B |
| not | U+00AC |
| notin | U+2209 |
| nsub | U+2284 |
| ntilde | U+00F1 |
| nu | U+03BD |
| oacute | U+00F3 |
| ocirc | U+00F4 |
| oelig | U+0153 |
| ograve | U+00F2 |
| oline | U+203E |
| omega | U+03C9 |
| omicron | U+03BF |
| oplus | U+2295 |
| or | U+2228 |
| ordf | U+00AA |
| ordm | U+00BA |
| oslash | U+00F8 |
| otilde | U+00F5 |
| otimes | U+2297 |
| ouml | U+00F6 |
| para | U+00B6 |
| part | U+2202 |
| permil | U+2030 |
| perp | U+22A5 |
| phi | U+03C6 |
| pi | U+03C0 |
| piv | U+03D6 |
| plusmn | U+00B1 |
| pound | U+00A3 |
| prime | U+2032 |
| prod | U+220F |
| prop | U+221D |
| psi | U+03C8 |
| quot | U+0022 |
| QUOT | U+0022 |
| rArr | U+21D2 |
| radic | U+221A |
| rang | U+232A |
| raquo | U+00BB |
| rarr | U+2192 |
| rceil | U+2309 |
| rdquo | U+201D |
| real | U+211C |
| reg | U+00AE |
| REG | U+00AE |
| rfloor | U+230B |
| rho | U+03C1 |
| rlm | U+200F |
| rsaquo | U+203A |
| rsquo | U+2019 |
| sbquo | U+201A |
| scaron | U+0161 |
| sdot | U+22C5 |
| sect | U+00A7 |
| shy | U+00AD |
| sigma | U+03C3 |
| sigmaf | U+03C2 |
| sim | U+223C |
| spades | U+2660 |
| sub | U+2282 |
| sube | U+2286 |
| sum | U+2211 |
| sup | U+2283 |
| sup1 | U+00B9 |
| sup2 | U+00B2 |
| sup3 | U+00B3 |
| supe | U+2287 |
| szlig | U+00DF |
| tau | U+03C4 |
| there4 | U+2234 |
| theta | U+03B8 |
| thetasym | U+03D1 |
| thinsp | U+2009 |
| thorn | U+00FE |
| tilde | U+02DC |
| times | U+00D7 |
| trade | U+2122 |
| uArr | U+21D1 |
| uacute | U+00FA |
| uarr | U+2191 |
| ucirc | U+00FB |
| ugrave | U+00F9 |
| uml | U+00A8 |
| upsih | U+03D2 |
| upsilon | U+03C5 |
| uuml | U+00FC |
| weierp | U+2118 |
| xi | U+03BE |
| yacute | U+00FD |
| yen | U+00A5 |
| yuml | U+00FF |
| zeta | U+03B6 |
| 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 `meta` **(page 86)** element in the `head` **(page 80)** element whose `name` **(page 86)** attribute has the value `generator` **(page 87)** and whose `content` **(page 86)** attribute's value ends with the string "`(WYSIWYG editor)`". Non-WYSIWYG authoring tools must not include this string in their generator string.

### 9.1.2. The `font` element

Transparent **(page 68)** block-level element **(page 67)**, and transparent **(page 68)** strictly inline-level content **(page 67)**.

**Contexts in which this element may be used:**

Where block-level content is allowed.
Where strictly inline-level content **(page 67)** is allowed.

**Content model:**

Transparent **(page 68)**.

**Element-specific attributes:**

`style` **(page 438)**

**Predefined classes that apply to this element:**

None.

**DOM interface:**

```
interface HTMLFontElement : HTMLElement (page 27) {
  readonly attribute CSSStyleDeclaration style (page 438);
};
```

The `font` **(page 437)** element doesn't represent anything. It must not be used except by WYSIWYG editors **(page 437)**, 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 **(page 437)**.

A `font` **(page 437)** element can only contain content that would still be conformant if all elements with transparent **(page 68)** content models were replaced by their contents.

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 `font` **(page 437)** element is conformant because `h1` **(page 98)** and `p` **(page 108)** elements are both allowed in `body` **(page 94)** elements. the next four are allowed because text and `em` **(page 122)** elements are allowed in `p` **(page 108)** elements.

The **`style`** attribute, if specified, must contain only a list of zero or more semicolon-separated (;) CSS declarations. [CSS21]

The declarations specified must be parsed and treated as the body of a declaration block whose selector matches just that `font` **(page 437)** 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 `style` **(page 438)** 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. [TBW] 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.

## 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` **(page 439)** interface.

```
interface MediaModeAbstractView {
  readonly attribute DOMString mediaMode (page 439);
};
```

The **`mediaMode`** attribute on objects implementing the `MediaModeAbstractView` **(page 439)** 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.

# 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. [SCS] Timers

This section is expected to be moved to the Window Object specification in due course.

```
interface WindowTimers {
  // timers
  long setTimeout (page 442)(in TimeoutHandler (page 442) handler, in long
timeout);
  long setTimeout (page 442)(in TimeoutHandler (page 442) handler, in long
timeout, arguments...);
  long setTimeout (page 442)(in DOMString code, in long timeout);
  long setTimeout (page 442)(in DOMString code, in long timeout, in
DOMString language);
  void clearTimeout (page 442)(in long handle);
  long setInterval (page 442)(in TimeoutHandler (page 442) handler, in long
timeout);
  long setInterval (page 442)(in TimeoutHandler (page 442) handler, in long
timeout, arguments...);
  long setInterval (page 442)(in DOMString code, in long timeout);
  long setInterval (page 442)(in DOMString code, in long timeout, in
DOMString language);
  void clearInterval (page 442)(in long handle);
};
```

```
interface TimeoutHandler {
  void handleEvent(arguments...);
};
```

The `WindowTimers` **(page 441)** interface must be obtainable from any `Window` object using binding-specific casting methods.

The `setTimeout` **(page 442)** and `setInterval` **(page 442)** methods allow authors to schedule timer-based events.

The **`setTimeout(`*`handler`*`, `*`timeout`*`[, `*`arguments...`*`])`** method takes a reference to a `TimeoutHandler` **(page 442)** 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` **(page 442)** interface such that invoking the `handleEvent()` method of that interface on the object from another language binding invokes the function itself, with the arguments passed to `handleEvent()` as the arguments passed to the function. In the ECMAScript DOM binding itself, however, the `handleEvent()` method of the interface is not directly accessible on `Function` objects. Such functions must be called in the global scope **(page 287)**.

Alternatively, **`setTimeout(`*`code`*`, `*`timeout`*`[, `*`language`*`])`** may be used. This variant takes a string instead of a `TimeoutHandler` **(page 442)** object. That string must be parsed using the specified *language* (defaulting to ECMAScript if the third argument is omitted) and executed in the global scope **(page 287)**.

Need to define *language* values.

The **`setInterval(...)`** variants must work in the same way as the `setTimeout` **(page 442)** variants except that the *handler* or `code` **(page 137)** must be invoked again every *timeout* milliseconds, not just the once.

The **`clearTimeout()`** and **`clearInterval()`** methods take one integer (the value returned by `setTimeout` **(page 442)** and `setInterval` **(page 442)** respectively) and must cancel the specified timeout. When called with a value that does not correspond to an active timeout or interval, the methods must return without doing anything.

Timeouts must never fire while another script is executing. (Thus the HTML scripting model is strictly single-threaded and not reentrant.)

## References

This section will be written in a future draft.

## Acknowledgements

Thanks to Aankhen, Aaron Leventhal, Alexey Feldgendler, Andrew Gove, Anne van Kesteren, Anthony Hickson, Asbjørn Ulsberg, Ben Godfrey, Ben Meadowcroft, Benjamin Hawkes-Lewis, Bert Bos, Bjoern Hoehrmann, Boris Zbarsky, Brad Fults, Brad Neuberg, Brendan Eich, Brett Wilson, Chao Cai, Channy Yun, Charl van Niekerk, Charles Iliya Krempeaux, Charles McCathieNevile, Christian Biesinger, Christian Johansen, Chriswa, Daniel Peng, Darin Fisher, Dave Townsend, David Baron, David Flanagan, David Håsäther, David Hyatt, Derek Featherstone, Dimitri Glazkov, dolphinling, Doron Rosenberg, Eira Monstad, Elliotte Harold, Erik Arvidsson, fantasai, Franck 'Shift' Quélain, Håkon Wium Lie, Henri Sivonen, Henrik Lied, Ignacio Javier, J. King, James Graham, James M Snell, James Perrett, Jan-Klaas Kollhof, Jasper Bryant-Greene, Jeff Cutsinger, Jens Bannmann, Joel Spolsky, John Harding, Johnny Stenback, Jon Perlow, Jonathan Worent, Jorgen Horstink, Joshua Randall, Jukka K. Korpela, Kai Hendry, Kornel Lesinski, 黒澤剛志 (KUROSAWA Takeshi), Lachlan Hunt, Larry Page, Laurens Holst, Lenny Domnitser, Léonard Bouchet, Leons Petrazickis, Logan, Maciej Stachowiak, Malcolm Rowe, Mark Nottingham, Mark Schenk, Martijn Wargers, Martin Atkins, Martin Honnen, Mathieu Henri, Matthew Mastracci, Matthew Raymond, Matthew Thomas, Mattias Waldau, Max Romantschuk, Michael 'Ratt' Iannarelli, Michael A. Nachbaur, Michael Gratton, Michael Powers, Michel Fortin, Mihai Şucan, Mike Dierken, Mike Dixon, Mike Schinkel, Mike Shaver, Mikko Rantalainen, Neil Deakin, Olav Junker Kjær, Rajas Moonka, Rimantas Liubertas, Robert O'Callahan, Robert Sayre, Roman Ivanov, S. Mike Dierken, Sam Ruby, Sean Knapp, Shadow2531, Shaun Inman, Simon Pieters, Stefan Haustein, Stephen Ma, Steve Runyon, Steven Garrity, Stewart Brodie, Stuart Parmenter, Tantek Çelik, Thomas Broyer, Thomas O'Connor, Tim Altman, Vladimir Vukićević, William Swanson, and everyone on the WHATWG mailing list for their useful and substantial comments.

Special thanks to Richard Williamson for creating the first implementation of `canvas` **(page 169)** 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` **(page 315)**, 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 **(page 414)** 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.