tabindex attribute needs to be checked for backwards-compatibility. hosts?
better way of doing this.
menus.

Web Applications 1.0
Working Draft — 1 July 2005

This version:

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

Latest version:

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

Editor:

Ian Hickson, Opera Software, 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, server-sent events, and more.

Status of this document

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

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

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

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

Table of contents

---

1. Introduction

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 specification is limited to providing a semantic-level markup language and associated semantic-level scripting APIs.

The scope of this specification does not include addressing presentation concerns.

The scope of this specification does not include documenting every HTML or DOM feature supporting 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, office productivity applications, image manipulation, and other applications that users would be expected to use with high-end workstations on a daily basis are out of scope. In terms of applications, this specification is targetted specifically at applications that would be expected to be used by users on an occasional basis, or regularly but from disparate locations. 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), 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. Requirements and ideas

HTML, CSS, DOM, and JavaScript provide enough power that Web developers have managed to base entire businesses on them. What is required are extensions to these technologies to provide much-needed features such as:

- Native pop-up menus and context menus.
- Inline markup for pop-up windows, for example for dialog boxes or tool palettes, so that dialogs need not be defined in separate files.
- Command updating: applications that have several access points for the same feature, for instance a menu item and a tool-bar button, would benefit from having to disable such commands only once, instead of having to keep each access point synchronized with the feature's availability at all times. Similarly menu items or tool-bar buttons that represent a toggle state could automatically stay synchronized whenever toggled.
- Server-sent events: triggering DOM3 Events from the server-side, for example for tickers or status updates.
- Client-server communications methods that do not require page loads, enabling on-demand data retrieval (where the UA automatically fetches data from the server as required), remote procedure calls (where script can invoke code on the server side and get an XML fragment in return), etc.
- More device-independent DOM events: The DOM event set needs device-independent events, such as events that fire when a button or link is activated, whether via the mouse or the keyboard. `DOMActivate` is a start, but it lacks equivalent HTML attributes, and additional events may be needed.
- Sortable and multicolumn tree views and list views with rich formatting.
- Rich text editing: an underlying architecture upon which domain-specific editors can be created, including things like control over the caret position.
- A predefined HTML editor based on the rich text editing architecture.
- Drag and drop APIs.
- Text selection manipulation APIs.
- Clipboard APIs (if the security and privacy concerns can be addressed).

Some less important features would be good to have as well:

- Window-based state management (so that new windows don't interfere

with existing sessions), for example implemented as a per-domain, per-window "file system". This would allow multiple instances of the same application (from the same site) to run without the instances overwriting each other's cookies.

- Elements for semantics commonly found in applications, such as <byline>, <footer>, <section>, <nav>, etc.
- Markup to denote mutually exclusive sections (as in the commonly seen wizard interfaces).
- Better defined user authentication state handling. (Being able to "log out" of sites reliably, for instance, or being able to integrate the HTTP authentication model into the Web page.)

Several of the features in these two lists have been supported in non-standard ways by some user agents for some time.

1.3. Relationship to HTML 4.01, XHTML 1.1, and DOM2 HTML

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.

1.4. Relationship to XHTML2

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.

1.5. Relationship to Web Forms 2.0

This specification is designed to complement Web Forms 2.0. [WF2] Where Web Forms concentrates on input controls, data validation, and form submission, this specification concentrates on client-side user interface features needed to create modern applications.

Eventually WF2 will simply be folded into this spec.

1.6. Relationship to CSS3 UI

The CSS3 UI specification [CSS3UI] introduces a number of properties suitable for Web-based application development. This specification expands on those properties and specifies their interaction with scripting-based environments and the DOM.

1.7. Relationship to XUL, Avalon/XAML, and other proprietary UI languages

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

1.8. Conformance requirements

As well as sections marked as non-normative, all diagrams, examples, and notes in this specification are 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.

This specification describes the conformance criteria for user agents (implementations and their implementors) and documents (and their authors).

Conformance requirements phrased as requirements on elements, attributes, methods or objects are conformance requirements on user agents.

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

Web browsers and other interactive user agents

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

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

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

Non-interactive presentation user agents

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

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

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

User agents with no scripting support

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

Note: Scripting can form an integral part of an application. Web browsers that do not support scripting, or that have

scripting disabled, might be unable to fully convey the author's intent.

Conformance checkers

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

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.

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.)

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

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

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

HTML documents that use the new features described in this specification and that are served over the wire (e.g. by HTTP) must be sent as `text/html` and must start with the following DOCTYPE: `<!DOCTYPE html>`.

1.9. 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 must 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**", 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).

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)".

The 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.

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`

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.

This specification uses the term *HTML documents* to generally 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.

1.10. Miscellaneous

As the specification evolves, these conformance requirements will most likely be moved to more appropriate places.

When a UA needs to convert a string to a number, algorithms equivalent to those specified in ECMA262 sections 9.3.1 ("ToNumber Applied to the String Type") and 8.5 ("The Number type") should be used (possibly after suitably altering the algorithms to handle numbers of the range that the UA can support). [ECMA262]

The `alt` attribute on images must not be shown in a tooltip in visual browsers.

DOM mutation events must not fire for changes caused by the UA parsing the document. (Conceptually, the parser is not mutating the DOM, it is constructing it.) This includes the parsing of any content inserted using `document.write()` and `document.writeln()` calls. Other changes, including fragment insertions involving `innerHTML` and similar attributes, must fire mutation events. [DOM3EVENTS]

The default value of `Content-Style-Type` and the default value of the `type` attribute of the `style` element is is `text/css`.

The default value of `Content-Script-Type` and the default value of the `type` attribute of the `script` element is the ECMAScript MIME type.

2. Semantics and structure of HTML elements

2.1. Introduction

2.1.1. Semantics

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

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

> For example, the following document is non-conforming, despite being syntactically correct:

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

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

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

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

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

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

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

2.1.2. Structure

All the elements in this specification have a defined content model, which describes what nodes are allows 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.

For the purposes of determining if an element matches its content model or not, CDATA nodes in the DOM must be treated as text nodes, and character entity reference nodes must be treated as if they were expanded in place.

The whitespace characters U+0020 SPACE, U+000A LINE FEED, and U+000D CARRIAGE RETURN are always allowed between elements. User agents must always represent these characters between elements in the source markup as text nodes in the DOM. Empty text nodes and text nodes consisting of just sequences of those characters are considered **inter-element whitespace** and must be ignored when establishing whether an element matches its content model or not.

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

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

2.1.3. The DOM

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

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

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

To ease migration from HTML to XHTML, UAs must assign the `http://www.w3.org/1999/xhtml` namespace to elements in that are parsed in documents labelled as `text/html`, at least for the purposes of the DOM and CSS.

In HTML documents, for HTML elements, the DOM APIs must return tag names and attributes names in uppercase, regardless of the case with which they were created. This does not apply to XML documents; in XML documents, the DOM APIs must always return tag names and attribute names in the original case used to create those nodes.

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

The values "`HTML`" and "`XHTML`" (both with version "`5.0`") may also be used with 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`".***

2.1.3.2. Common DOM interfaces

Still need to define `HTMLCollection`.

```
interface DOMTokenString {
  bool has(in DOMString token);
  void add(in DOMString token);
```

```
  void remove(in DOMString token);
}
```

Need to define those members.

2.1.3.3. The document

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

This object must also implement the document-level interface of any other namespaces found in the document that the UA supports. For example, if the implementation supports both HTML and SVG, then the `Document` object must also implement `HTMLDocument` and `SVGDocument`.

```
interface HTMLDocument : Document {
           attribute DOMString title;
  readonly attribute DOMString referrer;
  readonly attribute DOMString domain;
  readonly attribute DOMString URl;
           attribute HTMLElement body;
  readonly attribute HTMLCollection images;
  readonly attribute HTMLCollection applets;
  readonly attribute HTMLCollection links;
  readonly attribute HTMLCollection forms;
  readonly attribute HTMLCollection anchors;
           attribute DOMString cookie;

  void open();
  void close();
  void write(in DOMString text);
  void writeln(in DOMString text);
  NodeList getElementsByName(in DOMString elementName);
};
```

Need to define those members.

2.1.3.4. 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 returns the value of the content attribute, resolved to an absolute URI, and on setting, sets 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 is 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 is first 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 returns true if the attribute is set, and false if it is absent. On setting, the content attribute is removed if the DOM attribute is set to false, and is set to have the same value as its name if the DOM attribute is set to true.

If a reflecting DOM attribute is a numeric type (`long`) then the content attribute must be converted to a numeric type first (truncating any fractional part). If that fails, or if the attribute is absent, the default value should be returned instead, or 0 if there is no default value. On setting, the given value is converted to a string representing the number in base ten and then that string should be used as the new content attribute value.

2.1.3.5. The `textContent` attribute

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=""?

2.1.4. Kinds of elements

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

- Metadata elements
- Sectioning elements
- Block-level elements
- Strictly inline-level content
- Structured inline-level elements

- Interactive elements
- Form control elements

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

2.1.4.1. Block-level elements

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

There are several kinds of block-level elements:

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

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

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

2.1.4.2. Inline-level content

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

Inline-level content comes in various types:

Strictly inline-level content

Text, embedded content, and elements that annotate the text without introducing structural grouping. For example: `a`, `i`, `noscript`. Elements used in contexts allowing only strictly inline-level content must not contain anything other than strictly inline-level content.

Structured inline-level elements

Block-level elements that can also be used as inline-level content. For

example: `ol`, `blockquote`, `table`.

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

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

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

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

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

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

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

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

```
  <style>
   /* This example is illegal. */
  </style>
 </li>
</ol>
```

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

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

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

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

2.1.4.4. Interactive elements

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

When activation is performed via some method other than clicking the pointing device, the default action of the event that triggers the activation must, instead of being activating the element directly, be the dispatching of a new event, `click`, on the same element, with the mouse-specific fields (`button`, `screenX`, etc) set to zero, and the key fields set according to the current state of the key input device, if any (false for any keys that are not available). [DOM3EVENTS]

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

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

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

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

2.1.5. Global attributes

User agents must support the following common attributes on all elements in the HTML namespace (including elements that are not defined to exist by this specification).

`id`

The element's unique identifier. The value must be unique in the document 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 (e.g. for selectors in CSS or for the `getElementById()` method in the DOM).

Identifiers are opaque strings. Particular meanings should not be derived from the value of the `id` attribute.

When an element has an ID set through multiple methods (for example, if it has both `id` and `xml:id` attributes simultaneously [XMLID]), then the element has multiple identifiers. User agents must use all of an HTML element's identifiers (including those that are in error according to their relevant specification) for the purposes of ID matching.

`title`

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 caption or a description of the image; on a paragraph, it could be a footnote or commentary on the text; on a citation, it could be further information about the source; and so forth. The value is text.

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

Note: The `link`, `style`, `abbr`, and `dfn` elements define their own `title` attributes instead of using the global `title` attribute.

`lang` (HTML only) and `xml:lang` (XML only)

The primary language for the element's contents and for any of the element's attributes that contain text. The value must be a valid RFC 3066 language code, or the empty string. RFC3066

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

The `lang` attribute only applies to HTML documents. Authors must not use the `lang` attribute in XML documents. Authors must instead use the `xml:lang` attribute, defined in XML. [XML]

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

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

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

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

`dir`

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

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

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

`class`

The element's classes. The value must be a list of zero or more words (consisting of one or more non-space characters) separated by one or more spaces.

User agents must assign all the given classes to the element, for the purposes of class matching (e.g. for selectors in CSS or for the `getElementsByClassName()` method in the DOM).

Unless defined by one of the URIs given in the `profile` attribute, classes are opaque strings. Particular meanings must not be derived from undefined values in the `class` attribute.

Authors should bear in mind that using the `class` attribute does not convey any additional meaning to the element (unless using classes defined by a `profile`). There is no semantic difference between an element *with* a class attribute and one *without*. Authors that use classes that are not defined in a `profile` should make sure, therefore, that their documents make as much sense once all `class` attributes have been removed as they do with the attributes present.

Event handler attributes aren't handled yet.

The following DOM interface, common to elements in the HTML namespace, provides scripts with convenient access to the content attributes listed above:

```
interface HTMLElement : Element {
           attribute DOMString id;
           attribute DOMString title;
           attribute DOMString lang;
           attribute DOMString dir;
           attribute DOMString className;
};
```

The **`id`** attribute must reflect the content `id` attribute.

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

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

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

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

should also introduce a DOMTokenString accessor for the class attribute

2.1.6. The `html` element

Contexts in which this element may be used:

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

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

2.2. Document metadata

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

2.2.1. The `head` element

Contexts in which this element may be used:

As the first element in an `html` element.

Content model:

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

Element-specific attributes:

`profile` (optional)

DOM interface:

```
interface HTMLHeadElement : HTMLElement {
           attribute DOMString profile;
};
```

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

The **`profile`** attribute must, if specified, contain a list of zero or more URIs (or IRIs) representing definitions of classes, metadata names, and link relations. These URIs are opaque strings, like namespaces; user agents are not expected to determine any useful information from the resources that they reference.

Each time a class, metadata, or link relationship name that is not defined by this specification is found in a document, the UA must check whether any of the URIs in the `profile` attribute are known (to the UA) to define that name. The class, metadata, or link relationship shall then be interpreted using the semantics given by the first URI that is known to define the name. If the name is not defined by this specification and none of the specified URIs defines the name either, then the class, metadata, or link relationship is meaningless and the UA must not assign special meaning to that name.

If two profiles define the same name, then the semantic is given by the first URI specified in the `profile` attribute. There is no way to use the names from both profiles in one document.

User agents must ignore all the URIs given in the `profile` attribute that follow a URI that the UA does not recognise. (Otherwise, if a name is defined in two profiles, UAs would assign meanings to the document differently based on which profiles they supported.)

> ***Note: If a profile's definition introduces new definitions over time, documents that use multiple profiles can change defined meaning over time. So as to avoid this problem, authors are encouraged to avoid using multiple profiles.***

The **`profile`** DOM attribute must reflect the `profile` content attribute on getting and setting.

2.2.2. The `title` element

Metadata element.

Contexts in which this element may be used:

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

Content model:

Text (for details, see prose).

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

In HTML (as opposed to XHTML), the `title` element must not contain content other than text and entities; user agents must parse the element so that entities are recognised and processed, but all other markup is interpreted as literal text.

In XHTML, the `title` element must not contain any elements.

User agents must concatenate the contents of all the text nodes and CDATA nodes that are direct children of the `title` element (ignoring any other nodes such as comments or elements), in tree order, to get the string to use as the document's title. User agents should use the document's title when referring to the document in their user interface.

2.2.3. The `base` element

Metadata element.

Contexts in which this element may be used:

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

Content model:

Empty.

Element-specific attributes:

`href` (optional)

DOM interface:

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

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

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

User agents must use the value of the `href` attribute on the first `base` element in the document as the document entity's base URI for the purposes of section 5.1.1 of RFC 2396 ("Establishing a Base URI": "Base URI within Document Content"). [RFC2396] User agents must then follow the rules given by XML Base to resolve relative URIs in HTML and XHTML fragments. [XMLBASE] Note that the base URI from RFC 2396 is referred to by the algorithm given in XML Base.

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

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

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

2.2.4. The `link` element

Metadata element.

Contexts in which this element may be used:

In a `head` element.

Content model:

Empty.

Element-specific attributes:

`href` (optional)
`rel` (optional)
`media` (optional)
`hreflang` (optional)
`type` (optional)
`title` (optional)

DOM interface:

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

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

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

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

The type of link indicated (the relationship) is given by the value of the **`rel`** attribute. The allowed values and their meanings are defined in a later section. If the `rel` attribute is absent, or if the value used is not allowed according to the definitions in this specification, then the element does not define a link.

Two categories of links can be created using the `link` element. **Links to external resources** are links to resources that are to be used to augment the current document, and **hyperlinks** are links to other documents. The link types section defines whether a particular link type is an external resource or a hyperlink. One element can create multiple links (of which some might be external resource links and some might be hyperlinks). User agents should process the links on a per-link basis, not a per-element basis.

The exact behaviour for links to external resources depends on the exact relationship, as defined for the relevant link type. Some of the attributes control whether or not the external resource is to be applied (as defined below). For

external resources that are represented in the DOM (for example, style sheets), the DOM representation must be made available even if the resource is not applied. (However, user agents may opt to only fetch such resources when they are needed, instead of pro-actively downloading all the external resources that are not applied.)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Some versions of HTTP defined a `Link:` header, to be processed like a series of `link` elements. When processing links, those must be taken into consideration as well. For the purposes of ordering, links defined by HTTP headers must be assumed to come before any links in the document, in the order that they were given in the HTTP entity header. Relative URIs in these headers must be resolved according to the rules given in HTTP, not relative to base URIs set by the document (e.g. using a `base` element or `xml:base` attributes). [RFC2616]

[RFC2068]

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

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

2.2.5. The meta element

Metadata element.

Contexts in which this element may be used:

In a head element.

Content model:

Empty.

Element-specific attributes:

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

DOM interface:

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

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

To set metadata with meta elements, authors must first specify a profile that defines metadata names, using the profile attribute. The value of the name attribute must be defined by one of the profiles, and the value of the content attribute must conform to the syntax given by the profile.

How user agents handle metadata set in this way depends on the definitions of the profiles involved.

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

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

2.2.5.1. Specifying and establishing the document's character encoding

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

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

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).

For HTML, user agents must use the following algorithm in determining the character encoding of a document:

1. If the transport layer specifies an encoding, use that.

2. Otherwise, if the user agent can find a `meta` element that specifies character encoding information (as described above), then use that.

3. Otherwise, if the user agent can autodetect the character encoding from applying frequency analysis or other algorithms to the data stream, then use that.

4. Otherwise, use an implementation-defined or user-specified default character encoding (`ISO-8859-1`, `windows-1252`, and `UTF-8`) are recommended as defaults, and can in many cases be identified by inspection as they have different ranges of valid bytes).

For XML documents, the algorithm user agents must use to determine the character encoding is given by the XML specification. [XML]

2.2.6. The `style` element

Metadata element.

Contexts in which this element may be used:

In a `head` element.

Content model:

Depends on the value of the `type` attribute.

Element-specific attributes:

`type` (optional)
`media` (optional)
`title` (optional)

DOM interface:

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

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

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

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

value must be a valid MIME type, optionally with parameters. [RFC2046]

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

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

The **`media`** attribute says which media the styles apply to. The value must be a valid media query. [MQ] User agents must only apply the styles to views while their state match the listed media.

The default, if the `media` attribute is omitted, is `all`, meaning that by default

styles apply to all media.

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

For styling languages that consist of pure text, user agents must use a concatenation of the contents of all the text nodes and CDATA nodes that are direct children of the `style` element (ignoring any other nodes such as comments or elements), in tree order. For XML-based styling languages, user agents must use all the children nodes of the `style` element as the style.

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

The DOM **`disabled`** attribute behaves as defined for the alternate stylesheets DOM.

2.3. Sections

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

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

2.3.1. The `body` element

Sectioning element.

Contexts in which this element may be used:

As the second element in an `html` element.

Content model:

Zero or more block-level elements.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

Note: Some DOM operations (for example, parts of the drag and drop model) are defined in terms of "the `body` element". See the definition of the `document.body` DOM attribute for details.

2.3.2. The `section` element

Sectioning block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more block-level elements.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

2.3.3. The `nav` element

Sectioning block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

2.3.4. The `article` element

Sectioning block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Zero or more block-level elements.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

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

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

2.3.5. The `blockquote` element

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

Contexts in which this element may be used:

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

Content model:

Zero or more block-level elements.

Element-specific attributes:

`cite` (optional)

DOM interface:

```
interface HTMLQuoteElement : HTMLElement {
           attribute DOMString cite;
};
```

Note: The `HTMLQuoteElement` interface is also used by the `q` element.

The `blockquote` element represents a section that is quoted from another source.

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

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

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

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

> The `blockquote` element can be used with the `ol` and `cite` elements to mark up dialogue. This example demonstrates this using an extract from Abbot and Costello's famous sketch, *Who's on first*:

```
<ol>
 <li> <cite>Costello</cite>
      <blockquote> <p> Look, you gotta first baseman? </p> </blockq
 <li> <cite>Abbott</cite>
      <blockquote> <p> Certainly. </p> </blockquote>
 <li> <cite>Costello</cite>
      <blockquote> <p> Who's playing first? </p> </blockquote>
 <li> <cite>Abbott</cite>
      <blockquote> <p> That's right. </p> </blockquote>
 <li> <cite>Costello</cite>
      <blockquote> <p> When you pay off the first baseman every mon
 <li> <cite>Abbott</cite>
      <blockquote> <p> Every dollar of it. </p> </blockquote>
</ol>
```

2.3.6. The `aside` element

Sectioning block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

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

Block-level elements.

Contexts in which these elements may be used:

Where block-level elements are expected.

Content model:

Significant strictly inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

These elements define headers for their sections.

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

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

These elements must not be empty.

2.3.8. The `header` element

Block-level element.

Contexts in which this element may be used:

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

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

descendant.

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

Other heading elements indicate subheadings or subtitles.

> Here are some examples of valid headers. In each case, the emphasised text represents the text that would be used as the header in an application extracting header data and ignoring subheadings.
>
> ```
> <header>
>  <h1>The reality dysfunction</h1>
>  <h2>Space is not the only void</h2>
> </header>
>
> <header>
>  Welcome to...
>  <h1>Voidwars!</h1>
> </header>
>
> <header>
>  <h1>Scalable Vector Graphics (SVG) 1.2</h1>
>  <h2>W3C Working Draft 27 October 2004</h2>
>  <dl>
>   <dt>This version:</dt>
>   <dd><a href="http://www.w3.org/TR/2004/WD-SVG12-20041027/">http:/
>   <dt>Previous version:</dt>
>   <dd><a href="http://www.w3.org/TR/2004/WD-SVG12-20040510/">http:/
>   <dt>Latest version of SVG 1.2:</dt>
>   <dd><a href="http://www.w3.org/TR/SVG12/">http://www.w3.org/TR/SV
>   <dt>Latest SVG Recommendation:</dt>
>   <dd><a href="http://www.w3.org/TR/SVG/">http://www.w3.org/TR/SVG/
>   <dt>Editor:</dt>
>   <dd>Dean Jackson, W3C, <<a href="mailto:dean@w3.org">dean@w3.org<
>   <dt>Authors:</dt>
>   <dd>See <a href="#authors">Author List</a></dd>
>  </dl>
>  <a href="http://www.w3.org/Consortium/Legal/i
> </header>
> ```

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

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

2.3.9. The `footer` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

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

2.3.10. The `address` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

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

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

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

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

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

2.3.11. Headings and sections

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

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

Sectioning elements other than blockquote are always considered subsections of their nearest ancestor sectioning element, regardless of what implied

sections other headings may have created. However, `blockquote` elements *are* associated with implied sections. Effectively, `blockquote` elements act like sections on the inside, and act opaquely on the outside.

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

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

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

> For example, the following is correct:
>
> ```
> <body>
>  <h4>Apples</h4>
>  Apples are fruit.
>  <section>
>   <h2>Taste</h2>
>   They taste lovely.
> ```

```
  <h6>Sweet</h6>
  <p>Red apples are sweeter than green ones.</p>
  <h1>Colour</h1>
  <p>Apples come in various colours.</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>Colour</h2>
  <p>Apples come in various colours.</p>
 </section>
</body>
```

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

2.3.11.1. Creating an outline

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

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

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

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

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

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

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

Otherwise:

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

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

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

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

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

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

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

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

    // Skip any descendants of headings.
    if (n.parentNode && n.parentNode.namespaceURI == 'http://www.w3.o
        (n.parentNode.localName == 'h1' || n.parentNode.localName ==
         n.parentNode.localName == 'h3' || n.parentNode.localName ==
         n.parentNode.localName == 'h5' || n.parentNode.localName ==
         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 == 'navigation' ||
         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;
  }
```

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

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

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

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

3. If the node is an `h1`-`h6` element or a `header` element, then the associated heading is the element itself. The UA must then generate the hypothetical

section tree described in the previous section, rooted at the nearest section ancestor (or the root element if there is no such ancestor). If the parent of the heading in that hypothetical tree is an element in the real document tree, then that element is the associated section. Otherwise, there is no associated section element.

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

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

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

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

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

9. Repeat these steps (which effectively goes backwards through the node's previous siblings) until an answer is found:
    1. If *n* points to a node with no previous sibling, and *c* is null, then return the node's parent node as the answer. If the node has no parent node, return null as the answer.
    2. Otherwise, if *n* points to a node with no previous sibling, return *c* as the answer.
    3. Adjust *n* so that it points to the previous sibling of the current position.
    4. If *n* is pointing at an `h1` or `header` element, then return that element

as the answer.

5. If *n* is pointing at an h2-h6 element, and heading candidates are not being searched for, then return that element as the answer.
6. Otherwise, if *n* is pointing at an h2-h6 element, and either *c* is still null, or *c* is a heading of lower rank than this one, then set *c* to be this element, and continue going backwards through the previous siblings.
7. If *n* is pointing at a sectioning element, then from this point on top-level heading candidates are being searched for. (Specifically, we are looking for the nearest top-level header for the current section.) Continue going backwards through the previous siblings.

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

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

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

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

> For the following fragment:
>
> ```
> <body>
>  <h1>X</h1>
>  <h2>X</h2>
>  <blockquote>
>   <h3>X</h3>
>  </blockquote>
>  X
>  <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>

2.4. Paragraphs

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

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

2.4.1. The `p` element

Block-level element.

Contexts in which this element may be used:

Where block-level elements are expected.

Content model:

Significant inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

The `p` element represents a paragraph.

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

> The following examples are conforming HTML fragments:
>
> ```
> The little kitten gently seated himself on a piece of
> carpet. Later in his life, this would be referred to as the time th
> cat sat on the mat.
>
> <fieldset>
>  <legend>Personal information</legend>
>  
>    <label>Name: <input name="n"></label>
>    <label><input name="anon" type="checkbox"> Hide from other users
>  
>  <label>Address: <textarea name="a"></textarea></label>
> </fieldset>
>
> There was once an example from Femley,<br>
> Whose markup was of dubious quality.<br>
> The validator complained,<br>
> So the author was pained,<br>
> To move the error from the markup to the rhyming.
> ```

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

> The following example is technically correct:
>
> ```
> <section>
>  <!-- ... -->
>  Last modified: 2001-04-23
>  Author: fred@example.com
> </section>
> ```
>
> However, it would be better marked-up as:
>
> ```
> <section>
>  <!-- ... -->
>  <footer>Last modified: 2001-04-23</footer>
>  <address>Author: fred@example.com</address>
> </section>
> ```
>
> Or:
>
> ```
> <section>
>  <!-- ... -->
>  <footer>
> ```

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

2.4.2. The `hr` element

thematic separator. break. transition. hinge realignment. reconstruction, refinement, remodeling, reversal, revision, revolution. Maybe an 'html respite' or a 'hypertext rest'? .

2.5. Preformatted text

2.5.1. The `pre` element

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

Contexts in which this element may be used:

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

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

Some examples of cases where the `pre` element could be used:

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

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

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

2.6. Lists

2.6.1. The `ol` element

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

Contexts in which this element may be used:

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

Content model:

Zero or more `li` elements.

Element-specific attributes:

`start` (optional)

DOM interface:

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

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

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

If the `start` attribute is present, user agents must convert the value to a numeric type, truncating any fractional part, in order to determine the attribute's value. The default value, used if the attribute is missing or if the value cannot be converted to a number according to the referenced algorithm, is 1.

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

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

Each subsequent item in the list has the ordinal value given by its `value` attribute, if it has one, or, if it doesn't, the ordinal value of the previous item, plus one.

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

2.6.2. The `ul` element

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

Contexts in which this element may be used:

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

Content model:

Zero or more `li` elements.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

2.6.3. The `li` element

Contexts in which this element may be used:

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

Content model:

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

Element-specific attributes:

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

DOM interface:

```
interface HTMLLIElement : HTMLElement {
           attribute long value;
};
```

The `li` element represents a list item.

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

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

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

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

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

If the `value` attribute is present, user agents must convert the value to a numeric type, truncating any fractional part, in order to determine the attribute's value. If the attribute's value cannot be converted to a number, it is treated as if the attribute was absent. The attribute has no default value.

The `value` attribute is processed by the parent `ol` element, if there is one. If there is not, the attribute has no effect.

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

2.6.4. The `dl` element

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

Contexts in which this element may be used:

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

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

> The following are all conforming HTML fragments.
>
> In the following example, one entry ("Authors") is linked to two values ("John" and "Luke").
>
> ```
> <dl>
>  <dt> Authors
>  <dd> John
>  <dd> Luke
>  <dt> Editor
>  <dd> Frank
> </dl>
> ```

In the following example, one definition is linked to two terms.

```
<dl>
 <dt lang="en-US"> color </dt>
 <dt lang="en-GB"> colour </dt>
 <dd> A sensation which (in humans) derives from the ability of
 the fine structure of the eye to distinguish three differently
 filtered analyses of a view. </dd>
</dl>
```

The following example illustrates the use of the `dl` element to mark up metadata of sorts. At the end of the example, one group has two metadata labels ("Authors" and "Editors") and two values ("Robert Rothman" and "Daniel Jackson").

```
<dl>
 <dt> Last modified time </dt>
 <dd> 2004-12-23T23:33Z </dd>
 <dt> Recommended update interval </dt>
 <dd> 60s </dd>
 <dt> Authors </dt>
 <dt> Editors </dt>
 <dd> Robert Rothman </dd>
 <dd> Daniel Jackson </dd>
</dl>
```

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

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

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

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

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

2.6.5. The `dt` element

Contexts in which this element may be used:

Before `dd` elements inside `dl` elements.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

The `dt` element represents the term, or name, part of a name-value group in a `dl` element.

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

2.6.6. The `dd` element

Contexts in which this element may be used:

After `dt` elements inside `dl` elements.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

The `dd` element represents the definition, or value, part of a name-value group in a `dl` element.

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

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

2.7. Phrase elements

2.7.1. The `a` element

Interactive, strictly inline-level content.

Contexts in which this element may be used:

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

Content model:

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

Element-specific attributes:

`href` (optional)
`rel` (optional)
`media` (optional)
`hreflang` (optional)
`type` (optional)

DOM interface:

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

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

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

> If a site uses a consistent navigation toolbar on every page, then the link that would normally link to the page itself could be marked up using an `a` element:
>
> ```
> <nav>
>  <ul>
>   <li> <a href="/">Home</a> </li>
>   <li> <a href="/news">News</a> </li>
>   <li> <a>Examples</a> </li>
> ```

```
      <li> <a href="/legal">Legal</a> </li>
     </ul>
    </navigation>
```

The `href` attribute, if present, must have a value that is a URI (or IRI).

The relationship between the document containing the hyperlink and the destination resource indicated by the hyperlink is given by the value of the **`rel`** attribute. The allowed values and their meanings are defined in a later section. The `rel` attribute has no default value. If the attribute is omitted or if none of the values in the attribute are recognised by the UA, then the document has no particular relationship with the destination resource other than there being a hyperlink between the two.

Interactive user agents should allow users to follow hyperlinks created using the `a` element. The `rel`, `media`, `hreflang`, and `type` attributes may be used to indicate to the user the likely nature of the target resource.

The **`media`** attribute describes for which media the target document was designed. It is purely advisory. The value must be a valid media query. [MQ] The default, if the `media` attribute is omitted or has an invalid value, is `all`.

The **`hreflang`** attribute, if present, gives the language of the linked resource. It is purely advisory. The value must be a valid RFC 3066 language code. RFC3066 User agents must not consider this attribute authoritative — upon fetching the resource, user agents must only use language information associated with the resource to determine its language, not metadata included in the link to the resource.

The **`type`** attribute, if present, gives the MIME type of the linked resource. It is purely advisory. The value must be a valid MIME type, optionally with parameters. [RFC2046] User agents must not consider the `type` attribute authoritative — upon fetching the resource, user agents must only use the Content-Type information associated with the resource to determine its type, not metadata included in the link to the resource.

The `a` element must not be empty.

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

2.7.2. The `em` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

> These examples show how changing the emphasis changes the meaning. First, a general statement of fact, with no emphasis:
>
> ```
> Cats are cute animals.
> ```
>
> 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):
>
> ```
> Cats are cute animals.
> ```
>
> 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):
>
> ```
> Cats are cute animals.
> ```
>
> By moving it to the adjective, the exact nature of the the cats is reasserted (maybe someone suggested cats were *mean* animals):
>
> ```
> Cats are cute animals.
> ```
>
> Similarly, if someone asserted that cats were vegetables, someone correcting this might emphasise the last word:
>
> ```
> Cats are cute animals.
> ```
>
> 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>
```

2.7.3. The `strong` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

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

```
<p><strong>Warning.</strong> This dungeon is dangerous.
<strong>Avoid the ducks.</strong> Take any gold you find.
<strong><strong>Do not take any of the diamonds</strong>,
they are explosive and <strong>will destroy anything within
ten meters.</strong></strong> You have been warned.</p>
```

2.7.4. The `small` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

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

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

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

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

```
<p><strong><small>Continued use of this service will result in a ki
```

2.7.5. The `m` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

Should we just repurpose `u` or `b` for this semantic instead? What would they stand for?

In the following snippet, a paragraph of text refers to a specific part of a code fragment.

```
<p>The highlighted part below is where the error lies:</p>
<pre><code>var i: Integer;
begin
   i := <m>1.1</m>;
end.</code></pre>
```

Another example of the `m` element is highlighting parts of a document that are matching some search string. If someone looked at a document, and the server knew that the user was searching for the word "kitten", then the server might return the document with one paragraph modified as follows:

```
<p>I also have some <m>kitten</m>s who are visiting me
these days. They're really cute. I think they like my garden!</p>
```

2.7.6. The `abbr` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

`title` (optional)

DOM interface:

No difference from `HTMLElement`.

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

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

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

> In the example below, the word "Zat" is used as an abbreviation in the second paragraph. The abbreviation is defined in the first, so the explanatory `title` attribute has been omitted. Because of the way `dfn` elements are defined, the second `abbr` element in this example would be connected (in some UA-specific way) to the first.
>
> ```
> The <dfn><abbr>Zat</abbr></dfn>, short for Zat'ni'catel, is a we
> Jack used a <abbr>Zat</abbr> to make the boxes of evidence disap
> ```

2.7.7. The `dfn` element

Strictly inline-level content.

Contexts in which this element may be used:

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

Content model:

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

Element-specific attributes:

`title` (optional)

DOM interface:

No difference from `HTMLElement`.

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

definition for the term given by the contents of the `dfn` element.

`dfn` elements must not be nested.

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

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

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

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

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

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

2.7.8. The `i` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

The `i` element represents an instance of the use of a term, such as a taxonomic designation, technical term, an idiomatic phrase from another language, or similar.

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

> The examples below show uses of the `i` element:
>
> ```
> The felis silvestris catus is cute.
> The block-level elements are defined above.
> There is a certain je ne sais quoi in the air.<
> ```

Note: The `i` element is not appropriate for marking up names (e.g. of people, or of ships).

2.7.9. The `code` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

2.7.10. The `var` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

> In the paragraph below, the letter "n" is being used as a variable in prose:
>
> ```
> If there are <var>n</var> pipes leading to the ice
> cream factory then I expect at least <var>n</var>
> flavours of ice cream to be available for purchase!
> ```

2.7.11. The `samp` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

```
<p>The computer said <samp>Too much cheese in tray
two</samp> but I didn't know what that meant.</p>
```

This second example shows a block of sample output. Nested `samp` and `kbd` elements allow for the styling of specific elements of the sample output using a stylesheet.

```
<pre><samp><samp class="prompt">jdoe@mowmow:~$</samp> <kbd>ssh demo
Last login: Tue Apr 12 09:10:17 2005 from mowmow.example.com on pts
Linux demo 2.6.10-grsec+gg3+e+fhs6b+nfs+gr0501+++p3+c4a+gr2b-reslog

<samp class="prompt">jdoe@mowmow:~$</samp> <samp class="cursor">_</
```

2.7.12. The `kbd` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

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

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

> Here the `kbd` element is used to indicate keys to press:
>
> ```
> To make George eat an apple, press <kbd><kbd>Shift</kbd>+<kbd>F3
> ```
>
> In this second example, the user is told to pick a particular menu item. The outer `kbd` element marks up a block of input, with the inner `kbd` elements representing each individual step of the input, and the `samp` elements inside them indicating that the steps are input based on something being displayed by the system, in this case menu labels:
>
> ```
> To make George eat an apple, select
>     <kbd><kbd><samp>File</samp></kbd>|<kbd><samp>Eat Apple...</samp
> 
> ```

2.7.13. The `sup` and `sub` elements

Strictly inline-level content.

Contexts in which these elements may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

When the `sub` element is used inside a `var` element, it represents the subscript that identifies the variable in a family of variables.

```
<p>The coordinate of the <var>i</var>th point is
(<var>x<sub><var>i</var></sub></var>, <var>y<sub><var>i</var></sub>
For example, the 10th point has coordinate
(<var>x<sub>10</sub></var>, <var>y<sub>10</sub></var>).</p>
```

In certain languages, superscripts are part of the typographical conventions for some abbreviations.

```
<p>The most beautiful women are
<span lang="fr"><abbr>M<sup>lle</sup></abbr> Gwendoline</span> and
<span lang="fr"><abbr>M<sup>me</sup></abbr> Denise</span>.</p>
```

Mathematical expressions often use subscripts and superscripts. Authors are encouraged to use MathML for marking up mathematical, but authors may opt to use `sub` and `sup` if detailed mathematical markup is not desired. [MathML]

```
<var>E</var>=<var>m</var><var>c</var><sup>2</sup>

f(<var>x</var>, <var>n</var>) = log<sub>4</sub><var>x</var><sup><va
```

2.7.14. The `q` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

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

Element-specific attributes:

`cite` (optional)

DOM interface:

The `q` element uses the `HTMLQuoteElement` interface.

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

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

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

2.7.15. The `cite` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

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

This is incorrect usage:

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

This is the correct way to do it:

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

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

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

2.7.16. The `span` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

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

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

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

2.7.17. The `bdo` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Strictly inline-level content.

Element-specific attributes:

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

DOM interface:

No difference from `HTMLElement`.

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

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

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

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

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

2.7.18. The `br` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Empty.

Element-specific attributes:

None.

DOM interface:

No difference from `HTMLElement`.

The `br` element represents a line break.

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

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

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

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

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

2.8. Edits

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

2.8.1. The `ins` element

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

Contexts in which this element may be used:

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

Content model:

When the element is a child of an element with only one content model (i.e. an element that only allows strictly inline-level content, or only allows inline-level content, or only allows block-level elements): same content model as the parent element.
Otherwise, when the element is a child of an element that only contains inter-element whitespace, `ins` elements, and `del` elements: same content model as the parent element, with the additional restriction that if the parent element allows a choice in content models (e.g. block or inline) then if all the children of all the sibling `ins` elements were placed directly in the parent element, the document would still be conforming.
Otherwise, when the element is a child of an element that is being used as an inline-level content container: inline-level content.
Otherwise, when the element is a child of an element that is being used as a block-level element container: block-level elements.
Otherwise: zero or more block-level elements, or inline-level content (but not both).

Element-specific attributes:

`cite` (optional)
`datetime` (optional)

DOM interface:

Uses the `HTMLModElement` interface.

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

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

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

> The following would be syntactically legal:
>
> ```
> <aside>
>  <ins>
> ```

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

As would this:

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

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

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

2.8.2. The `del` element

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

Contexts in which this element may be used:

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

Content model:

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

Element-specific attributes:

`cite` (optional)
`datetime` (optional)

DOM interface:

Uses the `HTMLModElement` interface.

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

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

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

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

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

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

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

If the `datetime` attribute is present, it must have a value consisting of four digits representing the year, a literal hyphen, two digits representing the month, a literal hyphen, two digits representing the day, a literal T, two digits for the hour, a colon, two digits for the minutes, another colon, two digits for the seconds, optionally a decimal point followed by one or more digits for the fraction of a second, and finally either a literal Z, or, a plus sign or a minus sign followed by two digits for the hour offset, a colon, and two digits for the minute offset.

In other words: *`YYYY-MM-DDThh:mm:ss.sTZ`*

Digits must be in the range 0-9 (U+0030 to U+0039), interpreted in base ten. The hyphen must be U+002D, the T must be U+0054, the colon must be U+003A, the Z must be U+005A, the plus must be U+002B, and the minus U+002D (same as the hyphen).

To interpret this value, user agents must first check to see if the value matches the pattern described here. If it does, then the values must be extracted and interpreted as a date and time with a timezone offset, as per ISO 8601. [ISO8601]

If the attribute value does not match the format, or, if the date or time given is not a valid date and time (e.g. because the month is out of range) then the user agent must ignore the attribute (the modification has no associated timestamp).

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

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

The **`cite`** and **`datetime`** DOM attributes should reflect the elements' content attributes of the same name.

2.9. Embedded content

2.9.1. The `img` element

Strictly inline-level content.

Contexts in which this element may be used:

Where strictly inline-level content is allowed.

Content model:

Empty.

Element-specific attributes:

`src` (required)
`alt` (required)
`height` (optional)
`width` (optional)
`usemap` (optional)
`ismap` (optional)

DOM interface:

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

The `img` element represents a piece of text with an alternate graphical

representation. The text is given by the `alt` attribute, and the URI to the graphical representation of that text is given by the `src` attribute.

This section is (obviously) incomplete.

2.10. Tabular data

This section will contain definitions of the `table` element and so forth.

2.11. Forms

This section will contain definitions of the `form` element and so forth.

2.12. Scripting

2.12.1. The `script` element

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

Contexts in which this element may be used:

In a `head` element.
Where block-level elements are expected.
Where inline-level content is expected.

Content model:

If there is no `src` attribute, depends on the value of the `type` attribute.
If there *is* a `src` attribute, the element must be empty.

Element-specific attributes:

`src` (optional)
`type` (optional)

DOM interface:

```
interface HTMLScriptElement : HTMLElement {
           attribute DOMString text;
           attribute DOMString src;
           attribute DOMString type;
};
```

The `script` element allows authors to include dynamic script in their documents.

When the **src** attribute is set, the `script` element refers to an external file, which must (if it uses a supported scripting language) be downloaded and executed. The user agent must delay the execution of other scripts associated with the page that are invoked during the download (e.g. event handlers) until after the external script has been downloaded and executed.

The language of the script is given by the **type** attribute. The value must be a valid MIME type, optionally with parameters. [RFC2046]

For `script` elements that have the `src` attribute set, 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 as a scripting language, then the UA may opt not to download the script.

User agents must not consider the `type` attribute authoritative, however — upon fetching the script, user agents must only use the Content-Type information associated with it to determine whether or not to execute it; user agents must not use the `type` attribute in the document to determine the actual type of the script.

If the `type` attribute is omitted but the `src` attribute is set, then the UA must fetch the resource to determine its type and thus determine if it supports (and can execute) that external script.

If the `src` attribute is not set, then the script is given by the contents of the element. The language is given by the `type` attribute. If it is omitted, then the default is the ECMAScript MIME type.

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.

User agents that support scripting must execute scripts (written in languages that they support) immediately upon parsing a `script` element's end tag, and immediately upon having a dynamically created `script` element inserted into the DOM. Once a `script` element has been executed, it must be flagged as such and never re-executed again. When an element with this flag set is cloned, the new element must not have the flag set.

For scripting languages that consist of pure text, user agents must use the value of the DOM `text` attribute (defined below) as the script to execute. For XML-based scripting languages, user agents must use all the children nodes of the `script` element as the script.

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

The DOM attribute **`text`** must return a concatenation of the contents of all the text nodes and CDATA nodes that are direct children of the `script` element (ignoring any other nodes such as comments or elements), in tree order. On setting, it must act the same way as the `textContent` DOM attribute.

2.12.1.1. 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]

2.12.2. The `noscript` element

The `noscript` element needs to be defined too.

2.13. Other new elements

all the new things in WA1: menu, calendar, card, canvas, switch, gauge, progress, datagrid, datatree, switch, etc

2.14. Notes (draft sections to be moved elsewhere)

2.14.1. Classes

This section may somehow introduce some predefined classes with actual semantic meanings; possibly by defining a profile.

2.14.2. Link types

This section might at some future point list a small set of link `rel`ationship types and more exactly define their semantics than HTML4. This section (or indeed this specification in general) is unlikely to specify anything related to the `profile` attribute and how to extend the link types in HTML. Work in this area is currently being done by GMPG and others.

2.14.3. Document sections

User agents must support all of the common attributes and event handlers on the `section` element, as well as the `active` attribute (for use with mutually exclusive sections).

In CSS-aware user agents, the default presentation of this element should be achieved by including the following rules, or their equivalent, in the UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|section { display: block; margin: 1em 0; }
```

2.14.4. Section headers

For `h1` elements, CSS-aware visual user agents should derive the size of the header from the level of `section` nesting. This effect should be achieved by including the following rules, or their equivalent, in the UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|section xh|h1 { /* same styles as h2 */ }
xh|section xh|section xh|h1 { /* same styles as h4 */ }
xh|section xh|section xh|section xh|h1 { /* same styles as h4 */ }
xh|section xh|section xh|section xh|section xh|h1 { /* same styles as h5
xh|section xh|section xh|section xh|section xh|section xh|h1 { /* same s
```

Authors should use `h1` elements to denote headers in sections. Authors may instead use `h2` ... `h6` elements, for backwards compatibility with user agents that do not support `section` elements.

2.14.5. Section groups (tabs)

A group of related, order-neutral sections may be denoted using the **`tabbox`** element. The default presentation in a visual media (as described below) is to render each section as a separate tab in a tab box, allowing the user to switch between them. Sections can also be represented by links to other documents, instead of them being included literally in the markup.

The `tabbox` element is a block-level element that should only contain `section`, `fieldset`, and `a` elements.

Authors should only use `a` elements that cause the user agent to change the active page to a page with a similar structure. Other behaviours are likely to be highly confusing to users.

Each `section`, `fieldset`, and `a` child can have a title. If the element is a `section` element, then the title is taken from the `title` attribute of the element, if specified, or, if absent, from the `textContent` DOM attribute of the first element child of the `section` element, if that is an `h1` ... `h6` element. (If it is taken from a

header child, then that child is hidden from the rendering.) If the element is a `fieldset` element, then the title is taken from the the `textContent` DOM attribute of the first element child of the `fieldset` element, if that is an `legend` element. If the element is an `a` element, then the title is taken from the `textContent` DOM attribute of the element. (Titles may be the empty string.)

The titles obtained in this way, and the `section`, `fieldset`, and `a` elements from which they were derived, represent the list of sections in the `tabbox`. This list is *live*, in that dynamic changes to the DOM immediately affect the representation of the `tabbox` element.

All the other child nodes of the `tabbox` shall be ignored for the purposes of rendering the `tabbox`. Authors may use this in order to obtain acceptable renderings even in UAs that do not support `tabbox`.

In CSS-aware user agents, the default presentation of the `tabbox` element should, in part, be achieved by including the following rules, or their equivalent, in the UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|tabbox { display: block; }
xh|tabbox > xh|section:not([title]) > xh|h1:first-child,
xh|tabbox > xh|section:not([title]) > xh|h2:first-child,
xh|tabbox > xh|section:not([title]) > xh|h3:first-child,
xh|tabbox > xh|section:not([title]) > xh|h4:first-child,
xh|tabbox > xh|section:not([title]) > xh|h5:first-child,
xh|tabbox > xh|section:not([title]) > xh|h6:first-child,
xh|tabbox > xh|fieldset > xh|legend:first-child { display: none; }
```

These rules do not come even close to fully describing the full behaviour of a `tabbox` element, however.

The behaviour of the `tabbox` should be to provide quick access to any of the children of the `tabbox` that have a title (as described above). UAs may keep track of which section is the selected section, and report this information to the user.

When the user specifies a section to access, the relevant element must have a `click` event dispatched to it, whose default action is to further dispatch a `DOMActivate` event to the element.

For `section` and `fieldset` elements, the default action of `DOMActivate` events is to display, or jump to, the relevant section. For `a` elements, the default action is the normal default action for `a` elements (activating the link, command, or whatever). In addition to these default actions, when a child of a `tabbox` is accessed, it becomes the selected section.

If the `DOMActivate` event is canceled (or if the `click` event is canceled, causing

the `DOMActivate` event to never be fired in the first place), then the selected section does not change.

If an `a` element has a `command` attribute, it can be disabled. In such cases, the UA should not allow the user to select that section.

The initially selected section shall be the first element from the `tabbox` element's child list that is:

1. an `a` element whose `href` attribute matches the URI of the current document, if there is one,
2. otherwise, the first `a` element whose `href` attribute matches the URI given by the `href` attribute of the first `link` element in the document that has a `rel` attribute whose value contains the keyword `up` (treating that attribute as a space-separated list), if there is one,
3. otherwise, the first `section` or `fieldset` element that has a title, if there is one.

If no elements match, then initially no section shall be selected.

In the above algorithm, URI comparisons should be done after canonicalisation, and should ignore fragment identifiers unless the `a` element in question has one.

In non-interactive or non-spatial media (such as in print, on braille systems, or with speech synthesis) the UA may automatically switch the selected section to the next section once the selected section has been rendered.

Which section is selected if the element representing the currently selected section is dynamically removed from the document is up to the UA.

In interactive visual media, the `tabbox` element should be rendered as a tab box, with the section titles listed as the tabs, and the selected section (if it is a `section` or `fieldset` element) displayed in the tab panel area. When the selected section is an `a` element, the tab panel area should be empty.

This specification does not describe how CSS properties apply to `tabbox` elements when the UA uses this rendering, but the children rendered in the tab panel area must be styled using CSS, as if the tab panel area defined a new containing block and new block formatting context.

User agents must support all of the common attributes and event handlers on the `tabbox` element.

> Here is an example of a `tabbox` used to allow the user to read three different parts of the document:

```
<tabbox>
 <section>
  <h2>About</h2>
  <p><img src="logo" alt=""></p>
  <p>The Application.</p>
  <p>© copyright 2004 by The First Team.</p>
 </section>
 <section>
  <h2>Credits</h2>
  <ul>
   <li>Jack O'Neill</li>
   <li>Samantha Carter</li>
   <li>Daniel Jackson</li>
   <li>Teal'c</li>
   <li>Jonas Quinn</li>
  </ul>
 </section>
</tabbox>
```

Next, an example of a form that has been split into little groups of controls:

```
<tabbox>
 <fieldset>
  <legend>Identity</legend>
  <p><label>First name: <input name="fn"></label></p>
  <p><label>Last name: <input name="ln"></label></p>
  <p><label>Date of Birth: <input name="dob" type="date"></label></
 </fieldset>
 <fieldset>
  <legend>Food</legend>
  <p><label>Favourite appetizer: <input name="fa"></label></p>
  <p><label>Favourite meal: <input name="fm"></label></p>
  <p><label>Favourite desert: <input name="fd"></label></p>
 </fieldset>
</tabbox>
```

Finally, an example of a page using a `tabbox` to point to sections outside the document. Note the use of fallback content (elements and text in the `tabbox` element that are not `fieldset`, `section`, or `a` elements) for backwards compatibility.

```
<div>
 <tabbox>
  <strong>Navigation:</strong>
  <a href="/"><span>Home</span></a>,
  <a href="/news/"><span>News</span></a>,
  <a href="/games/"><span>Games</span></a>,
  <a href="/help/"><span>Help</span></a>,
  <a href="/contact/"><span>Contact</span></a>.
 </tabbox>
</div>
```

This would be semantically equivalent to the following:

```
<tabbox>
 <section><h2>Home</h2> ...content... </section>
 <section><h2>News</h2> ...content... </section>
 <section><h2>Games</h2> ...content... </section>
 <section><h2>Help</h2> ...content... </section>
 <section><h2>Contact</h2> ...content... </section>
</tabbox>
```

2.14.6. Mutually exclusive sections

The **switch** element represents a block of mutually exclusive sections.

> For example, in an application for an online mutiplayer game, there could be four mutually exclusive sections: one for the login page, one for the network status page displayed while the user is logging in, one for a "lobby" where players get together to organise a game, and one for the actual game. The different sections are the various states that the application can reach.

The `switch` element must contain only block-level elements. User agents must support all of the common attributes and event handlers on the `switch` element.

All child elements of a `switch` element shall be hidden except those that have `active` attributes (or, for non-XHTML elements, `active` attributes in the XHTML namespace).

In CSS-aware user agents, the default presentation of this element should be achieved by including the following rules, or their equivalent, in the UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|switch { display: block; }
xh|switch xh|*:not([active]) { display: none; }
xh|switch *:not([xh|active]) { display: none; }
```

2.14.7. Using **switch** and **section**

```
interface HTMLSwitchElement : HTMLElement {
  readonly attribute Element           activeElement;
  void setActive(in Element element);
};

interface HTMLSectionElement : HTMLElement {
  readonly attribute boolean           active;
  void setActive();
};
```

...

When an element is added to a `switch` element as a child (whether during parsing, or later), the element is examined. If the element has an `active` attribute (or, if it is a non-XHTML element, if it has an `active` attribute in the XHTML namespace), or, if the `switch` element's `activeElement` DOM attribute is null, then the `switch` element's `setActive` method is called with that element as the argument. This causes the element to be made the active element for the switch, and causes any other elements to be deactivated if needed.

A side-effect of this definition is that the first element in a `switch` element is the default element if none have been explicitly marked as active.

2.15. Calendars: event data

The **`calendar`** element may be used for indicating hCalendar fragments that should be processed and rendered, e.g. as inline calendars.

The `calendar` element is a block-level element whose content model is any block-level elements. User agents must support all the common attributes and event handlers on `calendar` elements.

Web browsers should render the `calendar` element by replacing the element by a representation of the calendar data contained within it.

2.15.1. Intepreting calendar data

UAs must process the contents of `calendar` data as described in the hCalendar specification. [HCALENDAR]

2.15.2. Rendering examples

> The following fragment:
>
> ```
> <calendar>
>  
>   -//hCalendar//EN
>   2.0
>   
>    <a href="http://www.web2con.com/">
>     20041005-
>     20041007
>     Web 2.0 Conference
>    </a>
>   
>  
> </calendar>
> ```
>
> ...might render as the following:

2.16. Business cards: personal data

The **`card`** element may be used for indicating hCard fragments that should be processed and rendered, e.g. as inline business cards.

The `card` element is a block-level element whose content model is any block-level elements. User agents must support all the common attributes and event handlers on `card` elements.

Web browsers should render the `card` element by replacing the element by a representation of the personal data contained within it.

2.16.1. Intepreting card data

UAs must process the contents of `card` data as described in the hCard specification. [HCARD]

2.16.2. Rendering examples

The following fragment:

```
<card>
 <p class="vcard">
  <a class="fn n" href="http://tantek.com/">
   <span class="Given-Name">Tantek</span>
   <span class="Family-Name">Çelik</span>
  </a>
 </p>
</card>
```

...might render as the following:

2.17. Inline data

This section is a place-holder for where elements such as <date> or <time> might be defined. But it will probably be moved up to the earlier section. This might also just be merged with the "Semantics and structure of HTML elements" section above, or dropped, based on demand.

2.18. Gauges

The `gauge` element is an inline element that represents a fractional value, such as the relative relevance of a search result, the fraction of a user's quota that is used, or the fraction of a voting population to have selected a particular candidate.

User agents must support all of the common attributes and event handlers on the `gauge` element, plus the following attributes:

...

The value should come from parsing the .textContent attribute and taking the first string of digits (possibly with a single dot) as the numerator and the second string of digits (possibly with another single dot) as the denominator, defaulting the denominator to 100 if it is absent, treating zero denominators as 100, and using the resulting fraction as the value, in the range 0 to 1, for the gauge. If the numerator is absent, default to 0.

Do we want something to say that "above 0.75 is bad"? "below 0.2 is bad"?

2.19. Progress meters

Similar to gauge, but renders as a progress bar. If the numerator is absent, default to an indeterminate progress bar (barber pole, bouncing blue box, etc).

2.20. Data grids and data trees

2.20.1. The `datagrid` element

It has been suggested that instead of this flattened-row API, we should have

all the "row" arguments in the API below be arrays of integers, and instead of getParentRow(), we would have getRowCount() get the number of children that a row had. A future version of this specification will make this change, along with adding a way to detect when a row/selection has been deleted, activated, etc.

Interactive, block-level element.

Contexts in which this element may be used:

Where block-level elements are expected, if there are no ancestor interactive elements.

Content model:

Zero or more block-level elements.

Element-specific attributes:

`multiple` (optional)
`disabled` (optional)

DOM interface:

```
interface HTMLDataGridElement : HTMLElement {
        attribute DataGridDataProvider data;
        attribute SelectionRanges selection;
        attribute boolean multiple;
        attribute boolean disabled;
  void updateEverything();
  void updateRowsChanged(in long row, in long count);
  void updateRowsInserted(in long row, in long count);
  void updateRowsRemoved(in long row, in long count);
  void updateRowChanged(in long row);
  void updateColumnChanged(in long column);
  void updateCellChanged(in long row, in long column);
};
```

The `datagrid` element represents an interactive representation of tree, list, or tabular data.

The data being presented can come either from the content, as elements given as children of the `datagrid` element, or from a scripted data provider given by the `data` DOM attribute.

The `multiple` attribute, if present, must be either empty or have the literal value `multiple`. Similarly, the `disabled` attribute, if present, must be either empty or have the literal value `disabled`. (The actual values do not have any effect on how these attributes are processed, only the presence or absence of the attributes is important.)

The **multiple** and **disabled** DOM attributes reflect the multiple and disabled content attributes respectively.

2.20.1.1. The datagrid data model

This section is non-normative.

In the datagrid data model, data is structured as a set of rows representing a tree, each row being split into a number of columns. The columns are always present in the data model, although individual columns may be hidden in the presentation.

Each row can have a parent row. If a row *r* has a parent row *p*, then all the rows between it and its parent will also have a parent row, and for each row *i* between *p* and *r* the parent row of *i* will be either *p* or another row between *p* and *i*.

Rows that have other rows claiming them as their parent row are open. Rows can be closed, hiding all the data that would form child rows, but when a row is closed its child data does not appear in the data model.

The columns can have captions. Those captions are not considered a row in their own right, they are obtained separately.

Selection of data in a datagrid operates at the row level. If the multiple attribute is present, multiple rows can be selected at once, otherwise the user can only select one row at a time.

The datagrid element can be disabled entirely by setting the disabled attribute.

Columns, rows, and cells can each have specific flags, known as classes, applied to them by the data provider. These classes affect the functionality of the datagrid element, and are also passed to the style system. They are similar in concept to the class attribute, except that they are not specified on elements but are given by scripted data providers.

2.20.1.2. The data provider interface

The conformance criteria in this section apply to any implementation of the DataGridDataProvider, including (and most commonly) the content author's implementation(s).

```
// To be implemented by Web authors as a JS object
interface DataGridDataProvider {
  void initialize(in HTMLDataGridElement datagrid);
  long getRowCount();
  long getColumnCount();
  DOMString getCaptionText(in long column);
```

```
  void getCaptionClasses(in long column, in DOMTokenString classes);
  long getRowParent(in long row);
  DOMString getRowImage(in long row);
  HTMLMenuElement getRowMenu(in long row);
  void getRowClasses(in long row, in DOMTokenString classes);
  DOMString getCellData(in long row, in long column);
  void getCellClasses(in long row, in long column, in DOMTokenString cl
  void toggleRowOpenState(in long row);
  void toggleColumnSortState(in long column);
  void setCellCheckedState(in long row, in long column, in int state);
  void cycleCell(in long row, in long column);
  void editCell(in long row, in long column, in DOMString data);
}
```

The `DataGridDataProvider` interface represents the interface that objects must implement to be used as custom data views for `datagrid` elements.

Not all the methods are required. The minimum number of methods that must be implemented in a useful view is two: the `getRowCount()` and `getCellData()` methods.

Once the object is written, it must be hooked up to the `datagrid` using the **`data`** DOM attribute.

The following methods may be usefully implemented:

`initialize(`*`datagrid`*`)`

Called by the `datagrid` element (the one given by the *datagrid* argument) after it has first populated itself. This would typically be used to set the initial selection of the `datagrid` element when it is first loaded. The data provider could also use this method call to register a `select` event handler on the `datagrid` in order to monitor selection changes.

`getRowCount()`

Must return the number of rows currently in the data model, including rows that are off-screen. If the value that this method would return changes, the relevant update methods on the `datagrid` must be called first. Otherwise, this method must always return the same number.

`getColumnCount()`

Must return the number of columns currently in the data model (including columns that might be hidden). May be omitted if there is only one column. If the value that this method would return changes, the `datagrid`'s `updateEverything()` method must be called.

`getCaptionText(`*`column`*`)`

Must return the caption, or label, for column *column*. May be omitted if the columns have no captions. If the value that this method would return

changes, the `datagrid`'s `updateColumnChanged()` method must be called with the appropriate column index.

`getCaptionClasses(column, classes)`

Must add the classes that apply to column *column* to the *classes* object. May be omitted if the columns have no special classes. If the classes that this method would add changes, the `datagrid`'s `updateColumnChanged()` method must be called with the appropriate column index. Some classes have predefined meanings.

`getRowParent(row)`

Must return the index to the row that is the parent of row *row*, or a negative number if this is a top-level row. If, for a row *r*, this method returns the index of parent row *p*, then for each row *i* between *p* and *r* the method must return either *p* or another row between *p* and *i*. May be omitted if the `datagrid` is a list and not a tree. If the value that this method would return changes, the `datagrid`'s update methods must be called to update all the rows in the range that covers the old parent, the new parent, and the row in question.

`getRowImage(row)`

Must return a URI to an image that represents row *row*, or the empty string if there is no applicable image. May be omitted if no rows have associated images. If the value that this method would return changes, the `datagrid`'s update methods must be called to update the row in question.

`getRowMenu(row)`

Must return an `HTMLMenuElement` object that is to be used as a context menu for row *row*, or null if there is no particular context menu. May be omitted if none of the rows have a special context menu. As this method is called immediately before showing the menu in question, no precautions need to be taken if the return value of this method changes.

`getRowClasses(row, classes)`

Must add the classes that apply to row *row* to the *classes* object. May be omitted if the rows have no special classes. If the classes that this method would add changes, the `datagrid`'s update methods must be called to update the row in question. Some classes have predefined meanings.

`getCellData(row, column)`

Must return the value of the cell on row *row* in column *column*. For text cells, this must be the text to show for that cell. For progress bar cells, this must be either a floating point number in the range 0.0 to 1.0 (converted to a string representation), indicating the fraction of the progress bar to show as full (1.0 meaning complete), or the empty string, indicating an

indeterminate progress bar. If the value that this method would return changes, the `datagrid`'s update methods must be called to update the rows that changed. If only one cell changed, the `updateCellChanged()` method may be used.

`getCellClasses(row, column, classes)`

Must add the classes that apply to cell on row *row* in column *column* to the *classes* object. May be omitted if the cells have no special classes. If the classes that this method would add changes, the `datagrid`'s update methods must be called to update the rows or cells in question. Some classes have predefined meanings.

`toggleRowOpenState(row)`

Called by the `datagrid` when the user tries to open or close a row. When it is called on a closed row, the data provider must update its state so that the rows now include the child rows, and must call the `updateRowsInserted()` method appropriately. Similarly, when it is called on an open row, the data provider must update its state so that the rows that were shown under the row in question are now removed from the data model, and must then call the `updateRowsRemoved()` method appropriately. There is no need to tell the `datagrid` that the row itself has changed (as it should; in particular its classes should change to reflect the new open/closed state), as the `datagrid` automatically assumes that the row will need updating.

`toggleColumnSortState(column)`

Called by the `datagrid` when the user tries to sort the data using a particular column *column*. The data provider must update its state so that the rows are in the new order, and update the classes of the columns to represent the new sort status. There is no need to tell the `datagrid` that it the data has changed, as the `datagrid` automatically assumes that the entire data model will need updating.

It is the data provider's responsibility to ensure that the user's selection persists through a sort. Typically this will involve taking a note of which rows were selected before the sort (using the `getRangeStart()` and `getRangeLength()` methods of the `selection` DOM attribute, for instance), and then clearing the selection and reselecting all the rows in their new positions (e.g. using the `addRange()` method).

`setCellCheckedState(row, column, state)`

Called by the `datagrid` when the user changes the state of a checkbox cell on row *row*, column *column*. The checkbox should be toggled to the state given by *state*, which is a positive integer (1) if the checkbox is to be checked, zero (0) if it is to be unchecked, and a negative number (-1) if it

is to be set to the indeterminate state. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

`cycleCell(row, column)`

Called by the `datagrid` when the user changes the state of a cyclable cell on row *row*, column *column*. The data provider should change the state of the cell to the new state, as appropriate. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

`editCell(row, column, data)`

Called by the `datagrid` when the user edits the cell on row *row*, column *column*. The new value of the cell is given by *data*. The data provider should update the cell accordingly. There is no need to tell the `datagrid` that the cell has changed, as the `datagrid` automatically assumes that the given cell will need updating.

The following classes (for rows, columns, and cells) may be usefully used in conjunction with this interface:

Class name	Applies to	Description
`checked`	Cells	The cell has a checkbox and it is checked. (The `cyclable` and `progress` classes override this, though.)
`closed`	Rows	The row can be opened and closed, and, unless the `open` class is also present, the row is currently closed.
`cyclable`	Cells	The cell can be cycled through multiple values. (The `progress` class overrides this, though.)
`editable`	Cells	The cell can be edited. (The `cyclable`, `progress`, `checked`, `unchecked` and `indeterminate` classes override this, though.)
`header`	Rows	The row is a heading, not a data row.
`indeterminate`	Cells	The cell has a checkbox, and it can be set to an indeterminate state. If neither the `checked` nor `unchecked` classes are present, then the checkbox is in that state, too. (The `cyclable` and `progress` classes override this, though.)
`initially-hidden`	Columns	The column will not be shown when the `datagrid` is initially rendered.

open	Rows	The row can be opened and closed, and is currently open.
progress	Cells	The cell is a progress bar.
reversed	Columns	If the cell is sorted, the sort direction is descending, instead of ascending.
selectable-separator	Rows	The row is a normal, selectable, data row, except that instead of having data, it only has a separator. (The `header` and `separator` classes override this, though.)
separator	Rows	The row is a separator row, not a data row. (The `header` class overrides this, though.)
sortable	Columns	The data can be sorted by this column.
sorted	Columns	The data is sorted by this column. Unless the `reversed` class is also present, the sort direction is ascending.
unchecked	Cells	The cell has a checkbox and, unless the `checked` class is present as well, it is unchecked. (The `cyclable` and `progress` classes override this, though.)

2.20.1.3. The default data provider

The user agent must supply a default data provider for the case where the `datagrid`'s `data` attribute is null. It must act as described in this section.

The behaviour of the default data provider depends on the nature of the first element child of the `datagrid`.

While the first element child is a `table`

`getRowCount()`: The number of rows returned by the default data provider must be the number of `tr` elements that are children of `tbody` elements that are children of the `table`, if there are any such child `tbody` elements. If there are no such `tbody` elements then the number of rows returned must be the number of `tr` elements that are children of the `table`.

> ***Note: Rows in `thead` elements do not contribute to the number of rows returned, although they do affect the columns and column captions. Rows in `tfoot` elements are ignored completely by this algorithm.***

`getColumnCount()`: The number of columns returned must be the number

of `td` element children in the first `tr` element child of the first `tbody` element child of the `table`, if there are any such `tbody` elements. If there are no such `tbody` elements, then it must be the number of `td` element children in the first `tr` element child of the `table`, if any, or otherwise 1. If the number that would be returned by these rules is 0, then 1 must be returned instead.

`getCaptionText(i)`: If the `table` has no `thead` element child, or if its first `thead` element child has no `tr` element child, the default data provider must return the empty string for all captions. Otherwise, the value of the `textContent` attribute of the *i*th `th` element child of the first `tr` element child of the first `thead` element child of the `table` element must be returned. If there is no such `th` element, the empty string must be returned.

`getCaptionClasses(i, classes)`: If the `table` has no `thead` element child, or if its first `thead` element child has no `tr` element child, the default data provider must not add any classes for any of the captions. Otherwise, each class in the `class` attribute of the *i*th `th` element child of the first `tr` element child of the first `thead` element child of the `table` element must be added to the *classes*. If there is no such `th` element, no classes must be added. The user agent must then:

1. Remove the `sorted` and `reversed` classes.
2. If the `table` element has a `class` attribute that includes the `sortable` class, add the `sortable` class.
3. If the column is the one currently being used to sort the data, add the `sorted` class.
4. If the column is the one currently being used to sort the data, and it is sorted in descending order, add the `reversed` class as well.

The various row- and cell- related methods operate relative to a particular element, the element of the row or cell specified by their arguments.

For rows: Since the view of the data can be sorted, the positions of the rows in the data model might not be the same as the positions of the real rows in the DOM. When the data is sorted, the row given by the method's argument has to be mapped to the real row. Initially, the mapping is the identity transform, but the mapping can be changed if the user sorts the rows.

Once the method's argument has been translated into an index for the real row, the row's element is found as follows. If the `table` has `tbody` element children, the element for the *i*th real row is the *i*th `tr` element that is a child

of a `tbody` element that is a child of the `table` element. If the `table` does not have `tbody` element children, then the element for the *i*th real row is the *i*th `tr` element that is a child of the `table` element.

For cells: Given a row and its element, the row's *i*th cell's element is the *i*th `td` element child of the row element.

> ***Note: The `colspan` and `rowspan` attributes are ignored by this algorithm.***

`getRowParent(i)`: The default data provider must always return -1 as the parent row of any row.

> ***Note: The `table`-based default data provider cannot represent a tree.***

`getRowImage(i)`: If the row's first cell's element has an `img` element child, then the URI of the row's image is the URI of the first `img` element child of the row's first cell's element. Otherwise, the URI of the row's image is the empty string.

`getRowMenu(i)`: If the row's first cell's element has a `menu` element child, then the row's menu is the first `menu` element child of the row's first cell's element. Otherwise, the row has no menu.

`getRowClasses(i, classes)`: The default data provider must never add a class to the row's classes.

`toggleColumnSortState(i)`: If the data is already being sorted on the given column, then the user agent must change the current sort mapping to be the inverse of the current sort mapping; if the sort order was ascending before, it is now descending, otherwise it is now ascending. Otherwise, if the current sort column is another column, or the data model is currently not sorted, the user agent must create a new mapping, which maps rows in the data model to rows in the DOM so that the rows in the data model are sorted by the specified column, in ascending order. (Which sort comparison operator to use is left up to the UA to decide.)

`getCellData(i, j)`, **`getCellClasses(i, j, classes)`**, **`getCellCheckedState(i, j, state)`**, **`cycleCell(i, j)`**, **and** **`editCell(i, j, data)`**: See the common definitions below.

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate. For example, if a `tr` is removed, then the `updateRowsRemoved()` methods would probably need to

be invoked, and any change to a cell or its descendants must cause the cell to be updated. If the `table` element stops being the first child of the `datagrid`, then the data provider must call the `updateEverything()` method on the `datagrid`. Any change to a cell that is in the column that the data provider is currently using as its sort column must also cause the sort to be reperformed, with a call to `updateEverything()` if the change did affect the sort order.

While the first element child is a `select`

The default data provider must return 1 for the column count, the empty string for the column's caption, and must not add any classes to the column's classes.

For the rows, assume the existence of a linear node iterator view of the children of the first `select` element child of the `datagrid` element, that skips all nodes other than `optgroup` and `option` elements, as well as any descendents of any `option` elements, and descendants of `optgroup` elements with the `closed` token in their `class` attribute.

Given this view, each element in the view represents a row in the data model: the *i*th element in the view is the *i*th row's element. The row of a particular method call is the row given by its arguments.

`getRowCount()` must return the number of elements in this view.

`getRowParent(i)` must return the index in the view of the nearest ancestor `optgroup` element of the row's element, -1 if there is no such ancestor.

`getRowImage(i)` must return the empty string, `getRowMenu(i)` must return null.

`getRowClasses(i, classes)` must add the classes from the following list to *classes* when their condition is met:

- If the row's element contains other elements that are also in the view, and the element's `class` attribute contains the `closed` class: `closed`
- If the row's element contains other elements that are also in the view, and the element's `class` attribute doesn't contain the `closed` class: `open`

The `toggleRowOpenState(i)` method must add a `closed` class to that row's element's `class` attribute and remove any `open` class, unless it already has a `closed` class and has no `open` class, in which case it must instead remove the `closed` class and add an `open` class. It must then invoke the

appropriate update methods to inform the `datagrid` of the newly added or removed rows.

The `getCellData(i, j)` method must return the value of the label attribute if the row's element is an `optgroup` element, otherwise, if the row's element is an `option`element, its `label` attribute if it has one, otherwise the value of its `textContent` DOM attribute.

The `getCellClasses(i, j, classes)` method must add no classes.

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate.

While the first element child is another element

The default data provider must return 1 for the column count, the empty string for the column's caption, and must not add any classes to the column's classes.

For the rows, assume the existence of a linear node iterator view of the children of the `datagrid` that skips all nodes other than `li`, `h1`-`h6`, and `hr` elements, and skips all elements that are descendants of elements with the `closed` token in their `class` attribute, and any descendants of `menu` elements.

Given this view, each element in the view represents a row in the data model: the *i*th element in the view is the *i*th row's element. The row of a particular method call is the row given by its arguments.

`getRowCount()` must return the number of elements in this view.

`getRowParent(i)` must return the index in the view of the nearest ancestor (in the real DOM) of the row's element that is also in the view, -1 if there is no such ancestor.

> In the following example, the row numbered 2 returns 1 as its parent, and the other rows return -1:
>
> ```
> <datagrid>
>  <ol>
>   <li> row 0 </li>
>   <li> row 1
>    <ol>
>     <li> row 2 </li>
>    </ol>
>   </li>
>   <li> row 3 </li>
>  </ol>
> </datagrid>
> ```

`getRowImage(i)` must return the URI of the image given by the first `img` element descendant (in the real DOM) of the row's element, that is not also a descendant of another element that has a later position in the view.

> In the following example, the row numbered 2 returns "http://example.com/a" as its image URI, and the other rows (including row 1) return the empty string:
>
> ```
> <datagrid>
>  <ol>
>   <li> row 0 </li>
>   <li> row 1
>    <ol>
>     <li> row 2 <img src="http://example.com/a" alt=""> </li>
>    </ol>
>   </li>
>   <li> row 3 </li>
>  </ol>
> </datagrid>
> ```

`getRowMenu(i)` must return the first `menu` element descendant (in the real DOM) of the row's element, that is not also a descendant of another element that has a later position in the view. (This is analogous to the image case above.)

`getRowClasses(i, classes)` must add the classes from the following list to *classes* when their condition is met:

- If the row's element contains other elements that are also in the view, and the element's `class` attribute contains the `closed` class: `closed`
- If the row's element contains other elements that are also in the view, and the element's `class` attribute doesn't contain the `closed` class: `open`
- If the row's element is an `h1`-`h6` element: `header`
- If the row's element is an `hr` element: `separator`

The `toggleRowOpenState(i)` method must add a `closed` class to that row's element's `class` attribute and remove any `open` class, unless it already has a `closed` class and has no `open` class, in which case it must instead remove the `closed` class and add an `open` class. It must then invoke the appropriate update methods to inform the `datagrid` of the newly added or removed rows.

The `getCellData(i, j)`, `getCellClasses(i, j, classes)`, `getCellCheckedState(i, j, state)`, `cycleCell(i, j)`, and `editCell(i, j,`

`data)` methods must act as described in the common definitions below, treating the row's element as being the cell's element.

The data provider must call the `datagrid`'s update methods appropriately whenever the descendants of the `datagrid` mutate.

Otherwise, while there is no element child

The data provider must return 0 for the number of rows, 1 for the number of columns, the empty string for the first column's caption, and must add no classes when asked for that column's classes. If the `datagrid`'s child list changes such that the first element child is one of the above, then the data provider must call the `updateEverything()` method on the `datagrid`.

2.20.1.3.1. 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 are the row and column indices respectively). Which element that is is defined in the previous section.

Cyclable cells

If the first element child of a cell's element is a `select` element that has a no `multiple` attribute and has at least one `option` element descendent, then the cell acts as a cyclable cell.

The "current" `option` element is the selected `option` element, or the first `option` element if none is selected.

The `getCellData()` method must return the `textContent` of the current `option` element (the `label` attribute is ignored in this context as the `optgroup`s are not displayed).

The `getCellClasses()` method must add the `cyclable` class and then all the classes of the current `option` element.

The `cycleCell()` method must change the selection of the `select` element such that the next `option` element after the current `option` element is the only one that is selected (in tree order). If the current `option` element is the last `option` element descendent of the `select`, then the first `option` element descendent must be selected instead.

The `setCellCheckedState()` and `editCell()` methods must do nothing.

Progress bar cells

If the first element child of a cell's element is a `progress` element, then the

cell acts as a progress bar cell.

The `getCellData()` method must return the value returned by the `progress` element's `position` DOM attribute.

The `getCellClasses()` method must add the `progress` class.

The `setCellCheckedState()`, `cycleCell()`, and `editCell()` methods must do nothing.

Checkbox cells

If the first element child of a cell's element is an `input` element that has a `type` attribute with the value `checkbox`, then the cell acts as a check box cell.

The `getCellData()` method must return the `textContent` of the cell element.

The `getCellClasses()` method must add the `checked` class if the `input` element is checked, and the `unchecked` class otherwise.

The `setCellCheckedState()` method must set the `input` element's checkbox state to checked if the method's third argument is 1, and to unchecked otherwise.

The `cycleCell()` and `editCell()` methods must do nothing.

Editable cells

If the first element child of a cell's element is an `input` element that has a `type` attribute with the value `text` or that has no `type` attribute at all, then the cell acts as an editable cell.

The `getCellData()` method must return the `value` of the `input` element.

The `getCellClasses()` method must add the `editable` class.

The `editCell()` method must set the `input` element's `value` DOM attribute to the value of the third argument to the method.

The `setCellCheckedState()` and `cycleCell()` methods must do nothing.

2.20.1.4. Populating the `datagrid` element

A `datagrid` must be disabled until its end tag has been parsed (in the case of a `datagrid` element in the original document markup) or until it has been inserted into the document (in the case of a dynamically created element). After that point, the element must fire a single `load` event at itself, which doesn't bubble

and cannot be canceled.

The `datagrid` must then populate itself using the data provided by the data provider assigned to the `data` DOM attribute. After the view is populated (using the methods described below), the `datagrid` must invoke the `initialize()` method on the data provider specified by the `data` attribute, passing itself (the `HTMLDataGridElement` object) as the only argument.

When the `data` attribute is null, the `datagrid` must use the default data provider described in the previous section.

To obtain data from the data provider, the element must invoke methods on the data provider object in the following ways:

To determine the total number of rows

Invoke the `getRowCount()` method with no arguments. The return value is the number of rows. If the return value 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 total number of columns

Invoke the `getColumnCount()` method with no arguments. The return value is the number of columns. If the return value is zero or negative, not an integer, or simply not a numeric type, or if the method is not defined, then 1 must be used instead.

To get the captions to use for the columns

Invoke the `getCaptionText()` method with the index of the column in question. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of columns. The return value is the string to use when referring to that column. If the method returns null or the empty string, the column has no caption. If the method is not defined, then none of the columns have any captions.

To establish what classes apply to a column

Invoke the `getCaptionClasses()` method with the index of the column in question, and an object implementing the `DOMTokenString` interface, initialised to empty. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of columns. The values contained in the `DOMTokenString` object when the method returns represent the classes that apply to the given column. If the method is not defined, no classes apply to the column.

To establish whether a column should be initially included in the visible columns

Check whether the `initially-hidden` class applies to the column. If it

does, then the column should not be initially included; if it does not, then the column should be initially included.

To establish whether the data can be sorted relative to a particular column

Check whether the `sortable` class applies to the column. If it does, then the user should be able to ask the UA to display the data sorted by that column; if it does not, then the user agent must not allow the user to ask for the data to be sorted by that column.

To establish if a column is a sorted column

If the user agent can handle multiple columns being marked as sorted simultaneously: Check whether the `sorted` class applies to the column. If it does, then that column is the sorted column, otherwise it is not.
If the user agent can only handle one column being marked as sorted at a time: Check each column in turn, starting with the first one, to see whether the `sorted` class applies to that column. The first column that has that class, if any, is the sorted column. If none of the columns have that class, there is no sorted column.

To establish the sort direction of a sorted column

Check whether the `reversed` class applies to the column. If it does, then the sort direction is descending (down; first rows have the highest values), otherwise it is ascending (up; first rows have the lowest values).

To establish a row's parent row

Invoke the `getRowParent()` method with the index of the row in question. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of rows. The return value *p* is the index of the parent row. If the method returns a number outside the range $0 \leq p < i$, or if the returned value is non-numeric, or if the method is not defined, then the row has no parent row (it is an unparented top-level row).

If a row *r* has a parent row *p*, but not all the rows between it and its parent also have a parent row, or if there is a row *i* between *p* and *r* the parent of which is neither *p* nor another row between *p* and *i*, then the user agent may present the tree structure in an inconsistent way instead of attempting to render the actual described tree structure.

To obtain a URI to an image representing a row

Invoke the `getRowImage()` method with the index of the row in question. The index *i* must be in the range $0 \leq i < N$, where *N* is the total number of rows. The return value is a string representing a URI or IRI to an image. Relative URIs must be interpreted relative to the `datagrid`'s base URI. If the method returns the empty string, null, or if the method is not defined, then the row has no associated image.

To obtain a context menu appropriate for a particular row

Invoke the `getRowMenu()` method with the index of the row in question. The index *i* must be in the range 0 ≤ *i* < *N*, where *N* is the total number of rows. The return value is a reference to an object implementing the `HTMLMenuElement` interface, i.e. a `menu` element DOM node. (This element must then be interpreted as described in the section on context menus to obtain the actual context menu to use.) If the method returns something that is not an `HTMLMenuElement`, or if the method is not defined, then the row has no associated context menu. User agents may provide their own default context menu, and may add items to the author-provided context menu. For example, such a menu could allow the user to change the presentation of the `datagrid` element.

To establish what classes apply to a row

Invoke the `getRowClasses()` method with the index of the row in question, and an object implementing the `DOMTokenString` interface, initialised to empty. The index *i* must be in the range 0 ≤ *i* < *N*, where *N* is the total number of rows. The values contained in the `DOMTokenString` object when the method returns represent the classes that apply to the row in question. If the method is not defined, no classes apply to the row.

To establish whether a row is a data row or a special row

Examine the classes that apply to the row. If the `header` class applies to the row, then it is not a data row, it is a subheading. The data from the first cell of the row is the text of the subheading, the rest of the cells must be ignored. Otherwise, if the `separator` class applies to the row, then in the place of the row, a separator should be shown. Otherwise, if the `selectable-separator` class applies to the row, then the row should be a data row, but represented as a separator. (The difference between a `separator` and a `selectable-separator` is that the former is not an item that can be actually selected, whereas the second can be selected and thus has a context menu that applies to it, and so forth.) For both kinds of separator rows, the data of the rows' cells must all be ignored. If none of those three classes apply then the row is a simple data row.

To establish whether a row is openable

Check whether one of the `open` or `closed` classes applies to the row. If one (or both) of these are present, then the row can be opened and closed, otherwise neither are present and the row cannot be opened or closed. (It might still have rows that consider this row a parent, however.)

To establish whether an openable row is open or closed

Check whether the `open` class applies to the row. If it does, the row is open. Otherwise, the row is closed. The `closed` class is not examined to make this determination (although either it or the `open` class must be

present to make the row openable in the first place). If a closed row has rows that consider it a parent, those rows must still be included in the rendering.

To establish the value of a particular cell

Invoke the `getCellData()` method with the first argument being the index of the cell's row and the second argument being the index of its column. The two arguments must be zero or positive integers less than the total number of rows and columns respectively. The return value is the value of the cell. If the return value is null or the empty string, or if the method is not defined, then the cell has no data. (For progress bar cells, the cell's value must be further interpreted, as described below.)

To establish what classes apply to a cell

Invoke the `getCellClasses()` method with the first argument being the index of the cell's row, the second argument being the index of its column, and the third being an object implementing the `DOMTokenString` interface, initialised to empty. The first two arguments must be zero or positive integers less than the total number of rows and columns respectively. The values contained in the `DOMTokenString` object when the method returns represent the classes that apply to that cell. If the method is not defined, no classes apply to the cell.

To establish how the type of a cell

Examine the classes that apply to the cell. If the `progress` class applies to the cell, it is a progress bar. Otherwise, if the `cyclable` class applies to the cell, it is a cycling cell whose value can be cycled between multiple states. Otherwise, none of these classes apply, and the cell is a simple text cell.

To establish the value of a progress bar cell

If the value *x* of the cell is a string that can be converted to a floating point number in the range $0.0 \leq x \leq 1.0$, then the progress bar has that value (0.0 means no progress, 1.0 means complete). Otherwise, the progress bar is an indeterminate progress bar.

To establish how a simple text cell should be presented

Check whether one of the `checked`, `unchecked`, or `indeterminate` classes applies to the cell. If any of these are present, then the cell has a checkbox, otherwise none are present and the cell does not have a checkbox. If the cell has no checkbox, check whether the `editable` class applies to the cell. If it does, then the cell value is editable, otherwise the cell value is static.

To establish the state of a cell's checkbox, if it has one

Check whether the `checked` class applies to the cell. If it does, the cell is

checked. Otherwise, check whether the `unchecked` class applies to the cell. If it does, the cell is unchecked. Otherwise, the `indeterminate` class appplies to the cell and the cell's checkbox is in an indeterminate state. When the `indeterminate` class appplies to the cell, the checkbox is a tristate checkbox, and the user can set it to the indeterminate state. Otherwise, only the `checked` and/or `unchecked` classes apply to the cell, and the cell can only be toggled betwen those two states.

If the data provider ever raises an exception while the `datagrid` is invoking one of its methods, the `datagrid` must act, for the purposes of that particular method call, as if the relevant method had not been defined.

The data model is considered stable: user agents may assume that subsequent calls to the data provider methods will return the same data, until one of the update methods is called on the `datagrid` element. If a user agent is returned inconsistent data, for example if the number of rows returned by `getRowCount()` varies in ways that do not match the calls made to the update methods, the user agent may disable the `datagrid`. User agents that do not disable the `datagrid` in inconsistent cases must honour the most recently returned values.

User agents may cache returned values so that the data provider is never asked for data that could contradict earlier data. User agents must not cache the return value of the `getRowMenu` method.

The exact algorithm used to populate the data grid is not defined here, since it will differ based on the presentation used. However, the behaviour of user agents must be consistent with the descriptions above. For example, it would be non-conformant for a user agent to make cells have both a checkbox and be editable, as the descriptions above state that cells that have a checkbox cannot be edited.

2.20.1.5. Updating the `datagrid`

Whenever the `data` attribute is set to a new value, the `datagrid` must clear the current selection, remove all the displayed rows, and plan to repopulate itself using the information from the new data provider at the earliest opportunity.

There are a number of update methods that can be invoked on the `datagrid` element to cause it to refresh itself in slightly less drastic ways:

When the **`updateEverything()`** method is called, the user agent must repopulate the entire `datagrid`. If the number of rows decreased, the selection must be updated appropriately. If the number of rows increased, the new rows should be left unselected.

When the **`updateRowsChanged(`*`row`*`, `*`count`*`)`** method is called, the user agent

must refresh the rendering of the rows in the range from row *row* to row *row*+*count*-1.

When the **`updateRowsInserted(`*`row`*`, `*`count`*`)`** method is called, the user agent must assume that *count* new rows have been inserted between what used to be row *row*-1 and row *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 row *row*. The user agent must update its rendering and the selection accordingly.

The **`updateRowChanged(`*`row`*`)`** method must be exactly equivalent to calling `updateRowsChanged(`*`row`*`, 1)`.

When the **`updateColumnChanged(`*`column`*`)`** method is called, the user agent must refresh the rendering of the specified column *column*, for all rows.

When the **`updateCellChanged(`*`row`*`, `*`column`*`)`** method is called, the user agent must refresh the rendering of the cell on row *row*, in column *column*.

Any effects the update methods have on the `datagrid`'s selection is not considered a change to the selection, and must therefore not fire the `select` event.

These update methods should only be called by the data provider, or code acting on behalf of the data provider. In particular, calling the `updateRowsInserted()` and `updateRowsRemoved()` methods without actually inserting or removing rows from the data provider is likely to result in inconsistent renderings.

2.20.1.6. Requirements for interactive user agents

This section only applies to interactive user agents.

If the `datagrid` element has a **`disabled`** attribute, then the user agent must disable the `datagrid`, preventing the user from interacting with it. The `datagrid` element should still continue to update itself when the data provider signals changes to the data, though. Obviously, conformance requirements stating that `datagrid` elements must react to users in particular ways do not apply when one is disabled.

If a row is openable, then the user must be able to toggle its open/closed state. When the user does so, then the `datagrid` must invoke the data provider's `toggleRowOpenState()` method, with the row's index as the only argument. The `datagrid` must *then* act as if the `datagrid`'s `updateRowChanged()` method had

been invoked with that row's index immediately before the provider's method was invoked.

If a cell is a cell whose value can be cycled between multiple states, then the user must be able to activate the cell to cycle its value. When the user activates this "cycling" behaviour of a cell, then the `datagrid` must invoke the data provider's `cycleCell()` method, with the cell's row index as the first argument and its column index as the second. The `datagrid` must then act as if the `datagrid`'s `updateCellChanged()` method had been invoked with those same arguments immediately before the provider's method was invoked.

When a cell has a checkbox, the user must be able to set the checkbox's state. When the user changes the state of a checkbox in such a cell, the `datagrid` must invoke the data provider's `setCellCheckedState()` method, with the cell's row index as the first argument, its column index as the second, and the checkbox's new state as the third. The state should be represented by the number 1 if the new state is checked, 0 if the new state is unchecked, and -1 if the new state is indeterminate (which must only be possible if the cell has the `indeterminate` class set). The `datagrid` must then act as if the `datagrid`'s `updateCellChanged()` method had been invoked, specifying the same cell, immediately before the provider's method was invoked.

If a cell is editable, the user must be able to edit the data for that cell, and doing so must cause the user agent to invoke the `editCell()` method of the data provider with three arguments: the row number and column number of the cell, and the new text entered by the user. The user agent must then act as if the `updateCellChanged()` method had been invoked, with the same row and column specified.

2.20.1.7. The selection

This section only applies to interactive user agents. For other user agents, the `selection` attribute must return null.

```
interface SelectionRanges {
  readonly attribute long count;
  long getRangeStart(in long index);
  long getRangeLength(in long index);
  void addRange(in long start, in long count);
  void removeRange(in long index);
  void setSelected(in long row, in bool selected);
  bool isSelected(in long row);
  void selectAll();
  void invert();
  void clear();
};
```

Each `datagrid` element must keep track of which rows are currently selected. Initially no rows are selected, but this can be changed via the methods described in this section.

The selection of a `datagrid` is represented by its **`selection`** DOM attribute, which must be a `SelectionRanges` object.

The `SelectionRanges` object represents the selection using ranges. Each range has a starting index and a length. The starting index is relative to the first row (index 0) of the `datagrid`. The length states how many of the rows are selected, starting from the starting index. A range of length one implies that only the row indicated by its starting index is selected.

The ranges in a selection must not overlap. Ranges may be adjacent (e.g. one range starting at index zero with length two, and a second range starting at index two) but user agents should coalesce adjacent ranges.

The start index of a range must not be negative, and must not be greater than the index of the last row. The length of a range must not be such that the range's start index plus its length yields a value greater than the number of rows.

The **`count`** attribute must return the number of ranges currently present in the selection. The **`getRangeStart()`** and **`getRangeLength()`** methods must return the starting index and length (respectively) of the range specified by their argument. If the argument is out of range (less than zero or greater than the number of ranges minus one), then they must raise an `INDEX_SIZE_ERR` exception. [DOM3CORE]

The ranges must be returned in ascending numerical order. That is, the value returned by the `getRangeStart()` method for an index *x* must always be greater than the value it returns for any index less than *x*.

The **`addRange()`** method takes two arguments, an index and a length, specifying a range of rows to select. If the specified range is invalid or would contain rows outside the `datagrid` (e.g. the starting index is negative, or the length would take the selection beyond the end of the `datagrid`), then the method must raise an `INDEX_SIZE_ERR` exception. Otherwise, the specified range must be added to the selection. If the range overlaps, grows, or joins existing selections, the user agent must adjust the ranges so that no two ranges overlap, and should adjust them so that no two ranges are adjacent. Thus, calling `addRange()` may actually reduce the total number of ranges in the selection.

The **`removeRange()`** method takes two arguments, an index and a length, specifying a range of rows to unselect. If the specified range is invalid or would contain rows outside the `datagrid` (e.g. the starting index is negative, or the

length would take the selection beyond the end of the `datagrid`), then the method must raise an `INDEX_SIZE_ERR` exception. Otherwise, the specified rows must be removed from the selection. Calling `removeRange()` may actually increase the total number of ranges in the selection, e.g. if a range had to be split in order to unselect a row in the middle.

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, by adjusting the selection's ranges accordingly. If *row* is less than zero or greater than the index of the last row then the method must raise an `INDEX_SIZE_ERR` exception.

The **`isSelected()`** method must return the selected state of the row specified by its argument. If the specified row exists and is in one of the ranges of the selection, it must return true, otherwise it must return false.

The **`selectAll()`** method must replace all the current ranges in the selection with a single selection range having index zero and a length equal to the number of rows in the `datagrid`. If there are no rows in the `datagrid` then this method must instead only remove all the current ranges. (In a compliant UA, there would not be any ranges to remove.)

The **`invert()`** method must adjust the selections such that the selection is inverted. That is, the ranges must be adjusted such that only the rows that were previously not a part of the selection must be made a part of the new selection.

The **`clear()`** method must remove all the ranges in the selection.

If the `datagrid` element has a **`multiple`** attribute, then the user must be able to select any number of rows (zero or more). If the attribute is not present, then the user must only be able to select a single row at a time, and selecting another one must unselect all the other rows.

> ***Note: This only applies to the user. Scripts can select multiple rows even when the `multiple` attribute is absent.***

Whenever the selection of a `datagrid` changes, whether due to the user interacting with the element, or as a result of calls to methods of the `selection` object, a **`select`** event that bubbles but is not cancelable must be fired on the `datagrid` element. If multiple changes are made to the selection via calls to the object's methods during a single execution of a script, then the `select` events should be coalesced into one (which later fires once the script execution has completed).

2.20.1.8. Columns and captions

This section only applies to interactive user agents.

Each `datagrid` element must keep track of which columns are currently being rendered. User agents should initially show all the columns except those with the `initially-hidden` class, but may allow users to hide or show columns. User agents should initially display the columns in the order given by the data provider, but may allow this order to be changed by the user.

If columns are not being used, as might be the case if the data grid is being presented in an icon view, or if an overview of data is being read in an aural context, then the text of the first column of each row should be used to represent the row.

If none of the columns have any captions (i.e. if the data provider does not provide a `getCaptionText()` method), then user agents may avoid showing the column headers at all. This may prevent the user from performing actions on the columns (such as reordering them, changing the sort column, and so on).

> ***Note: Whatever the order used for rendering, and irrespective of what columns are being shown or hidden, the "first column" as referred to in this specification is always the column with index zero, and the "last column" is always the column with the index one less than the value returned by the `getColumnCount()` method of the data provider.***

If a column is sortable, then the user must be able to invoke it to sort the data. When the user does so, then the `datagrid` must invoke the data provider's `toggleColumnSortState()` method, with the column's index as the only argument. The `datagrid` must *then* act as if the `datagrid`'s `updateEverything()` method had been invoked.

3. Menus, buttons and commands

This section will probably be moved to the structure details section near the top. Parts of it need a lot of work. We need to decide if we care about some of it, e.g. the menu bar nonsense should probably die. The main use case is context menus and drop down menus on buttons, probably (although the latter is dubious at best).

3.1. Tutorial

This section still needs to be written. For now, here are some markup snippets to show how this should work:

```
<menubar>
 <li>
  <a href="#file">File</a>
  <menu id="file">
   <li><button type="button" onclick="fnew()">New...</button></li>
   <li><button type="button" onclick="fopen()">Open...</button></li>
   <li><button type="button" onclick="fsave()" id="save">Save</button></
   <li><button type="button" onclick="fsaveas()">Save as...</button></li
  </menu>
 </li>
 <li>
  <a href="#edit">Edit</a>
  <menu id="edit">
   <li><button type="button" onclick="ecopy()">Copy</button></li>
   <li><button type="button" onclick="ecut()">Cut</button></li>
   <li><button type="button" onclick="epaste()">Paste</button></li>
  </menu>
 </li>
 <li>
  <a href="#help">Help</a>
  <menu id="help">
   <li><a href="help.html">Help</a></li>
   <li><a href="about.html">About</a></li>
  </menu>
 </li>
</menubar>

...

<input command="save"/> <!-- This will act exactly like the
                             Save button above, including reflecting
                             its disabled state dynamically -->
```

Here's a way of doing something similar. This menu bar would not display inline in the page, but could be made to display in the browser's menu bar or as the window's only menu bar if the application is running standalone. How to do that hasn't yet been decided.

```
<menubar id="appmenu">
 <menulabel label="File"/>
 <menu>
  <command label="New..." onclick="fnew()"/>
  <command label="Open..." onclick="fopen()"/>
  <command label="Save" onclick="fsave()" id="save"/>
  <command label="Save as..." onclick="fsaveas()"/>
 </menu>
 <menulabel label="Edit"/>
 <menu>
```

```
  <command label="Copy" onclick="ecopy()"/>
  <command label="Cut" onclick="ecut()"/>
  <command label="Paste" onclick="epaste()"/>
 </menu>
 <menulabel label="Help"/>
 <menu>
  <a href="help.html">Help</a>
  <a href="about.html">About</a>
 </menu>
</menubar>
```

Here's some markup that falls back on the traditional abuse of the `select` element as a navigation menu, but which is implemented as a semi-correct menu using the new techniques of this document:

```
<form action="redirect.cgi">
 <menubar>
  <menulabel><label for="goto">Go to...</label></menulabel>
  <menu>
   <select id="goto"
           onchange="if (this.options[this.selectedIndex].value)
                       window.location = this.options[this.selectedIndex].
    <option value="" selected="selected"> Select site: </option>
    <option value="http://www.apple.com/"> Apple </option>
    <option value="http://www.mozilla.org/"> Mozilla </option>
    <option value="http://www.opera.com/"> Opera </option>
   </select>
   <span><input type="submit" value="Go"></span>
  </menu>
 </menubar>
</form>
```

3.2. Commands

A command is the abstraction behind menu items, buttons, and keyboard shortcuts. Once a command is defined, it can be referred to by menu items, buttons, keyboard shortcut declarations, script, and so forth. The advantage of this is that it allows many access points to a single feature to share features such as their disabled state.

Commands have the following facets:

Type

The kind of command: "command", meaning it is a normal command; "radio", meaning that triggering the command will, amongst other things, set the Checked State to true (and probably uncheck some other commands); or "checkbox", meaning that triggering the command will, amongst other things, toggle the value of the Checked State.

ID

The name of the command, for referring to the command from the markup or from script. If a command has no ID, it is an **anonymous command**.

Label

The name of the command as seen by the user.

Hint

A helpful or descriptive string that can be shown to the user.

Icon

A graphical image that represents the action.

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.

Hidden State

Whether the command is hidden or not (basically, whether it should be shown in menus).

Disabled State

Whether the command can be triggered or not. If the Hidden State is true (hidden) then the Disabled state will be true (disabled) regardless.

Checked State

Whether the command is checked or not.

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.

Note: The distinction between Disabled State and Hidden State is subtle. A command should be Disabled if, in the same context, it could be enabled if only certain aspects of the situation were changed. A command should be marked as Hidden if, in that situation, the command will never be enabled. For example, in the context menu for a water faucet, the command "open" might be Disabled if the faucet is already open, but the command "eat" would be marked Hidden since the faucet could never be eaten.

In the DOM, the following interface is used to represent a command. (The comments describing each member of this interface are normative.)

```
interface Command {

  // The command's ID, null if the element defines an anonymous commanc
  readonly attribute DOMString               id;

  // The command's Label, null if the element does not specify one.
  readonly attribute DOMString               label;

  // The command's Hint, null if the element does not specify one.
  readonly attribute DOMString               title;

  // The absolute URI to the command's Icon, or, if the element
  // defining the command has no explicit icon, the computed value
  // of the CSS 'icon' property on that element. [CSS3UI]
  // Null if the element does not specify an icon and the computed
  // value of the CSS 'icon' property is 'auto'.
  readonly attribute DOMString               icon;

  // The Hidden State of the command. True if the element is
  // hidden, false otherwise.
  readonly attribute boolean                 hidden;

  // The Disabled State of the command. True if the element is
  // disabled or hidden, false otherwise.
  readonly attribute boolean                 disabled;

  // The Checked State of the command. True if the element is
  // checked, false otherwise.
  readonly attribute boolean                 checked;

  // The type of command. Either "command", "radio", or "checkbox".
  // Null if the element does not define a command.
  readonly attribute DOMString               commandType;

  // The Action of the command: a method that triggers the action for
  // the command. Has no effect if the element does not define a commar
  void triggerCommand();

  // The list of elements that can trigger this command (the Triggers
  // for the command), null if the element does not define a command.
  readonly attribute HTMLCollection          triggers;

  // The element referred to by the command attribute (if
  // specified), which is the element that actually defines the
  // command for this element. (See: the command attribute.)
  // If the element defines a command, this must point to the element
  // itself (as in commandElement.command == commandElement).
  // Null if the element does not have a command attribute and
  // does not define a command.
  readonly attribute Command                 command;

};
```

The `Command` interface is implemented by any element capable of defining a command. All the attributes of the `Command` interface are readonly. Elements implementing this interface may implement other interfaces that have attributes with identical names but that are writable; in bindings that simply flatten all supported interfaces on the object, the writable attributes have priority over the readonly attributes defined above.

All the commands defined in a document that have IDs are listed in the **`document.commands`** attribute:

```
interface DocumentCommands {
  readonly attribute HTMLCollection          commands;
}
```

The collection represented by this attribute is *live*. As commands are defined in or removed from the document, the attribute is updated.

The following elements may define commands: `a`, `button`, `input`, `option`, `command`.

3.2.1. The `command` attribute

Any element that can define a command can also, instead, have a `command` attribute that specifies the ID of a command that the element should defer to. In this case the element does not define a command, but, in the absence of attributes to the contrary, reflects the state of the element specified.

If the `command` attribute specifies an ID that is not the ID of an element that defines a command, then the `command` DOM attribute is set to the null value, and the element acts as if it was linked to an element that defined a command with no Label, no Hint, no Icon, no Action, that was not Hidden, not Disabled, not Checked, and that was of Type "command".

3.2.2. The `a` element and commands

3.2.2.1. Using the `a` element to define a command

To define a command, an `a` element must have an appropriate `href` attribute, and must not have a `command` attribute. An appropriate `href` attribute is one whose URI does not contain a fragment identifier that points to a `menu` element in the same document as the `a` element.

Note: An `a` element with an `href` attribute that points to a `menu` element in the same file can be used to open a menu.

The Type of the command is "command".

The ID of the command is the ID of the `a` element, if present. Otherwise it is an anonymous command.

The Label of the command is the string given by the element's `textContent` DOM attribute. [DOM3CORE]

The Hint of the command is the string given by the `title` attribute, if any, and the empty string if the attribute is absent.

The Icon of the command is the absolute URI of the first image in the `a` element. Specifically, in a depth-first search of the children of the element, the first element that is either an `img` element with a `src` attribute, or an `object` element with a `data` attribute. If it is an `img` element then the URI is taken from the `src` attribute. If it is an `object` element then the URI is taken from the `data` attribute. Relative URIs must be resolved.

The Action of the command is that a `{"http://www.w3.org/2001/xml-events", "click"}` event is fired on the `a` element.

The Hidden State and Disabled State facets of the command are always false. (The command is always enabled.)

The Checked State of the command is always false. (The command is never checked.)

3.2.2.2. Using the `a` element with the `command` attribute

If an `a` element has a `command` attribute, then:

If the element's `title` attribute is absent, then when the UA attempts to display the element's hint, it must instead use the specified command's Hint.

Even if the element's `href` attribute is absent, the element must still match the CSS `:link` or `:visited` pseudo-classes. It must match the `:visited` pseudo-class if the command's action is to follow a link that has already been visited by the user, and must match the `:link` pseudo-class otherwise.

If a `DOMActivate` event is dispatched on the element and is not canceled, and the event has no other default action, and the command's Disabled State is false (enabled), then the command's Action must be triggered as the default action.

> ***Note: The `DOMActivate` event is fired as the default action of the `click` event.***

If the command's Disabled State is true (disabled) then the element must be disabled and must therefore match the `:disabled` pseudo-class. UAs should style disabled links in such a way as to clearly convey their disabled state.

The Label, Icon, Checked State and Type facets of the command are ignored by the `a` element (except for matching CSS pseudo-classes).

3.2.3. The `button` element and commands

3.2.3.1. Using the `button` element to define a command

To define a command, a `button` element must not have a `command` attribute.

The Type of the command is "command".

The ID of the command is the ID of the `button` element, if present. Otherwise it is an anonymous command.

The Label, Hint, Icon, and Action facets of the command are determined as for `a` elements.

The Hidden State of the command is always false.

The Disabled State of the command mirrors the disabled state of the button. Typically this is given by the element's `disabled` attribute, but certain button types become disabled at other times too — for example, the Web Forms 2.0 `move-up` button type is disabled when it would have no effect. [WF2]

The Checked State of the command is always false.

3.2.3.2. Using the `button` element with the `command` attribute

If a `button` element has a `command` attribute, then:

If the element's `title` attribute is absent, then when the UA attempts to display the element's hint, it must instead use the specified command's Hint.

If a `DOMActivate` event is dispatched on the element and is not canceled, and the event has no other default action, and the command's Disabled State is false (enabled), and the button's `disabled` attribute is absent, then the command's Action must be triggered as the default action.

> ***Note: The `DOMActivate` event is fired as the default action of the `click` event.***

If the command's Disabled State is true (disabled) then the element must be

disabled. The `button` element must also be disabled if the element's `disabled` attribute is set.

The Label, Icon, Checked State and Type facets of the command are ignored by the `button` element (except for matching CSS pseudo-classes).

3.2.4. The `input` element and commands

3.2.4.1. Using the `input` element to define a command

To define a command, an `input` element must have a `type` attribute specifying a button, radio button, or check box type (In HTML4: `submit`, `reset`, `button`, `radio`, `checkbox` (but not `image`); in WF2: `move-up`, `move-down`, `add`, `remove`), and must not have a `command` attribute.

The Type of the command is "radio" if the `type` attribute has the value `radio`, "checkbox" if the `type` attribute has the value `checkbox`, and "command" otherwise.

The ID of the command is the ID of the `input` element, if present. Otherwise it is an anonymous command.

The Label of the command depends on the Type of the command. If the Type is "command", then it is the string given by the `value` attribute, if any, and a UA-dependent value that the UA uses to label the button itself if the attribute is absent.

If the Type is "radio" or "checkbox", then, if the element has a `label` element associated with it, the `textContent` of the first such element is used as the Label (in DOM terms, `this.labels[0].textContent` [WF2] [DOM3CORE]). Otherwise, the value of the `value` attribute, if present, used is as the Label. Otherwise, the Label is the empty string.

The Hint of the command is the string given by the `title` attribute, if any, and the empty string if the attribute is absent.

There is no Icon for the command.

The Action of the command is that a `{"http://www.w3.org/2001/xml-events", "click"}` event is fired on the `input` element.

The Hidden State and Disabled State facets of the command are as determined for `button` elements.

The Checked State of the command is true if the command is of Type "radio" or "checkbox" and the element has a `checked` attribute, and false otherwise.

3.2.4.2. Using the `input` element with the `command` attribute

If an `input` element has no `type` attribute and no `name` attribute, and it has a `command` attribute, then:

If the command is of Type "command" then the element must generally be styled and behave as if it was of type `button`; if the Type of the command is "radio" then the element must generally be styled and behave as if it was of type `radio`; and if the Type of the command is "checkbox" then the element must generally be styled and behave as if it was of type `checkbox`.

If the command is of Type "command" and the element's `value` attribute is absent, then when the UA attempts to display the element's caption, it must instead use the specified command's Label. The Label facet is ignored if the command is not of Type "command".

The UA may use the Icon facet of the command to render an icon in the control, if appropriate for the UI used.

If the element's `title` attribute is absent, then when the UA attempts to display the element's hint, it must instead use the specified command's Hint.

If a `DOMActivate` event is dispatched on the element and is not canceled, and the event has no other default action, and the command's Disabled State is false (enabled), and the element's `disabled` attribute is absent, then the command's Action must be triggered as the default action.

> ***Note: The `DOMActivate` event is fired as the default action of the `click` event.***

If the command's Disabled State is true (disabled) then the element must be disabled. The `input` element must also be disabled if the element's `disabled` attribute is set.

If the command's Checked State is true (checked) then the element must be checked. The `input` element must also be checked if the element's `checked` attribute is set.

3.2.5. The `option` element and commands

3.2.5.1. Using the `option` element to define a command

To define a command, an `option` element must have an ancestor `select` element and either no `value` attribute or a `value` attribute that is not the empty string.

The Type of the command is "radio" if the `option`'s `select` element has no `multiple` attribute, and "checkbox" if it does.

The ID of the command is the ID of the `option` element, if present. Otherwise it is an anonymous command.

The Label of the command is the value of the `option` element's `label` attribute, if there is one, or the value of the `option` element's `textContent` DOM attribute if it doesn't.

The Hint of the command is the string given by the `title` attribute, if any, and the empty string if the attribute is absent.

There is no Icon for the command.

The Action of the command is that the element be selected in its `select` element. If the command is of Type "radio" then this must unselect all the other options, otherwise it must toggle the selection state of the current option. Once the selection has changed, a `change` event must be fired on the `select` element, as if the selection had been changed directly.

The Hidden State facet of the command is always false (shown).

The Disabled State of the command is true (disabled) when the `option` element is disabled, and false otherwise.

The Checked State of the command is true (checked) when the element is selected in its `select` element.

3.2.5.2. Using the `option` element with the `command` attribute

The `command` attribute cannot be used with `option` elements.

3.2.6. The `command` element and commands

3.2.6.1. Using the `command` element to define a command

The most direct way to represent a command is by using the `command` element. A `command` element defines a command if it does not have a `command` attribute.

```
...
 <command id="c_stop" label="Emergency Stop" onclick="dostop()"/>
 <command id="c_go" label="Go" onclick="dogo()"/>
 <command id="c_lamp" label="Headlamps" onclick="dof2()" disabled="
...
```

This element should not be directly displayed. In CSS-aware user agents, this should be achieved by including the following rules, or their equivalent, in the

UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|command { display: none; }
```

The `command` element, in addition to the core and internationalisation attributes, may have the following attributes specified:

`type`

The command's Type. If present, this attribute must either have the value `radio`, in which case the command is of Type "radio", or the value `checkbox`, in which case the command is (amazingly) of Type "checkbox". Any other value, or the absence of the attribute altogether, means that the command is of Type "command".

`id`

The command's ID. If this attribute is not specified, then the command is anonymous.

`label`

The command's Label. If the attribute is not specified, the command's Label is given by the element's `textContent` DOM attribute.

`title`

The command's Hint. If the attribute is not specified, the command's Hint is the empty string.

`icon`

A URI to the command's Icon. If the attribute is not specified, then the command has no Icon.

onclick

An event handler attribute that listens for `click` events.

hide

The command's Hidden State. If the attribute is present, the command is hidden (and also disabled, regardless of the value of the `disabled` attribute), otherwise, the command is shown. If the attribute is present, it should have the value "`hide`".

disabled

The command's Disabled State. If the attribute is present, the command is disabled, otherwise, the command is enabled. If the attribute is present, it should have the value "`disabled`".

checked

The command's Checked State. If the attribute is present, the command is

checked, otherwise, the command is not. If the attribute is present, it should have the value "`checked`".

radiogroup

An attribute indicating the name of the group of commands that will be toggled when the command itself is toggled. (Described below.)

default

An attribute indicating whether the command is the default command. If the attribute is present, the command is the default command, otherwise it is not. If it is set, it should have the value `default`. Used by context menus to indicate what the default option would be. The :default pseudo-class matches `command` elements with this attribute.

In addition, `command` elements may also have a `command` attribute, as described below.

The Type, ID, Label, Hint, Icon, Hidden State, Disabled State, and Checked State of the command defined by a `command` element are as described above. The Action of a `command` element is that a `{"http://www.w3.org/2001/xml-events", "click"}` event is fired on the element.

If the Type of the command is "checkbox", when a `click` event is dispatched on the element, user agents must toggle the value of the `checked` attribute before the event is dispatched in the document. (If the attribute is absent, then it is set to the value `checked`, and if the attribute is present, it is removed.) If the default action of the event is canceled, the value of the attribute must be changed back to the value it had before the event was dispatched.

If the Type of the command is "radio", when a `click` event is dispatched on the element, user agents must set the value of the `checked` attribute on the element to `checked`, and remove the attribute from any `command` elements with `type` set to `radio` and the same parent element and same `radiogroup` attribute, before the event is dispatched in the document. (If the element has no `radiogroup` attribute, then the elements "with the same `radiogroup` attribute" are those elements with *no* `radiogroup` attribute.) If the default action of the event is canceled, the value of the attributes that were changed must be changed back to the values they had before the event was dispatched.

In HTML the `command` element is an empty element with no end tag.

Authors should put `command` elements inside the `head` element, inside any element that may contain block-level elements or inline-level content, or inside `commandset` elements.

Authors should not put elements or text inside `command` elements.

3.2.6.2. Using the `command` element with the `command` attribute

If a `command` element has a `command` attribute, then:

If the element's `label` attribute is absent, then when the UA attempts to display the element's caption, it must instead use the specified command's Label.

If the element's `icon` attribute is absent, then when the UA attempts to display the element's icon, it must instead use the specified command's Icon.

If the element's `title` attribute is absent, then when the UA attempts to display the element's hint, it must instead use the specified command's Hint.

If a `click` event is dispatched on the element and is not canceled, and the command's Disabled State is false (enabled), and the element's own `disabled` attribute is absent, then the command's Action must be triggered as the default action.

If the command's Disabled State is true (disabled) then the element must be disabled. The `command` element must also be disabled if the element's `disabled` attribute is set.

If the command's Checked State is true (checked) then the element must be checked. The `command` element must also be checked if the element's `checked` attribute is set.

When a `command` element has a `command` attribute, any `type` and `radiogroup` attribute is ignored.

3.2.6.3. Command Sets

Authors may place related commands together inside a **`commandset`** element.

Apart from the core and internationalisation attributes, `commandset` elements have no attributes.

Authors may use `commandset` elements wherever `command` elements are allowed. `commandset` elements may contain any number of `command` and `commandset` elements.

3.2.7. The 'icon' property

UAs should use the command's Icon as the default generic icon provided by the user agent when the 'icon' property computes to 'auto' on an element that either defines a command or refers to one using the `command` attribute.

3.2.8. CSS pseudo-classes and commands

When an element uses the `command` attribute, any UI pseudo-classes from the following list that apply to the element defining the command also apply to the elements that refer to that command.

:enabled, :disabled

Matches commands whose Disabled State facet is False and True respectively.

:checked

Matches commands whose Type facet is either "radio" or "checkbox", and whose Checked State facet is true.

3.3. Menus

This section is horrible. Feel free to coment on this section, but be aware that the current state does not represent anything more than a step along the way to what this section will eventually become.

3.3.1. The `menu` element

Menus are defined using the `menu` element. The semantic of the `menu` element is a structured list of navigation links and commands. The element can be used either as a list or as a block-level container. User agents must support all the common attributes and event handlers, plus the `label` attribute, on `menu` elements.

`menu` elements with explicit `label` attributes, and `menu` elements following `menulabel` elements, should be hidden. In CSS-aware UAs, this effect should be achieved by including the following rules, or their equivalent, in the UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|menu[label], xh|menulabel + xh|menu { display: none; }
```

All other `menu` elements should be rendered identically to `ul` elements. In CSS-aware UAs, this effect may be achieved by including rules similar to the following in the UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|menu { display: block; margin: 0 0 0 40px; list-style: disc; }
```

3.3.1.1. Menu labels

The `label` attribute sets the label of the menu.

If the attribute is not specified, and the element immediately preceding the `menu` element (with the same parent node, ignoring sibling nodes that are not elements) is a `menulabel` element, then that element provides the label.

Otherwise, if the `menu` element has no `label` attribute and the element that immediately precedes it is not a `menulabel` element, not an `hr` element, not a `commandset` element, not a `select` element, and not an element that defines or refers to a command, then the label of the menu is the value of the `textContent` DOM attribute of that previous sibling element.

Otherwise, the `menu` element has no label.

3.3.1.1.1. The `menulabel` element

Menus may be labelled by `menulabel` elements. The semantic of the `menulabel` element is that it labels its following sibling element, which must be a `menu` element. It must only contain inline elements. User agents must support all the common attributes and event handlers, plus the `label` attribute, on the `menulabel` element.

A `menulabel` whose next sibling element is not a `menu` element is semantically meanginless.

The label of `menulabel` elements with explicit `label` attributes is given by that attribute; the label of `menulabel` elements with no `label` attribute is given by the DOM `textContent` attribute.

The default rendering of `menulabel` elements in visual UAs should be a block. In CSS-aware UAs, this effect should be achieved by including the following rules, or their equivalent, in the UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|menulabel { display: block; }
```

Menu bars cause `menulabel` elements to be styled further.

3.3.1.2. Content model of menus

When used as a list, a `menu` element must only contain `li` elements. When used in this way, each `li` element represents at most one item in the menu. What kind of item is represented depends on the children of the `li`.

When used as a block-level container, a `menu` element must only contain block-level markup. Each child element represents at most one item in the menu, depending on which kind of element it is.

Each item in a menu is either a group of commands, a single command, a

separator, a submenu, or legacy fallback content. A menu is built up from these items.

3.3.1.3. Using `optgroup`s as menus

When an `optgroup` element is a descendant of a `menu` element, and the `optgroup` element has a `label` attribute, then it defines a submenu. The label of such a menu is given by the `label` attribute.

When defining a submenu, an `optgroup` element must be a child node of either a `select` element or another `optgroup` element, must only contain `option` elements or other `optgroup` elements. Despite this, however, the processing model for constructing menus, as described in the next section, is the same whether the menu is defined by a `menu` element or an `optgroup` element.

3.3.1.4. Building menus

Menus shall be built up from the children of their `menu` (or `optgroup`) element by processing each child node of that element as follows:

1. If the node is not an element node, it is ignored. (Fallback content.)

2. If the node is an element node but is not in the XHTML namespace, it is ignored. (Fallback content.)

3. If the node is an `li` element, then:
    1. If the first element node defines or refers to a command, or if the first element node is a `menu` element, an `hr` element, a `commandset` element, a `select` element, or an `optgroup` element, then continue the steps as if the `li` was actually this element.
    2. Otherwise, if the first element node is an `a` element with an `href` attribute, then continue the steps as if the `li` was actually that `menu` element. (This can only happen if the `a` element is a menu link, otherwise it would have defined a command and be processed in the first item in this list.)
    3. Otherwise, if the *second* element node is a `menu` element, then continue the steps as if the `li` was actually *that* element. (The first element will probably be used to get the label of the menu.)
    4. Otherwise, this `li` element is ignored.

    Non-element child nodes of the `li` element must be ignored. (Fallback content.)

4. If the node is a command of some sort, then add that command to the menu. The item can be further annotated as follows:
    1. If the node is a `command` element with a `default` attribute, then the command is a default command and this should be reflected in the

resulting interface.

For example, on Windows, context menus can have one menu item marked as being the default item. That item is usually highlighted in bold.

2. Each of the Triggers for the command must be checked in turn (in document order). If any of these triggers has an access key then the first such access key should be used as the shortcut key shown in the menu.

5. If the node is an `a` element with an `href` attribute whose URI points to the current document and contains a fragment identifier that points to a `menu` element that is not the `menu` element for which the menu is currently being built, nor any of the `menu` elements for which any of the higher-level menus were created, then continue the steps as if the `a` was actually that `menu` element. If that `menu` element does not have a label then for the purposes of the current menu's creation, the `a` element's `textContent` is used as the label instead.

6. If the node is a `menu` element, then, if it has a label, add that menu to the menu as a submenu. Otherwise, if it is an unlabelled `menu` element, ignore the node. (Note that a temporary label that applies just for this step may have been assigned by the previous step.)

7. If the node is an `optgroup` element and it has a `label` attribute, then add that menu to the menu as a submenu.

8. If the node is a `commandset` element or a `select` element, then add a separator to the menu, process all the children nodes of the element as if they were children of the `menu` element, then add another separator.

9. If the node is an `hr` element, then add a separator to the menu.

10. If the node is an `option` element that does not define a command, that is disabled, and whose label (either from its `label` attribute, or, if it doesn't have one, from its `textContent` DOM attribute) consists of nothing but one or more hyphens (U+002D HYPHEN-MINUS), then add a separator to the menu.

11. Otherwise, ignore the node. (Fallback content.)

Once all the nodes have been processed in this way, any separators at the top of the menu and at the bottom of the menu shall be removed, and any consecutive separators shall be collapsed to just a single separator.

Commands of Type "radio" or "checkbox" should be represented appropriately.

Commands whose Hidden State is true (hidden) must not be shown in the menu at all. Similarly, the Label, Icon, Hint, Disabled State and Checked State facets of the command should be appropriately reflected in the user interface created for the menu. The default state and access key for each menu item, if any, should similarly be reflected in the UI.

Menus are live: changes to the underlying document structure must be reflected in the menu visible to the user immediately.

Immediately prior to a menu or submenu being opened or made visible, a `click` event that cannot be canceled must be fired on the menu's `menu` (or `optgroup`) element. This event allows menus and submenus to be populated dynamically if needed.

When commands are selected from the menu, their associated Action should be triggered.

3.3.1.5. Displaying menus

When a `menu` element is activated, the associated menu should be constructed and shown. (For details on how a `menu` element can be activated, see the sections on menu links and menu bars.)

The styles applied to each element in the `menu` element, as well as the element itself, may be applied when constructing a menu. UAs are recommended to not apply styling to context menus and menus for application menu bars, and to only use styles for in-page menus.

If user agents support styling of menus, they should only support the '`background`', '`color`', '`border`', '`padding`' and '`font`' properties on menus and menu items. (This list might be incomplete; in general, properties that merely affect the appearance of the element should work, but properties that affect the layout should not.)

As the user interacts with a menu, the elements from which the menu was created should have appropriate pseudo-classes (:hover, :focus, :active) applied.

The menu items must only consider the computed styles of the elements from which they were derived, not other elements.

> For example, take this menu:
>
> ```
> <menu>
> <li><command label="a"/></li>
> <menu>
> ```

> The menu has one menu item, labelled "a".
>
> Styles applied to the `li` element in this menu would have no effect on the rendered menu, except in so far as styles inherit from that element to the `command` element.
>
> Styles applied to the `command` element could affect the menu. While the user is hovering over the menu item, the `:hover` pseudo-class matches the `command` element and any appropriate newly matching rules could be applied.

When activated from a menu link, a menu must be placed in an Appropriate Place. Specifically, if the `a` element is displayed as a vertically-stacked box (as is typically seen for elements with '`display: block`', '`list-item`', or '`table`'), then the menu should appear vertically below the element, anchored so that one of its top corners coincides with a bottom corner of the box so that the menu and the box each have a horizontal sides in common (or a bottom corner of the menu coincides with a top corner of the box, if there isn't enough room for the menu to drop down); otherwise, if the element is displayed as a horizontally stacked box ('`display: inline`', '`table-cell`', etc), the menu should appear to the *side* of the box in an analogous way. If the element is on the right of the page, the menu should drop to the left, and vice versa.

UAs should implement the drop-down behaviour in more platform-appropriate ways if the platform conventions differ from the behaviour described above.

3.3.2. Menu bars: the `menubar` element

Menu bars are defined using the `menubar` element. The semantic of the `menubar` element is a structured list of menus. The element can be used either as a list or as a menu container. User agents must support all the common attributes and event handlers on `menubar` elements.

When used as a list, a `menubar` element must only contain `li` elements. When used in this way, each `li` element represents at most one item in the menu bar. What kind of item is represented depends on the children of the `li`.

When used as a menu container, a `menubar` element must only contain elements that define commands, `menulabel` and `menu` elements, `hr` elements, `commandset` elements, plus any other inline content needed for fallback. Each child element represents at most one item in the menu, depending on which kind of element it is.

3.3.2.1. Displaying menu bars inline

When a `menubar` is displayed inline in the content of the document in a style-

sheet-capable UA, it should be rendered according to the rules of the appropriate style sheet language.

Any `a` elements with `href` attributes that are children of `menubar` elements or children of `li` elements that are themselves children of `menubar` elements should be rendered in a way that indicates that they are not normal links, but can show menus, just like `menulabel` elements. Any `menu` elements that are children of `menubar` elements or children of `li` elements that are themselves children of `menubar` elements should be hidden until they are activated.

In a CSS-aware UA this could be achieved with rules such as:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|menubar > xh|menu, xh|menubar > xh|li > xh|menu { display: none; }
xh|menubar > xh|a[href], xh|menubar > xh|li > xh|a[href],
xh|menulabel { /* styling */ }
```

3.3.2.2. Displaying menu bars as menu bars

If the UA does not render a `menubar` element using a style sheet language's rendering model, then it should use the rendering model described in this section.

This model should also be used when a `menubar` element is to be shown as an actual menu bar in native UI.

First, menu bars shall be built up from the children of their `menubar` element by processing each child node of that element as follows:

1. If the node is not an element node, it is ignored. (Fallback content.)
2. If the node is an element node but is not in the XHTML namespace, it is ignored. (Fallback content.)
3. If the node is an `li` element, then if the first element node in that element is one of the following:
   - An element that defines or refers to a command.
   - An `a` element with an `href` attribute whose URI points to the current document and contains a fragment identifier that points to a `menu` element.
   - A `menulabel` element whose next sibling element is a `menu` element.
   - A `commandset` element.
   - An `hr` element.

   ...then then continue the steps as if the `li` was that element. Otherwise, this `li` element is ignored. Non-element child nodes of the `li` element must be ignored. (Fallback content.)

4. If the node is a command of type Command, and the command's Hidden State is not hidden, then add that command to the menu.

5. If the node is an `a` element with an `href` attribute whose URI points to the current document and contains a fragment identifier that points to a `menu` element, then add that menu to the menu bar as a submenu, using the contents of the `textContent` DOM attribute of the `a` element as the menu label.

6. If the node is a `menulabel` element whose next sibling element is a `menu` element, then add that menu to the menu bar as a submenu, using the contents of the `menulabel` element's `label` attribute (if there is one) or of its `textContent` DOM attribute (if there isn't) as the menu label.

7. If the node is a `commandset` element, then add a separator to the menu, process all the children nodes of the element as if they were children of the `menu` element, then add another separator.

8. If the node is an `hr` element, then add a separator to the menu.

9. Otherwise, ignore the node. (Fallback content.)

Note: This processing model, while similar to the processing model for constructing menus, is intentionally different in many respects.

Once all the nodes have been processed in this way, any separators at the top of the menu and at the bottom of the menu shall be removed, and any consecutive separators shall be collapsed to just a single separator. If the menu is to be rendered in a way that does not support separators, then all separators should be dropped.

The Label, Icon, Hint, and Disabled State facets of the command should be appropriately reflected in the user interface created for the menu bar. (Checkbox and Radio commands cannot be added to a menu bar, so the Checked facet is ignored.)

Menu bars are live: changes to the underlying document structure must be reflected in the menu visible to the user immediately.

When commands are selected from the menu bar, their associated Action should be triggered.

3.3.3. Menu links

The default action of the `DOMActivate` event of `a` elements that do not define or

refer to commands is as follows:

1. If the `a` element has an `href` attribute, and that attribute points to the `a` element's document, and contains a fragment identifier that points to a `menu` element, then activate the menu element.

2. Otherwise, if the `a` element has an `href` attribute, then follow that link, taking into account any other relevant attributes on the element as appropriate.

Thus, any `a` element can be made to activate a menu by making it point to a `menu` element in the same document.

Note: By default, such `a` elements look like links, not like buttons or menus, unless they are placed inside `menubar` elements.

3.3.4. Context menus

This section will probably describe a `context-menu` attribute (or similar) which would be a common attribute and would refer to a `menu` element, allowing any element to get a context menu. This section would have to define how the context menu commands determine which element the menu was triggered on. It would also have to ensure that UAs can show their own context menu alongside the author-provided menu (or at least, give access to it).

3.4. Keyboard shortcuts

Support for the `accesskey` attribute is optional. User agents may use the attribute as a suggestion for a suitable shortcut key, or may ignore the attribute altogether. User agents should avoid letting author-specified access keys prevent users from accessing the UA's features.

Interactive user agents that support keyboard input devices should allow users to conveniently access or activate hyperlinks, form controls, and other interactive parts of Web content using the keyboard, without having to cycle through all such content.

Note: The `accesskey` attribute has numerous problems, such as not being discoverable by users, not being consistent with the interface on certain platforms, clashing with the user agent's own access keys or requiring unusual modifiers, being unable to handle the differing needs of platforms with varying keyboard

types, etc. Authors are discouraged from relying on this feature.

It is unclear what new features will be supported in Web Apps with respect to key handling, if any. Some sort of declarative way of listing key listeners that would take effect while a particular element has focus is possible, maybe with the key being given in a style sheet instead of the markup, allowing for a model where the user has final say, and allowing for per-device style sheets to be used to change the key based on the available input device(s).

4. Editing

4.1. The `contentEditable` attribute

This section will be based on the **`contentEditable`** attribute.

The **`contentEditable`** attribute is a common attribute. User agents must support this attribute on all HTML elements.

If an HTML element has a `contentEditable` attribute set to exactly the literal value `true`, or if its nearest ancestor with the `contentEditable` attribute set has its attribute set to exactly the literal value `true`, then the UA must treat the element as **editable** (as described below).

If an HTML element has a `contentEditable` attribute set but the value of the attribute is not exactly the literal value `true`, or if its nearest ancestor with the `contentEditable` attribute set is not *editable*, or if it has no ancestor with the `contentEditable` attribute set, then the element is not editable.

Authors must only use the values `true` and `false` with the `contentEditable` attribute.

If an element is editable and its parent element is not, then the element is an **editing host**. Editable elements can be nested, meaning the user can edit through them (see below). User agents must make editing hosts focusable (which typically means it enters 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. These nested editing hosts are not handled any differently to top-level editing hosts — they ...

How editable elements act depends on their CSS '`display`' type. (For non-CSS user agents, analogous rules should be followed.)

If an editable element is an inline box ('`display`' has the value '`inline`' or '`run-in`' and the result is an inline box), ...

4.2. Undo history

...

4.3. Drag and drop

This section defines an event-based drag-and-drop mechanism.

This specification does not define exactly what a *drag and drop operation* actually is.

On a visual medium with a pointing device, a drag operation could be the default action of a `mousedown` event that is followed by a series of `mousemove` events, and the drop could be triggered by the mouse being released.

On media without a pointing device, the user would probably have to explicitly indicate his intention to perform a drag-and-drop operation, stating what he wishes to drag and what he wishes to drop, respectively.

However it is implemented, drag and drop operations must have a starting point (e.g. where the mouse was clicked, or the start of the selection or element that was selected for the drag), may have any number of intermediate steps (elements that the mouse moves over during a drag, or elements that the user picks as possible drop points as he cycles through possibilities), and must either have an end point (the element above which the mouse button was released, or the element that was finally selected), or be canceled. The end point must be the last element selected as a possible drop point before the drop occurs (so if the operation is not canceled, there must be at least one element in the middle step).

4.3.1. Drag-and-drop processing model

There are two processing models for drag-and-drop: one for when a drag is initiated within the document, and one for when a drag is initiated in another (DOM-based) document or another application altogether, but the user has selected a node in the document as a drop target.

4.3.1.1. For drags initiated within the document

When the user attempts to begin a drag operation, the user agent must first

determine what is being dragged. If the drag operation was invoked on a selection, then it is the selection that is being dragged. Otherwise, it is the first element, going up the ancestor chain, starting at the node that the user tried to drag, that has the DOM attribute `draggable` set to true. If there is no such element, then nothing is being dragged, the drag-and-drop operation is never started, and the user agent must not continue with this algorithm.

If the user agent determines that something can be dragged, a `dragstart` event must then be fired.

If it is a selection that is being dragged, then this event must be fired on the node that the user started the drag on (typically the text node that the user originally clicked). If the user did not specify a particular node, for example if the user just told the user agent to begin a drag of "the selection", then the event must be fired on the deepest node that is a common ancestor of all parts of the selection.

If it is not a selection that is being dragged, then the event must be fired on the element that is being dragged.

The node on which the event is fired is the **source node**. Multiple events are fired on this node during the course of the drag-and-drop operation.

The `dataTransfer` member of the event must initially contain no nodes if a selection is being dragged, and just the source node otherwise.

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 selection if the user is dragging a selection, or from the nodes that were in the `dataTransfer` object's list after the event has been handled otherwise. In visual media, if an image was specified, then that image must be used instead.

The user agent must take a note of the data that was placed in the `dataTransfer` object.

From this point until the end of the drag-and-drop operation, mouse and key events must be suppressed. In addition, the user agent must track all DOM changes made during the drag-and-drop operation, and add them to its undo

history as one atomic operation once the drag-and-drop operation has ended.

During the drag operation, the element directly indicated by the user as the drop target is called the **immediate user selection**. (Only elements can be selected by the user; other nodes must not be made available as drop targets.)

However, the immediate user selection is not necessarily the element the **current target element**, the element currently selected for the drop part of the drag-and-drop operation.

The immediate user selection changes as the user selects different elements (either by pointing at them with a pointing device, or by selecting them in some other way). The current target element changes when the immediate user selection changes, based on the results of event handlers in the document, as described below.

Both the current target element and the immediate user selection can be null, which means no target element is selected. They can also both be elements in other (DOM-based) documents, or other (non-Web) programs altogether. (For example, a user could drag text to a word-processor.) The current target element is initially null.

In addition, there is also a **current drag operation**, which can take on the values "none", "copy", "link", and "move". Initially it has the value "none". It is updated by the user agent as described in the steps below.

User agents must, every 350ms (±200ms), perform the following steps in sequence. (If the user agent is still performing the previous iteration of the sequence when the next iteration becomes due, the user agent must not execute the overdue iteration, effectively "skipping missed frames" of the drag and drop operation.)

1. First, the user agent must fire a `drag` event at the source node.

2. Next, if the `drag` event was not canceled and the user has not ended the drag-and-drop operation, the user agent must check the state of the drag-and-drop operation, as follows:

    1. First, if the user is indicating a different immediate user selection than during the last iteration (or if this is the first iteration), and if this immediate user selection is not the same as the current target element, then the current target element must be updated, as follows:

        1. If the new immediate user selection is null, or is in a non-DOM document or application, then set the current target element to the same value.

2. Otherwise, the user agent must fire a `dragenter` event at the immediate user selection.

3. If the event is canceled, then the current target element must be set to the immediate user selection.

4. Otherwise, if the current target element is not the `body` element, the user agent must fire a `dragenter` event at the `body` element, and the current target element must be set to the `body` element, regardless of whether the event was canceled or not.

2. If the previous step caused the current target element to change, and if the previous target element was not null or a part of a non-DOM document, the user agent must fire a `dragleave` event at the previous target element.

3. If the current target element is a DOM element, the user agent must fire a `dragover` event at this current target element.

   If the `dragover` event is not canceled, the `dataTransfer` object's `dropEffect` attribute must then be reset to the value it was given when the event was fired.

   Then, regardless of whether the event was canceled or not, the drag feedback (e.g. the mouse cursor) must be updated to match the kind of drag-and-drop operation indicated by the event's `dataTransfer` object's `dropEffect` attribute, as follows:

`dropEffect`	Drag operation
`none`	No operation allowed.
`copy`	Data will be copied.
`link`	Data will be linked.
`move`	Data will be moved.

   The drag operation in question is the new *current drag operation*.

4. Otherwise, if the current target element is not a DOM element, the user agent must use platform-specific mechanisms to determine what drag operation is being performed (none, copy, link, or move). This sets the *current drag operation*.

3. Otherwise, if the user ended the drag and drop operation (e.g. by releasing the mouse button in a mouse-driven drag-and-drop interface), then this will be the last iteration. The user agent should follow the following steps, then stop looping.

1. If the current drag operation is none (no drag operation), or, if the user ended the drag-and-drop operation by canceling it (e.g. by hitting the `Escape` key), or if the current target element is null, then the drag operation failed. If the current target element is a DOM element, the user agent must fire a `dragleave` event at it; otherwise, if it is not null, it must use platform-specific conventions for drag cancellation.

2. Otherwise, the drag operation was as success. If the current target element is a DOM element, the user agent must fire a `drop` event at it; otherwise, it must use platform-specific conventions for indicating a drop.

   When the target is a DOM element, the `dropEffect` attribute of the event's `dataTransfer` object must be given the value representing the current drag operation (`copy`, `link`, or `move`), and the object must be set up so that the `getData()` method will return the data that was added during the `dragstart` event.

   Some elements have a default behaviour for "drop", e.g. textarea receives new text. Cancelable.

3. Finally, the user agent must fire a `dragend` event at the source node.

   Some elements have a default behaviour for "dragend", e.g. textarea deletes source text in a move. NOT cancelable.

The events must be fired as described above, even if the nodes are in different documents (assuming those are DOM-based). User agents should handle cases where the target is not in a DOM-based document according to the platform conventions.

4.3.1.2. For drags initiated in other documents or applications

...

4.3.2. The `draggable` attribute

...

The **`draggable`** DOM attribute reflects the `draggable` content attribute. However, the default value varies based on the element type For `img` elements, the default is true. For `a` elements, the default is true if the element has an `href` content

attribute, and false otherwise. For all other elements, the default is false.

4.3.3. The `DragEvent` interface and the `dataTransfer` object

Need to define DragEvent interface.

```
interface DataTransfer {
           attribute DOMString dropEffect;
           attribute DOMString effectAllowed;
  void clearData(in DOMString format);
  void setData(in DOMString format, in DOMString data);
  DOMString getData(in DOMString format);

  // XXX addElement, dragImage, etc
};
```

Need to define DataTransfer members

When a `DragEvent` event object is initialised by the user agent for the purposes of the drag-and-drop model described above (as opposed to when a custom `DragEvent` event object is created by author script), the object must be initialised as follows.

- Its `dataTransfer` member must be initialised to a new instance of a `DataTransfer` object.
- That object must initially contain no elements and have no associated image.
- The `dataTransfer` object's `effectAllowed` attribute must be set to "`uninitialized`" for `dragstart` events, and to whatever value the field had after the last drag-and-drop event was fired for all other events (only counting events fired by the user agent for the purposes of the drag-and-drop model described above).
- The `dropEffect` attribute must be set to "`none`" for `dragstart`, `drag`, `dragleave`, and `dragend` events (except when stated otherwise in the algorithms given in the earlier sections), and to a value based on the `effectAllowed` attribute's value and to the drag-and-drop source, as given by the following table, for other events:

effectAllowed	**dropEffect**
`none`	`none`
`copy`, `copyLink`, `copyMove`, `all`	`copy`

effectAllowed	dropEffect
link, linkMove	link
move	move
uninitialised when what is being dragged is a selection from a text field	move
uninitialised when what is being dragged is a selection	copy
uninitialised when what is being dragged is an a element with an href attribute	link
Any other case	copy

4.3.4. Events fired during a drag-and-drop action

This section is non-normative. It merely summarises the preceeding sections.

The following events are involved in the drag-and-drop model. They all use the DragEvent interface.

Event Name	Target	Bubbles?	Cancelable?	dataTransfer	effectAllowed	d
dragstart	Source node	✓ Bubbles	✓ Cancelable	Contains source node unless a selection is being dragged, in which case it is empty	uninitialized	nor
drag	Source node	✓ Bubbles	✓ Cancelable	—	Same as last event	nor
dragenter	Immediate user selection or the body element	✓ Bubbles	✓ Cancelable	—	Same as last event	Ba eff val
dragleave	Previous target element	✓ Bubbles	—	—	Same as last event	nor

Event Name	Target	Bubbles?	Cancelable?	dataTransfer	effectAllowed	d
dragover	Current target element	✓ Bubbles	✓ Cancelable	—	Same as last event	Ba eff val
drop	Current target element	✓ Bubbles	✓ Cancelable	getData() returns data set in dragstart event	Same as last event	Cu op
dragend	Source node	✓ Bubbles	✓ Cancelable	—	Same as last event	nor

5. Script and the DOM

Applications typically involve an element of interactivity implemented programmatically. This section defines some APIs that complement the APIs defined by the W3C DOM specifications.

5.1. Bootstrapping the DOM and the Window interface

The object implementing the AbstractView interface that represents the default view of a document (as obtained through the defaultView attribute of the DocumentView interface) must also implement the Window interface defined below. [DOM2VIEWS]

The following equality must always hold (assuming appropriate casting has been applied, as required by the binding):

```
window.document.defaultView == window
```

In this equality, window is the object implementing the Window interface, the document property of that object is the document attribute of the AbstractView interface implemented by the window object, and the defaultView property of *that* object is the defaultView attribute of the DocumentView interface, which must, as described by the equality, be the original window object. The object returned by the document property of the AbstractView interface must implement the Document interface as well.

```
interface Window {
  readonly attribute Window window;

  // (part of AbstractView interface)
  // readonly attribute Document document;

           attribute ErrorHandler       onerror;

  readonly attribute History history;
  readonly attribute Location location;

  // timers
  long setTimeout(in TimeoutHandler handler, in long timeout);
  long setTimeout(in TimeoutHandler handler, in long timeout, arguments
  long setTimeout(in DOMString code, in long timeout);
  long setTimeout(in DOMString code, in long timeout, in DOMString lang
  void clearTimeout(in long handle);
  long setInterval(in TimeoutHandler handler, in long timeout);
  long setInterval(in TimeoutHandler handler, in long timeout, argument
  long setInterval(in DOMString code, in long timeout);
  long setInterval(in DOMString code, in long timeout, in DOMString lan
  void clearInterval(in long handle);


};

interface ErrorHandler {
  void handleEvent(in DOMString errorMessage, in DOMString fileName, in
};

interface TimeoutHandler {
  void handleEvent(arguments...);
};
```

In UAs that expose the DOM to ECMAScript [ECMA262] scripts, the global scope object must implement the `Window` interface described above.

The **`window`** attribute of an object implementing the `Window` interface must always point to the object itself. In other words, the following equality must also always hold:

```
x.window == x
```

...where *x* is an object implementing the `Window` interface.

Thus, in ECMAScript, the ECMAScript global object must have a property `window` pointing at the global object itself.

5.1.1. Error handling

The **`onerror`** attribute takes a reference to an object implementing the

ErrorHandler interface. In ECMAScript, such an interface is implemented by any function that takes three arguments and returns a boolean value, as well as by the `null` value and the `undefined` value.

The function to which the `onerror` attributes points is invoked whenever a runtime script error occurs in the context of the `window` object, before the error is reported to the user. If the function is `null` or if the function returns true then the error is not reported to the user. If the function is `undefined` or if the function doesn't returns true, then the message is reported as normal.

The three arguments passed to the function are all `DOMString`s; the first gives the message that the UA is considering reporting, the second gives the URI to the resource in which the error occured, and the third gives the line number in tha resource on which the error occured.

The initial value of `onerror` is `undefined`.

5.1.2. Timers

The `setTimeout` and `setInterval` methods allow authors to schedule timer-based events.

The **setTimeout(*handler*, *timeout*[, *arguments...*])** method takes a reference to a `TimeoutHandler` object and a length of time in milliseconds. It returns a handle to the timeout created, and then asynchronously waits *timeout* milliseconds and then invokes `handleEvent()` on the *handler* object. If any *arguments...* were provided, they are passed to the *handler* as arguments to the `handleEvent()` function.

In the ECMAScript DOM binding, the ECMAScript native `Function` type must implement the `TimeoutHandler` interface such that invoking the `handleEvent()` method of that interface on the object invokes the function itself, with the arguments passed to `handleEvent()` as the arguments passed to the function. Such functions must be called in the global scope.

Alternatively, **setTimeout(*code*, *timeout*[, *language*])** may be used. This variant takes a string instead of a `TimeoutHandler` object. That string is then parsed using the specified language (defaulting to ECMAScript if the third argument is omitted) and executed in the global scope.

The **setInterval(...)** variants work in the same way as the code>setTimeout variants except that the *handler* or `code` is invoked again every *timeout* milliseconds, not just the once.

The **clearTimeout()** and **clearInteval()** methods take one integer (the value returned by `setTimeout` and `setInterval` respectively) and 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.

5.1.3. Session history

the History object and window.history; also the new pushState() method (push a JS object onto the session history) and the onpopstate="" event handler

In user agents that support a sequential browsing model, where following a link in one document can cause the document to be pushed onto a history stack and replaced by the new document, the history stack must be represented in the scripting environment by an object implementing the `History` interface. The **`history`** attribute of the `Window` interface must return this object.

```
interface History {
  readonly attribute long length;
  readonly attribute DOMString current;
  readonly attribute DOMString previous;
  readonly attribute DOMString next;
  void back();
  void forward();
  void go();
  void go(in unsigned long index);
  void pushState(in DOMObject data);
}
```

`History` objects provide a representation of the list of pages in the user's session history *for that* *`window`*. Each `frame`, `iframe`, etc, has a distinct session history.

5.1.3.1. Implementaton notes

This section is non-normative.

The `History` interface is not meant to place restrictions on how implementations represent the session history to the user.

For example, session history could be implemented in a tree-like manner, with each page having multiple "forward" pages. This specification doesn't define how the linear list of pages in the `history` object are derived from the actual session history as seen from the user's perspective.

Similarly, a page containing two `iframe`s has a `history` object distinct from the `iframe`s' `history` objects, but 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.

It is suggested that to avoid letting a page "hijack" the history navigation facilities of a UA by abusing `pushState()`, the UA provide the user with a way to jump back to the previous page (rather than just going back to the previous state). For example, the back button could have a drop down showing just the pages in the session history, and not showing any of the states. Similarly, an aural browser could have two "back" commands, one that goes back to the previous state, and one that jumps straight back to the previous page.

5.1.4. Navigation

the Location object and window.location

5.2. Selecting elements

Both `Document`s and `Element`s shall also implement the `GetElementsByClassName` interface:

```
interface GetElementsByClassName {
  NodeList getElementsByClassName(in DOMString className1 [, in DOMStri
}
```

This interface defines one method, **`getElementsByClassName()`**, which takes one or more strings representing classes and returns all the elements in that document or below that element that are of all those classes. HTML, XHTML, SVG and MathML elements define which classes they are in by having an attribute in the per-element partition with the name `class` containing a space-separated list of classes to which the element belongs. Other specifications may also allow elements in their namespaces to be labelled as being in specific classes. UAs must not assume that all attributes of the name `class` for elements in any namespace work in this way, however, and must not assume that such attributes, when used as global attributes, label other elements as being in specific classes.

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

> `document.getElementById('example').getElementsByClassName('aaa')` would return a `NodeList` with the two paragraphs `p1` and `p2` in it. A call to `getElementsByClassName('ccc', 'bbb')` would only return one node, however, namely `p3`.

5.3. Navigating DOM trees

All objects that implement the `Node` interface shall also implement the `ElementTraversal` interface:

```
// Originally defined in SVG 1.2 Appendix A
interface ElementTraversal {
  readonly attribute Element                 firstElementChild;
  readonly attribute Element                 lastElementChild;
  readonly attribute Element                 nextElementSibling;
  readonly attribute Element                 previousElementSibling;
};
```

The **`firstElementChild`** and **`lastElementChild`** attributes shall return the first element child and last element child (respectively) of their node. If there is no such child, they shall return null.

The **`nextElementSibling`** and **`previousElementSibling`** attributes shall return the first element to follow the current node and the first element to precede the current node (respectively). If there is no such element, they shall return null.

5.4. Serialization and parsed fragment replacement

This section will try to explain how `document.write()` actually works (HTML only), and will define the `innerHTML` attribute, for both HTML and XML contexts. Wish us luck.

5.5. Alternate style sheets

```
// Introduced in DOM Level 2: [DOM2STYLE]
interface DocumentStyle {
  readonly attribute StyleSheetList   styleSheets;

  // New in this specification:
           attribute DOMString        selectedStylesheetSet;
  readonly attribute DOMString        lastStylesheetSet;
  readonly attribute DOMString        preferredStylesheetSet;
  readonly attribute DOMStringList    stylesheetSets;
```

```
  void                                enableStylesheetsForSet(in DOMStr
};
```

For this interface, the `DOMString` values "null" and "the empty string" are distinct, and must not be considered equivalent.

The new members are defined as follows:

`selectedStylesheetSet` of type `DOMString`

This attribute indicates which style sheet set ([HTML4]) is in use. This attribute is live; changing the disabled attribute on style sheets directly will change the value of this attribute.

If all the sheets that are enabled have the same title (by case insensitive comparisons) then the value of this attribute shall be exactly equal to the title of the first enabled style sheet with a title in the `styleSheets` list. If style sheets from different sets are enabled, then the return value shall be null (there is no way to determine what the currently selected style sheet set is in those conditions). Otherwise, either all style sheets are disabled, or there are no alternate style sheets, and `selectedStylesheetSet` must return the empty string.

Setting this attribute to the null value shall have no effect.

Setting this attribute to a non-null value must call `enableStylesheetsForSet()` with that value as the function's argument, then set `lastStylesheetSet` to that value.

From the DOM's perspective, all views have the same `selectedStylesheetSet`. If a UA supports multiple views with different selected alternate style sheets, then this attribute (and the `StyleSheet` interface's `disabled` attribute) must return and set the value for the default view.

`lastStylesheetSet` of type `DOMString`, readonly

This property shall return the last value that `selectedStylesheetSet` was set to, or, if none, null.

`preferredStylesheetSet` of type `DOMString`, readonly

This attribute shall indicate the preferred style sheet set as set by the author. It is determined from the order of style sheet declarations and the `Default-Style` HTTP headers. [HTML4]. If there is no preferred style sheet set, this attribute must return the empty string. The case of this attribute must exactly match the case given by the author where the preferred style sheet is specified or implied. This attribute must never return null.

`stylesheetSets` of type `DOMStringList`, readonly

This must return the live list of the currently available style sheet sets. This list is constructed by enumerating all the style sheets for this document available to the implementation, in the order they are listed in the `styleSheets` attribute, adding the title of each style sheet with a title to the list, avoiding duplicates by dropping titles that match (case insensitively) titles that have already been added to the list.

`enableStylesheetsForSet(`*`name`*`)`, method

Calling this method must change the `disabled` attribute on each `StyleSheet` object with a title attribute with a length greater than 0 in the `styleSheets` attribute, so that all those whose title matches the *name* argument are enabled, and all others are disabled. Title matches are case insensitive. Calling this method with the empty string disables all alternate and preferred style sheets (but does not change the state of persistent style sheets, that is those with no title attribute).

Calling this method with a null value must have no effect.

Style Sheets that have no title are never affected by this method. This method does not change the values of the `lastStylesheetSet` or `preferredStylesheetSet` attributes.

5.5.1. Dynamically adding new style sheets

If new style sheets with titles are added to the document, the UA must decide whether or not the style sheets should be initially enabled or not. How this happens depends on the exact state of the document at the time the style sheet is added, as follows.

5.5.1.1. Adding style sheets

First, if the style sheet is a preferred style sheet (it has a title, but is not marked as alternate), and there is no current preferred style sheet (the `preferredStylesheetSet` attribute is equal to the empty string) then the `preferredStylesheetSet` attribute is set to the exact value of this style sheet's title. (This changes the preferred style sheet set, which causes further changes — see below.)

Then, for all sheets, if any of the following is true, then the style sheet must be enabled:

- The style sheet has an empty title.
- The `lastStylesheetSet` is null, and the style sheet's title matches (by case insensitive match) the value of the `preferredStylesheetSet` attribute.

- The style sheet's title matches (by case insensitive match) the value of the `lastStylesheetSet` attribute.

Otherwise, the style sheet must be disabled.

5.5.1.2. Changing the preferred style sheet set

If the UA has the preferred style sheet set changed, for example if it receives a "Default-Style:" HTTP header after it receives HTTP "Link:" headers implying another preferred style sheet, then the `preferredStylesheetSet` attribute's value must be changed appropriately, and, if the `lastStylesheetSet` is null, the `enableStylesheetsForSet()` method must be called with the new `preferredStylesheetSet` value. (The `lastStylesheetSet` attribute is not changed.)

5.5.1.3. Examples

> Thus, in the following HTML snippet:
>
> ```
> <link rel="alternate stylesheet" title="foo" href="a">
> <link rel="alternate stylesheet" title="bar" href="b">
> <script>
>   document.selectedStylesheetSet = 'foo';
>   document.styleSheets[1].disabled = false;
> </script>
> <link rel="alternate stylesheet" title="foo" href="c">
> <link rel="alternate stylesheet" title="bar" href="d">
> ```
>
> ...the style sheets that end up enabled are style sheets "a", "b", and "c", the `selectedStylesheetSet` attribute would return null, `lastStylesheetSet` would return "foo", and `preferredStylesheetSet` would return "".
>
> Similarly, in the following HTML snippet:
>
> ```
> <link rel="alternate stylesheet" title="foo" href="a">
> <link rel="alternate stylesheet" title="bar" href="b">
> <script>
>   var before = document.preferredStylesheetSet;
>   document.styleSheets[1].disabled = false;
> </script>
> <link rel="stylesheet" title="foo" href="c">
> <link rel="alternate stylesheet" title="bar" href="d">
> <script>
>   var after = document.preferredStylesheetSet;
> </script>
> ```
>
> ...the "before" variable will be equal to the empty string, the "after" variable will be equal to "foo", and style sheets "a" and "c" will be enabled. This is the case even though the first script block sets style sheet "b" to be enabled, because upon parsing the following `<link>` element, the

> `preferredStylesheetSet` is set and the `enableStylesheetsForSet()` method is called (since `selectedStylesheetSet` was never set explicitly, leaving `lastStylesheetSet` at null throughout), which changes which style sheets are enabled and which are not.

5.5.2. Interaction with the User Interface

The user interface of Web browsers that support style sheets should list the style sheet titles given in the `stylesheetSets` list, showing the `selectedStylesheetSet` as the selected style sheet set, leaving none selected if it is null or the empty string, and selecting an extra option "Basic Page Style" (or similar) if it is the empty string and the `preferredStylesheetSet` is the empty string as well.

Selecting a style sheet from this list should set the `selectedStylesheetSet` attribute.

5.5.2.1. Persisting the selected style sheet set

If UAs persist the selected style sheet set, they should use the value of the `selectedStylesheetSet` attribute, or if that is null, the `lastStylesheetSet` attribute, when leaving the page (or at some other time) to determine the set name to store. If that is null then the style sheet set should not be persisted.

When re-setting the style sheet set to the persisted value (which can happen at any time, typically at the first time the style sheets are needed for styling the document, after the `<head>` of the document has been parsed, after any scripts that are not dependent on computed style have executed), the style sheet set should be set by setting the `selectedStylesheetSet` attribute as if the user had selected the set manually.

> ***Note: This specification does not give any suggestions on how UAs should decide to persist the style sheet set or whether or how to persist the selected set across pages.***

5.5.3. Future compatibility

Future versions of CSS may introduce ways of having alternate style sheets declared at levels lower than the top level, i.e. embedded within other style sheets. Implementations of this specification that also support this proposed declaration of alternate style sheets are expected to perform depth-first traversals of the `styleSheets` list, not simply enumerations of the `styleSheets` list that only contains the top level.

5.6. Events

We need a section to define how events all work, default actions, etc. For example, how does clicking on a span in a link that is in another link actually cause a link to be followed? which one?

5.6.1. Event listeners

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 invokes the function itself, with the `event` argument as its only argument. Such functions must be called in the global scope. If the function returns false, the event's `preventDefault()` method must then invoked. Exception: for historical reasons, for the HTML `mouseover` event, the `preventDefault()` method must be called when the function returns true instead.

In HTML, event handler attributes (such as `onclick`) are invoked as if they were functions implementing `EventListener`, with the argument called `event`. Such attributes are added as non-capture event listeners of the type given by their name (without the leading `on` prefix). Only attributes actually defined to exist by specifications implemented by the UA (e.g. HTML, Web Forms 2, Web Apps) are actually registered, however; for example if an author created an `onfoo` attribute, it would not be fired for `foo` events.

The scope chain for ECMAScript executed in HTML event handler attributes must link from the activation object for the handler, to its `this` parameter (the event target), to the element's `form` element if it is a form control, to the document, to the default view (the `window`).

Note: This definition is intentionally backwards compatible with DOM Level 0. See also ECMA262 Edition 3, sections 10.1.6 and 10.2.3, for more details on activation objects. [ECMA262]

6. Multimedia

should we move all the img, object, embed, iframe, etc, elements here?

6.1. Dynamic graphics: The bitmap canvas

The **canvas** element represents a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, or other visual images on the fly.

When authors use the canvas element, they should also provide content that, when presented to the user, conveys essentially the same function or purpose as the bitmap canvas. This content may be placed as content of the canvas element.

Authors should not use the canvas element in a document when a more suitable element is available. For example, it is inappropriate to use a canvas element to render a page heading: if the desired presentation of the heading is graphically intense, it should be marked up using appropriate elements (typically h1) and then styled using CSS and supporting technologies such as XBL.

In non-visual media, and in visual media with scripting disabled, the canvas element should be treated as an ordinary block-level element and the fallback content should therefore be used instead.

In non-interactive, static, visual media, if the canvas element has been previously painted on (e.g. if the page was viewed in an interactive visual media and is now being printed, or if some script that ran during the page layout process painted on the element), then the canvas element should be treated as a replaced block-level element with the current image and size. Otherwise, the element should be treated as an ordinary block-level element and the fallback content should therefore be used instead.

In interactive visual media with scripting enabled, the canvas element is a block-level replaced element.

In CSS-aware user agents, this should be achieved by including the following rules, or their equivalent, in the UA's user agent style sheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|canvas { display: block; }
```

The canvas element has two attributes to control the size of the coordinate space: height and width. These attributes each take a positive integer value (one digit in the range 1-9 followed by zero or more digits in the range 0-9, interpreted in base ten). If an attribute is missing, or if it has a value that does not match this syntax, then the default values must be used instead. The width attribute defaults to 300, and the height attribute defaults to 150.

The intrinsic dimensions of the canvas element equal the size of the coordinate space, with the numbers interpreted in CSS pixels. However, the element can be sized arbitrarily by a style sheet. During rendering, the image is scaled to fit

this layout size.

The size of the coordinate space does not necessarily represent the size of the actual bitmap that the user agent will use internally or during rendering. On high-definition displays, for instance, the user agent may internally use a bitmap with two device pixels per unit in the coordinate space, so that the rendering remains at high quality throughout.

If the `width` and `height` attributes are dynamically modified, the bitmap and any associated contexts must be cleared back to their initial state and reinitialised with the newly specified coordinate space dimensions.

The canvas is initially fully transparent black. Whenever the `width` and `height` attributes are changed, the canvas must be cleared back to this state.

As with any replaced element, the CSS background properties do apply to `canvas` elements; they are rendered below the canvas image.

```
interface HTMLCanvasElement : HTMLElement {

  // returns the values of the width and height attributes, or the assu
  // defaults if the attributes were not specified or invalid
  // sets the relevant content attributes on setting
         attribute long                    width;
         attribute long                    height;

  // returns a data: URI representing the current image as a PNG
  DOMString toDataURL();

  // returns the context with which to paint, see below
  DOMObject getContext(in DOMString contextID);

};
```

To draw on the canvas, authors must first obtain a reference to a **context** using the **`getContext`** method of the `canvas` element.

This specification only defines one context, with the name "`2d`". If `getContext()` is called with that exact string, then the UA must return a reference to an object implementing `CanvasRenderingContext2D`. Other specifications may define their own contexts, which would return different objects.

Vendors may also define experimental contexts using the syntax *`vendorname-context`*, for example, `moz-3d`.

When the UA is passed an empty string or a string specifying a context that it does not support, then it must return null. String comparisons should be literal and case sensitive.

Note: A future version of this specification will probably define a `3d` context (probably based on the OpenGL ES API).

The **`toDataURL()`** method must return a `data:` URI containing a representation of the image as a PNG file. [PNG].

6.1.1. The 2D context

When the `getContext()` method of a `canvas` element is invoked with **`2d`** as the argument, a `CanvasRenderingContext2D` object is returned.

There is only one `CanvasRenderingContext2D` object per canvas, so calling the `getContext()` method with the `2d` argument a second time must return the same object.

```
interface CanvasRenderingContext2D {

  // back-reference to the canvas
  readonly attribute HTMLCanvasElement       canvas;

  // state
  void save(); // push state on state stack
  void restore(); // pop state stack and restore state

  // transformations (default transform is the identity matrix)
  void scale(in float x, in float y);
  void rotate(in float angle);
  void translate(in float x, in float y);

  // compositing
           attribute float                   globalAlpha; // (default 1
           attribute DOMString               globalCompositeOperation;

  // colours and styles
           attribute DOMObject               strokeStyle; // (default b
           attribute DOMObject               fillStyle; // (default bla
  CanvasGradient createLinearGradient(in float x0, in float y0, in floa
  CanvasGradient createRadialGradient(in float x0, in float y0, in floa
  CanvasPattern createPattern(in HTMLImageElement image, DOMString repe
  CanvasPattern createPattern(in HTMLCanvasElement image, DOMString rep

  // line caps/joins
           attribute float                   lineWidth; // (default 1)
           attribute DOMString               lineCap; // "butt", "round
           attribute DOMString               lineJoin; // "round", "bev
           attribute float                   miterLimit; // (default 10

  // shadows
           attribute float                   shadowOffsetX; // (default
           attribute float                   shadowOffsetY; // (default
```

```
           attribute float                  shadowBlur; // (default 0)
           attribute DOMString              shadowColor; // (default b

  // rects
  void clearRect(in float x, in float y, in float w, in float h);
  void fillRect(in float x, in float y, in float w, in float h);
  void strokeRect(in float x, in float y, in float w, in float h);

  // path API
  void beginPath();
  void closePath();
  void moveTo(in float x, in float y);
  void lineTo(in float x, in float y);
  void quadraticCurveTo(in float cpx, in float cpy, in float x, in floa
  void bezierCurveTo(in float cp1x, in float cp1y, in float cp2x, in fl
  void arcTo(in float x1, in float y1, in float x2, in float y2, in flo
  void rect(in float x, in float y, in float w, in float h);
  void arc(in float x, in float y, in float radius, in float startAngle
  void fill();
  void stroke();
  void clip();

  // drawing images
  void drawImage(in HTMLImageElement image, in float dx, in float dy);
  void drawImage(in HTMLImageElement image, in float dx, in float dy, i
  void drawImage(in HTMLImageElement image, in float sx, in float sy, i
  void drawImage(in HTMLCanvasElement image, in float dx, in float dy);
  void drawImage(in HTMLCanvasElement image, in float dx, in float dy,
  void drawImage(in HTMLCanvasElement image, in float sx, in float sy,

  // drawing text is not supported in this version of the API
  // (there is no way to predict what metrics the fonts will have,
  // which makes fonts very hard to use for painting)

};

interface CanvasGradient {
  // opaque object
  void addColorStop(in float offset, in DOMString color);
}

interface CanvasPattern {
  // opaque object
}
```

The **canvas** attribute returns the canvas element that the context paints on.

6.1.1.1. The canvas state

Each context maintains a stack of drawing states. **Drawing states** consist of:

- The current transformation matrix.

- The current clip region.
- The current values of the following attributes: `strokeStyle`, `fillStyle`, `globalAlpha`, `lineWidth`, `lineCap`, `lineJoin`, `miterLimit`, `shadowOffsetX`, `shadowOffsetY`, `shadowBlur`, `shadowColor`, `globalCompositeOperation`.

Note: The current path and the current bitmap are not part of the drawing state. The current path is persistent, and can only be reset using the `beginPath()` method. The current bitmap is a property of the canvas, not the context.

The **`save()`** method pushes a copy of the current drawing state onto the drawing state stack.

The **`restore()`** method pops the top entry in the drawing state stack, and resets the drawing state it describes. If there is no saved state, the method does nothing.

6.1.1.2. 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 is initially the identity transform. It may then be adjusted using the three transformation methods.

The transformations are 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 adds a scaling transformation 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 adds a rotation transformation to the transformation matrix. The `angle` argument represents a clockwise rotation angle expressed in radians.

The **`translate(x, y)`** method adds a translating transformation 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.

6.1.1.3. Compositing

All drawing operations are affected by the global compositing attributes, `globalAlpha` and `globalCompositeOperation`.

The **`globalAlpha`** attribute gives an alpha value that is applied to shapes and images before they are composited onto the canvas. The valid range of values is from 0.0 (fully transparent) to 1.0 (no additional transparency). If the attribute is set to values outside this range, they are ignored. When the context is created, the `globalAlpha` attribute initially has the value 1.0.

The **`globalCompositeOperation`** attribute sets how shapes and images are drawn onto the existing bitmap, once they have had `globalAlpha` and the current transformation matrix applied. It may be set to any of the values in the following list. In the descriptions below, the source image is the shape or image being rendered, and the destination image is the current state of the bitmap.

`source-atop`

Display the source image wherever both images are opaque. Display the destination image wherever the destination image is opaque but the source image is transparent. Display transparency elsewhere.

`source-in`

Display the source image wherever both the source image and destination image are opaque. Display transparency elsewhere.

`source-out`

Display the source image wherever the source image is opaque and the destination image is transparent. Display transparency elsewhere.

`source-over` **(default)**

Display the source image wherever the source image is opaque. Display the destination image elsewhere.

`destination-atop`

Same as `source-atop` but using the destination image instead of the source image and vice versa.

`destination-in`

Same as `source-in` but using the destination image instead of the source image and vice versa.

`destination-out`

Same as `source-out` but using the destination image instead of the source image and vice versa.

`destination-over`

Same as `source-over` but using the destination image instead of the source image and vice versa.

`darker`

Display the sum of the source image and destination images, with color values approaching 0 as a limit.

`lighter`

Display the sum of the source image and destination image, with color values approaching 1 as a limit.

`copy`

Display the source image instead of the destination image.

`xor`

Exclusive OR of the source and destination images.

`vendorName-operationName`

Vendor-specific extensions to the list of composition operators should use this syntax.

If the user agent does not recognise the specified value, it must be ignored, leaving the value of `globalCompositeOperation` unaffected.

When the context is created, the `globalCompositeOperation` attribute initially has the value `source-over`.

6.1.1.4. Colours and styles

The **`strokeStyle`** attribute represents the colour or style to use for the lines around shapes, and the **`fillStyle`** attribute represents the colour or style to use inside the shapes.

Both attributes can be either strings, `CanvasGradient`s, or `CanvasPattern`s. On setting, strings should be parsed as CSS <color> values. [CSS3COLOR] If the value is a string but is not a valid colour, or is neither a string, a `CanvasGradient`, nor a `CanvasPattern`, then it must be ignored, and the attribute must retain its previous value.

On getting, if the value is a color, then: if it has alpha equal to 1.0, then the colour 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), a literal U+0020 SPACE and U+002C COMMA, an integer for the green component, a space and a comma, an integer for the blue component, another space and comma, a U+0030 DIGIT ZERO, a U+002E FULL STOP (representing the decimal point), one or more digits in the range 0-9 (U+0030 to U+0039) representing the fractional part of the alpha value, and finally a U+0029 RIGHT PARENTHESIS.

Otherwise, if it is not a color but a `CanvasGradient` or `CanvasPattern`, then an object supporting those interfaces must be returned. Such objects are opaque and therefore only useful for assigning to other attributes or for comparison to other gradients or patterns.

When the context is created, the `strokeStyle` and `fillStyle` attributes 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 colours are distributed along the gradient. Between each such stop, the colours and the alpha component are interpolated over the RGBA space to find the colour to use at that offset. Immediately before the 0 offset and immediately after the 1 offset, transparent black stops are assumed.

The **`addColorStop(offset, color)`** method on the `CanvasGradient` interface adds a new stop to a gradient. If the *offset* is less than 0 or greater than 1 then an `INDEX_SIZE_ERR` exception is raised. If the *color* cannot be parsed as a CSS colour, then a `SYNTAX_ERR` exception is raised. Otherwise, the gradient is 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 returns a linear `CanvasGradient` initialised with that line.

Linear gradients are rendered such that at the starting point on the canvas the colour 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 colour at the point where those two lines cross, and that any points beyond the start or end points are a transparent black. (Of course, the colours are only painted where the shape they are being painted on needs them.)

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 returns a radial `CanvasGradient` initialised with those two circles.

Radial gradients are rendered such that a cone is created from the two circles, so that at the circumference of the starting circle the colour 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 colour at that offset, that the end circle appear to be above the start circle when the end circle is not completely enclosed by the start circle, and that any points not described by the gradient are a transparent black.

If a gradient has no stops defined, then the gradient is 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 colour. However, `createLinearGradient()` and `createRadialGradient()` must always return objects when passed valid arguments.

Patterns are represented by objects implementing the opaque **`CanvasPattern`** interface.

To create objects of this type, the **`createPattern(image, repetition)`** method is used. The first argument gives the image to use as the pattern (either an `HTMLImageElement` or an `HTMLCanvasElement`). Modifying this image after calling the `createPattern()` method must not affect the pattern. The second argument must be a string with one of the following values: `repeat`, `repeat-x`, `repeat-y`, `no-repeat`. If the empty string or null is specified, `repeat` is assumed. If an unrecognised value is given, then the user agent must raise a `SYNTAX_ERR` exception. The method returns a `CanvasPattern` object suitably initialised.

Patterns are painted so that the first image is centered in the middle 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 shall not be scaled by this process; one CSS pixel of the image is painted on one coordinate space unit. Of course, patterns must only actually painted where the stroking or filling effect requires that they be drawn.

Support for patterns is optional. If the user agent doesn't support patterns, then

`createPattern()` must return null.

6.1.1.5. Line styles

The **`lineWidth`** attribute gives the default width of lines, in coordinate space units. On setting, zero and negative values are ignored, and leave the value unchanged.

When the context is created, the `lineWidth` attribute initially has 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` are ignored and leave the value unchanged.

When the context is created, the `lineCap` attribute initially has the value `butt`.

The **`lineJoin`** attribute defines the type of corners that that UAs shall 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` are ignored and leave the value unchanged.

When the context is created, the `lineJoin` attribute initially has 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, is rendered at the join. 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, is rendered at the join. The `miter` value means that a filled four- or five-sided polygon is 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 is the maximum allowed ratio of the miter length to the line width. If the miter limit would be exceeded, then a fifth line is 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 are ignored, and leave the value unchanged.

When the context is created, the `miterLimit` attribute initially has the value `10.0`.

6.1.1.6. 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 colour of the shadow.

When the context is created, the `shadowColor` attribute initially is fully-transparent black.

The **`shadowOffsetX`** and **`shadowOffsetY`** attributes specify the distance that the shadow should 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 should cover. On setting, negative numbers are ignored and leave the attribute unmodified.

When the context is created, the `shadowBlur` attribute initially has the value `0`.

Support for shadows is optional.

6.1.1.7. Shapes

There are three methods that immediately draw rectangles to the bitmap. They each take four arguments; the first two give the *x* and *y* coordinates of the top left of the rectangle, and the second two give the width and height of the rectangle, respectively.

Shapes are painted without affecting the current path, and are subject to transformations, shadow effects, global alpha, clipping paths, and global composition operators.

Negative values for width and height must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`clearRect()`** method clears the pixels in the specified rectangle to a fully transparent black, erasing any previous image.

The **`fillRect()`** method paints the specified rectangular area using the `fillStyle`.

The **`strokeRect()`** method draws a rectangular outline of the specified size using the `strokeStyle`, `lineWidth`, `lineJoin`, and (if appropriate) `miterLimit` attributes.

6.1.1.8. 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 subpaths and a current position. Each subpath consists of a list of points, some of which may be connected by straight and curved lines, and a flag indicating whether the subpath is closed or not.

The **`beginPath()`** method resets the list of subpaths to an empty list, and calls `moveTo()` with the point (0,0). When the context is created, a call to `beginPath()` is implied.

The **`moveTo(x, y)`** method sets the current position to the given coordinate and creates a new subpath with that point as its first (and only) point. If there was a previous subpath, and it consists of just one point, then that subpath is removed from the path.

The **`closePath()`** method adds a straight line from the current position to the first point in the last subpath and marks the subpath as closed, if the last subpath isn't closed, and if it has more than one point in its list of points. If the last subpath is not open or has only one point, it does nothing.

The **`lineTo(x, y)`** method adds the given coordinate (*x*, *y*) to the list of points of the subpath, and connects the current position to that point with a straight line. It then sets the current position to the given coordinate (*x*, *y*).

The **`quadraticCurveTo(cpx, cpy, x, y)`** method adds the given coordinate (*x*, *y*) to the list of points of the subpath, and connects the current position to that point with a quadratic curve with control point (*cpx*, *cpy*). It then sets the current position to the given coordinate (*x*, *y*).

The **`bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)`** method adds the given coordinate (*x*, *y*) to the list of points of the subpath, and connects the two points with a bezier curve with control points (*cp1x*, *cp1y*) and (*cp2x*, *cp2y*). It then sets the current position to the given coordinate (*x*, *y*).

The **`arcTo(x1, y1, x2, y2, radius)`** method adds an arc to the current path.

The arc is given by the circle that has one point tangent to the line from the current position to point (*x1*, *y1*), one point tangent to the line from from the point (*x1*, *y1*) to the point (*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 from the current position to point (*x1*, *y1*) then this method does nothing. Otherwise, the arc is the shortest path along the circle's circumference between those two points. If the first tangent point is not equal to the current position then the first tangent point is added to the list of points of the subpath and the current position is joined to that point by a straight line. Then, the second tangent point is added to the list of points and the two tangent points are joined by the arc described above. Finally, the current position is set to the second tangent point.

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 adds an arc to the current path. The arc is given by the circle that has its origin at (*x*, *y*) and that has radius *radius*. The points at *startAngle* and *endAngle* along the circle, measured in radians clockwise from the positive x-axis, are the start and end points. The arc is the path along the circumference of the circle from the start point to the end point going anti-clockwise if the *anticlockwise* argument is true, and clockwise otherwise.

The start point is added to the list of points of the subpath and the current position is joined to that point by a straight line. Then, the end point is added to the list of points and these last two points are joined by the arc described above. Finally, the current position is set to the end point.

Negative or zero values for *radius* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`rect(x, y, w, h)`** method creates a new subpath containing just the rectangle with top left coordinate (*x*, *y*), width *w* and height *h*, and marks it as closed. It then calls `moveTo` with the point (0,0).

Negative values for *w* and *h* must cause the implementation to raise an `INDEX_SIZE_ERR` exception.

The **`fill()`** method fills each subpath of the current path in turn, using `fillStyle`, and using the non-zero winding number rule. Open subpaths are implicitly closed when being filled (without affecting the actual subpaths).

The **`stroke()`** method strokes each subpath of the current path in turn, using the

`strokeStyle`, `lineWidth`, `lineJoin`, and (if appropriate) `miterLimit` attributes.

Paths, when filled or stroked, are painted without affecting the current path, and are subject to transformations, shadow effects, global alpha, clipping paths, and global composition operators.

The **`clip()`** method creates a new **clipping path** by calculating the intersection of the current clipping path and the area described by the current path, using the non-zero winding number rule. Open subpaths are implicitly closed 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.

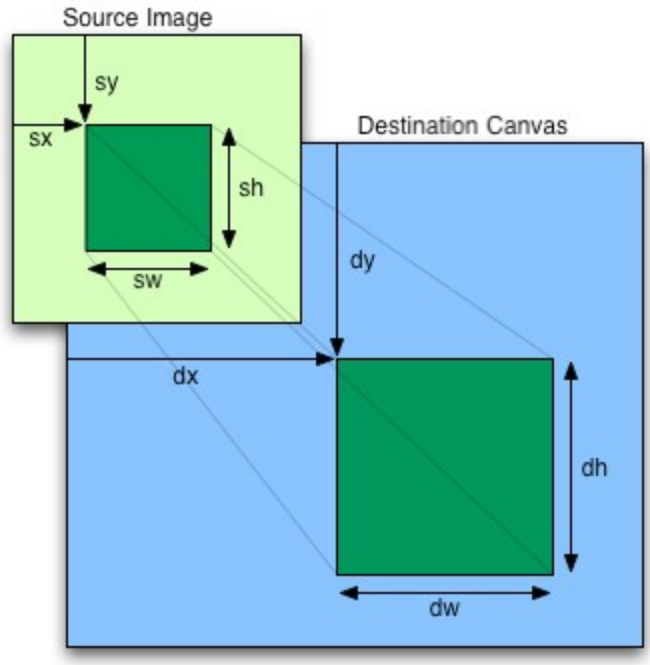
6.1.1.9. Images

To draw images onto the canvas, the **`drawImage`** method may be used.

This method is overloaded with three variants: `drawImage(image, dx, dy)`, `drawImage(image, dx, dy, dw, dh)`, and `drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)`. (Actually it is overloaded with six; each of those three can take either an `HTMLImageElement` or an `HTMLCanvasElement` for the *image* argument.) If not specified, the *dw* and *dh* arguments default to the values of *sw* and *sh*, interpreted such that one CSS pixel in the image is treated as one unit in the canvas coordinate space. If the *sx*, *sy*, *sw*, and *sh* arguments are omitted, they default to 0, 0, the image's intrinsic width in image pixels, and the image's intrinsic height in image pixels, respectively.

The *image* argument must be an instance of an `HTMLImageElement` or `HTMLCanvasElement`. If the *image* is of the wrong type, the implementation must raise a `TYPE_MISMATCH_ERR` exception. If one of the *sy*, *sw*, *sw*, and *sh* arguments is outside the size of the image, or if one of the *dw* and *dh* arguments is negative, the implementation must raise an `INDEX_SIZE_ERR` exception.

When `drawImage` is invoked, the specified region of the image specified by the source rectangle (*sx*, *sy*, *sw*, *sh*) is painted on the region of the canvas specified by the destination rectangle (*dx*, *dy*, *dw*, *dh*).



Images are painted without affecting the current path, and are subject to transformations, shadow effects, global alpha, clipping paths, and global composition operators.

6.1.1.10. Drawing model

When a shape or image is painted, user agents shall 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*.
3. The shadow is rendered from image *A*, creating image *B*.
4. Image *A* is composited over image *B* creating the source image.
5. The source image has its alpha adjusted by `globalAlpha`.
6. Within the clip region, the source image is composited over the current canvas bitmap using the composition operator.

6.2. Sound

The `Audio` interface allows scripts to play sound clips.

There is no markup element that corresponds to `Audio` objects, they are only accessible from script.

User agents should allow users to dynamically enable and disable sound output, but doing so must not affect how `Audio` objects act in any way other than whether sounds are physically played back or not. For instance, sound files must still be downloaded, `load` 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;
  void play();
  void loop();
  void loop(in unsigned long repeatCount);
  void stop();
}
```

`Audio` objects must also implement the `EventTarget` interface. [DOM3EVENTS]

In ECMAScript, an instance of `Audio` can be created using the `Audio(`*`uri`*`)` constructor:

```
var a = new Audio("test.wav");
```

The **`Audio()` constructor** takes a single argument, a URI (or IRI), which is resolved using the script context's `window.location.href` value as the base, and which returns an `Audio` object that will, at the completion of the current script, start loading that URI.

Once the URI is loaded, a `load` event must be fired on the `Audio` object.

`Audio` objects have a current position and a repeat count. Both are initially zero.

The `Audio` interface has the following members:

onload

An event listener that is invoked along with any other appropriate event listeners that are registered on this object when a `load` event is fired on it.

play()

Begins playing the sound at the current position, setting the repeat count to 1.

loop()

Begins playing the sound at the current position, setting the repeat count to infinity.

loop(*repeatCount*)

Begins playing the sound at the current position, setting the repeat count to *repeatCount*.

stop()

Stops playing the clip and resets the current position and repeat 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 repeat count is decreased by one (unless it is infinite). If the repeat count is greater than zero, then the sound is played again.

7. Communication

7.1. Server-sent DOM events

This section describes a mechanism for allowing servers to dispatch DOM events into documents that expect it.

7.1.1. The `event-source` element

To specify an event source in an HTML document authors use a new (empty) element `event-source`, with an attribute `src=""` that takes a URI (or IRI) to open as a stream and, if the data found at that URI is of the appropriate type, treat as an event source.

The `event-source` element may also have an `onevent=""` attribute. If present, the attribute must be treated as script representing an event handler registered as non-capture listener of events with name `event` and the namespace `uuid:755e2d2d-a836-4539-83f4-16b51156341f` or null, that are targetted at or bubble through the element.

UAs must also support all the common attributes on the `event-source` element.

7.1.2. The `RemoteEventTarget` interface

Any object that implements the `EventTarget` interface shall also implement the `RemoteEventTarget` interface.

```
interface RemoteEventTarget {
  void addEventSource(in DOMString src);
  void removeEventSource(in DOMString src);
```

```
};
```

The **`addEventSource(`*`src`*`)`** method shall register the URI (or IRI) specified in *src* as an event source on the object. The **`removeEventSource(`*`src`*`)`** method shall remove the URI (or IRI) specified in *src* from the list of event sources for that object. If a single URI is added multiple times, each instance must be handled individually. Removing a URI must only remove one instance of that URI. If the specified URI cannot be added or removed, the method must return without doing anything or raising an exception.

7.1.3. Processing model

When an `event-source` element in a document has a `src` attribute set, the UA should fetch the resource indicated by the attribute's value.

Similarly, when the `addEventSource()` method is invoked on an object, the UA should, at the completion of the script's current execution, fetch the resource identified by the method's argument (unless the `removeEventSource()` was called removing the URI from the list first).

When an `event-source` element is removed from the document, or when an event source is removed from the list of event sources for an object using the `removeEventSource()` method, the relevant connection must be closed (and not reopened unless the element is returned to the document or the `addEventSource()` method is called with the same URI again).

Since connections established to remote servers for such resources are expected to be long-lived, UAs should ensure that appropriate buffering is used. In particular, while line buffering may be safe if lines are defined to end with a single U+000A LINE FEED character, block buffering or line buffering with different expected line endings can cause delays in event dispatch.

In general, the semantics of the transport protocol specified by the "src" attribute must be followed. Clients should re-open `event-source` connections that get closed after a short interval (such as 5 seconds), unless they were closed due to problems that aren't expected to be resolved, as described in this section.

DNS errors must be considered fatal, and cause the user agent to not open any connection for the event-source.

HTTP 200 OK responses that have a Content-Type other than `application/x-dom-event-stream` must be ignored and must prevent the user agent from reopening the connection for that event-source. HTTP 200 OK responses with the right MIME type, however, should, when closed, be reopened after a small delay.

Resource with the type `application/x-dom-event-stream` must be processed line by line as described below.

HTTP 201 Created, 202 Accepted, 203 Non-Authoritative Information, and 206 Partial Content responses must be treated like HTTP 200 OK responses for the purposes of reopening event-source connections. They are, however, likely to indicate an error has occurred somewhere and may cause the user agent to emit a warning.

HTTP 204 No Content, and 205 Reset Content responses must be treated as if they were 200 OK responses with the right MIME type but no content, and should therefore cause the user agent to reopen the connection after a short delay.

HTTP 300 Multiple Choices responses should be handled automatically if possible (treating the responses as if they were 302 Moved Permanently responses pointing to the appropriate resource), and otherwise must be treated as HTTP 404 responses.

HTTP 301 Moved Permanently responses must cause the user agent to use the server specified URI instead of the one specified in the event-source's "src" attribute for future connections.

HTTP 302 Found, 303 See Other, and 307 Temporary Redirect responses must cause the user agent to use the server specified URI instead of the one specified in the event-source's "src" attribute for the next connection, but if the user agent needs to reopen the connection at a later point, it must once again start from the "src" attribute (or the last URI given by a 301 Moved Permanently response in complicated cases where such responses are chained).

HTTP 304 Not Modified responses should be handled like HTTP 200 OK responses, with the content coming from the user agent cache. A new connection attempt should then be made after a short wait.

HTTP 305 Use Proxy, HTTP 401 Unauthorized, and 407 Proxy Authentication Required should be treated transparently as for any other subresource.

HTTP 400 Bad Request, 403 Forbidden, 404 Not Found, 405 Method Not Allowed, 406 Not Acceptable, 408 Request Timeout, 409 Conflict, 410 Gone, 411 Length Required, 412 Precondition Failed, 413 Request Entity Too Large, 414 Request-URI Too Long, 415 Unsupported Media Type, 416 Requested Range Not Satisfiable, 417 Expectation Failed, 500 Internal Server Error, 501 Not Implemented, 502 Bad Gateway, 503 Service Unavailable, 504 Gateway Timeout, and 505 HTTP Version Not Supported responses, and any other HTTP response code not listed here, should cause the user agent to stop trying to process this event-source element.

For non-HTTP protocols, UAs should act in equivalent ways.

7.1.4. The event stream format

The event stream MIME type is `application/x-dom-event-stream`.

The event stream must always be encoded as UTF-8. Line must always be terminated by a single U+000A LINE FEED character.

The event stream format is (in pseudo-BNF):

```
<stream>  ::= <event>*
<event>   ::= [ <comment> | <command> | <field> ]* <newline>

<comment> ::= ';' <data> <newline>
<special> ::= ':' <data> <newline>
<field>   ::= <name> [ ':' <space>? <data> ]? <newline>

<name>    ::= one or more UNICODE characters other than ':', ';', and U+
<data>    ::= zero or more UNICODE characters other than U+000A LINE FEE
<space>   ::= a single U+0020 SPACE character (' ')
<newline> ::= a single U+000A LINE FEED character
```

Bytes that are not valid UTF-8 sequences must be interpreted as the U+FFFD REPLACEMENT CHARACTER.

The stream is parsed by reading everything line by line, in blocks separated by blank lines (blank lines are those consisting of just a single lone line feed character). Comment lines (those starting with the character ';') and command lines (those starting with the character ':') are ignored. Command lines are reserved for future use and should not be used.

For each non-blank, non-comment line, the field name is first taken. This is everything on the line up to but not including the first colon (':') or the line feed, whichever comes first. Then, if there was a colon, the data for that line is taken. This is everything after the colon, ignoring a single space after the colon if there is one, up to the end of the line. If there was no colon the data is the empty string.

> Examples:
>
> ```
> Field name: Field data
>
> This is a blank field
>
> 1. These two lines: have the same data
> 2. These two lines:have the same data
>
> 1. But these two lines:  do not
> 2. But these two lines: do not
> ```

If a field name occurs multiple times, the data values for those lines are concatenated with a newline between them.

> For example, the following:
>
> ```
> Test: Line 1
> Foo:  Bar
> Test: Line 2
> ```
>
> ...is treated as having two fields, one called `Test` with the value `Line 1\nLine 2` (where `\n` represents a newline), and one called `Foo` with the value `Bar`.

Note: (Since any random stream of characters matches the above format, there is no need to define any error handling.)

7.1.5. Event stream interpretation

Once the fields have been parsed, they are interpreted as follows (these are case sensitive exact comparisons):

- `Event` is the name of the event. For example, `load`, `DOMActivate`, `updateTicker`. If there is no field with this name, then no event will be synthesised, and the other data will be ignored.
- `Namespace` is the DOM3 namespace for the event. For normal DOM events this would be `http://www.w3.org/2001/xml-events`. If it isn't specified the event namespace is null.
- `Class` is the interface used for the event, for instance `Event`, `UIEvent`, `MutationEvent`, `KeyboardEvent`, etc. For compatibility with DOM3 Events, the values `UIEvents`, `MouseEvents`, `MutationEvents`, and `HTMLEvents` are valid values and must be treated respectively as meaning the interfaces `UIEvent`, `MouseEvent`, `MutationEvent`, and `Event`. (This value can therefore be used as the argument to `createEvent()`.) If the value is not specified it is defaulted based on the event name as follows:
  - If `Namespace` is `http://www.w3.org/2001/xml-events` or null and the `Event` field exactly matches one of the events specified by DOM3 Events in section 1.4.2 "Complete list of event types", then the Class defaults to the interface relevant for that event type. [DOM3EVENTS]

    > For example:
    >
    > ```
    > Event: click
    > ```

> ...would cause `Class` to be treated as `MouseEvent`.

- If `Namespace` is `uuid:755e2d2d-a836-4539-83f4-16b51156341f` or null and the `Event` doesn't match any of the known events, then the `RemoteEvent` interface (described below) is used.

- Otherwise, if the UA doesn't have special knowledge of which class to use for the given event in the given namespace, then the `Event` interface is used.

It is quite possible to give the wrong class for an event. This is equivalent to creating an event in the DOM using the DOM Event APIs, but using the wrong interface for it.

- `Bubbles` specifies whether the event is to bubble. If it is specified and has the value `No`, the event does not bubble. If it is specified and has any other value (including `no` or `No\n`) then the event bubbles. If it is not specified it is defaulted based on the event name as follows:

  - If `Namespace` is `http://www.w3.org/2001/xml-events` or null and the `Event` field exactly matches one of the events specified by DOM3 Events in section 1.4.2 "Complete list of event types", then whether the event bubbles depends on whether the DOM3 Events spec specifies that that event should bubble or not. [DOM3EVENTS]

    > For example:
    >
    > ```
    > Event: load
    > ```
    >
    > ...would cause `Bubbles` to be treated as `No`.

  - Otherwise, if the UA doesn't have special knowledge of which class to use for the given event in the given namespace, then the event bubbles.

- `Cancelable` specifies whether the event may have its default action prevented. If it is specified and has the value `No`, the event may not have its default action prevented. If it is specified and has any other value (including `no` or `No\n`) then the event may be canceled. If it is not specified it is defaulted based on the event name as follows:

  - If `Namespace` is `http://www.w3.org/2001/xml-events` or null and the `Event` field exactly matches one of the events specified by DOM3 Events in section 1.4.2 "Complete list of event types", then whether the event is cancelable depends on whether the DOM3 Events spec specifies that that event should be cancelable or not. [DOM3EVENTS]

> For example:
>
> ```
> Event: load
> ```
>
> ...would cause `Cancelable` to be treated as `No`.

  - Otherwise, if the UA doesn't have special knowledge of which class to use for the given event in the given namespace, then the event may be canceled.

- `Target` is the element that the event is to be dispatched on. If its value starts with a `#` character then the remainder of the value represents an ID, and the event must be dispatched on the same node as would be obtained by the getElementById() method on the ownerDocument of the event-source element responsible for the event being dispatched.

  > For example,
  >
  > ```
  > Target: #test
  > ```
  >
  > ...would target the element with ID `test`.

  If the value does not start with a `#` but has the literal value `Document`, then the event is dispatched at the `ownerDocument` of the `event-source` element responsible for the event being dispatched.

  Otherwise, the event is dispatched at the `event-source` element itself.

- Other fields depend on the interface specified (or possibly implied) by the `Class` field. If the specified interface has an attribute that exactly matches the name of the field, and the value of the field can be converted (using the type conversions defined in ECMAScript) to the type of the attribute, then it must be used. Any attributes (other than the `Event` interface attributes) that do not have matching fields are initialised to zero, null, false, or the empty string.

  > For example:
  >
  > ```
  > ; ...some other fields...
  > Class: MouseEvent
  > button: 2
  > ```
  >
  > ...would result in a MouseEvent event that had `button` set to `2` but `screenX`, `screenY`, etc, set to 0, false, or null as appropriate.

  If a field does not match any of the attributes on the event, it is ignored.

  > For example:

```
Event: keypress
Class: MouseEvent
keyIdentifier: 0
```

...would result in a `MouseEvent` event with its fields all at their default values, with the event name being `keypress`. The `ctrlKey` field would be ignored. (If the author had not included the `Class` field explicitly, it would have just worked, since the class would have defaulted as described above.)

Once a blank line is reached, an event of the appropriate type is synthesized and dispatched to the appropriate node as described by the fields above. No event is dispatched until a blank line has been received.

If the `Event` field was omitted, then no event is synthesised and the data is ignored.

The following stream contains four blocks yet synthesises no events, since none of the blocks have a field called `Event`. (The first block has just a comment, the second block has two fields with names "load" and "Target" respectively, the third block is empty, and the fourth block has two comments.)

```
; test

load
Target: #image1


; if any real events follow this block, they will not be affected b
; the "Target" and "load" fields above.
```

7.1.6. The `RemoteEvent` interface

The `RemoteEvent` interface is defined as follows:

```
interface RemoteEvent : Event {
  readonly attribute DOMString       data;
  void               initRemoteEvent(in DOMString typeArg,
                                     in boolean canBubbleArg,
                                     in boolean cancelableArg,
                                     in DOMString dataArg);
  void               initRemoteEventNS(in DOMString namespaceURI,
                                       in DOMString typeArg,
                                       in boolean canBubbleArg,
                                       in boolean cancelableArg,
                                       in DOMString dataArg);
};
```

Events that use the `RemoteEvent` interface never have any default action

associated with them.

7.1.7. Example

> The following event description, once followed by a blank line:
>
> ```
> Event: stock change
> data: YHOO
> data: -2
> data: 10
> ```
>
> ...would cause an event `stock change` with the interface `RemoteEvent` to be dispatched on the `event-source` element, which would then bubble up the DOM, and whose `data` attribute would contain the string `YHOO\n-2\n10` (where `\n` again represents a newline).
>
> This could be used as follows:
>
> ```
> <event-source src="http://stocks.example.com/ticker.php" id="stock"
> <script type="text/javascript">
> document.getElementById('stock').addEventListener('stock change',
>   function () {
>     var data = event.data.split('\n');
>     updateStocks(data[0], data[1], data[2]);
>   }, false);
> </script>
> ```
>
> ...where updateStocks is a function defined as:
>
> ```
> function updateStocks(symbol, delta, value) { ... }
> ```
>
> ...or some such.

7.2. Scripted HTTP: `XMLHttpRequest`

To allow scripts to programmatically connect back to their originating server via HTTP, the following interface may be used.

```
interface XMLHttpRequest {
           attribute EventListener           onreadystatechange;
  readonly attribute int                     readyState;
  void open(in DOMString method, in DOMString uri);
  void open(in DOMString method, in DOMString uri, in boolean async);
  void open(in DOMString method, in DOMString uri, in boolean async, in
  void open(in DOMString method, in DOMString uri, in boolean async, in
  void setRequestHeader(in DOMString header, in DOMString value);
  void send();
  void send(in DOMString body);
  void send(in Document body);
  void abort();
```

```
  DOMString getAllResponseHeaders();
  DOMString getResponseHeader(in DOMString header);
  readonly attribute DOMString               responseText;
  readonly attribute Document                responseXML
  readonly attribute int                     status;
  readonly attribute DOMString               statusText;
};
```

`XMLHttpRequest` objects must also implement the `EventTarget` interface. [DOM3EVENTS]

In ECMAScript, an instance of `XMLHttpRequest` can be created using the `XMLHttpRequest()` constructor:

```
var r = new XMLHttpRequest();
```

The `XMLHttpRequest` interface has the following members:

onreadystatechange

An event listener that is invoked along with any other appropriate event listeners that are registered on this object when a `readystatechange` event is fired on it.

readyState

The state of the object. The values have the following meanings:

0 Uninitialised

The initial value.

1 Open

The `open()` method has been successfully called.

2 Sent

The `send()` method has been successfully called, but no data has yet been received.

3 Receiving

Data is being received, but the data transfer is not yet complete.

4 Loaded

The data transfer has been completed.

A `readystatechange` event shall immediately be dispatched at the object whenever the `readyState` attribute changes value. The `readystatechange` event must never be dispatched by the UA if the `readyState` attribute did not change. The `readystatechange` event has no default action.

open(*method*, *uri*, [*async*, [*user*, [*password*]]])

Initialises the object by remembering the *method*, *uri*, *async* (defaulting to true if omitted), *user* (defaulting to null if omitted), and *password* (defaulting to null if omitted) arguments, setting the `readyState` attribute to 1 (Open), resetting the `responseText`, `responseXML`, `status`, and `statusText`

attributes to their initial values, and resetting the list of request headers. The *uri* argument is resolved to an absolute URI using the script context's `window.location.href` value as the base.

Same-origin security restrictions should apply.

If the URI given to this method contains a username and a password (the latter potentially being the empty string), then these must be used if the *user* and *password* arguments are omitted. If the arguments are not omitted, they take precedence, even if they are null.

setRequestHeader(*header*, *value*)

If the `readyState` attribute has a value other than 1 (Open), raises an exception. If the *header* or *value* arguments contain any U+000A LINE FEED or U+000D CARRIAGE RETURN characters, or if the *header* argument contains any U+0020 SPACE or U+003A COLON charecters, does nothing. Otherwise, the request header *header* is set to *value*. If the request header *header* had already been set, then the new *value* is concatenated to the existing value after a comma and a space.

> The following script:
>
> ```
> var r = new XMLHttpRequest;
> r.open('get', 'demo.cgi');
> r.setRequestHeader('X-Test', 'one');
> r.setRequestHeader('X-Test', 'two');
> r.send(null);
> ```
>
> ...would result in the following header being sent:
>
> ```
> ...
> X-Test: one, two
> ...
> ```

The list of request headers must be reset when the `open()` method is called.

User agents must not set any headers other than the headers set by the author using this method, with the following exceptions:

- UAs must set the `Host` header appropriately (see `open()`) and not allow it to be overridden.
- UAs must set the `Authorization` header according to the values passed to the `open()` method (but must allow calls to `setRequestHeader()` to append values to it).
- UAs may set the `Accept-Charset` and `Accept-Encoding` headers and must not allow them to be overridden.

- UAs may set the `If-Modified-Since`, `If-None-Match`, `If-Range`, and `Range` headers if the resource is cached and has not expired (as allowed by HTTP), and must not allow those headers to be overridden.
- UAs must set the `Connection` and `Keep-Alive` headers as described by the HTTP specification, and must not allow those headers to be overridden.
- UAs should set the proxy-related headers according to proxy settings of the environment, and must not allow those headers to be overridden.
- UAs may give the `User-Agent` header an initial value, but must allow authors to append values to it.
- UAs should set `Cookie` and `Cookie2` headers appropriately for the given URI and given the user's current cookies, and must allow authors to append values to these headers.

In particular, UAs must not automatically set the `Cache-Control` or `Pragma` headers to defeat caching. [HTTP]

send([*data*])

If the `readyState` attribute has a value other than 1 (Open), raises an exception. Otherwise, sets the `readyState` attribute to 2 (Sent) and sends a request to *uri* using method *method*, authenticating using *user* and *password* as appropriate. If the *async* flag is set to false, then the method does not return until the request has completed. Otherwise, it returns immediately. (See: `open()`.)
If the *method* is `post` or `put`, then the *data* passed to the `send()` method is used for the entity body. If *data* is a string, the data is encoded as UTF-8 for transmission. If the *data* is a Document, then the document is serialised using the encoding given by *data*`.xmlEncoding`, if specified, or UTF-8 otherwise. [DOM3CORE]

If the response is an HTTP redirect, then it should be transparently followed (unless it violates security or infinite loop precautions). Any other error (including a 401) must cause the object to use that error page as the response.

Once the final HTTP status line has been received, the `readyState` attribute should be set to to 3 (Receiving). When the request has completed loading, the `readyState` attribute should be set to 4 (Loaded).

abort

Cancels any network activity for which the object is responsible and returns `readyState` to 0 (Uninitialised).

getAllResponseHeaders()

If the `readyState` attribute has a value other than 3 (Receiving) or 4 (Loaded), returns null. Otherwise, returns the HTTP headers that have been received so far for the last request sent, as a single string, with each header line separated by a CR (U+000D) LF (U+000A) pair. The status line is not included.

> The following script:
>
> ```
> var r = new XMLHttpRequest;
> r.open('get', 'test.txt', false);
> r.send();
> alert(r.getAllResponseHeaders());
> ```
>
> ...should display a dialog with text similar to the following:
>
> ```
> Date: Sun, 24 Oct 2004 04:58:38 GMT
> Server: Apache/1.3.31 (Unix)
> Keep-Alive: timeout=15, max=99
> Connection: Keep-Alive
> Transfer-Encoding: chunked
> Content-Type: text/plain; charset=utf-8
> ```

getResponseHeader(*header*)

If the `readyState` attribute has a value other than 3 (Receiving) or 4 (Loaded), returns an empty string. Otherwise, returns the value of the given HTTP header in the data received so far for the last request sent, as a single string. If more than one header of the given name was received, then the values should be concatenated, separated from each other by a comma and a space. If no headers of that name were received, then returns the empty string. Header names must be compared case-insensitively to the method argument (*header*).

responseText

If the `readyState` attribute has a value other than 3 (Receiving) or 4 (Loaded), returns an empty string. Otherwise, returns the body of the data received so far, interpreted using the character encoding specified in the response, or UTF-8 if no character encoding was specified. Invalid bytes must be converted to U+FFFD REPLACEMENT CHARACTER.

responseXML

If the `readyState` attribute has a value other than 4 (Loaded), returns null. Otherwise, if the `Content-Type` header is either `text/xml`, `application/xml`, or ends in `+xml`, returns an object that implements the Document interface representing the parsed document. If the document was not an XML document, or if the document could not be parsed (due to an XML well-

formedness error or unsupported character encoding, for instance), returns null.

status

If the `readyState` attribute has a value other than 3 (Receiving) or 4 (Loaded), raises an exception. Otherwise, returns the HTTP status code (typically 200 for a successful connection).

statusText

If the `readyState` attribute has a value other than 3 (Receiving) or 4 (Loaded), raises an exception. Otherwise, returns the HTTP status text sent by the server after the status code.

If an exception is raised due to an attribute or method being used when `readyState` has an inappropriate value, it should be a `INVALID_STATE_ERR` DOM Exception.

HTTP requests sent from multiple different `XMLHttpRequest` objects in succession should be pipelined into shared HTTP connections.

7.3. Network connections

This section needs much more work before being ready for review. At the moment it mostly consists of a place for ideas to be described.

To enable Web applications to communicate with each other in local area networks, and to maintain bidirectional communications with their originating server, this specification introduces the `Connection` interface.

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.

```
interface Connection {
  readonly attribute DOMString network;
  readonly attribute DOMString peer;
           attribute EventListener onopen;
           attribute EventListener onread;
           attribute EventListener onclose;
  readonly attribute int readyState;
  void send(in DOMString data);
  void disconnect();
};

interface ConnectionReadEvent : Event {
```

```
  readonly attribute DOMString data;
  readonly attribute DOMString source;
  void               initUIEvent(in DOMString typeArg,
                                 in boolean canBubbleArg,
                                 in boolean cancelableArg,
                                 in DOMString dataArg);
  void               initUIEventNS(in DOMString namespaceURI,
                                   in DOMString typeArg,
                                   in boolean canBubbleArg,
                                   in boolean cancelableArg,
                                   in DOMString dataArg);
};
```

`Connection` objects must also implement the `EventTarget` interface. [DOM3EVENTS]

In ECMAScript, an instance of `Connection` can be created using one of the connection constructors:

`TCPConnection(`*`subdomain`*`, `*`port`*`)`

Establishes a TCP connection to the specified subdomain using the specified port. If the *subdomain* is null or the empty string, the connection shall be established to the script context's origin host (`window.location.host`). Otherwise, the *subdomain* string is prepended to the `window.location.host` string with a dot separating the two strings, and if that is a valid host name, the connection shall be established to that host. If the `window.location.host` is not a valid host, or if prepending the *subdomain* does not yield an valid host, then a security exception should be thrown.

If the port is not equal to 80 or greater than 1024, raises a security exception. Otherwise, the given port shall be used to establish a connection.

Returns a `Connection` object with its `network` attribute set to the name of the host to which the connection was established and its `peer` attribute set to "*host*:*port*".

`LocalBroadcastConnection(`*`topic`*`)`

Prompts the user to confirm that a connection should be made. Such a prompt could look like this:

```
|:: New Connection ::::::::::::::::::::::::::::::::::::::::|
|                                                          |
|  Would you like to open a connection called "Chess" for  |
|  this Web site?:                                         |
|                                                          |
|    example.org                                           |
```

```
|                                                                |
|  Select connection to use: [ Bluetooth      | v ]              |
|                                                                |
|                          (( Open connection ))  ( Cancel )     |
|________________________________________________________________|
```

Returns null if the prompt was canceled. Otherwise, returns a `Connection` object with its `network` attribute set to *topic* and its `peer` attribute set to null, and begins broadcasting on the relevant network. (See: broadcast formats.)

> The following script creates a connection to a local party line:
>
> ```
> var a = new LocalBroadcastConnection("Party Line");
>   a.onread = function(s, f) { alert(f + ' wrote ' + s); }
>   a.send('hello');
> ```

`LocalPeerConnection(`*`topic`*`)`

Prompts the user to select a connection to make, which could look like this:

```
|:: New Connection ::::::::::::::::::::::::::::::::::::::::::::::|
|                                                                |
|  Select the peer to connect to:                                |
|                                                                |
|    JohnSmith_Series60   via Bluetooth      (( Connect ))       |
|    Patrick's Phone      via Bluetooth       ( Connect )        |
|    John Smith           via UDP             ( Connect )        |
|                                                                |
|                                             ( Cancel )         |
|________________________________________________________________|
```

While the prompt is displayed, the UA should broadcast on all supported networks, as described below.

Returns null if the prompt was canceled. Otherwise, returns a `Connection` object with its `network` attribute set to *topic* and its `peer` attribute set to a string uniquely identifying the selected peer, and opens a connection to that peer. (See: peer connection formats.)

...

Events: One event when the connection is first established, one event for when data is received, one event for when the connection is closed for good, one event for when the connection is cut temporarily, one event for when the connecton is restored.

When data is received for a connection, a `read` event must be fired on the `Connection` object, using the `ConnectionReadEvent` interface for the event. The

event has no default action.

Data that is received during script execution (e.g. between the connection object being created — and thus the connection being established — and the current script completing, or during the execution of a `read` event handler) must be buffered, and `read` events queued up to be fired after the script has completed.

The `data` attribute of the event object contains the string representing the data received from the network.

For connections established using `TCPConnection` and `LocalPeerConnection`, the `source` attribute of the event is equal to the `peer` attribute of the connection object. For `LocalBroadcastConnection` connections, the `source` attribute of the event contains the string uniquely identifying the source of the message.

...

7.3.1. TCP connections

All TCP connections should have a handshake to ensure the server is expecting a TCPConnection. TCP connections should attempt to automatically re-connect when they get disconnected. All text is sent as UTF-8.

7.3.2. Broadcast formats

...

7.3.3. Peer connection formats

...

7.3.4. Announcing peer connections

...

7.4. Cross-document messaging

Web browsers, for security and privacy reasons, prevent documents in different domains from affecting each other; that is, cross-site scripting is disallowed.

While this is an important security feature, it prevents pages from different domains from communicating even when those pages are not hostile. This section introduces a messaging system that allows documents to communicate with each other regardless of their source domain, in a way designed to not

enable cross-site scripting attacks.

7.4.1. Definitions

Any `Document` object that supports this cross-document messaging API must implement the `DocumentMessaging` interface.

```
interface DocumentMessaging {
  void postMessage(in DOMString message);
};
```

Such `Document` objects must also implement the `EventTarget` interface. [DOM3EVENTS]

The postMessage() method causes an event to be dispatched (as defined below). This event uses the following interface:

```
interface CrossDocumentMessageEvent : Event {
  readonly attribute DOMString data;
  readonly attribute DOMString domain;
  readonly attribute DOMString uri;
  readonly attribute Document source;
  void initCrossDocumentMessageEvent(in DOMString typeArg,
                                     in boolean canBubbleArg,
                                     in boolean cancelableArg,
                                     in DOMString dataArg,
                                     in DOMString domainArg,
                                     in DOMString uriArg,
                                     in Document documentArg);
  void initCrossDocumentMessageEventNS(in DOMString namespaceURI,
                                       in DOMString typeArg,
                                       in boolean canBubbleArg,
                                       in boolean cancelableArg,
                                       in DOMString dataArg,
                                       in DOMString domainArg,
                                       in DOMString uriArg,
                                       in Document documentArg);
};
```

7.4.2. Processing model

When a script invokes the **`postMessage()`** method on a document, the user agent must create an event that uses the `CrossDocumentMessageEvent` interface, with the event name **`message`** in the `uuid:7f37e11a-3a5c-4f3d-a82e-83b611439f37` namespace, which bubbles, is cancelable, and has no default action. The **`data`** attribute must be set to the value passed as the argument to the `postMessage()` method, the **`domain`** attribute must be set to the domain of the document that the script that invoked the methods is associated with, the **`uri`** attribute must be set to the URI of that document, and the **`source`**

attribute must be set to the object representing that document.

⚠ Warning! Authors should check the `domain` attribute to ensure that messages are only accepted from domains that they expect to receive messages from. Otherwise, bugs in the author's message handling code could be exploited by hostile sites.

> For example, if document A contains an `object` element that contains document B, and script in document A calls `postMessage()` on document B, then a message event will be fired on that element, marked as originating from document A. The script in document A might look like:
>
> ```
> var o = document.getElementsByTagName('object')[0];
> o.contentDocument.postMessage('Hello world');
> ```
>
> To register an event handler for incoming events, the script would use `addEventListener()` (or similar mechanisms). For example, the script in document B might look like:
>
> ```
> document.addEventListener('message', receiver, false);
> function receiver(e) {
>   if (e.domain == 'example.com') {
>     if (e.data == 'Hello world') {
>       e.source.postMessage('Hello');
>     } else {
>       alert(e.data);
>     }
>   }
> }
> ```
>
> This script first checks the domain is the expected domain, and then looks at the message, which it either displays to the user, or responds to by sending a message back to the document which sent the message in the first place.

Note: Implementors are urged to take extra care in the implementation of this feature. It allows authors to transmit information from one domain to another domain, which is normally disallowed for security reasons. It also requires that UAs be careful to allow access to certain properties but not others.

The **`initCrossDocumentMessageEvent()`** and **`initCrossDocumentMessageEventNS()`** methods must initialise an event object in a manner analogous to other `initXXXEvent` metheds.

8. User interaction

8.1. Focus

This entire section will be merged with earlier sections in due course.

When an element is focused, key events are targetted at that element instead of at the document's root element.

8.1.1. The `tabindex` Attribute

The `tabindex` attribute defined in HTML4 is extended to apply to all HTML elements by defining it as a common attribute.

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

The `tabindex` attribute can take any integer (an optional U+002D HYPHEN-MINUS representing negativity followed by one or more digits in the range 0-9, U+0030 to U+0039, interpreted as base ten).

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

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

Other values are ignored, as if the attribute was absent. Certain elements may default absent `tabindex` attributes to zero, at the user agent's discretion. (In other words, some elements are focusable by default, and they are assumed to have tab index 0. Text fields will typically be in the tab order by default, for instance.)

When an element that does not normally take focus has the `tabindex` attribute specified with a positive value, then it is added to the tab order and is made focusable. When focused, the element matches the CSS `:focus` pseudo-class and key events are dispatched on that element when appropriate, just like focusing a link.

Since all HTML elements can thus be focused and unfocusd, the `onfocus` and `onblur` attributes shall also apply to all HTML elements.

8.1.2. The `ElementFocus` interface

The `ElementFocus` interface contains methods for moving focus to and from an element. It can be obtained from objects that implement the `Element` interface using binding-specific casting methods.

```
interface ElementFocus {
           attribute long                    tabIndex;
  void focus();
  void blur();
};
```

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

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

8.1.3. The `DocumentFocus` interface

The `DocumentFocus` interface contains methods for moving focus around the document. It can be obtained from objects that implement the `Document` interface using binding-specific casting methods.

```
interface DocumentFocus {
  readonly attribute Element                currentFocus;
  void moveFocusForward();
  void moveFocusBackward();
  void moveFocusUp();
  void moveFocusRight();
  void moveFocusDown();
  void moveFocusLeft();
};
```

The **`currentFocus`** attribute returns the element to which key events will be sent when the document receives key events.

The **`moveFocusForward`** method uses the `'nav-index'` property and the `tabindex` attribute to find the next focusable element and focuses it.

The **`moveFocusBackward`** method uses the `'nav-index'` property and the `tabindex` attribute to find the previous focusable element and focuses it.

The **`moveFocusUp`** method uses the `'nav-up'` property and the `tabindex` attribute to find an appropriate focusable element and focuses it.

In a similar manner, the **`moveFocusRight`**, **`moveFocusDown`**, and **`moveFocusLeft`** methods use the `'nav-right'`, `'nav-down'`, and `'nav-left'` properties (respectively), and the `tabindex` attribute, to find an appropriate focusable element and focus it.

The `'nav-index'`, `'nav-up'`, `'nav-right'`, `'nav-down'`, and `'nav-left'` properties are defined in [CSS3UI].

9. Parsing

The rules for parsing XHTML documents into DOM trees are covered by the XML and Namespaces in XML specifications, and are out of scope of this specification.

For HTML documents, user agents must use the parsing rules described in this section to generate the DOM trees.

This specification mainly defines the parsing rules for syntactically valid HTML documents. When user agents encounter something described as a **hard parse error** in the rules below, they may use any error correction algorithm to handle the error. Such error correction should not result in DOMs that are not strictly trees. User agents are reluctantly encouraged to reverse engineer the error handling behaviour of the prevalent user agent in order to foster interoperability.

This specification also speaks of **easy parse errors**; these are parse errors where the error handling is well-defined, and user agents must act as described below when encountering such problems.

Conformance checkers must report all parse error conditions (both hard and easy errors) to the user, but may apply error correction algorithms (those described in the spec for easy errors, and those reverse-engineered from other UAs for hard errors) in an attempt to continue past the location of the error and find the remaining errors.

> ***While the HTML form of HTML5 bears a close resemblance to SGML and XML, it is a separate language with its own parsing rules.***
>
> ***Past 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 section needs to be written, obviously.

10. Rendering

This section will probably include details on how to render DATAGRID, drag-and-drop, etc, in a visual media, in concert with CSS.

10.1. Rendering and the DOM

Any object implement the `AbstractView` interface must also implement the `MediaModeAbstractView` interface.

```
interface MediaModeAbstractView {
  readonly attribute DOMString mediaMode;
}
```

The **`mediaMode`** attribute on objects implementing the `MediaModeAbstractView` interface must return the string that represents the canvas' current rendering mode (`screen`, `print`, etc). This is a lowercase string, as defined by the CSS specification. [CSS21]

Some user agents may support multiple media, in which case there will exist multiple objects implementing the `AbstractView` interface. Only the default view implements the `Window` interface. The other views can be reached using the `view` attribute of the `UIEvent` inteface, during event propagation. There is no way currently to enumerate all the views.

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.

References

This section will be written in a future draft.

Acknowledgements

Thanks to Aaron Leventhal, Anne van Kesteren, Asbjørn Ulsberg, Ben Godfrey, Ben Meadowcroft, Bjoern Hoehrmann, Boris Zbarsky, Brad Fults, Brad Neuberg, Brendan Eich, Chriswa, Darin Fisher, David Hyatt, Derek

Featherstone, Dimitri Glazkov, dolphinling, Doron Rosenberg, fantasai, Franck 'Shift' Quélain, Henri Sivonen, Henrik Lied, Håkon Wium Lie, James Graham, James Perrett, Jan-Klaas Kollhof, Jens Bannmann, Joel Spolsky, Jukka K. Korpela, Kai Hendry, Kornel Lesinski, Lachlan Hunt, Laurens Holst, Maciej Stachowiak, Malcolm Rowe, Mark Nottingham, Mark Schenk, Martijn Wargers, Martin Honnen, Matthew Mastracci, Matthew Raymond, Matthew Thomas, Mattias Waldau, Max Romantschuk, Michael A. Nachbaur, Michael Gratton, Michael 'Ratt' Iannarelli, Mike Shaver, Mikko Rantalainen, Olav Junker Kjær, Rimantas Liubertas, Shaun Inman, Steven Garrity, Stuart Parmenter, Tantek Çelik, Thomas O'Connor, Vladimir Vukićević, and everyone on the WHATWG mailing list for their useful and substantial comments.

Special thanks to Richard Williamson for creating the first implementation of `canvas` in Safari, from which the canvas feature was designed.

Special thanks also to the Microsoft employees who first implemented the `XMLHttpRequest` interface, the event-based drag-and-drop mechanism, and other features first widely deployed by the Windows Internet Explorer browser.

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 #mozilla crew, the #opera crew, and the #mrt crew for their ideas and support.