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

Web Applications 1.0
Working Draft — 8 December 2004

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.

It is very wrong to cite this as anything other than a work in progress. Do not implement this in a production product. It is not ready yet! At all!

This document is the result of a loose collaboration between interested parties in the context of the Web Hypertext Application Technology Working Group. To become involved in the development of this document, please send comments

to the address given above. **Your input will be taken into consideration.**

This is a working draft and may therefore be updated, replaced or rendered obsolete by other documents at any time. It is inappropriate to use Working Drafts as reference material or to cite them as other than "work in progress".

This draft may contain namespaces that use the `uuid:` URI scheme. These are temporary and will be changed before this specification is ready to be implemented.

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.

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. 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.

1.1. 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.
- Ability to define custom widgets cleanly, for example using XBL and APIs to query and control focus state, widget state, the position and state of input devices, etc.
- 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>, <navigation>, 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.

1.2. Relationship to HTML 4.01 and XHTML 1.1

This specification adds a number of features to HTML. For XML-based documents, these are added to the XHTML 1.x namespace. This ensures backwards compatibility with, and a simple migration path from, existing HTML and XHTML content.

1.3. Relationship to XHTML2

XHTML2 [XHTML2] updates HTML with better features for hyperlinks, multimedia content, annotating document edits, and introduces elements for better describing the semantics of human literary works such as poems.

Unfortunately, it lacks elements to express the semantics of many of the non-document types of content most often seen on the Web. For instance, the very popular forum sites, auction sites, search engines, online shops, and the like, do

not fit the document metaphor well.

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

XHTML2 and this specification therefore address the needs of two different authoring scenarios, and are expected to be used side by side. XHTML2 is optimised for *documents* — for example help files, essays, articles — whereas this specification is optimised for applications.

1.4. 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.

1.5. 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.

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

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

1.7. Conformance requirements

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.

Diagrams, examples, and notes are non-normative. All other content in this specification is normative.

This specification includes by reference all of the HTML4, XHTML1.1, DOM2

HTML, DOM3 Core, and DOM3 Events specifications ([HTML4], [XHTML1], [DOM2HTML], [DOM3CORE], [DOM3EVENTS]). Compliant UAs must implement all the requirements of those specifications to claim compliance to this one. Implementations may optionally implement only one of HTML4 and XHTML1.1.

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. User agents that do not support scripting, or that have scripting disabled, might be unable to fully convey the author's intent.***

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 use the following DOCTYPE: `<!DOCTYPE html PUBLIC "-//WHATWG //NONSGML HTML5//EN">`.

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.

> ***Note: Documents that use the new features described in this specification cannot be strictly conforming XHTML or HTML4 documents, since they contain features not defined in those specifications.***

1.8. Miscellaneous

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 default media for `link` and `style` elements shall be `all`, not `screen` as defined by HTML4.

The `title` attribute on `style` elements shall have the same semantics as on `link` elements with `rel="stylesheet"`: the name of the alternate stylesheet set

to which the stylesheet belongs.

On all other elements, the `title` attribute has the semantic of advisory information for the element, such as would be appropriate for a tooltip.

The `id`, `title`, and `class` attributes apply to all HTML elements.

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

In XHTML, the `script` and `style` elements, when they are designated as containing a language that is pure text (e.g. CSS or ECMAScript, but not XForms Actions), must be processed by taking all the contents of text and CDATA nodes that are direct children of the `script` or `style` elements, skipping over any other nodes (e.g. elements or comments, and their children).

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.

In XHTML documents, user agents must ignore the `http-equiv` attribute of `meta` elements. In HTML documents, user agents must treat all `meta` elements with `http-equiv` attributes as if the document had been served with the given HTTP headers following any real HTTP ones. For "Link:" headers introduced by such `meta` elements, the order of `link` and `meta` elements must be honoured. (However, for `Content-Type` headers, HTML user agents must only extract the encoding information from `meta` elements, and only if the HTTP headers failed to provide that information. See HTML4, section 5.2.2. [HTML4])

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]

2. Semantics and structure of HTML elements

This section is incomplete, and is mostly a placeholder for clarifications to the meanings of HTML elements.

2.1. Block-level context

Documents that place inline-level content in inappropriate contexts are non-conformant.

> For example, the following document is non-conformant, despite being syntactically correct:
>
> ```
> <!DOCTYPE html PUBLIC "-//WHATWG//NONSGML HTML5//EN">
> <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 PUBLIC "-//WHATWG//NONSGML HTML5//EN">
> <html lang="en-GB">
>  <head> <title> Demonstration </title> </head>
>  <body>
>   <blockquote>
>     My favourite animal is the cat. 
>   </blockquote>
>   
>    —<a href="http://example.org/~ernest/"><cite>Ernest</cite></a>,
>    in an essay from 1992
>   
>  </body>
> </html>
> ```
>
> This next document fragment, intended to represent the heading of a corporate site, is similarly non-conformant because the second line is not the heading of a subsection, but merely a subtitle. The `header` element should be used in these kinds of situations.
>
> ```
> <body>
>  <h1>ABC Company</h1>
>  <h2>Leading the way in widget design since 1432</h2>
>  .../pre>
> ```

2.2. Block-level meaning

2.2.1. The `p` element

The `p` element represents a paragraph.

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.

> The following examples are conformant HTML fragments:
>
> ```
> The little kitten gently seated himself on a piece of
> carpet. Later in his life, this would be refered to as the time the
> cat sat on the mat.
> ```
>
> ```
> <fieldset>
>  <legend>Personal information</legend>
>  <label>Name: <input name="n"></label>
>  <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.
> ```

2.2.2. The `dl`, `dt`, and `dd` elements

The `dl` element introduces an 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). `dl` elements must not contain non-whitespace text nodes or elements other than `dt` and `dd` elements.

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

> The following are all conformant HTML fragments.
>
> In this first one, the groups represent lines in a play, with the groups' names (the `dt` elements) representing the character names, and the groups' values (the `dd` elements) representing what they say.
>
> ```
> <dl>
>  <dt> John </dt>
>  <dd> Joan! Let's go back home. </dd>
>  <dt> Joan </dt>
>  <dd> Ok, if you want. </dd>
> ```

```
</dl>
```

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-conformant.

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

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

2.3. Inline-level meaning

2.3.1. The `em` and `strong` elements

We should define exactly what these mean, if anyone can come to an agreement.

2.3.2. The `q` element

Need to deal with the quotemark problem without adding verbose markup, breaking existing documents, or adding redundant elements.

2.3.3. The `br` element

Need some text here that basically says that `br` must only be used for the line breaks in poems and addresses, and that it is quite blatently not appropriate for markup like:

```
<p><a ...>34 comments</a>.<br>
<a ...>Add a comment.<a></p>
```

2.4. 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 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 on the same element, `click`, 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]

2.5. Link types

This section might at some future point list a small set of link relationship 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.

3. Sections

This section is not finished. The content models of the various elements defined here need work, and in particular the handling of headers is very broken right now. Feel free to comment 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.

3.1. Document sections

The **`section`** element represents a section of a document, such as a chapter.

The `section` element is a block level element that must contain either nothing, block level content, or inline level content. 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 stylesheet:

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

3.1.1. Section headers

The semantics of `h1` ... `h6` elements that have ancestor `section` elements are all the same: they simply specify the header of that section.

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 stylesheet:

```
@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 h3 */ }
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.

3.1.2. Section footers

Need to clarify that the `address` element is purely for a section's contact information and *not* for any arbitrary address.

3.2. Specific section types

The following elements define sections with specific semantics as described:

`navigation`

A block of navigation links. User agents may offer users the option to skip past such blocks, for example speech user agents typically would not read out navigation sections unless specifically requested to do so by the user.

`header`

The page header if the `header` element has no ancestor `section` elements, otherwise, the header for the nearest ancestor section.
The highest-level header in the `header` element is the title of the page or section. Heading elements with lower levels represent subtitles, not titles of lower level sections.

When presenting outlines, or otherwise extracting header information from documents, the contents of the first highest-level header descendant of a `header` element must be considered as being equivalent to an `h1` element replacing the entire `header` element.

> 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 subtitles.

```
<header>
 <h1>The reality dysfunction</h1>
 <h2>Space is not the only void</h2>
</header>

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

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

`header` elements must only be used where `h1` elements would be allowed.

`footer`

The footer for the nearest ancestor `section` element, or if none, for the page.

`article`

An article, for instance a Web log post, a magazine article, or a forum post. The first header element (`h1` to `h6`) in a depth first search of the `article` element represents the article header. The first `a` element with a `rel` attribute having as one of its values the keyword `bookmark` in a depth first search of the `article` element is the article's permalink.
Articles that contain further articles, as for example a Web log post that contains comments on that post, should be represented by nesting `article` elements inside the main `article` element.

`sidebar`

A section that is not directly part of the main flow of text, but is on a related topic.

All of these elements are block level element that must contain either nothing,

block level content, or inline level content. User agents must support all of the common attributes and event handlers on these elements.

In CSS-aware user agents, the default presentation of these elements should be achieved by including the following rules, or their equivalent, in the UA's user agent stylesheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|navigation, xh|header, xh|footer, xh|article, xh|sidebar
  { display: block; margin: 0; }
```

The rendering of `h1` elements according to the UA stylesheet should not be affected by these elements.

3.3. 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 stylesheet:

```
@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 cancelled (or if the click event is cancelled, 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>
>   <img src="logo" alt="">
>   The Application.
>   © copyright 2004 by The First Team.
>  </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>
```

3.4. 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 stylesheet:

```
@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; }
```

3.4.1. 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.

4. Structured data

This entire section will need serious fleshing out in due course. Currently it is pending the completion of specifications from Technorati.

4.1. 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 content. 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.

4.1.1. Intepreting calendar data

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

4.1.2. 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:

4.2. 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 content. 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.

4.2.1. Intepreting card data

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

4.2.2. 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:

4.3. Inline data

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

5. Widgets

This section merely indicates what new widgets are expected to be introduced in due course.

5.1. 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"?

5.2. 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).

5.3. Data grids

5.4. Data trees

6. Menus, buttons and commands

6.1. 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>
```

6.2. 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 command
  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 comman
```

```
  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`.

6.2.1. 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".

6.2.2. The `a` element and commands

6.2.2.1. 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.)

6.2.2.2. 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 cancelled, 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).

6.2.3. The `button` element and commands

6.2.3.1. 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.

6.2.3.2. 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 cancelled, 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).

6.2.4. The `input` element and commands

6.2.4.1. 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.

6.2.4.2. 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 cancelled, 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.

6.2.5. The `option` element and commands

6.2.5.1. 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.

6.2.5.2. Using the `option` element with the `command` attribute

The `command` attribute cannot be used with `option` elements.

6.2.6. The `command` element and commands

6.2.6.1. 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 stylesheet:

```
@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 or inline-level elements, or inside `commandset` elements.

Authors should not put elements or text inside `command` elements.

6.2.6.2. 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.

6.2.6.3. 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.

6.2.7. 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.

6.2.8. 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.

6.3. 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.

6.3.1. 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 stylesheet:

```
@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 stylesheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|menu { display: block; margin: 0 0 0 40px; list-style: disc; }
```

6.3.1.1. 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.

6.3.1.1.1. 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 stylesheet:

```
@namespace xh url(http://www.w3.org/1999/xhtml);
xh|menulabel { display: block; }
```

Menu bars cause `menulabel` elements to be styled further.

6.3.1.2. 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.

6.3.1.3. Using `optgroup`s as menus

When an `optgroup` element is a descendent 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.

6.3.1.4. 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), 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.

6.3.1.5. 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.

6.3.2. 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.

6.3.2.1. Displaying menu bars inline

When a `menubar` is displayed inline in the content of the document in a stylesheet-capable UA, it should be rendered according to the rules of the appropriate stylesheet 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 */ }
```

6.3.2.2. Displaying menu bars as menu bars

If the UA does not render a `menubar` element using a stylesheet 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.

6.3.3. 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.

6.3.4. 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).

6.4. 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 stylesheet instead of the markup, allowing for a model where the user has final say, and allowing for per-device stylesheets to be used to change the key based on the available input device(s).

7. Editing

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 typicially 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),

If an editable host is a block box ('`display`' has the value '`block`', '`list-item`', '`inline-block`', '`table-cell`', '`table-caption`', or '`run-in`' and the result is a block box), then the user agent must place '`table`', '`inline-table`', '`table-row-group`', '`table-header-group`', '`table-footer-group`', '`table-row`', '`table-column-group`', '`table-column`' '`none`'

8. 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.

8.1. Event listeners

In the ECMAScript DOM binding, the ECMAScript native `Function` type

implements the `EventListener` interface such that invoking the `handleEvent()` method of the object invokes the function itself, with the `evt` argument as its only argument. 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]

8.2. Bootstrapping the DOM and the `Window` interface

```
interface Window {
  readonly attribute Window            window;

  // (part of AbstractView interface)
  // readonly attribute Document          document;

  readonly attribute DOMString          mediaMode;

           attribute ErrorHandler       onerror;

  long setTimeout(in TimeoutHandler handler, in long timeout);
  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 DOMString code, in long timeout);
  long setInterval(in DOMString code, in long timeout, in DOMString lar
  void clearInterval(in long handle);
```

```
};

interface ErrorHandler {
  void handleEvent(in DOMString errorMessage, in DOMString fileName, in
};

interface TimeoutHandler {
  void handleEvent();
};
```

The **window** interface represents the chrome into which the document is rendered.

In UAs that expose the DOM to ECMAScript [ECMA262] scripts, the global scope object must implement the `Window` interface described above.

The object implementing the `Window` interface must also implement the `AbstractView` interface. [DOM2VIEWS]

The following equality must hold (assuming appropriate casting has been applied, as required by the binding):

```
window.document.defaultView == window
```

In this equality, `window` is a property of the ECMAScript global object pointing at the global object itself, the `document` property of that object is the `document` attribute of the `AbstractView` interface implemented by the global object, and the `defaultView` property of *that* object is the `defaultView` attribute of the `DocumentView` interface. The object returned by the `document` property of the `AbstractView` interface must implement the `Document` interface as well.

The **mediaMode** attribute on the `window` object returns the string that represents the canvas' current rendering mode (`screen`, `print`, etc). This is a lowercase string, as defined by the CSS specification. [CSS21]

8.2.1. 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`.

8.2.2. Timeouts

The `setTimeout` and `setInterval` methods allow authors to schedule timer-based events.

The **`setTimeout(`*`handler`*`, `*`timeout`*`)`** 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.

In the ECMAScript binding, any Function object implements `TimeoutHandler`. Such functions are 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.

8.3. Selecting elements

Both `Document`s and `Element`s shall also implement the `GetElementsByClass` interface:

```
interface GetElementsByClass {
  NodeList           getElementsByClass(in DOMString className1 [, in D
}
```

This interface defines one method, **`getElementsByClass()`**, 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`.

8.4. 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.

8.5. Serialization and parsed fragment replacement

This section will try to explain how `document.write()` actually works, and will define the `innerHTML` attribute, for both HTML and XML contexts. Wish me luck.

9. Multimedia

9.1. 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.

In non-visual media, the `canvas` element should be treated as an ordinary block-level element and the fallback content should therefore be used instead.

In non-interactive 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, 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 stylesheet:

```
@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 are used instead. The `width` attribute defaults to 300, and the `height` attribute defaults to 200.

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 stylesheet. 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` element also supports the `usemap` and `ismap` attributes. These must be handled exactly as they would if the element was an `img` element. (See HTML4 section 13.6.1 (for `usemap`) and section 13.6.2 (for `ismap`.)

As with any replaced element, the CSS background properties do apply to `canvas` elements; they are rendered below the canvas image.

```
interface HTMLCanvasElement : HTMLImageElement {

  // 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
  // (defined in HTMLImageElement)
  //       attribute long                    width;
  //       attribute long                    height;

  // returns a data: uri representing the current image as a PNG
  // does nothing on setting
  // (defined in HTMLImageElement)
  //       attribute DOMString               src;

  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.***

The `alt` DOM attribute shall return the same value as the `textContent` DOM attribute. On setting it must do nothing.

The `src` attribute must return a `data:` URI containing a representation of the image as a PNG file. [PNG] On setting it must do nothing.

The `useMap` and `isMap` DOM attributes inherited from the `HTMLImageElement` interface shall do the same as for `img` elements.

All the other DOM attributes — `align`, `border`, `hspace`, `vspace`, `longDesc`, and `name` — must do nothing on setting and return the empty string on getting.

9.1.1. 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

  // 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

  // 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.

9.1.1.1. 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 resets the context's drawing state to its initial values.

9.1.1.2. 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` method represents the scale factor in the horizontal direction and the `y` factor represents the scale factor in the vertical direction.

The **`rotate(angle)`** method adds a rotation transformation to the transformation matrix. The `angle` method represents an anti-clockwise rotation angle expressed in radians.

The **`translate(x, y)`** method adds a translating transformation to the transformation matrix. The `x` method represents the translation distance in the horizontal direction and the `y` factor represents the translation distance in the vertical direction.

9.1.1.3. 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`.

9.1.1.4. 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 they 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), 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. If the value has alpha less than 1.0, then the value must be returned in the CSS `rgba()` functional-notation format: the literal string `rgba` followed by an open parenthesis (U+0028), a base-ten integer in the range 0-255 representing the red component, a literal space and comma (U+0020 and U+002C), an integer for the green component, a space and a comma, an integer for the blue component, another space and comma, a zero (U+0030), a decimal point (U+002E), one or more digits in the range 0-9 representing the fractional part of the alpha value, and finally a close parenthesis (U+0029).

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, 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. 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` objecy 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 are 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.

9.1.1.5. 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`.

9.1.1.6. 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.

9.1.1.7. 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.

9.1.1.8. 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, clockwise)`** 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 anti-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 clockwise if the *clockwise* argument is true, and anti-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 thout 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.

9.1.1.9. 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)`. 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`. 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 regison 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.

Note: All canvases implement `HTMLImageElement`, so canvas elements can be passed to the `drawImage` methods.

9.1.1.10. 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.

9.2. 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, 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.

10. Networking

10.1. Server-sent DOM events

This specification describes a mechanism for allowing servers to dispatch DOM events into documents that expect it.

10.1.1. 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 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.

10.1.2. 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()** method shall register the specified URI as an event source on the object. The **removeEventSource()** method shall remove a URI 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 and raising an exception.

10.1.3. 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 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.

10.1.4. 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
<space>   ::= a single U+0020 character (' ')
<newline> ::= a single U+000A 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.)

10.1.5. 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 cancelled. 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 cancelled.

- `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.
> ```

10.1.6. 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.

10.1.7. 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.

10.2. 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. 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.

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.

10.3. 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 ConnectionReadEventListener 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 host used and its *peer* attribute set to "*host*:*port*".

`LocalBroadcastConnection(`*`topic`*`)`

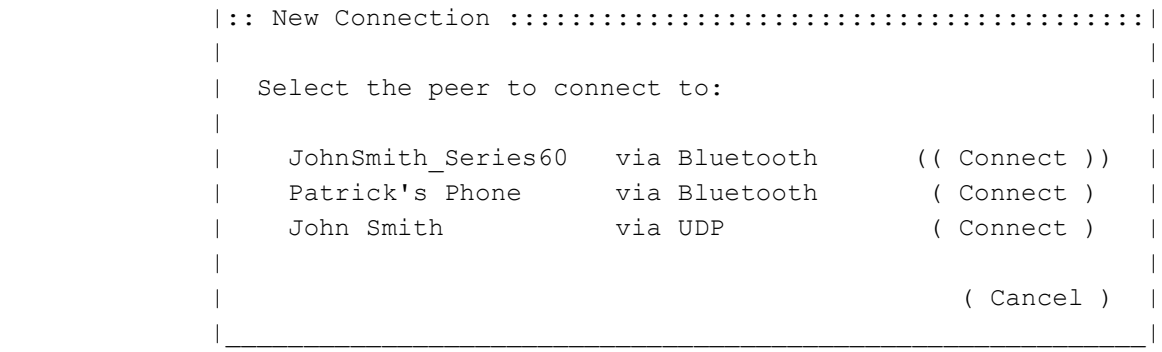
Prompts the user to confirm that a connection should be made. Such a prompt could look like this:

```
|:: 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 cancelled. 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.)

`LocalPeerConnection(`*`topic`*`)`

Prompts the user to select a connection to make, which could look like this:



While the prompt is displayed, the UA should broadcast on all supported networks, as described below.

Returns null if the prompt was cancelled. 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.)

> 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');
> ```

...

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.

The **`onread`** attribute takes a reference to an object implementing the **`ConnectionReadEventListener`** interface. In ECMAScript, such an interface is implemented by any function that takes one or two arguments.

Whenever a `read` event is invoked on a `Connection` object, if the `onread` attribute is not null, then it is invoked along with any other appropriate event listeners registered on the object, except that instead of passing the function the event object, the first argument contains the event's `data`, and the second, if any, contains the event's `source`.

...

10.3.1. 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.

10.3.2. Broadcast formats

...

10.3.3. Peer connection formats

...

10.3.4. Announcing peer connections

...

11. Focus

When an element is focused, key events are targetted at that element instead of at the root element.

11.1. 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 hyphen-minus (U+002D) representing negativity followed by one or more digits in the range 0-9 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.

11.2. 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 `Elemnet` 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.

11.3. 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].

12. Things that you can't do with this specification because they are better handled using other technologies that are further described herein

There are certain features that are not handled by this specification because a client side markup language is not the right level for them. This section covers some of the more common requests.

12.1. 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.

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, Brad Fults, Brendan Eich, Chriswa, Darin Fisher, David Hyatt, Derek Featherstone, Doron Rosenberg, fantasai, Franck 'Shift' Quélain, Henri Sivonen, Håkon Wium Lie, James Graham, James Perrett, Jan-Klaas Kollhof, Joel Spolsky, Jukka K. Korpela, Kai Hendry, 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 'Ratt' Iannarelli, Mike Shaver, Mikko Rantalainen, Olav Junker Kjær, 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 employee who first invented the `XMLHttpRequest` interface.

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.