Add developer-view styles

HTML5

A vocabulary and associated APIs for HTML and XHTML

W3C Recommendation 28 October 2014

← 6 Web application APIs – Table of contents – 8 The HTML syntax →

- 7 User interaction
  1. 7.1 The `hidden` attribute
  2. 7.2 Inert subtrees
  3. 7.3 Activation
  4. 7.4 Focus
     1. 7.4.1 Sequential focus navigation and the `tabindex` attribute
     2. 7.4.2 Focus management
     3. 7.4.3 Document-level focus APIs
     4. 7.4.4 Element-level focus APIs
  5. 7.5 Assigning keyboard shortcuts
     1. 7.5.1 Introduction
     2. 7.5.2 The `accesskey` attribute
     3. 7.5.3 Processing model
  6. 7.6 Editing
     1. 7.6.1 Making document regions editable: The `contenteditable` content attribute
     2. 7.6.2 Making entire documents editable: The `designMode` IDL attribute
     3. 7.6.3 Best practices for in-page editors
     4. 7.6.4 Editing APIs
     5. 7.6.5 Spelling and grammar checking

7 User interaction

7.1 The `hidden` attribute

All HTML elements may have the `hidden` content attribute set. The `hidden` attribute is a boolean attribute. When specified on an element, it indicates that the element is not yet, or is no longer, directly relevant to the page's current state, or that it is being used to declare content to be reused by other parts of the page as opposed to being directly accessed by the user. User agents should not render elements that have the `hidden` attribute specified. This requirement may be implemented indirectly through the style layer. For example, an HTML+CSS user agent could implement these requirements using the rules suggested in the

Rendering section.

Because this attribute is typically implemented using CSS, it's also possible to override it using CSS. For instance, a rule that applies 'display: block' to all elements will cancel the effects of the `hidden` attribute. Authors therefore have to take care when writing their style sheets to make sure that the attribute is still styled as expected.

In the following skeletal example, the attribute is used to hide the Web game's main screen until the user logs in:

```
  <h1>The Example Game</h1>
  <section id="login">
   <h2>Login</h2>
   <form>
    ...
    <!-- calls login() once the user's credentials have
been checked -->
   </form>
   <script>
    function login() {
      // switch screens
      document.getElementById('login').hidden = true;
      document.getElementById('game').hidden = false;
    }
   </script>
  </section>
  <section id="game" hidden>
   ...
  </section>
```

The `hidden` attribute must not be used to hide content that could legitimately be shown in another presentation. For example, it is incorrect to use `hidden` to hide panels in a tabbed dialog, because the tabbed interface is merely a kind of overflow presentation — one could equally well just show all the form controls in one big page with a scrollbar. It is similarly incorrect to use this attribute to hide content just from one presentation — if something is marked `hidden`, it is hidden from all presentations, including, for instance, printers.

Elements that are not themselves `hidden` must not hyperlink to elements that are `hidden`. The `for` attributes of `label` and `output` elements that are not themselves `hidden` must similarly not refer to elements that are `hidden`. In both cases, such references would cause user confusion.

Elements and scripts may, however, refer to elements that are `hidden` in other contexts.

For example, it would be incorrect to use the `href` attribute to link to a section marked with the `hidden` attribute. If the content is not applicable or relevant,

then there is no reason to link to it.

It would be fine, however, to use the ARIA `aria-describedby` attribute to refer to descriptions that are themselves `hidden`. While hiding the descriptions implies that they are not useful alone, they could be written in such a way that they are useful in the specific context of being referenced from the images that they describe.

Similarly, a `canvas` element with the `hidden` attribute could be used by a scripted graphics engine as an off-screen buffer, and a form control could refer to a hidden `form` element using its `form` attribute.

Accessibility APIs are encouraged to provide a way to expose structured content while marking it as hidden in the default view. Such content should not be perceivable to users in the normal document flow in any modality, whether using Assistive Technology (AT) or mainstream User Agents.

When such features are available, User Agents may use them to expose the full semantics of `hidden` elements to AT when appropriate, if such content is referenced indirectly by an ID reference or valid hash-name reference. This allows ATs to access the structure of these `hidden` elements upon user request, while keeping the content hidden in all presentations of the normal document flow. Authors who wish to prevent user-initiated viewing of a `hidden` element should not reference the element with such a mechanism.

Because some User Agents have flattened hidden content when exposing such content to AT, authors should not reference `hidden` content which would lose essential meaning when flattened.

For example, it would be incorrect to use the `href` attribute to link to a section marked with the `hidden` attribute. If the content is not applicable or relevant, then there is no reason to link to it.

It would be fine, however, to use the ARIA `aria-describedby` attribute to refer to descriptions that are themselves `hidden`. While hiding the descriptions implies that they are not useful alone, they could be written in such a way that they are useful in the specific context of being referenced from the images that they describe.

Similarly, a `canvas` element with the `hidden` attribute could be used by a scripted graphics engine as an off-screen buffer, and a form control could refer to a hidden `form` element using its `form` attribute.

Elements in a section hidden by the `hidden` attribute are still active, e.g. scripts and form controls in such sections still execute and submit respectively. Only their presentation to the user changes.

The **`hidden`** IDL attribute must reflect the content attribute of the same name.

7.2 Inert subtrees

A node (in particular elements and text nodes) can be marked as **inert**. When a node is inert, then the user agent must act as if the node was absent for the purposes of targeting user interaction events, may ignore the node for the purposes of text search user interfaces (commonly known as "find in page"), and may prevent the user from selecting text in that node. User agents should allow the user to override the restrictions on search and text selection, however.

> For example, consider a page that consists of just a single inert paragraph positioned in the middle of a `body`. If a user moves their pointing device from the `body` over to the inert paragraph and clicks on the paragraph, no `mouseover` event would be fired, and the `mousemove` and `click` events would be fired on the `body` element rather than the paragraph.

When a node is inert, it also can't be focusable.

An entire `Document` can be marked as **blocked by a modal dialog** *subject*. While a `Document` is so marked, every node that is in the `Document`, with the exception of the *subject* element and its descendants, must be marked inert. (The elements excepted by this paragraph can additionally be marked inert through other means; being part of a modal dialog does not "protect" a node from being marked inert.)

Only one element at a time can mark a `Document` as being blocked by a modal dialog.

7.3 Activation

***element*`.click()`**
Acts as if the element was clicked.

The **`click()`** method must run synthetic click activation steps on the element.

7.4 Focus

When an element is *focused*, key events received by the document must be targeted at that element. There may be no element focused; when no element is focused, key events received by the document must be targeted at the body element, if there is one, or else at the `Document`'s root element, if there is one. If there is no root element, key events must not be fired.

User agents may track focus for each browsing context or `Document` individually, or may support only one focused element per top-level browsing context — user agents should follow platform conventions in this regard.

Which elements within a top-level browsing context currently have focus must be independent of whether or not the top-level browsing context itself has the *system focus*.

When a child browsing context is focused, its browsing context container must also have focus.

When an element is focused, the element matches the CSS `:focus` pseudo-class.

7.4.1 Sequential focus navigation and the `tabindex` attribute

The **`tabindex`** content attribute allows authors to control whether an element is supposed to be focusable, whether it is supposed to be reachable using sequential focus navigation, and what is to be the relative order of the element for the purposes of sequential focus navigation. The name "tab index" comes from the common use of the "tab" key to navigate through the focusable elements. The term "tabbing" refers to moving forward through the focusable elements that can be reached using sequential focus navigation.

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

Each element can have a **tabindex focus flag** set, as defined below. This flag is a factor that contributes towards determining whether an element is focusable, as described in the next section.

If the attribute is specified, it must be parsed using the rules for parsing integers. The attribute's values have the following meanings:

If the attribute is omitted or parsing the value returns an error
The user agent should follow platform conventions to determine if the element's tabindex focus flag is set and, if so, whether the element can be reached using sequential focus navigation, and if so, what its relative order should be.

Modulo platform conventions, it is suggested that for the following elements, the tabindex focus flag be set:

- `a` elements that have an `href` attribute
- `link` elements that have an `href` attribute
- `button` elements
- `input` elements whose `type` attribute are not in the Hidden state
- `select` elements
- `textarea` elements

- Editing hosts
- Browsing context containers

One valid reason to ignore the platform conventions and always allow an element to be focused (by setting its tabindex focus flag) would be if the user's only mechanism for activating an element is through a keyboard action that triggers the focused element.

If the value is a negative integer
The user agent must set the element's tabindex focus flag, but should not allow the element to be reached using sequential focus navigation.

One valid reason to ignore the requirement that sequential focus navigation not allow the author to lead to the element would be if the user's only mechanism for moving the focus is sequential focus navigation. For instance, a keyboard-only user would be unable to click on a text field with a negative `tabindex`, so that user's user agent would be well justified in allowing the user to tab to the control regardless.

If the value is a zero
The user agent must set the element's tabindex focus flag, should allow the element to be reached using sequential focus navigation, and should follow platform conventions to determine the element's relative order.

If the value is greater than zero
The user agent must set the element's tabindex focus flag, should allow the element to be reached using sequential focus navigation, and should place the element in the sequential focus navigation order so that it is:

- before any focusable element whose `tabindex` attribute has been omitted or whose value, when parsed, returns an error,
- before any focusable element whose `tabindex` attribute has a value equal to or less than zero,
- after any element whose `tabindex` attribute has a value greater than zero but less than the value of the `tabindex` attribute on the element,
- after any element whose `tabindex` attribute has a value equal to the value of the `tabindex` attribute on the element but that is earlier in the document in tree order than the element,
- before any element whose `tabindex` attribute has a value equal to the value of the `tabindex` attribute on the element but that is later in the document in tree order than the element, and
- before any element whose `tabindex` attribute has a value greater than

the value of the `tabindex` attribute on the element.

An element that has its tabindex focus flag set but does not otherwise have an activation behavior defined has an activation behavior that does nothing.

This means that an element that is only focusable because of its `tabindex` attribute will fire a `click` event in response to a non-mouse activation (e.g. hitting the "enter" key while the element is focused).

The **`tabIndex`** IDL attribute must reflect the value of the `tabindex` content attribute. Its default value is 0 for elements that are focusable and –1 for elements that are not focusable.

7.4.2 Focus management

An element is **focusable** if all of the following conditions are met:

- The element's tabindex focus flag is set.
- The element is either being rendered or is a descendant of a `canvas` element that represents embedded content.
- The element is not inert.
- The element is not disabled.

In addition, each shape that is generated for an `area` element, any user-agent-provided interface components of media elements (e.g. a play button), and distinct user interface components of form controls (e.g. "up" and "down" buttons on an `<input type=number>` spin control), should be focusable, unless platform conventions dictate otherwise or unless their corresponding element is disabled. (A single `area` element can correspond to multiple shapes, since image maps can be reused with multiple images on a page.)

Notwithstanding the above, user agents may make *any* element or part of an element focusable, especially to aid with accessibility or to better match platform conventions.

The **focusing steps** for an element are as follows:

1. If the element is not in a `Document`, or if the element's `Document` has no browsing context, or if the element's `Document`'s browsing context has no top-level browsing context, or if the element is not focusable, or if the element is already focused, then abort these steps.

2. If focusing the element will remove the focus from another element, then run the unfocusing steps for that element.

3. Make the element the currently focused element in its top-level browsing context.
   Some elements, most notably `area`, can correspond to more than one distinct focusable area. If a particular area was indicated when the element was focused, then that is the area that must get focus; otherwise, e.g. when using the `focus()` method, the first such region in tree order is the one that must be focused.

4. The user agent may apply relevant platform-specific conventions for focusing widgets.
   For example, some platforms select the contents of a text field when that field is focused.

5. Fire a simple event named `focus` at the element.

User agents must synchronously run the focusing steps for an element whenever the user moves the focus to a focusable element.

The **unfocusing steps** for an element are as follows:

1. If the element is an `input` element, and the `change` event applies to the element, and the element does not have a defined activation behavior, and the user has changed the element's value or its list of selected files while the control was focused without committing that change, then fire a simple event that bubbles named `change` at the element.

2. Unfocus the element.

3. Fire a simple event named `blur` at the element.

When an element that is focused stops being a focusable element, or stops being focused without another element being explicitly focused in its stead, the user agent should synchronously run the unfocusing steps for the affected element only.

For example, this might happen because the element is removed from its `Document`, or has a `hidden` attribute added. It would also happen to an

`input` element when the element gets disabled.

7.4.3 Document-level focus APIs

***document*`.activeElement`**
Returns the currently focused element.

***document*`.hasFocus()`**
Returns true if the document has focus; otherwise, returns false.

***window*`.focus()`**
Focuses the window. Use of this method is discouraged. Allow the user to control window focus instead.

***window*`.blur()`**
Unfocuses the window. Use of this method is discouraged. Allow the user to control window focus instead.

The **`activeElement`** attribute on `Document` objects must return the element in the document that is focused. If no element in the `Document` is focused, this must return the body element.

When a child browsing context is focused, its browsing context container is also focused, by definition. For example, if the user moves the focus to a text field in an `iframe`, the `iframe` is the element with focus in the parent browsing context.

The **`hasFocus()`** method on `Document` objects must return true if the `Document`'s browsing context is focused, and all its ancestor browsing contexts are also focused, and the top-level browsing context has the *system focus*. If the `Document` has no browsing context or if its browsing context has no top-level browsing context, then the method will always return false.

The **`focus()`** method on the `Window` object, when invoked, provides a hint to the user agent that the script believes the user might be interested in the contents of the browsing context of the `Window` object on which the method was invoked.

User agents are encouraged to have this `focus()` method trigger some kind of notification.

The **`blur()`** method on the `Window` object, when invoked, provides a hint to the user agent that the script believes the user probably is not currently interested in the contents of the browsing context of the `Window` object on which the method was invoked, but that the contents might become interesting again in the future.

User agents are encouraged to ignore calls to this `blur()` method entirely.

Historically the `focus()` and `blur()` methods actually affected the system focus, but hostile sites widely abuse this behavior to the user's detriment.

7.4.4 Element-level focus APIs

***element*`.focus()`**
Focuses the element.

***element*`.blur()`**
Unfocuses the element. Use of this method is discouraged. Focus another element instead.

Do not use this method to hide the focus ring. Do not use any other method that hides the focus ring from keyboard users, in particular do not use a CSS rule to override the 'outline' property. Removal of the focus ring leads to serious accessibility issues for users who navigate and interact with interactive content using the keyboard.

The **`focus()`** method, when invoked, must run the following algorithm:

1. If the element is marked as *locked for focus*, then abort these steps.
2. Mark the element as **locked for focus**.
3. Run the focusing steps for the element.
4. Unmark the element as *locked for focus*.

The **`blur()`** method, when invoked, should run the unfocusing steps for the element on which the method was called instead. User agents may selectively or uniformly ignore calls to this method for usability reasons.

> For example, if the `blur()` method is unwisely being used to remove the focus ring for aesthetics reasons, the page would become unusable by keyboard users. Ignoring calls to this method would thus allow keyboard users to interact with the page.

7.5 Assigning keyboard shortcuts

7.5.1 Introduction

This section is non-normative.

Each element that can be activated or focused can be assigned a single key combination to activate it, using the `accesskey` attribute.

The exact shortcut is determined by the user agent, based on information about the user's keyboard, what keyboard shortcuts already exist on the platform, and what other shortcuts have been specified on the page, using the information

provided in the accesskey attribute as a guide.

In order to ensure that a relevant keyboard shortcut is available on a wide variety of input devices, the author can provide a number of alternatives in the accesskey attribute.

Each alternative consists of a single character, such as a letter or digit.

User agents can provide users with a list of the keyboard shortcuts, but authors are encouraged to do so also. The accessKeyLabel IDL attribute returns a string representing the actual key combination assigned by the user agent.

In this example, an author has provided a button that can be invoked using a shortcut key. To support full keyboards, the author has provided "C" as a possible key. To support devices equipped only with numeric keypads, the author has provided "1" as another possibly key.

To tell the user what the shortcut key is, the author has this script here opted to explicitly add the key combination to the button's label:

```
function addShortcutKeyLabel(button) {
  if (button.accessKeyLabel != '')
    button.value += ' (' + button.accessKeyLabel + ')';
}
addShortcutKeyLabel(document.getElementById('c'));
```

Browsers on different platforms will show different labels, even for the same key combination, based on the convention prevalent on that platform. For example, if the key combination is the Control key, the Shift key, and the letter C, a Windows browser might display "Ctrl+Shift+C", whereas a Mac browser might display "^⇧C", while an Emacs browser might just display "C-C". Similarly, if the key combination is the Alt key and the Escape key, Windows might use "Alt+Esc", Mac might use "⌥⎋", and an Emacs browser might use "M-ESC" or "ESC ESC".

In general, therefore, it is unwise to attempt to parse the value returned from the accessKeyLabel IDL attribute.

7.5.2 The accesskey attribute

All HTML elements may have the accesskey content attribute set. The accesskey attribute's value is used by the user agent as a guide for creating a keyboard shortcut that activates or focuses the element.

If specified, the value must be an ordered set of unique space-separated tokens that are case-sensitive, each of which must be exactly one Unicode code point in length.

In the following example, a variety of links are given with access keys so that keyboard users familiar with the site can more quickly navigate to the relevant

pages:

```
<nav>
 <p>
  <a title="Consortium Activities" accesskey="A" href="/
Consortium/activities">Activities</a> |
  <a title="Technical Reports and Recommendations"
accesskey="T" href="/TR/">Technical Reports</a> |
  <a title="Alphabetical Site Index" accesskey="S" href="/
Consortium/siteindex">Site Index</a> |
  <a title="About This Site" accesskey="B" href="/
Consortium/">About Consortium</a> |
  <a title="Contact Consortium" accesskey="C" href="/
Consortium/contact">Contact</a>
 </p>
</nav>
```

In the following example, the search field is given two possible access keys, "s" and "0" (in that order). A user agent on a device with a full keyboard might pick `Ctrl+Alt+S` as the shortcut key, while a user agent on a small device with just a numeric keypad might pick just the plain unadorned key `0`:

```
<form action="/search">
 <label>Search: <input type="search" name="q" accesskey="s
0"></label>
 <input type="submit">
</form>
```

In the following example, a button has possible access keys described. A script then tries to update the button's label to advertise the key combination the user agent selected.

```
<input type=submit accesskey="N @ 1" value="Compose">
...
<script>
 function labelButton(button) {
   if (button.accessKeyLabel)
     button.value += ' (' + button.accessKeyLabel + ')';
 }
 var inputs = document.getElementsByTagName('input');
 for (var i = 0; i < inputs.length; i += 1) {
   if (inputs[i].type == "submit")
     labelButton(inputs[i]);
 }
</script>
```

On one user agent, the button's label might become "`Compose (⌘N)`". On another, it might become "`Compose (Alt+⇧+1)`". If the user agent doesn't assign a key, it will be just "`Compose`". The exact string depends on what the assigned access key is, and on how the user agent represents that key

combination.

7.5.3 Processing model

An element's **assigned access key** is a key combination derived from the element's `accesskey` content attribute. Initially, an element must not have an assigned access key.

Whenever an element's `accesskey` attribute is set, changed, or removed, the user agent must update the element's assigned access key by running the following steps:

1. If the element has no `accesskey` attribute, then skip to the *fallback* step below.

2. Otherwise, split the attribute's value on spaces, and let *keys* be the resulting tokens.

3. For each value in *keys* in turn, in the order the tokens appeared in the attribute's value, run the following substeps:

    1. If the value is not a string exactly one Unicode code point in length, then skip the remainder of these steps for this value.

    2. If the value does not correspond to a key on the system's keyboard, then skip the remainder of these steps for this value.

    3. If the user agent can find a mix of zero or more modifier keys that, combined with the key that corresponds to the value given in the attribute, can be used as the access key, then the user agent may assign that combination of keys as the element's assigned access key and abort these steps.

4. *Fallback*: Optionally, the user agent may assign a key combination of its choosing as the element's assigned access key and then abort these steps.

5. If this step is reached, the element has no assigned access key.

Once a user agent has selected and assigned an access key for an element, the user agent should not change the element's assigned access key unless the `accesskey` content attribute is changed or the element is moved to another

`Document`.

User agents might expose elements that have an `accesskey` attribute in other ways as well, e.g. in a menu displayed in response to a specific key combination.

The **`accessKey`** IDL attribute must reflect the `accesskey` content attribute.

The **`accessKeyLabel`** IDL attribute must return a string that represents the element's assigned access key, if any. If the element does not have one, then the IDL attribute must return the empty string.

7.6 Editing

7.6.1 Making document regions editable: The `contenteditable` content attribute

The **`contenteditable`** attribute is an enumerated attribute whose keywords are the empty string, `true`, and `false`. The empty string and the `true` keyword map to the *true* state. The `false` keyword maps to the *false* state. In addition, there is a third state, the *inherit* state, which is the *missing value default* (and the *invalid value default*).

The *true* state indicates that the element is editable. The *inherit* state indicates that the element is editable if its parent is. The *false* state indicates that the element is not editable.

***element*`.contentEditable` [`=` *value*]**
Returns "`true`", "`false`", or "`inherit`", based on the state of the `contenteditable` attribute.

Can be set, to change that state.

Throws a `SyntaxError` exception if the new value isn't one of those strings.

***element*`.isContentEditable`**
Returns true if the element is editable; otherwise, returns false.

The **`contentEditable`** IDL attribute, on getting, must return the string "`true`" if the content attribute is set to the true state, "`false`" if the content attribute is set to the false state, and "`inherit`" otherwise. On setting, if the new value is an ASCII case-insensitive match for the string "`inherit`" then the content attribute must be removed, if the new value is an ASCII case-insensitive match for the string "`true`" then the content attribute must be set to the string "`true`", if the new value is an ASCII case-insensitive match for the string "`false`" then the content attribute must be set to the string "`false`", and otherwise the attribute setter must throw a `SyntaxError` exception.

The **`isContentEditable`** IDL attribute, on getting, must return true if the

element is either an editing host or editable, and false otherwise.

7.6.2 Making entire documents editable: The `designMode` IDL attribute

Documents have a **`designMode`**, which can be either enabled or disabled.

***document*`.designMode` [= *value*]**
Returns "`on`" if the document is editable, and "`off`" if it isn't.

Can be set, to change the document's current state. This focuses the document and resets the selection in that document.

The `designMode` IDL attribute on the `Document` object takes two values, "`on`" and "`off`". On setting, the new value must be compared in an ASCII case-insensitive manner to these two values; if it matches the "`on`" value, then `designMode` must be enabled, and if it matches the "`off`" value, then `designMode` must be disabled. Other values must be ignored.

On getting, if `designMode` is enabled, the IDL attribute must return the value "`on`"; otherwise it is disabled, and the attribute must return the value "`off`".

The last state set must persist until the document is destroyed or the state is changed. Initially, documents must have their `designMode` disabled.

When the `designMode` changes from being disabled to being enabled, the user agent must synchronously reset the document's active range's start and end boundary points to be at the start of the `Document` and then run the focusing steps for the root element of the `Document`, if any.

7.6.3 Best practices for in-page editors

Authors are encouraged to set the 'white-space' property on editing hosts and on markup that was originally created through these editing mechanisms to the value 'pre-wrap'. Default HTML whitespace handling is not well suited to WYSIWYG editing, and line wrapping will not work correctly in some corner cases if 'white-space' is left at its default value.

As an example of problems that occur if the default 'normal' value is used instead, consider the case of the user typing "`yellow␣␣ball`", with two spaces (here represented by "␣␣") between the words. With the editing rules in place for the default value of 'white-space' ('normal'), the resulting markup will either consist of "`yellow  ball`" or "`yellow  ball`"; i.e., there will be a non-breaking space between the two words in addition to the regular space. This is necessary because the 'normal' value for 'white-space' requires adjacent regular spaces to be collapsed together.

In the former case, "`yellow␣`" might wrap to the next line ("␣" being used here to represent a non-breaking space) even though "`yellow`" alone might fit at the end of the line; in the latter case, "`␣ball`", if wrapped to the start of the line,

would have visible indentation from the non-breaking space.

When 'white-space' is set to 'pre-wrap', however, the editing rules will instead simply put two regular spaces between the words, and should the two words be split at the end of a line, the spaces would be neatly removed from the rendering.

7.6.4 Editing APIs

The definition of the terms **active range**, **editing host**, and **editable**, the user interface requirements of elements that are editing hosts or editable, the **`execCommand()`**, **`queryCommandEnabled()`**, **`queryCommandIndeterm()`**, **`queryCommandState()`**, **`queryCommandSupported()`**, and **`queryCommandValue()`** methods, text selections, and the **delete the selection** algorithm are defined in the HTML Editing APIs specification. The interaction of editing and the undo/redo features in user agents is defined by the UndoManager and DOM Transaction specification. [EDITING] [UNDO]

7.6.5 Spelling and grammar checking

User agents can support the checking of spelling and grammar of editable text, either in form controls (such as the value of `textarea` elements), or in elements in an editing host (e.g. using `contenteditable`).

For each element, user agents must establish a **default behavior**, either through defaults or through preferences expressed by the user. There are three possible default behaviors for each element:

true-by-default
The element will be checked for spelling and grammar if its contents are editable.
false-by-default
The element will never be checked for spelling and grammar.
inherit-by-default
The element's default behavior is the same as its parent element's. Elements that have no parent element cannot have this as their default behavior.

The **`spellcheck`** attribute is an enumerated attribute whose keywords are the empty string, `true` and `false`. The empty string and the `true` keyword map to the *true* state. The `false` keyword maps to the *false* state. In addition, there is a third state, the *default* state, which is the *missing value default* (and the *invalid value default*).

The true state indicates that the element is to have its spelling and grammar checked. The default state indicates that the element is to act according to a default behavior, possibly based on the parent element's own `spellcheck` state, as defined below. The false state indicates that the element is not to be checked.

***element*.spellcheck [= *value*]**
Returns true if the element is to have its spelling and grammar checked; otherwise, returns false.

Can be set, to override the default and set the `spellcheck` content attribute.

The **`spellcheck`** IDL attribute, on getting, must return true if the element's `spellcheck` content attribute is in the *true* state, or if the element's `spellcheck` content attribute is in the *default* state and the element's default behavior is true-by-default, or if the element's `spellcheck` content attribute is in the *default* state and the element's default behavior is inherit-by-default and the element's parent element's `spellcheck` IDL attribute would return true; otherwise, if none of those conditions applies, then the attribute must instead return false.

The `spellcheck` IDL attribute is not affected by user preferences that override the `spellcheck` content attribute, and therefore might not reflect the actual spellchecking state.

On setting, if the new value is true, then the element's `spellcheck` content attribute must be set to the literal string "`true`", otherwise it must be set to the literal string "`false`".

User agents must only consider the following pieces of text as checkable for the purposes of this feature:

- The value of `input` elements whose `type` attributes are in the Text, Search, URL, or E-mail states and that are *mutable* (i.e. that do not have the `readonly` attribute specified and that are not disabled).
- The value of `textarea` elements that do not have a `readonly` attribute and that are not disabled.
- Text in `Text` nodes that are children of editing hosts or editable elements.
- Text in attributes of editable elements.

For text that is part of a `Text` node, the element with which the text is associated is the element that is the immediate parent of the first character of the word, sentence, or other piece of text. For text in attributes, it is the attribute's element. For the values of `input` and `textarea` elements, it is the element itself.

To determine if a word, sentence, or other piece of text in an applicable element (as defined above) is to have spelling- and grammar-checking enabled, the UA

must use the following algorithm:

1. If the user has disabled the checking for this text, then the checking is disabled.

2. Otherwise, if the user has forced the checking for this text to always be enabled, then the checking is enabled.

3. Otherwise, if the element with which the text is associated has a `spellcheck` content attribute, then: if that attribute is in the *true* state, then checking is enabled; otherwise, if that attribute is in the *false* state, then checking is disabled.

4. Otherwise, if there is an ancestor element with a `spellcheck` content attribute that is not in the *default* state, then: if the nearest such ancestor's `spellcheck` content attribute is in the *true* state, then checking is enabled; otherwise, checking is disabled.

5. Otherwise, if the element's default behavior is true-by-default, then checking is enabled.

6. Otherwise, if the element's default behavior is false-by-default, then checking is disabled.

7. Otherwise, if the element's parent element has *its* checking enabled, then checking is enabled.

8. Otherwise, checking is disabled.

If the checking is enabled for a word/sentence/text, the user agent should indicate spelling and grammar errors in that text. User agents should take into account the other semantics given in the document when suggesting spelling and grammar corrections. User agents may use the language of the element to determine what spelling and grammar rules to use, or may use the user's preferred language settings. UAs should use `input` element attributes such as `pattern` to ensure that the resulting value is valid, where possible.

If checking is disabled, the user agent should not indicate spelling or grammar errors for that text.

Even when checking is enabled, user agents may opt to not report spelling or grammar errors in text that the user agent deems the user has no interest in having checked (e.g. text that was already present when the page was loaded, or that the user did not type, or text in controls that the user has not focused, or in parts of e-mail addresses that the user agent is not confident were misspelt).

The element with ID "a" in the following example would be the one used to determine if the word "Hello" is checked for spelling errors. In this example, it

would not be.

```
<div contenteditable="true">
 <span spellcheck="false" id="a">Hell</span><em>o!</em>
</div>
```

The element with ID "b" in the following example would have checking enabled (the leading space character in the attribute's value on the input element causes the attribute to be ignored, so the ancestor's value is used instead, regardless of the default).

```
<p spellcheck="true">
 <label>Name: <input spellcheck=" false" id="b"></label>
</p>
```

This specification does not define the user interface for spelling and grammar checkers. A user agent could offer on-demand checking, could perform continuous checking while the checking is enabled, or could use other interfaces.