W3C Working Group Note

WAI-ARIA Authoring Practices 1.1

W3C Working Group Note 14 August 2019

This version:
https://www.w3.org/TR/2019/NOTE-wai-aria-practices-1.1-20190814/

Latest published version:
https://www.w3.org/TR/wai-aria-practices-1.1/

Latest editor's draft:
https://w3c.github.io/aria-practices/

Previous version:
https://www.w3.org/TR/2019/NOTE-wai-aria-practices-1.1-20190207/

Editors:
Matt King (Facebook)
JaEun Jemma Ku (University of Illinois)
James Nurthen (Adobe)
Zoë Bijl (Invited Expert)
Michael Cooper (W3C)

Former editors:
Joseph Scheuhammer (Inclusive Design Research Centre, OCAD University) (Editor until October 2014)
Lisa Pappas (SAS) (Editor until October 2009)
Rich Schwerdtfeger (IBM Corporation) (Editor until October 2014)

Abstract

This document provides readers with an understanding of how to use *WAI-ARIA 1.1* [WAI-ARIA] to create accessible rich internet applications. It describes considerations that might not be evident to most authors from the WAI-ARIA specification alone and recommends approaches to make widgets, navigation, and behaviors accessible using WAI-ARIA roles, states, and properties. This document is directed primarily to Web application developers, but the guidance is also useful for user agent and assistive technology developers.

This document is part of the WAI-ARIA suite described in the WAI-ARIA Overview.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This is the WAI-ARIA Authoring Practices 1.1 Working Group Note by the Accessible Rich Internet Applications Working Group. It supports the Accessible Rich Internet Applications 1.1 W3C Recommendation [wai-aria-1.1], providing detailed advice and examples beyond what would be appropriate to a technical specification but which are important to understand the specification.

WAI-ARIA Authoring Practices 1.1 was previously published as a Working Group Note in December 2017, to accompany the WAI-ARIA 1.1 Recommendation, and was republished in July 2018 and February 2019 with additional design pattern and examples, quality improvements, and improveed support for WAI-ARIA 1.1. Details of changes are described in the change log. Separately, WAI-ARIA Authoring Practices 1.2 includes the improvements in this document plus additional features specific to WAI-ARIA 1.2.

To comment, file an issue in the W3C ARIA Practices GitHub repository, or if that is not possible, send email to public-aria@w3.org (comment archive).

This document was published by the Accessible Rich Internet Applications Working Group as a Working Group Note.

Comments regarding this document are welcome. Please send them to public-aria@w3.org (archives).

Publication as a Working Group Note does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C Patent Policy.

This document is governed by the 1 March 2019 W3C Process Document.

Table of Contents

1. Introduction §

This section is *informative.*

WAI-ARIA Authoring Practices is a guide for understanding how to use *WAI-ARIA 1.1* to create an accessible Rich Internet Application. It provides guidance on the appropriate application of WAI-ARIA, describes recommended WAI-ARIA usage patterns, and explains concepts behind them.

Languages used to create rich and dynamic web sites, e.g., HTML, JavaScript, CSS, and SVG, do not natively include all the features required to make sites usable by people who use assistive technologies (AT) or who rely on keyboard navigation. The W3C Web Accessibility Initiative's (WAI) Accessible Rich Internet Applications working group (ARIA WG) is addressing these deficiencies through several W3C standards efforts. The WAI-ARIA Overview provides additional background on WAI-ARIA, summarizes those efforts, and lists the other documents included in the WAI-ARIA suite.

After a brief “Read Me First” section, the guide begins with ARIA implementation patterns for common widgets that both enumerate expected behaviors and demonstrate those behaviors with working code. The implementation patterns and examples refer to detailed explanations of supporting concepts in subsequent guidance sections. The guidance sections cover more general topics such as use of ARIA landmarks, practices for keyboard interfaces, grid and table properties, and the effects of role `presentation`.

2. Read Me First §

2.1 No ARIA is better than Bad ARIA §

Functionally, ARIA roles, states, and properties are analogous to a CSS for assistive technologies. For screen reader users, ARIA controls the rendering of their non-visual experience. Incorrect ARIA misrepresents visual experiences, with potentially devastating effects on their corresponding non-visual experiences.

Before using ARIA or any of the guidance in this document, please take time to understand the following two essential principles.

Principle 1: A role is a promise §

This code:

```
<div role="button">Place Order</div>
```

Is a promise that the author of that has also incorporated JavaScript that provides the keyboard interactions expected for a button. Unlike HTML input elements, ARIA roles do not cause browsers to provide keyboard behaviors or styling.

Using a role without fulfilling the promise of that role is similar to making a "Place Order" button that abandons an order and empties the shopping cart.

One of the objectives of this guide is to define expected behaviors for each ARIA role.

Principle 2: ARIA Can Both Cloak and Enhance, Creating Both Power and Danger §

The information assistive technologies need about the meaning and purpose of user interface elements is called accessibility semantics. From the perspective of assistive technologies, ARIA gives authors the ability to dress up HTML and SVG elements with critical accessibility semantics that the assistive technologies would not otherwise be able to reliably derive.

Some of ARIA is like a cloak; it covers up, or overrides, the original semantics or content.

```
ıitem">Assistive tech users perceive this element as an item in a menu, not a link.</a>
l="Assistive tech users can only perceive the contents of this aria-label, not the link text">Link Tex
```

On the other hand, some uses of ARIA are more like suspenders or belts; they add meaning that provides essential support to the original content.

```
<button aria-pressed="false">Mute</button>
```

This is the power of ARIA. It enables authors to describe nearly any user interface component in ways that assistive technologies can reliably interpret, thus making components accessible to assistive technology users.

This is also the danger of ARIA. Authors can inadvertently override accessibility semantics.

```
<table role="log">
  <!--
    Table that assistive technology users will not perceive as a table.
    The log role tells browser this is a log, not a table.
  -->
</table>
<ul role="navigation">
  <!-- This is a navigation region, not a list. -->
  <li><a href="uri1">nav link 1</li>
  <li><a href="uri2">nav link 2</li>
  <!-- ERROR! Previous list items are not in a list! -->
</ul>
```

2.2 Browser and Assistive Technology Support §

Testing assistive technology interoperability is essential before using code from this guide in production. Because the purpose of this guide is to illustrate appropriate use of ARIA 1.1 as defined in the ARIA specification, the design patterns,

reference examples, and sample code intentionally **do not** describe and implement coding techniques for working around problems caused by gaps in support for ARIA 1.1 in browsers and assistive technologies. It is thus advisable to test implementations thoroughly with each browser and assistive technology combination that is relevant within a target audience.

Similarly, JavaScript and CSS in this guide is written to be compatible with the most recent version of Chrome, Firefox, Internet Explorer, and Safari at the time of writing. In particular, some JavaScript and CSS may not function correctly in Internet Explorer version 10 or earlier.

Except in cases where the ARIA Working Group and other contributors have overlooked an error, examples in this guide that do not function well in a particular browser or with a specific assistive technology are demonstrating browser or assistive technology bugs. Browser and assistive technology developers can thus utilize code in this guide to help assess the quality of their support for ARIA 1.1.

2.3 Mobile and Touch Support §

Currently, this guide does not indicate which examples are compatible with mobile browsers or touch interfaces. While some of the examples include specific features that enhance mobile and touch support, some ARIA features are not supported in any mobile browser. In addition, there is not yet a standardized approach for providing touch interactions that work across mobile browsers.

More guidance about touch and mobile support is planned for future releases of the guide.

3. Design Patterns and Widgets §

This section demonstrates how to make common rich internet application patterns and widgets accessible by applying WAI-ARIA roles, states, and properties and implementing keyboard support.

3.1 Accordion (Sections With Show/Hide Functionality) §

An accordion is a vertically stacked set of interactive headings that each contain a title, content snippet, or thumbnail representing a section of content. The headings function as controls that enable users to reveal or hide their associated sections of content. Accordions are commonly used to reduce the need to scroll when presenting multiple sections of content on a single page.

Terms for understanding accordions include:

Accordion Header:
: Label for or thumbnail representing a section of content that also serves as a control for showing, and in some implementations, hiding the section of content.

Accordion Panel:
: Section of content associated with an accordion header.

In some accordions, there are additional elements that are always visible adjacent to the accordion header. For instance, a menubutton may accompany each accordion header to provide access to actions that apply to that section. And, in some cases, a snippet of the hidden content may also be visually persistent.

Example §

Accordion Example: demonstrates a form divided into three sections using an accordion to show one section at a time.

Keyboard Interaction §

- `Enter` or `Space`:
  - When focus is on the accordion header for a collapsed panel, expands the associated panel. If the implementation allows only one panel to be expanded, and if another panel is expanded, collapses that panel.
  - When focus is on the accordion header for an expanded panel, collapses the panel if the implementation supports collapsing. Some implementations require one panel to be expanded at all times and allow only one panel to be expanded; so, they do not support a collapse function.
- `Tab`: Moves focus to the next focusable element; all focusable elements in the accordion are included in the page `Tab` sequence.
- `Shift + Tab`: Moves focus to the previous focusable element; all focusable elements in the accordion are included in the page `Tab` sequence.
- `Down Arrow` (Optional): If focus is on an accordion header, moves focus to the next accordion header. If focus is on the last accordion header, either does nothing or moves focus to the first accordion header.
- `Up Arrow` (Optional): If focus is on an accordion header, moves focus to the previous accordion header. If focus is on the first accordion header, either does nothing or moves focus to the last accordion header.
- `Home` (Optional): When focus is on an accordion header, moves focus to the first accordion header.
- `End` (Optional): When focus is on an accordion header, moves focus to the last accordion header.

WAI-ARIA Roles, States, and Properties: §

- The title of each accordion header is contained in an element with role button.
- Each accordion header `button` is wrapped in an element with role heading that has a value set for aria-level that is appropriate for the information architecture of the page.
  - If the native host language has an element with an implicit `heading` and `aria-level`, such as an HTML heading tag, a native host language element may be used.
  - The `button` element is the only element inside the `heading` element. That is, if there are other visually persistent elements, they are not included inside the `heading` element.
- If the accordion panel associated with an accordion header is visible, the header `button` element has aria-expanded set to `true`. If the panel is not visible, aria-expanded is set to `false`.
- The accordion header `button` element has aria-controls set to the ID of the element containing the accordion panel content.
- If the accordion panel associated with an accordion header is visible, and if the accordion does not permit the panel to be collapsed, the header `button` element has aria-disabled set to `true`.
- Optionally, each element that serves as a container for panel content has role region and aria-labelledby with a value that refers to the button that controls display of the panel.
  - Avoid using the `region` role in circumstances that create landmark region proliferation, e.g., in an accordion that contains more than approximately 6 panels that can be expanded at the same time.

- Role `region` is especially helpful to the perception of structure by screen reader users when panels contain heading elements or a nested accordion.

3.2 Alert §

An alert is an element that displays a brief, important message in a way that attracts the user's attention without interrupting the user's task. Dynamically rendered alerts are automatically announced by most screen readers, and in some operating systems, they may trigger an alert sound. It is important to note that, at this time, screen readers do not inform users of alerts that are present on the page before page load completes.

Because alerts are intended to provide important and potentially time-sensitive information without interfering with the user's ability to continue working, it is crucial they do not affect keyboard focus. The alert dialog is designed for situations where interrupting work flow is necessary.

It is also important to avoid designing alerts that disappear automatically. An alert that disappears too quickly can lead to failure to meet WCAG 2.0 success criterion 2.2.3. Another critical design consideration is the frequency of interruption caused by alerts. Frequent interruptions inhibit usability for people with visual and cognitive disabilities, which makes meeting the requirements of WCAG 2.0 success criterion 2.2.4 more difficult.

Example §

Alert Example

Keyboard Interaction §

An alert (WAI-ARIA live region) does not require any keyboard interaction.

WAI-ARIA Roles, States, and Properties §

The widget has a role of alert.

3.3 Alert and Message Dialogs §

An alert dialog is a modal dialog that interrupts the user's workflow to communicate an important message and acquire a response. Examples include action confirmation prompts and error message confirmations. The alertdialog role enables assistive technologies and browsers to distinguish alert dialogs from other dialogs so they have the option of giving alert dialogs special treatment, such as playing a system alert sound.

Example §

Alert Dialog Example: A confirmation prompt that demonstrates an alert dialog.

Keyboard Interaction §

See the keyboard interaction section for the modal dialog pattern.

WAI-ARIA Roles, States, and Properties §

- The element that contains all elements of the dialog, including the alert message and any dialog buttons, has role alertdialog.
- The element with role `alertdialog` has either:
  - A value for aria-labelledby that refers to the element containing the title of the dialog if the dialog has a visible label.
  - A value for aria-label if the dialog does not have a visible label.
- The element with role `alertdialog` has a value set for aria-describedby that refers to the element containing the alert message.

3.4 Breadcrumb §

A breadcrumb trail consists of a list of links to the parent pages of the current page in hierarchical order. It helps users find their place within a website or web application. Breadcrumbs are often placed horizontally before a page's main content.

Example §

Breadcrumb design pattern example

Keyboard Interaction §

Not applicable.

WAI-ARIA Roles, States, and Properties §

- Breadcrumb trail is contained within a navigation landmark region.
- The landmark region is labelled via aria-label or aria-labelledby.
- The link to the current page has aria-current set to `page`. If the element representing the current page is not a link, aria-current is optional.

3.5 Button §

A button is a widget that enables users to trigger an action or event, such as submitting a form, opening a dialog, canceling an action, or performing a delete operation. A common convention for informing users that a button launches a dialog is to append "…" (ellipsis) to the button label, e.g., "Save as…".

In addition to the ordinary button widget, WAI-ARIA supports 2 other types of buttons:

- Toggle button: A two-state button that can be either off (not pressed) or on (pressed). To tell assistive technologies that a button is a toggle button, specify a value for the attribute aria-pressed. For example, a button labelled mute in an

audio player could indicate that sound is muted by setting the pressed state true. **Important:** it is critical the label on a toggle does not change when its state changes. In this example, when the pressed state is true, the label remains "Mute" so a screen reader would say something like "Mute toggle button pressed". Alternatively, if the design were to call for the button label to change from "Mute" to "Unmute," the aria-pressed attribute would not be needed.

- Menu button: as described in the menu button pattern, a button is revealed to assistive technologies as a menu button if it has the property aria-haspopup set to either `menu` or `true`.

> NOTE
>
> The types of actions performed by buttons are distinctly different from the function of a link (see link pattern). It is important that both the appearance and role of a widget match the function it provides. Nevertheless, elements occasionally have the visual style of a link but perform the action of a button. In such cases, giving the element role `button` helps assistive technology users understand the function of the element. However, a better solution is to adjust the visual design so it matches the function and ARIA role.

Examples §

Button Examples: Examples of clickable HTML `div` and `span` elements made into accessible command and toggle buttons.

Keyboard Interaction §

When the button has focus:

- `Space`: Activates the button.
- `Enter`: Activates the button.
- Following button activation, focus is set depending on the type of action the button performs. For example:
  - If activating the button opens a dialog, the focus moves inside the dialog. (see dialog pattern)
  - If activating the button closes a dialog, focus typically returns to the button that opened the dialog unless the function performed in the dialog context logically leads to a different element. For example, activating a cancel button in a dialog returns focus to the button that opened the dialog. However, if the dialog were confirming the action of deleting the page from which it was opened, the focus would logically move to a new context.
  - If activating the button does not dismiss the current context, then focus typically remains on the button after activation, e.g., an Apply or Recalculate button.
  - If the button action indicates a context change, such as move to next step in a wizard or add another search criteria, then it is often appropriate to move focus to the starting point for that action.
  - If the button is activated with a shortcut key, the focus usually remains in the context from which the shortcut key was activated. For example, if `Alt + U` were assigned to an "Up" button that moves the currently focused item in a list one position higher in the list, pressing `Alt + U` when the focus is in the list would not move the focus from the list.

WAI-ARIA Roles, States, and Properties §

- The button has role of button.

- The `button` has an accessible label. By default, the accessible name is computed from any text content inside the button element. However, it can also be provided with aria-labelledby or aria-label.
- If a description of the button's function is present, the button element has aria-describedby set to the ID of the element containing the description.
- When the action associated with a button is unavailable, the button has aria-disabled set to `true`.
- If the button is a toggle button, it has an aria-pressed state. When the button is toggled on, the value of this state is `true`, and when toggled off, the state is `false`.

3.6 Carousel (Slide Show or Image Rotator) §

A carousel presents a set of items, referred to as slides, by sequentially displaying a subset of one or more slides. Typically, one slide is displayed at a time, and users can activate a next or previous slide control that hides the current slide and "rotates" the next or previous slide into view. In some implementations, rotation automatically starts when the page loads, and it may also automatically stop once all the slides have been displayed. While a slide may contain any type of content, image carousels where each slide contains nothing more than a single image are common.

Ensuring all users can easily control and are not adversely effected by slide rotation is an essential aspect of making carousels accessible. For instance, the screen reader experience can be confusing and disorienting if slides that are not visible on screen are incorrectly hidden, e.g., displayed off-screen. Similarly, if slides rotate automatically and a screen reader user is not aware of the rotation, the user may read an element on slide one, execute the screen reader command for next element, and, instead of hearing the next element on slide one, hear an element from slide 2 without any knowledge that the element just announced is from an entirely new context.

Features needed to provide sufficient rotation control include:

- Buttons for displaying the previous and next slides.
- Optionally, a control, or group of controls, for choosing a specific slide to display. For example, slide picker controls can be marked up as tabs in a tablist with the slide represented by a tabpanel element.
- If the carousel can automatically rotate, it also:
  - Has a button for stopping and restarting rotation. This is particularly important for supporting assistive technologies operating in a mode that does not move either keyboard focus or the mouse.
  - Stops rotating when keyboard focus enters the carousel. It does not restart unless the user explicitly requests it to do so.
  - Stops rotating whenever the mouse is hovering over the carousel.

Example §

Auto-Rotating Image Carousel Example: A basic image carousel that demonstrates the accessibility features necessary for carousels that rotate automatically on page load.

Terms §

The following terms are used to describe components of a carousel.

Slide

A single content container within a set of content containers that hold the content to be presented by the carousel.

Rotation Control
An interactive element that stops and starts automatic slide rotation.

Next Slide Control
An interactive element, often styled as an arrow, that displays the next slide in the rotation sequence.

Previous Slide Control
An interactive element, often styled as an arrow, that displays the previous slide in the rotation sequence.

Slide Picker Controls
A group of elements, often styled as small dots, that enable the user to pick a specific slide in the rotation sequence to display.

Keyboard Interaction §

- If the carousel has an auto-rotate feature, automatic slide rotation stops when any element in the carousel receives keyboard focus. It does not resume unless the user activates the rotation control.
- `Tab` and `Shift + Tab`: Move focus through the interactive elements of the carousel as specified by the page tab sequence -- scripting for `Tab` is not necessary.
- Button elements implement the keyboard interaction defined in the button pattern. Note: Activating the rotation control, next slide, and previous slide do not move focus, so users may easily repetitively activate them as many times as desired.
- If present, the rotation control is the first element in the `Tab` sequence inside the carousel. It is essential that it precede the rotating content so it can be easily located.
- If tab elements are used for slide picker controls, they implement the keyboard interaction defined in the Tabs Pattern.

WAI-ARIA Roles, States, and Properties §

This section describes the element composition for three styles of carousels:

- Basic: Has rotation, previous slide, and next slide controls but no slide picker controls.
- Tabbed: Has basic controls plus a single tab stop for slide picker controls implemented using the tabs pattern.
- Grouped: Has basic controls plus a series of tab stops in a group of slide picker controls where each control implements the button pattern. Because each slide selector button adds an element to the page tab sequence, this style is the least friendly for keyboard users.

Basic carousel elements §

- A carousel container element that encompasses all components of the carousel, including both carousel controls and slides, has either role region or role group. The most appropriate role for the carousel container depends on the information architecture of the page. See the landmark regions guidance to determine whether the carousel warrants being designated as a landmark region.
- The carousel container has the aria-roledescription property set to `carousel`.
- If the carousel has a visible label, its accessible label is provided by the property aria-labelledby on the carousel container set to the ID of the element containing the visible label. Otherwise, an accessible label is provided by the

property aria-label set on the carousel container. Note that since the `aria-roledescription` is set to "carousel", the label does not contain the word "carousel".

- The rotation control, next slide control, and previous slide control are either native button elements (recommended) or implement the button pattern.
- The rotation control has an accessible label provided by either its inner text or aria-label. The label changes to match the action the button will perform, e.g., "Stop slide rotation" or "Start slide rotation". A label that changes when the button is activated clearly communicates both that slide content can change automatically and when it is doing so. Note that since the label changes, the rotation control does not have any states, e.g., `aria-pressed`, specified.
- Each slide container has role group with the property aria-roledescription set to `slide`.
- Each slide has an accessible name:
  - If a slide has a visible label, its accessible label is provided by the property aria-labelledby on the slide container set to the ID of the element containing the visible label.
  - Otherwise, an accessible label is provided by the property aria-label set on the slide container.
  - If unique names that identify the slide content are not available, a number and set size can serve as a meaningful alternative, e.g., "3 of 10". Note: Normally, including set position and size information in an accessible name is not appropriate. An exception is helpful in this implementation because group elements do not support aria-setsize or aria-posinset. The tabbed carousel implementation pattern does not have this limitation.
  - Note that since the `aria-roledescription` is set to "slide", the label does not contain the word "slide."
- Optionally, an element wrapping the set of slide elements has aria-atomic set to `false` and aria-live set to:
  - `off`: if the carousel is automatically rotating.
  - `polite`: if the carousel is **NOT** automatically rotating.

Tabbed Carousel Elements §

The structure of a tabbed carousel is the same as a basic carousel except that:

- Each slide container has role tabpanel in lieu of `group`, and it does not have the `aria-roledescription` property.
- It has slide picker controls implemented using the tabs pattern where:
  - Each control is a `tab` element, so activating a tab displays the slide associated with that tab.
  - The accessible name of each `tab` indicates which slide it will display by including the name or number of the slide, e.g., "Slide 3". Slide names are preferable if each slide has a unique name.
  - The set of controls is grouped in a `tablist` element with an accessible name provided by the value of aria-label that identifies the purpose of the tabs, e.g., "Choose slide to display."
  - The `tab`, `tablist`, and `tabpanel` implement the properties specified in the tabs pattern.

Grouped Carousel Elements §

A grouped carousel has the same structure as a basic carousel, but it also includes slide picker controls where:

- The set of slide picker controls is contained in an element with role group.
- The group containing the picker controls has an accessible label provided by the value of aria-label that identifies the purpose of the controls, e.g., "Choose slide to display."

- Each picker control is a native button element (recommended) or implements the button pattern.
- The accessible name of each picker button matches the name of the slide it displays. One technique for accomplishing this is to set aria-labelledby to a value that references the slide `group` element.
- The picker button representing the currently displayed slide has the property aria-disabled set to `true`. Note: `aria-disabled` is preferable to the HTML `disabled` attribute because this is a circumstance where screen reader users benefit from the disabled button being included in the page `Tab` sequence.

3.7 Checkbox §

WAI-ARIA supports two types of checkbox widgets:

1. Dual-state: The most common type of checkbox, it allows the user to toggle between two choices -- checked and not checked.
2. Tri-state: This type of checkbox supports an additional third state known as partially checked.

One common use of a tri-state checkbox can be found in software installers where a single tri-state checkbox is used to represent and control the state of an entire group of install options. And, each option in the group can be individually turned on or off with a dual state checkbox.

- If all options in the group are checked, the overall state is represented by the tri-state checkbox displaying as checked.
- If some of the options in the group are checked, the overall state is represented with the tri-state checkbox displaying as partially checked.
- If none of the options in the group are checked, the overall state of the group is represented with the tri-state checkbox displaying as not checked.

The user can use the tri-state checkbox to change all options in the group with a single action:

- Checking the overall checkbox checks all options in the group.
- Unchecking the overall checkbox will uncheck all options in the group.
- And, In some implementations, the system may remember which options were checked the last time the overall status was partially checked. If this feature is provided, activating the overall checkbox a third time recreates that partially checked state where only some options in the group are checked.

Examples §

- Simple Two-State Checkbox Example: Demonstrates a simple 2-state checkbox.
- Tri-State Checkbox Example: Demonstrates how to make a widget that uses the `mixed` value for `aria-checked` and group collection of checkboxes with a field set.

Keyboard Interaction §

When the checkbox has focus, pressing the `Space` key changes the state of the checkbox.

WAI-ARIA Roles, States, and Properties §

- The checkbox has role checkbox.
- The checkbox has an accessible label provided by one of the following:
  - Visible text content contained within the element with role `checkbox`.
  - A visible label referenced by the value of aria-labelledby set on the element with role `checkbox`.
  - aria-label set on the element with role `checkbox`.
- When checked, the checkbox element has state aria-checked set to `true`.
- When not checked, it has state aria-checked set to `false`.
- When partially checked, it has state aria-checked set to `mixed`.
- If a set of checkboxes is presented as a logical group with a visible label, the checkboxes are included in an element with role group that has the property aria-labelledby set to the ID of the element containing the label.
- If the presentation includes additional descriptive static text relevant to a checkbox or checkbox group, the checkbox or checkbox group has the property aria-describedby set to the ID of the element containing the description.

3.8 Combo Box §

A combobox is a widget made up of the combination of two distinct elements: 1) a single-line textbox, and 2) an associated pop-up element for helping users set the value of the textbox. The popup may be a listbox, grid, tree, or dialog. Many implementations also include a third optional element -- a graphical button adjacent to the textbox, indicating the availability of the popup. Activating the button displays the popup if suggestions are available.

The popup is hidden by default, and the conditions that trigger its display are specific to each implementation. Some possible popup display conditions include:

- It is displayed only if a certain number of characters are typed in the textbox and those characters match some portion of one of the suggested values.
- It is displayed as soon as the textbox is focused, even if the textbox is empty.
- It is displayed when the `Down Arrow` key is pressed or the “show” button is activated, possibly with a dependency on the content of the textbox.
- It is displayed if the value of the textbox is altered in a way that creates one or more partial matches to a suggested value.

Combobox widgets are useful for setting the value of a single-line textbox in one of two types of scenarios:

1. The value for the textbox must be chosen from a predefined set of allowed values, e.g., a location field must contain a valid location name. Note that the listbox and menu button patterns are also useful in this scenario; differences between combobox and alternative patterns are described below.
2. The textbox may contain any arbitrary value, but it is advantageous to suggest possible values to the user, e.g., a search field may suggest similar or previous searches to save the user time.

The nature of the suggested values and the way the suggestions are presented is called the autocomplete behavior. Comboboxes can have one of four forms of autocomplete:

1. **No autocomplete:** When the popup is triggered, the suggested values it contains are the same regardless of the characters typed in the textbox. For example, the popup suggests a set of recently entered values, and the suggestions do not change as the user types.

2. **List autocomplete with manual selection:** When the popup is triggered, it presents suggested values that complete or logically correspond to the characters typed in the textbox. The character string the user has typed will become the value of the textbox unless the user selects a value in the popup.
3. **List autocomplete with automatic selection:** When the popup is triggered, it presents suggested values that complete or logically correspond to the characters typed in the textbox, and the first suggestion is automatically highlighted as selected. The automatically selected suggestion becomes the value of the textbox when the combobox loses focus unless the user chooses a different suggestion or changes the character string in the textbox.
4. **List with inline autocomplete:** This is the same as list with automatic selection with one additional feature. The portion of the selected suggestion that has not been typed by the user, a completion string, appears inline after the input cursor in the textbox. The inline completion string is visually highlighted and has a selected state.

With any form of list autocomplete, the popup may appear and disappear as the user types. For example, if the user types a two character string that triggers five suggestions to be displayed but then types a third character that forms a string that does not have any matching suggestions, the popup may close and, if present, the inline completion string disappears.

When constructing a widget that is both visually compact and enables users to choose one value from a set of discrete values, often either a listbox or menu button is simpler to implement and use. One feature of combobox that distinguishes it from both listbox and menu button is that the value of the combobox is presented in an edit field. Thus, the combobox gives users one function that both listbox and menu button lack, namely the ability to select some or all of the value for copying to the clipboard. One feature that distinguishes both combobox and menu button widgets from listbox widgets is their ability to provide an undo mechanism. In many implementations, users can navigate the set of allowed values in a combobox or menu and then decide to revert to the value the widget had before navigating by pressing `escape`. In contrast, navigating a listbox immediately changes its value, and escape does not provide an undo mechanism.

> NOTE
>
> The options for a combobox to popup a grid, tree, or dialog were introduced in ARIA 1.1. Changes made in the ARIA 1.1 specification also add support for a code pattern that enables assistive technologies to present the textbox and popup as separately perceivable elements. both ARIA 1.0 and 1.1 patterns are described in the following sections. While using the ARIA 1.1 pattern is recommended as soon as assistive technology support is sufficient, there are no plans to deprecate the ARIA 1.0 pattern.

Examples §

- Examples of ARIA 1.1 Combobox with Listbox Popup: Comboboxes that demonstrate the various forms of autocomplete behavior using a listbox popup and use the ARIA 1.1 implementation pattern.
- Example of ARIA 1.1 Combobox with Grid Popup: A combobox that presents suggestions in a grid, enabling users to navigate descriptive information about each suggestion.
- ARIA 1.0 Combobox with Both List and Inline Autocomplete: A combobox that demonstrates the autocomplete behavior known as “list with inline autocomplete” and uses the ARIA 1.0 implementation pattern.
- ARIA 1.0 Combobox with List Autocomplete: A combobox that demonstrates the autocomplete behavior known as “list with manual selection” and uses the ARIA 1.0 implementation pattern.
- ARIA 1.0 Combobox Without Autocomplete: A combo box that demonstrates the behavior associated with `aria-autocomplete=none` and uses the ARIA 1.0 implementation pattern.

Keyboard Interaction §

- `Tab`: The textbox is in the page `Tab` sequence.
- Note: The popup indicator icon or button (if present), the popup, and the popup descendants are excluded from the page `Tab` sequence.

Textbox Keyboard Interaction §

When focus is in the textbox:

- `Down Arrow`: If the popup is available, moves focus into the popup:
  - If the autocomplete behavior automatically selected a suggestion before `Down Arrow` was pressed, focus is placed on the suggestion following the automatically selected suggestion.
  - Otherwise, places focus on the first focusable element in the popup.
- `Up Arrow` (Optional): If the popup is available, places focus on the last focusable element in the popup.
- `Escape`: Dismisses the popup if it is visible. Optionally, clears the textbox.
- `Enter`: If an autocomplete suggestion is automatically selected, accepts the suggestion either by placing the input cursor at the end of the accepted value in the textbox or by performing a default action on the value. For example, in a messaging application, the default action may be to add the accepted value to a list of message recipients and then clear the textbox so the user can add another recipient.
- Printable Characters: Type characters in the textbox. Note that some implementations may regard certain characters as invalid and prevent their input.
- Standard single line text editing keys appropriate for the device platform (see note below).
- `Alt + Down Arrow` (Optional): If the popup is available but not displayed, displays the popup without moving focus.
- `Alt + Up Arrow` (Optional): If the popup is displayed:
  1. If the popup contains focus, returns focus to the textbox.
  2. Closes the popup.

> NOTE
>
> Standard single line text editing keys appropriate for the device platform:

1. include keys for input, cursor movement, selection, and text manipulation.
2. Standard key assignments for editing functions depend on the device operating system.
3. The most robust approach for providing text editing functions is to rely on browsers, which supply them for HTML inputs with type text and for elements with the `contenteditable` HTML attribute.
4. **IMPORTANT:** Be sure that JavaScript does not interfere with browser-provided text editing functions by capturing key events for the keys used to perform them.

Listbox Popup Keyboard Interaction §

When focus is in a listbox popup:

- `Enter`: Accepts the focused option in the listbox by closing the popup and placing the accepted value in the textbox with the input cursor at the end of the value.

- `Escape`: Closes the popup and returns focus to the textbox. Optionally, clears the contents of the textbox.
- `Right Arrow`: Returns focus to the textbox without closing the popup and moves the input cursor one character to the right. If the input cursor is on the right-most character, the cursor does not move.
- `Left Arrow`: Returns focus to the textbox without closing the popup and moves the input cursor one character to the left. If the input cursor is on the left-most character, the cursor does not move.
- Any printable character: Returns the focus to the textbox without closing the popup and types the character.
- `Backspace` (Optional): Returns focus to the textbox and deletes the character prior to the cursor.
- `Delete` (Optional): Returns focus to the textbox, removes the selected state if a suggestion was selected, and removes the inline autocomplete string if present.
- `Down Arrow`: Moves focus to and selects the next option. If focus is on the last option, either returns focus to the textbox or does nothing.
- `Up Arrow`: Moves focus to and selects the previous option. If focus is on the first option, either returns focus to the textbox or does nothing.
- `Home` (Optional): Either moves focus to and selects the first option or returns focus to the textbox and places the cursor on the first character.
- `End` (Optional): Either moves focus to the last option or returns focus to the textbox and places the cursor after the last character.

NOTE

1. DOM Focus is maintained on the combobox textbox and the assistive technology focus is moved within the listbox using `aria-activedescendant` as described in Managing Focus in Composites Using aria-activedescendant.
2. Selection follows focus in the listbox; the listbox allows only one suggested value to be selected at a time for the textbox value.

Grid Popup Keyboard Interaction §

In a grid popup, each suggested value may be represented by either a single cell or an entire row. See notes below for how this aspect of grid design effects the keyboard interaction design and the way that selection moves in response to focus movements.

- `Enter`: Accepts the currently selected suggested value by closing the popup and placing the selected value in the textbox with the input cursor at the end of the value.
- `Escape`: Closes the popup and returns focus to the textbox. Optionally, clears the contents of the textbox.
- Any printable character: Returns the focus to the textbox without closing the popup and types the character.
- `Backspace` (Optional): Returns focus to the textbox and deletes the character prior to the cursor.
- `Delete` (Optional): Returns focus to the textbox, removes the selected state if a suggestion was selected, and removes the inline autocomplete string if present.
- `Right Arrow`: Moves focus one cell to the right. Optionally, if focus is on the right-most cell in the row, focus may move to the first cell in the following row. If focus is on the last cell in the grid, either does nothing or returns focus to the textbox.

- `Left Arrow`: Moves focus one cell to the left. Optionally, if focus is on the left-most cell in the row, focus may move to the last cell in the previous row. If focus is on the first cell in the grid, either does nothing or returns focus to the textbox.
- `Down Arrow`: Moves focus one cell down. If focus is in the last row of the grid, either does nothing or returns focus to the textbox.
- `Up Arrow`: Moves focus one cell up. If focus is in the first row of the grid, either does nothing or returns focus to the textbox.
- `Page Down` (Optional): Moves focus down an author-determined number of rows, typically scrolling so the bottom row in the currently visible set of rows becomes one of the first visible rows. If focus is in the last row of the grid, focus does not move.
- `Page Up` (Optional): Moves focus up an author-determined number of rows, typically scrolling so the top row in the currently visible set of rows becomes one of the last visible rows. If focus is in the first row of the grid, focus does not move.
- `Home` (Optional): **Either:**
  - Moves focus to the first cell in the row that contains focus. Or, if the grid has fewer than three cells per row or multiple suggested values per row, focus may move to the first cell in the grid.
  - Returns focus to the textbox and places the cursor on the first character.
- `End` (Optional): **Either:**
  - Moves focus to the last cell in the row that contains focus. Or, if the grid has fewer than three cells per row or multiple suggested values per row, focus may move to the last cell in the grid.
  - Returns focus to the textbox and places the cursor after the last character.
- `Control + Home` (optional): moves focus to the first row.
- `Control + End` (Optional): moves focus to the last row.

NOTE

1. DOM Focus is maintained on the combobox textbox and the assistive technology focus is moved within the grid using `aria-activedescendant` as described in Managing Focus in Composites Using aria-activedescendant.
2. The grid allows only one suggested value to be selected at a time for the textbox value.
3. In a grid popup, each suggested value may be represented by either a single cell or an entire row. This aspect of design effects focus and selection movement:
   1. If every cell contains a different suggested value:
      - Selection follows focus so that the cell containing focus is selected.
      - Horizontal arrow key navigation typically wraps from one row to another.
      - Vertical arrow key navigation typically wraps from one column to another.
   2. If all cells in a row contain information about the same suggested value:
      - Either the row containing focus is selected or a cell containing a suggested value is selected when any cell in the same row contains focus.
      - Horizontal key navigation may wrap from one row to another.
      - Vertical arrow key navigation **does not** wrap from one column to another.

Tree Popup Keyboard Interaction §

In some implementations of tree popups, some or all parent nodes may serve as suggestion category labels so may not be selectable values. See notes below for how this aspect of the design effects the way selection moves in response to focus movements.

When focus is in a vertically oriented tree popup:

- `Enter`: Accepts the currently selected suggested value by closing the popup and placing the selected value in the textbox with the input cursor at the end of the value.
- `Escape`: Closes the popup and returns focus to the textbox. Optionally, clears the contents of the textbox.
- Any printable character: Returns the focus to the textbox without closing the popup and types the character.
- `Right arrow`:
  - When focus is on a closed node, opens the node; focus and selection do not move.
  - When focus is on a open node, moves focus to the first child node and selects it if it is selectable.
  - When focus is on an end node, does nothing.
- `Left arrow`:
  - When focus is on an open node, closes the node.
  - When focus is on a child node that is also either an end node or a closed node, moves focus to its parent node and selects it if it is selectable.
  - When focus is on a root node that is also either an end node or a closed node, does nothing.
- `Down Arrow`: Moves focus to the next node that is focusable without opening or closing a node and selects it if it is selectable.
- `Up Arrow`: Moves focus to the previous node that is focusable without opening or closing a node and selects it if it is selectable.
- `Home`: Moves focus to the first node in the tree without opening or closing a node and selects it if it is selectable.
- `End`: Moves focus to the last node in the tree that is focusable without opening a node and selects it if it is selectable.

NOTE

1. DOM Focus is maintained on the combobox textbox and the assistive technology focus is moved within the tree using `aria-activedescendant` as described in Managing Focus in Composites Using aria-activedescendant.
2. The tree allows only one suggested value to be selected at a time for the textbox value.
3. In a tree popup, some or all parent nodes may not be selectable values; they may serve as category labels for suggested values. If focus moves to a node that is not a selectable value, either:
   - The previously selected node, if any, remains selected until focus moves to a node that is selectable.
   - There is no selected value.
   - In either case, focus is visually distinct from selection so users can readily see if a value is selected or not.
4. If the nodes in a tree are arranged horizontally:
   1. `Down Arrow` performs as `Right Arrow` is described above, and vice versa.
   2. `Up Arrow` performs as `Left Arrow` is described above, and vice versa.

Dialog Popup Keyboard Interaction §

When focus is in a dialog popup:

- There are two ways to close the popup and return focus to the textbox:
  1. Perform an action in the dialog, such as activate a button, that specifies a value for the textbox.
  2. Cancel out of the dialog, e.g., press `Escape` or activate the cancel button in the dialog. Canceling either returns focus to the text box without changing the textbox value or returns focus to the textbox and clears the textbox.
- The dialog implements the keyboard interaction defined in the modal dialog pattern.

> NOTE
>
> Unlike other combobox popups, dialogs do not support `aria-activedescendant` so DOM focus moves into the dialog from the textbox.

WAI-ARIA Roles, States, and Properties §

The role, state, and property guidance where the ARIA 1.1 and ARIA 1.0 patterns differ is listed first. The subsequent guidance applies to both patterns.

- In a combobox implementing the ARIA 1.1 pattern:
  - The element that serves as the combobox container has role combobox.
  - The element with role `combobox` contains or owns a textbox element that has either role textbox or role searchbox.
  - When the combobox popup is visible, the combobox element contains or owns an element that has role listbox, tree, grid, or dialog.
  - If the combobox popup has a role other than `listbox`, the element with role `combobox` has aria-haspopup set to a value that corresponds to the popup type. That is, `aria-haspopup` is set to `grid`, `tree`, or `dialog`. Note that elements with role `combobox` have an implicit `aria-haspopup` value of `listbox`.
  - When the combobox popup is visible, the textbox element has aria-controls set to a value that refers to the combobox popup element.
- In a combobox implementing the ARIA 1.0 pattern:
  - The element that serves as the textbox has role combobox.
  - When the combobox popup is visible, the element with role combobox has aria-owns set to a value that refers to an element with role listbox.
  - the element with role `combobox` has a value for aria-haspopup of `listbox`. Note that elements with role `combobox` have an implicit `aria-haspopup` value of `listbox`.
- The textbox element has a value for aria-multiline of `false`. Note that the default value of `aria-multiline` is `false`.
- When the combobox popup is not visible, the element with role `combobox` has aria-expanded set to `false`. When the popup element is visible, `aria-expanded` is set to `true`. Note that elements with role `combobox` have a default value for `aria-expanded` of `false`.
- When a combobox receives focus, DOM focus is placed on the textbox element.
- When a descendant of a listbox, grid, or tree popup is focused, DOM focus remains on the textbox and the textbox has aria-activedescendant set to a value that refers to the focused element within the popup.

- In a combobox with a listbox, grid, or tree popup, when a suggested value is visually indicated as the currently selected value, the `option`, `gridcell`, `row`, or `treeitem` containing that value has aria-selected set to `true`.
- If the combobox has a visible label, the element with role combobox has aria-labelledby set to a value that refers to the labelling element. Otherwise, the combobox element has a label provided by aria-label.
- The textbox element has aria-autocomplete set to a value that corresponds to its autocomplete behavior:
  - `none`: When the popup is displayed, the suggested values it contains are the same regardless of the characters typed in the textbox.
  - `list`: When the popup is triggered, it presents suggested values that complete or logically correspond to the characters typed in the textbox.
  - `both`: When the popup is triggered, it presents suggested values that complete or logically correspond to the characters typed in the textbox. In addition, the portion of the selected suggestion that has not been typed by the user, known as the "completion string", appears inline after the input cursor in the textbox. The inline completion string is visually highlighted and has a selected state.

NOTE

1. When referring to the roles, states, and properties documentation for the below list of patterns used for popups, keep in mind that a combobox is a single-select widget where selection always follows focus in the popup.
2. The roles, states, and properties for popup elements are defined in their respective design patterns:
   - Listbox Roles, States, and Properties
   - Grid Roles, States, and Properties
   - Tree Roles, States, and Properties
   - Dialog Roles, States, and Properties

3.9 Dialog (Modal) §

A dialog is a window overlaid on either the primary window or another dialog window. Windows under a modal dialog are inert. That is, users cannot interact with content outside an active dialog window. Inert content outside an active dialog is typically visually obscured or dimmed so it is difficult to discern, and in some implementations, attempts to interact with the inert content cause the dialog to close.

Like non-modal dialogs, modal dialogs contain their tab sequence. That is, `Tab` and `Shift + Tab` do not move focus outside the dialog. However, unlike most non-modal dialogs, modal dialogs do not provide means for moving keyboard focus outside the dialog window without closing the dialog.

The alertdialog role is a special-case dialog role designed specifically for dialogs that divert users' attention to a brief, important message. Its usage is described in the alert dialog design pattern.

Examples §

- Modal Dialog Example: Demonstrates multiple layers of modal dialogs with both small and large amounts of content.
- Date Picker Dialog Example: Demonstrates a dialog containing a calendar grid for choosing a date.

Keyboard Interaction §

In the following description, the term “tabbable element” refers to any element with a `tabindex` value of zero or greater. Note that values greater than 0 are strongly discouraged.

- When a dialog opens, focus moves to an element inside the dialog. See notes below regarding initial focus placement.
- `Tab`:
  - Moves focus to the next tabbable element inside the dialog.
  - If focus is on the last tabbable element inside the dialog, moves focus to the first tabbable element inside the dialog.
- `Shift + Tab`:
  - Moves focus to the previous tabbable element inside the dialog.
  - If focus is on the first tabbable element inside the dialog, moves focus to the last tabbable element inside the dialog.
- `Escape`: Closes the dialog.

> NOTE
>
> 1. When a dialog opens, focus placement depends on the nature and size of the content.
>   - In all circumstances, focus moves to an element contained in the dialog.
>   - Unless a condition where doing otherwise is advisable, focus is initially set on the first focusable element.
>   - If content is large enough that focusing the first interactive element could cause the beginning of content to scroll out of view, it is advisable to add `tabindex="-1"` to a static element at the top of the dialog, such as the dialog title or first paragraph, and initially focus that element.
>   - If a dialog contains the final step in a process that is not easily reversible, such as deleting data or completing a financial transaction, it may be advisable to set focus on the least destructive action, especially if undoing the action is difficult or impossible. The Alert Dialog Pattern is often employed in such circumstances.
>   - If a dialog is limited to interactions that either provide additional information or continue processing, it may be advisable to set focus to the element that is likely to be most frequently used, such as an “OK” or “Continue” button.
> 2. When a dialog closes, focus returns to the element that invoked the dialog unless either:
>   - The invoking element no longer exists. Then, focus is set on another element that provides logical work flow.
>   - The work flow design includes the following conditions that can occasionally make focusing a different element a more logical choice:
>   1. It is very unlikely users need to immediately re-invoke the dialog.
>   2. The task completed in the dialog is directly related to a subsequent step in the work flow.
>
>   For example, a grid has an associated toolbar with a button for adding rows. the Add Rows button opens a dialog that prompts for the number of rows. After the dialog closes, focus is placed in the first cell of the first new row.
> 3. It is strongly recommended that the tab sequence of all dialogs include a visible element with role `button` that closes the dialog, such as a close icon or cancel button.

WAI-ARIA Roles, States, and Properties §

- The element that serves as the dialog container has a role of dialog.
- All elements required to operate the dialog are descendants of the element that has role `dialog`.
- The dialog container element has aria-modal set to `true`.
- The dialog has either:
  - A value set for the aria-labelledby property that refers to a visible dialog title.
  - A label specified by aria-label.
- Optionally, the aria-describedby property is set on the element with the `dialog` role to indicate which element or elements in the dialog contain content that describes the primary purpose or message of the dialog. Specifying descriptive elements enables screen readers to announce the description along with the dialog title and initially focused element when the dialog opens.

NOTE

- Because marking a dialog modal by setting aria-modal to `true` can prevent users of some assistive technologies from perceiving content outside the dialog, users of those technologies will experience severe negative ramifications if a dialog is marked modal but does not behave as a modal for other users. So, mark a dialog modal **only when both:**
  1. Application code prevents all users from interacting in any way with content outside of it.
  2. Visual styling obscures the content outside of it.
- The `aria-modal` property introduced by ARIA 1.1 replaces aria-hidden for informing assistive technologies that content outside a dialog is inert. However, in legacy dialog implementations where `aria-hidden` is used to make content outside a dialog inert for assistive technology users, it is important that:
  1. `aria-hidden` is set to `true` on each element containing a portion of the inert layer.
  2. The dialog element is not a descendant of any element that has `aria-hidden` set to `true`.

3.10 Disclosure (Show/Hide) §

A disclosure is a button that controls visibility of a section of content. When the controlled content is hidden, it is often styled as a typical push button with a right-pointing arrow or triangle to hint that activating the button will display additional content. When the content is visible, the arrow or triangle typically points down.

Examples §

- Disclosure (Show/Hide) of Image Description
- Disclosure (Show/Hide) of Answers to Frequently Asked Questions
- Disclosure (Show/Hide) for Navigation Menus

Keyboard Interaction §

When the disclosure control has focus:

- `Enter`: activates the disclosure control and toggles the visibility of the disclosure content.
- `Space`: activates the disclosure control and toggles the visibility of the disclosure content.

WAI-ARIA Roles, States, and Properties §

- The element that shows and hides the content has role button.
- When the content is visible, the element with role `button` has aria-expanded set to `true`. When the content area is hidden, it is set to `false`.
- Optionally, the element with role `button` has a value specified for aria-controls that refers to the element that contains all the content that is shown or hidden.

3.11 Feed §

A feed is a section of a page that automatically loads new sections of content as the user scrolls. The sections of content in a feed are presented in article elements. So, a feed can be thought of as a dynamic list of articles that often appears to scroll infinitely.

The feature that most distinguishes feed from other ARIA patterns that support loading data as users scroll, e.g., a grid, is that a feed is a structure, not a widget. Consequently, assistive technologies with a reading mode, such as screen readers, default to reading mode when interacting with feed content. However, unlike most other WAI-ARIA structures, a feed establishes an interoperability contract between the web page and assistive technologies. The contract governs scroll interactions so that assistive technology users can read articles, jump forward and backward by article, and reliably trigger new articles to load while in reading mode.

For example, a product page on a shopping site may have a related products section that displays five products at a time. As the user scrolls, more products are requested and loaded into the DOM. While a static design might include a next button for loading five more products, a dynamic implementation that automatically loads more data as the user scrolls simplifies the user experience and reduces the inertia associated with viewing more than the first five product suggestions. But, unfortunately when web pages load content dynamically based on scroll events, it can cause usability and interoperability difficulties for users of assistive technologies.

The feed pattern enables reliable assistive technology reading mode interaction by establishing the following interoperability agreement between the web page and assistive technologies:

1. In the context of a feed, the web page code is responsible for:
    - Appropriate visual scrolling of the content based on which article contains DOM focus.
    - Loading or removing feed articles based on which article contains DOM focus.
2. In the context of a feed, assistive technologies with a reading mode are responsible for:
    - Indicating which article contains the reading cursor by ensuring the article element or one of its descendants has DOM focus.
    - providing reading mode keys that move DOM focus to the next and previous articles.
    - Providing reading mode keys for moving the reading cursor and DOM focus past the end and before the start of the feed.

Thus, implementing the feed pattern allows a screen reader to reliably read and trigger the loading of feed content while staying in its reading mode.

Another feature of the feed pattern is its ability to facilitate skim reading for assistive technology users. Web page authors may provide both an accessible name and description for each article. By identifying the elements inside of an article that provide the title and the primary content, assistive technologies can provide functions that enable users to jump from article to article and efficiently discern which articles may be worthy of more attention.

Example §

Example Implementation of Feed Pattern

Keyboard Interaction §

The feed pattern is not based on a desktop GUI widget so the `feed` role is not associated with any well-established keyboard conventions. Supporting the following, or a similar, interface is recommended.

When focus is inside the feed:

- `Page Down`: Move focus to next article.
- `Page Up`: Move focus to previous article.
- `Control + End`: Move focus to the first focusable element after the feed.
- `Control + Home`: Move focus to the first focusable element before the feed.

NOTE

1. Due to the lack of convention, providing easily discoverable keyboard interface documentation is especially important.
2. In some cases, a feed may contain a nested feed. For example, an article in a social media feed may contain a feed of comments on that article. To navigate the nested feed, users first move focus inside the nested feed. Options for supporting nested feed navigation include:
   - Users move focus into the nested feed from the content of the containing article with `Tab`. This may be slow if the article contains a significant number of links, buttons, or other widgets.
   - Provide a key for moving focus from the elements in the containing article to the first item in the nested feed, e.g., `Alt + Page Down`.
   - To continue reading the outer feed, `Control + End` moves focus to the next article in the outer feed.
3. In the rare circumstance that a feed article contains a widget that uses the above suggested keys, the feed navigation key will operate the contained widget, and the user needs to move focus to an element that does not utilize the feed navigation keys in order to navigate the feed.

WAI-ARIA Roles, States, and Properties §

- The element that contains the set of feed articles has role feed.
- If the feed has a visible label, the `feed` element has aria-labelledby referring to the element containing the title. Otherwise, the `feed` element has a label specified with aria-label.

- Each unit of content in a feed is contained in an element with role article. All content inside the feed is contained in an `article` element.
- Each `article` element has aria-labelledby referring to elements inside the article that can serve as a distinguishing label.
- It is optional but strongly recommended for each `article` element to have aria-describedby referring to one or more elements inside the article that serve as the primary content of the article.
- Each `article` element has aria-posinset set to a value that represents its position in the feed.
- Each `article` element has aria-setsize set to a value that represents either the total number of articles that have been loaded or the total number in the feed, depending on which value is deemed more helpful to users. If the total number in the feed is undetermined, it can be represented by a `aria-setsize` value of `-1`.
- When `article` elements are being added to or removed from the `feed` container, and if the operation requires multiple DOM operations, the `feed` element has aria-busy set to `true` during the update operation. Note that it is extremely important that `aria-busy` is set to `false` when the operation is complete or the changes may not become visible to some assistive technology users.

3.12 Grids : Interactive Tabular Data and Layout Containers §

A grid widget is a container that enables users to navigate the information or interactive elements it contains using directional navigation keys, such as arrow keys, `Home`, and `End`. As a generic container widget that offers flexible keyboard navigation, it can serve a wide variety of needs. It can be used for purposes as simple as grouping a collection of checkboxes or navigation links or as complex as creating a full-featured spreadsheet application. While the words "row" and "column" are used in the names of WAI-ARIA attributes and by assistive technologies when describing and presenting the logical structure of elements with the `grid` role, using the `grid` role on an element does not necessarily imply that its visual presentation is tabular.

When presenting content that is tabular, consider the following factors when choosing between implementing this `grid` pattern or the table pattern.

- A `grid` is a composite widget so it:
  - Always contains multiple focusable elements.
  - Only one of the focusable elements contained by the grid is included in the page tab sequence.
  - Requires the author to provide code that manages focus movement inside it.
- All focusable elements contained in a table are included in the page tab sequence.

Uses of the `grid` pattern broadly fall into two categories: presenting tabular information (data grids) and grouping other widgets (layout grids). Even though both data grids and layout grids employ the same ARIA roles, states, and properties, differences in their content and purpose surface factors that are important to consider in keyboard interaction design. To address these factors, the following two sections describe separate keyboard interaction patterns for data and layout grids.

Examples §

- Layout Grid Examples: Three example implementations of grids that are used to lay out widgets, including a collection of navigation links, a message recipients list, and a set of search results.
- Data Grid Examples: Three example implementations of grid that include features relevant to presenting tabular information, such as content editing, sort, and column hiding.

- Advanced Data Grid Example: Example of a grid with behaviors and features similar to a typical spreadsheet, including cell and row selection.

Data Grids For Presenting Tabular Information §

A `grid` can be used to present tabular information that has column titles, row titles, or both. The `grid` pattern is particularly useful if the tabular information is editable or interactive. For example, when data elements are links to more information, rather than presenting them in a static table and including the links in the tab sequence, implementing the `grid` pattern provides users with intuitive and efficient keyboard navigation of the grid contents as well as a shorter tab sequence for the page. A `grid` may also offer functions, such as cell content editing, selection, cut, copy, and paste.

In a grid, every cell contains a focusable element or is itself focusable, regardless of whether the cell content is editable or interactive. There is one exception: if column or row header cells do not provide functions, such as sort or filter, they do not need to be focusable. One reason it is important for all cells to be able to receive or contain keyboard focus is that screen readers will typically be in their application reading mode, rather than their document reading mode, when users are interacting with the grid. While in application mode, a screen reader user hears only focusable elements and content that labels focusable elements. So, screen reader users may unknowingly overlook elements contained in a grid that are either not focusable or not used to label a column or row.

Keyboard Interaction For Data Grids §

The following keys provide grid navigation by moving focus among cells of the grid. Implementations of grid make these key commands available when an element in the grid has received focus, e.g., after a user has moved focus to the grid with `Tab`.

- `Right Arrow`: Moves focus one cell to the right. If focus is on the right-most cell in the row, focus does not move.
- `Left Arrow`: Moves focus one cell to the left. If focus is on the left-most cell in the row, focus does not move.
- `Down Arrow`: Moves focus one cell down. If focus is on the bottom cell in the column, focus does not move.
- `Up Arrow`: Moves focus one cell Up. If focus is on the top cell in the column, focus does not move.
- `Page Down`: Moves focus down an author-determined number of rows, typically scrolling so the bottom row in the currently visible set of rows becomes one of the first visible rows. If focus is in the last row of the grid, focus does not move.
- `Page Up`: Moves focus up an author-determined number of rows, typically scrolling so the top row in the currently visible set of rows becomes one of the last visible rows. If focus is in the first row of the grid, focus does not move.
- `Home`: moves focus to the first cell in the row that contains focus.
- `End`: moves focus to the last cell in the row that contains focus.
- `Control + Home`: moves focus to the first cell in the first row.
- `Control + End`: moves focus to the last cell in the last row.

NOTE

- When the above grid navigation keys move focus, whether the focus is set on an element inside the cell or the grid cell depends on cell content. See Whether to Focus on a Cell or an Element Inside It.
- While navigation keys, such as arrow keys, are moving focus from cell to cell, they are not available to do something like operate a combobox or move an editing caret inside of a cell. If this functionality is needed, see Editing and Navigating Inside a Cell.
- If navigation functions can dynamically add more rows or columns to the DOM, key events that move focus to the beginning or end of the grid, such as `control + End`, may move focus to the last row in the DOM rather than the last available row in the back-end data.

If a grid supports selection of cells, rows, or columns, the following keys are commonly used for these functions.

- `Control + Space`: selects the column that contains the focus.
- `Shift + Space`: Selects the row that contains the focus. If the grid includes a column with checkboxes for selecting rows, this key can serve as a shortcut for checking the box when focus is not on the checkbox.
- `Control + A`: Selects all cells.
- `Shift + Right Arrow`: Extends selection one cell to the right.
- `Shift + Left Arrow`: Extends selection one cell to the left.
- `Shift + Down Arrow`: Extends selection one cell down.
- `Shift + Up Arrow`: Extends selection one cell Up.

NOTE

See § 6.8 Key Assignment Conventions for Common Functions for cut, copy, and paste key assignments.

Layout Grids for Grouping Widgets §

The `grid` pattern can be used to group a set of interactive elements, such as links, buttons, or checkboxes. Since only one element in the entire grid is included in the tab sequence, grouping with a grid can dramatically reduce the number of tab stops on a page. This is especially valuable if scrolling through a list of elements dynamically loads more of those elements from a large data set, such as in a continuous list of suggested products on a shopping site. If elements in a list like this were in the tab sequence, keyboard users are effectively trapped in the list. If any elements in the group also have associated elements that appear on hover, the `grid` pattern is also useful for providing keyboard access to those contextual elements of the user interface.

Unlike grids used to present data, A `grid` used for layout does not necessarily have header cells for labelling rows or columns and might contain only a single row or a single column. Even if it has multiple rows and columns, it may present a single, logically homogenous set of elements. For example, a list of recipients for a message may be a grid where each cell contains a link that represents a recipient. The grid may initially have a single row but then wrap into multiple rows as recipients are added. In such circumstances, grid navigation keys may also wrap so the user can read the list from beginning to end by pressing either `Right Arrow` or `Down Arrow`. While This type of focus movement wrapping can be very helpful in a layout grid, it would be disorienting if used in a data grid, especially for users of assistive technologies.

Because arrow keys are used to move focus inside of a `grid`, a `grid` is both easier to build and use if the components it contains do not require the arrow keys to operate. If a cell contains an element like a listbox, then an extra key command to focus and activate the listbox is needed as well as a command for restoring the grid navigation functionality. Approaches to supporting this need are described in the section on Editing and Navigating Inside a Cell.

Keyboard Interaction For Layout Grids §

The following keys provide grid navigation by moving focus among cells of the grid. Implementations of grid make these key commands available when an element in the grid has received focus, e.g., after a user has moved focus to the grid with `Tab`.

- `Right Arrow`: Moves focus one cell to the right. Optionally, if focus is on the right-most cell in the row, focus may move to the first cell in the following row. If focus is on the last cell in the grid, focus does not move.
- `Left Arrow`: Moves focus one cell to the left. Optionally, if focus is on the left-most cell in the row, focus may move to the last cell in the previous row. If focus is on the first cell in the grid, focus does not move.
- `Down Arrow`: Moves focus one cell down. Optionally, if focus is on the bottom cell in the column, focus may move to the top cell in the following column. If focus is on the last cell in the grid, focus does not move.
- `Up Arrow`: Moves focus one cell up. Optionally, if focus is on the top cell in the column, focus may move to the bottom cell in the previous column. If focus is on the first cell in the grid, focus does not move.
- `Page Down` (Optional): Moves focus down an author-determined number of rows, typically scrolling so the bottom row in the currently visible set of rows becomes one of the first visible rows. If focus is in the last row of the grid, focus does not move.
- `Page Up` (Optional): Moves focus up an author-determined number of rows, typically scrolling so the top row in the currently visible set of rows becomes one of the last visible rows. If focus is in the first row of the grid, focus does not move.
- `Home`: moves focus to the first cell in the row that contains focus. Optionally, if the grid has a single column or fewer than three cells per row, focus may instead move to the first cell in the grid.
- `End`: moves focus to the last cell in the row that contains focus. Optionally, if the grid has a single column or fewer than three cells per row, focus may instead move to the last cell in the grid.
- `Control + Home` (optional): moves focus to the first cell in the first row.
- `Control + End` (Optional): moves focus to the last cell in the last row.

NOTE

- When the above grid navigation keys move focus, whether the focus is set on an element inside the cell or the grid cell depends on cell content. See Whether to Focus on a Cell or an Element Inside It.
- While navigation keys, such as arrow keys, are moving focus from cell to cell, they are not available to do something like operate a combobox or move an editing caret inside of a cell. If this functionality is needed, see Editing and Navigating Inside a Cell.
- If navigation functions can dynamically add more rows or columns to the DOM, key events that move focus to the beginning or end of the grid, such as `control + End`, may move focus to the last row in the DOM rather than the last available row in the back-end data.

It would be unusual for a layout grid to provide functions that require cell selection. If it did, though, the following keys are commonly used for these functions.

- `Control + Space`: selects the column that contains the focus.
- `Shift + Space`: Selects the row that contains the focus. If the grid includes a column with checkboxes for selecting rows, this key can serve as a shortcut for checking the box when focus is not on the checkbox.
- `Control + A`: Selects all cells.
- `Shift + Right Arrow`: Extends selection one cell to the right.
- `Shift + Left Arrow`: Extends selection one cell to the left.
- `Shift + Down Arrow`: Extends selection one cell down.
- `Shift + Up Arrow`: Extends selection one cell Up.

> NOTE
>
> See § 6.8 Key Assignment Conventions for Common Functions for cut, copy, and paste key assignments.

Keyboard Interaction - Setting Focus and Navigating Inside Cells §

This section describes two important aspects of keyboard interaction design shared by both data and layout grid patterns:

1. Choosing whether a cell or an element inside a cell receives focus in response to grid navigation key events.
2. Enabling grid navigation keys to be used to interact with elements inside of a cell.

Whether to Focus on a Cell Or an Element Inside It §

For assistive technology users, the quality of experience when navigating a grid heavily depends on both what a cell contains and on where keyboard focus is set. For example, if a cell contains a button and a grid navigation key places focus on the cell instead of the button, screen readers announce the button label but do not tell users a button is present.

There are two optimal cell design and focus behavior combinations:

1. A cell contains one widget whose operation does not require arrow keys and grid navigation keys set focus on that widget. Examples of such widgets include link, button, menubutton, toggle button, radio button (not radio group), switch, and checkbox.
2. A cell contains text or a single graphic and grid navigation keys set focus on the cell.

While any combination of widgets, text, and graphics may be included in a single cell, grids that do not follow one of these two cell design and focus movement patterns add complexity for authors or users or both. The reference implementations included in the example section below demonstrate some strategies for making other cell designs as accessible as possible, but the most widely accessible experiences are likely to come by applying the above two patterns.

Editing and Navigating Inside a Cell §

While navigation keys, such as arrow keys, are moving focus from cell to cell, they are not available to perform actions like operate a combobox or move an editing caret inside of a cell. The user may need keys that are used for grid navigation to

operate elements inside a cell if a cell contains:

1. Editable content.
2. Multiple widgets.
3. A widget that utilizes arrow keys in its interaction model, such as a radio group or slider.

Following are common keyboard conventions for disabling and restoring grid navigation functions.

- `Enter`: Disables grid navigation and:
  - If the cell contains editable content, places focus in an input field, such as a textbox. If the input is a single-line text field, a subsequent press of `Enter` may either restore grid navigation functions or move focus to an input field in a neighboring cell.
  - If the cell contains one or more widgets, places focus on the first widget.
- `F2`:
  - If the cell contains editable content, places focus in an input field, such as a textbox. A subsequent press of `F2` restores grid navigation functions.
  - If the cell contains one or more widgets, places focus on the first widget. A subsequent press of `F2` restores grid navigation functions.
- Alphanumeric keys: If the cell contains editable content, places focus in an input field, such as a textbox.

When grid navigation is disabled, conventional changes to navigation behaviors include:

- `Escape`: restores grid navigation. If content was being edited, it may also undo edits.
- `Right Arrow` or `Down Arrow`: If the cell contains multiple widgets, moves focus to the next widget inside the cell, optionally wrapping to the first widget if focus is on the last widget. Otherwise, passes the key event to the focused widget.
- `Left Arrow` or `Up Arrow`: If the cell contains multiple widgets, moves focus to the previous widget inside the cell, optionally wrapping to the first widget if focus is on the last widget. Otherwise, passes the key event to the focused widget.
- `Tab`: moves focus to the next widget in the grid. Optionally, the focus movement may wrap inside a single cell or within the grid itself.
- `Shift + Tab`: moves focus to the previous widget in the grid. Optionally, the focus movement may wrap inside a single cell or within the grid itself.

WAI-ARIA Roles, States, and Properties §

- The grid container has role grid.
- Each row container has role row and is either a DOM descendant of or owned by the `grid` element or an element with role rowgroup.
- Each cell is either a DOM descendant of or owned by a `row` element and has one of the following roles:
  - columnheader if the cell contains a title or header information for the column.
  - rowheader if the cell contains title or header information for the row.
  - gridcell if the cell does not contain column or row header information.

- If there is an element in the user interface that serves as a label for the grid, aria-labelledby is set on the grid element with a value that refers to the labelling element. Otherwise, a label is specified for the grid element using aria-label.
- If the grid has a caption or description, aria-describedby is set on the grid element with a value referring to the element containing the description.
- If the grid provides sort functions, aria-sort is set to an appropriate value on the header cell element for the sorted column or row as described in the section on grid and table properties.
- If the grid supports selection, when a cell or row is selected, the selected element has aria-selected set `true`. If the grid supports column selection and a column is selected, all cells in the column have `aria-selected` set to `true`.
- If the grid provides content editing functionality and contains cells that may have edit capabilities disabled in certain conditions, aria-readonly may be set `true` on cells where editing is disabled. If edit functions are disabled for all cells, `aria-readonly` may be set `true` on the grid element. Grids that do not provide editing functions do not include the `aria-readonly` attribute on any of their elements.
- If there are conditions where some rows or columns are hidden or not present in the DOM, e.g., data is dynamically loaded when scrolling or the grid provides functions for hiding rows or columns, the following properties are applied as described in the section on grid and table properties.
  - aria-colcount or aria-rowcount is set to the total number of columns or rows, respectively.
  - aria-colindex or aria-rowindex is set to the position of a cell within a row or column, respectively.
- If the grid includes cells that span multiple rows or multiple columns, and if the `grid` role is NOT applied to an HTML `table` element, then aria-rowspan or aria-colspan is applied as described in grid and table properties.

> NOTE
>
> - If the element with the `grid` role is an HTML `table` element, then it is not necessary to use ARIA roles for rows and cells because the HTML elements have implied ARIA semantics. For example, an HTML `<TR>` has an implied ARIA role of `row`. A `grid` built from an HTML `table` that includes cells that span multiple rows or columns must use HTML `rowspan` and `colspan` and must not use `aria-rowspan` or `aria-colspan`.
> - If rows or cells are included in a grid via aria-owns, they will be presented to assistive technologies after the DOM descendants of the `grid` element unless the DOM descendants are also included in the `aria-owns` attribute.

3.13 Link §

A link widget provides an interactive reference to a resource. The target resource can be either external or local, i.e., either outside or within the current page or application.

> NOTE
>
> Authors are strongly encouraged to use a native host language link element, such as an HTML `<A>` element with an `href` attribute. As with other WAI-ARIA widget roles, applying the link role to an element will not cause browsers to enhance the element with standard link behaviors, such as navigation to the link target or context menu actions. When using the `link` role, providing these features of the element is the author's responsibility.

Examples §

Link Examples: Link widgets constructed from HTML `span` and `img` elements.

Keyboard Interaction §

- `Enter`: Executes the link and moves focus to the link target.
- `Shift + F10` (Optional): Opens a context menu for the link.

WAI-ARIA Roles, States, and Properties §

The element containing the link text or graphic has role of link.

3.14 Listbox §

A listbox widget presents a list of options and allows a user to select one or more of them. A listbox that allows a single option to be chosen is a single-select listbox; one that allows multiple options to be selected is a multi-select listbox.

When screen readers present a listbox, they may render the name, state, and position of each option in the list. The name of an option is a string calculated by the browser, typically from the content of the option element. As a flat string, the name does not contain any semantic information. Thus, if an option contains a semantic element, such as a heading, screen reader users will not have access to the semantics. In addition, the interaction model conveyed by the listbox role to assistive technologies does not support interacting with elements inside of an option. Because of these traits of the listbox widget, it does not provide an accessible way to present a list of interactive elements, such as links, buttons, or checkboxes. To present a list of interactive elements, see the grid pattern.

Avoiding very long option names facilitates understandability and perceivability for screen reader users. The entire name of an option is spoken as a single unit of speech when the option is read. When too much information is spoken as the result of a single key press, it is difficult to understand. Long names inhibit perception by increasing the impact of interrupted speech because users typically have to re-read the entire option. And, if the user does not understand what is spoken, reading the name by character, word, or phrase may be a difficult operation for many screen reader users in the context of a listbox widget.

Sets of options where each option name starts with the same word or phrase can also significantly degrade usability for keyboard and screen reader users. Scrolling through the list to find a specific option becomes inordinately time consuming for a screen reader user who must listen to that word or phrase repeated before hearing what is unique about each option. For example, if a listbox for choosing a city were to contain options where each city name were preceded by a country name, and if many cities were listed for each country, a screen reader user would have to listen to the country name before hearing each city name. In such a scenario, it would be better to have 2 list boxes, one for country and one for city.

Examples §

- Scrollable Listbox Example: Single-select listbox that scrolls to reveal more options, similar to HTML `select` with `size` attribute greater than one.
- Collapsible Dropdown Listbox Example: Single-select collapsible listbox that expands when activated, similar to HTML `select` with the attribute `size="1"`.
- Example Listboxes with Rearrangeable Options: Examples of both single-select and multi-select listboxes with accompanying toolbars where options can be added, moved, and removed.

Keyboard Interaction §

For a vertically oriented listbox:

- When a single-select listbox receives focus:
  - If none of the options are selected before the listbox receives focus, the first option receives focus. Optionally, the first option may be automatically selected.
  - If an option is selected before the listbox receives focus, focus is set on the selected option.
- When a multi-select listbox receives focus:
  - If none of the options are selected before the listbox receives focus, focus is set on the first option and there is no automatic change in the selection state.
  - If one or more options are selected before the listbox receives focus, focus is set on the first option in the list that is selected.
- `Down Arrow`: Moves focus to the next option. Optionally, in a single-select listbox, selection may also move with focus.
- `Up Arrow`: Moves focus to the previous option. Optionally, in a single-select listbox, selection may also move with focus.
- `Home` (Optional): Moves focus to first option. Optionally, in a single-select listbox, selection may also move with focus. Supporting this key is strongly recommended for lists with more than five options.
- `End` (Optional): Moves focus to last option. Optionally, in a single-select listbox, selection may also move with focus. Supporting this key is strongly recommended for lists with more than five options.
- Type-ahead is recommended for all listboxes, especially those with more than seven options:
  - Type a character: focus moves to the next item with a name that starts with the typed character.
  - Type multiple characters in rapid succession: focus moves to the next item with a name that starts with the string of characters typed.
- **Multiple Selection**: Authors may implement either of two interaction models to support multiple selection: a recommended model that does not require the user to hold a modifier key, such as `Shift` or `Control`, while navigating the list or an alternative model that does require modifier keys to be held while navigating in order to avoid losing selection states.
  - Recommended selection model -- holding modifier keys is not necessary:
    - `Space`: changes the selection state of the focused option.
    - `Shift + Down Arrow` (Optional): Moves focus to and toggles the selected state of the next option.
    - `Shift + Up Arrow` (Optional): Moves focus to and toggles the selected state of the previous option.
    - `Shift + Space` (Optional): Selects contiguous items from the most recently selected item to the focused item.
    - `Control + Shift + Home` (Optional): Selects the focused option and all options up to the first option. Optionally, moves focus to the first option.
    - `Control + Shift + End` (Optional): Selects the focused option and all options down to the last option. Optionally, moves focus to the last option.
    - `Control + A` (Optional): Selects all options in the list. Optionally, if all options are selected, it may also unselect all options.
  - Alternative selection model -- moving focus without holding a `Shift` or `Control` modifier unselects all selected nodes except the focused node:
    - `Shift + Down Arrow`: Moves focus to and toggles the selection state of the next option.

- `Shift + Up Arrow`: Moves focus to and toggles the selection state of the previous option.
- `Control + Down Arrow`: Moves focus to the next option without changing its selection state.
- `Control + Up Arrow`: Moves focus to the previous option without changing its selection state.
- `Control + Space` Changes the selection state of the focused option.
- `Shift + Space` (Optional): Selects contiguous items from the most recently selected item to the focused item.
- `Control + Shift + Home` (Optional): Selects the focused option and all options up to the first option. Optionally, moves focus to the first option.
- `Control + Shift + End` (Optional): Selects the focused option and all options down to the last option. Optionally, moves focus to the last option.
- `Control + A` (Optional): Selects all options in the list. Optionally, if all options are selected, it may also unselect all options.

NOTE

1. DOM focus (the active element) is functionally distinct from the selected state. For more details, see this description of differences between focus and selection.
2. The `listbox` role supports the aria-activedescendant property, which provides an alternative to moving DOM focus among `option` elements when implementing keyboard navigation. For details, see Managing Focus in Composites Using aria-activedescendant.
3. In a single-select listbox, moving focus may optionally unselect the previously selected option and select the newly focused option. This model of selection is known as "selection follows focus". Having selection follow focus can be very helpful in some circumstances and can severely degrade accessibility in others. For additional guidance, see Deciding When to Make Selection Automatically Follow Focus.
4. If selecting or unselecting all options is an important function, implementing separate controls for these actions, such as buttons for "Select All" and "Unselect All", significantly improves accessibility.
5. If the options in a listbox are arranged horizontally:
    1. `Down Arrow` performs as `Right Arrow` is described above, and vice versa.
    2. `Up Arrow` performs as `Left Arrow` is described above, and vice versa.

WAI-ARIA Roles, States, and Properties §

- An element that contains or owns all the listbox options has role listbox.
- Each option in the listbox has role option and is a DOM descendant of the element with role `listbox` or is referenced by an aria-owns property on the listbox element.
- If the listbox is not part of another widget, then it has a visible label referenced by aria-labelledby on the element with role `listbox`.
- In a single-select listbox, the selected option has aria-selected set to `true`.
- if the listbox supports multiple selection:
    - The element with role `listbox` has aria-multiselectable set to `true`.
    - All selected options have aria-selected set to `true`.
    - All options that are not selected have aria-selected set to `false`.

- If the complete set of available options is not present in the DOM due to dynamic loading as the user scrolls, their aria-setsize and aria-posinset attributes are set appropriately.
- If options are arranged horizontally, the element with role `listbox` has aria-orientation set to `horizontal`. The default value of `aria-orientation` for `listbox` is `vertical`.

3.15 Menu or Menu bar §

A menu is a widget that offers a list of choices to the user, such as a set of actions or functions. A menu is usually opened, or made visible, by activating a menu button, choosing an item in a menu that opens a sub menu, or by invoking a command, such as `Shift + F10` in Windows, that opens a context specific menu. When a user activates a choice in a menu, the menu usually closes unless the choice opened a submenu.

A menu that is visually persistent is a menubar. A menubar is typically horizontal and is often used to create a menu bar similar to those found near the top of the window in many desktop applications, offering the user quick access to a consistent set of commands.

A common convention for indicating that a menu item launches a dialog box is to append "…" (ellipsis) to the menu item label, e.g., "Save as …".

Examples §

- Navigation Menubar Example: Demonstrates a menubar that provides site navigation.
- Editor Menubar Example: Demonstrates menu radios and menu checkboxes in submenus of a menubar that provides text formatting commands for a text field.

Keyboard Interaction §

The following description of keyboard behaviors assumes:

1. A horizontal `menubar` containing several `menuitem` elements.
2. All items in the `menubar` have child submenus that contain multiple vertically arranged items.
3. Some of the `menuitem` elements in the submenus have child submenus with items that are also vertically arranged.

When reading the following descriptions, also keep in mind that:

1. Focusable elements, which may have role `menuitem`, `menuitemradio`, or `menuitemcheckbox`, are referred to as items.
2. If a behavior applies to only certain types of items, e.g., `menuitem` elements, the specific role name is used.
3. Submenus, also known as pop-up menus, are elements with role `menu`.
4. Except where noted, menus opened from a menubutton behave the same as menus opened from a menubar.

- When a `menu` opens, or when a `menubar` receives focus, keyboard focus is placed on the first item. All items are focusable as described in § 6.6 Keyboard Navigation Inside Components.
- `Enter`:
  - When focus is on a `menuitem` that has a submenu, opens the submenu and places focus on its first item.
  - Otherwise, activates the item and closes the menu.

- `Space`:
  - (Optional): When focus is on a `menuitemcheckbox`, changes the state without closing the menu.
  - (Optional): When focus is on a `menuitemradio` that is not checked, without closing the menu, checks the focused `menuitemradio` and unchecks any other checked `menuitemradio` element in the same group.
  - (Optional): When focus is on a `menuitem` that has a submenu, opens the submenu and places focus on its first item.
  - (Optional): When focus is on a `menuitem` that does not have a submenu, activates the `menuitem` and closes the menu.
- `Down Arrow`:
  - When focus is on a `menuitem` in a `menubar`, opens its submenu and places focus on the first item in the submenu.
  - When focus is in a `menu`, moves focus to the next item, optionally wrapping from the last to the first.
- `Up Arrow`:
  - When focus is in a `menu`, moves focus to the previous item, optionally wrapping from the first to the last.
  - (Optional): When focus is on a `menuitem` in a `menubar`, opens its submenu and places focus on the last item in the submenu.
- `Right Arrow`:
  - When focus is in a `menubar`, moves focus to the next item, optionally wrapping from the last to the first.
  - When focus is in a `menu` and on a `menuitem` that has a submenu, opens the submenu and places focus on its first item.
  - When focus is in a `menu` and on an item that does not have a submenu, performs the following 3 actions:
    1. Closes the submenu and any parent menus.
    2. Moves focus to the next `menuitem` in the `menubar`.
    3. Either: (Recommended) opens the submenu of that `menuitem` without moving focus into the submenu, or opens the submenu of that `menuitem` and places focus on the first item in the submenu.

    Note that if the `menubar` were not present, e.g., the menus were opened from a menubutton, `Right Arrow` would not do anything when focus is on an item that does not have a submenu.
- `Left Arrow`:
  - When focus is in a `menubar`, moves focus to the previous item, optionally wrapping from the first to the last.
  - When focus is in a submenu of an item in a `menu`, closes the submenu and returns focus to the parent `menuitem`.
  - When focus is in a submenu of an item in a `menubar`, performs the following 3 actions:
    1. Closes the submenu.
    2. Moves focus to the previous `menuitem` in the `menubar`.
    3. Either: (Recommended) opens the submenu of that `menuitem` without moving focus into the submenu, or opens the submenu of that `menuitem` and places focus on the first item in the submenu.
- `Home`: If arrow key wrapping is not supported, moves focus to the first item in the current `menu` or `menubar`.
- `End`: If arrow key wrapping is not supported, moves focus to the last item in the current `menu` or `menubar`.
- Any key that corresponds to a printable character (Optional): Move focus to the next menu item in the current menu whose label begins with that printable character.
- `Escape`: Close the menu that contains focus and return focus to the element or context, e.g., menu button or parent `menuitem`, from which the menu was opened.

- `Tab`: Moves focus to the next element in the tab sequence, and if the item that had focus is not in a `menubar`, closes its `menu` and all open parent `menu` containers.
- `Shift + Tab`: Moves focus to the previous element in the tab sequence, and if the item that had focus is not in a `menubar`, closes its `menu` and all open parent `menu` containers.

> NOTE
>
> 1. Disabled menu items are focusable but cannot be activated.
> 2. A separator in a menu is not focusable or interactive.
> 3. If a menu is opened or a menubar receives focus as a result of a context action, `Escape` or `Enter` may return focus to the invoking context. For example, a rich text editor may have a menubar that receives focus when a shortcut key, e.g., `alt + F10`, is pressed while editing. In this case, pressing `Escape` or activating a command from the menu may return focus to the editor.
> 4. Although it is recommended that authors avoid doing so, some implementations of navigation menubars may have `menuitem` elements that both perform a function and open a submenu. In such implementations, `enter` and `Space` perform a navigation function, e.g., load new content, while `Down Arrow`, in a horizontal menubar, opens the submenu associated with that same `menuitem`.
> 5. When items in a `menubar` are arranged vertically and items in `menu` containers are arranged horizontally:
>   1. `Down Arrow` performs as `Right Arrow` is described above, and vice versa.
>   2. `Up Arrow` performs as `Left Arrow` is described above, and vice versa.

WAI-ARIA Roles, States, and Properties §

- A menu is a container of items that represent choices. The element serving as the menu has a role of either menu or menubar.
- The items contained in a menu are child elements of the containing menu or menubar and have any of the following roles:
  - menuitem
  - menuitemcheckbox
  - menuitemradio
- If activating a menuitem opens a submenu, the menuitem is known as a parent menuitem. A submenu's `menu` element is:
  - Contained inside the same `menu` element as its parent `menuitem`.
  - Is the sibling element immediately following its parent `menuitem`.
- A parent menuitem has aria-haspopup set to either `menu` or `true`.
- A parent menuitem has aria-expanded set to `false` when its child menu is not visible and set to `true` when the child menu is visible.
- One of the following approaches is used to enable scripts to move focus among items in a menu as described in § 6.6 Keyboard Navigation Inside Components:
  - The menu container has `tabindex` set to `-1` or `0` and aria-activedescendant set to the ID of the focused item.
  - Each item in the menu has `tabindex` set to `-1`, except in a menubar, where the first item has `tabindex` set to `0`.

- When a menuitemcheckbox or menuitemradio is checked, aria-checked is set to `true`.
- When a menu item is disabled, aria-disabled is set to `true`.
- Items in a menu may be divided into groups by placing an element with a role of separator between groups. For example, this technique should be used when a menu contains a set of menuitemradio items.
- All separators should have aria-orientation consistent with the separator's orientation.
- If a menubar has a visible label, the element with role `menubar` has aria-labelledby set to a value that refers to the labelling element. Otherwise, the menubar element has a label provided by aria-label.
- If a menubar is vertically oriented, it has aria-orientation set to `vertical`. The default value of `aria-orientation` for a menubar is `horizontal`.
- An element with role `menu` either has:
  - aria-labelledby set to a value that refers to the menuitem or button that controls its display.
  - A label provided by aria-label.
- If a menu is horizontally oriented, it has aria-orientation set to `horizontal`. The default value of `aria-orientation` for a menu is `vertical`.

> NOTE
>
> If aria-owns is set on the menu container to include elements that are not DOM children of the container, those elements will appear in the reading order in the sequence they are referenced and after any items that are DOM children. Scripts that manage focus need to ensure the visual focus order matches this assistive technology reading order.

3.16 Menu Button §

A menu button is a button that opens a menu. It is often styled as a typical push button with a downward pointing arrow or triangle to hint that activating the button will display a menu.

Examples §

- Navigation Menu Button: A menu button made from an HTML `a` element that opens a menu of items that behave as links.
- Action Menu Button Example Using element.focus(): A menu button made from an HTML `button` element that opens a menu of actions or commands where focus in the menu is managed using `element.focus()`.
- Action Menu Button Example Using aria-activedescendant: A button that opens a menu of actions or commands where focus in the menu is managed using aria-activedescendant.

Keyboard Interaction §

- With focus on the button:
  - `Enter`: opens the menu and places focus on the first menu item.
  - `Space`: Opens the menu and places focus on the first menu item.
  - (Optional) `Down Arrow`: opens the menu and moves focus to the first menu item.
  - (Optional) `Up Arrow`: opens the menu and moves focus to the last menu item.

- The keyboard behaviors needed after the menu is open are described in § 3.15 Menu or Menu bar.

WAI-ARIA Roles, States, and Properties §

- The element that opens the menu has role button.
- The element with role `button` has aria-haspopup set to either `menu` or `true`.
- When the menu is displayed, the element with role `button` has aria-expanded set to `true`. When the menu is hidden, it is recommended that `aria-expanded` is not present. If `aria-expanded` is specified when the menu is hidden, it is set to `false`.
- The element that contains the menu items displayed by activating the button has role menu.
- Optionally, the element with role `button` has a value specified for aria-controls that refers to the element with role `menu`.
- Additional roles, states, and properties needed for the menu element are described in § 3.15 Menu or Menu bar.

3.17 Radio Group §

A radio group is a set of checkable buttons, known as radio buttons, where no more than one of the buttons can be checked at a time. Some implementations may initialize the set with all buttons in the unchecked state in order to force the user to check one of the buttons before moving past a certain point in the workflow.

Examples §

- Radio Group Example Using Roving tabindex
- Radio Group Example Using aria-activedescendant

Keyboard Interaction §

For Radio Groups Not Contained in a Toolbar §

This section describes the keyboard interaction implemented for most radio groups. For the special case of a radio group nested inside a toolbar, use the keyboard interaction described in the following section.

- `Tab` and `Shift + Tab`: Move focus into and out of the radio group. When focus moves into a radio group :
  - If a radio button is checked, focus is set on the checked button.
  - If none of the radio buttons are checked, focus is set on the first radio button in the group.
- `Space`: checks the focused radio button if it is not already checked.
- `Right Arrow` and `Down Arrow`: move focus to the next radio button in the group, uncheck the previously focused button, and check the newly focused button. If focus is on the last button, focus moves to the first button.
- `Left Arrow` and `Up Arrow`: move focus to the previous radio button in the group, uncheck the previously focused button, and check the newly focused button. If focus is on the first button, focus moves to the last button.

NOTE

The initial focus behavior described above differs slightly from the behavior provided by some browsers for native HTML radio groups. In some browsers, if none of the radio buttons are selected, moving focus into the radio group with `Shift+Tab` will place focus on the last radio button instead of the first radio button.

For Radio Group Contained in a Toolbar §

Because arrow keys are used to navigate among elements of a toolbar and the `Tab` key moves focus in and out of a toolbar, when a radio group is nested inside a toolbar, the keyboard interaction of the radio group is slightly different from that of a radio group that is not inside of a toolbar. For instance, users need to be able to navigate among all toolbar elements, including the radio buttons, without changing which radio button is checked. So, when navigating through a radio group in a toolbar with arrow keys, the button that is checked does not change. The keyboard interaction of a radio group nested in a toolbar is as follows.

- `Space`: If the focused radio button is not checked, unchecks the currently checked radio button and checks the focused radio button. Otherwise, does nothing.
- `Enter` (optional): If the focused radio button is not checked, unchecks the currently checked radio button and checks the focused radio button. Otherwise, does nothing.
- `Right Arrow`:
  - When focus is on a radio button and that radio button is **not** the last radio button in the radio group, moves focus to the next radio button.
  - When focus is on the last radio button in the radio group and that radio button is **not** the last element in the toolbar, moves focus to the next element in the toolbar.
  - When focus is on the last radio button in the radio group and that radio button is also the last element in the toolbar, moves focus to the first element in the toolbar.
- `Left Arrow`:
  - When focus is on a radio button and that radio button is **not** the first radio button in the radio group, moves focus to the previous radio button.
  - When focus is on the first radio button in the radio group and that radio button is **not** the first element in the toolbar, moves focus to the previous element in the toolbar.
  - When focus is on the first radio button in the radio group and that radio button is also the first element in the toolbar, moves focus to the last element in the toolbar.
- `Down Arrow` (optional): Moves focus to the next radio button in the radio group. If focus is on the last radio button in the radio group, moves focus to the first radio button in the group.
- `Up Arrow` (optional): Moves focus to the previous radio button in the radio group. If focus is on the first radio button in the radio group, moves focus to the last radio button in the group.

NOTE

Radio buttons in a toolbar are frequently styled in a manner that appears more like toggle buttons. For an example, See the Simple Editor Toolbar Example

WAI-ARIA Roles, States, and Properties §

- The radio buttons are contained in or owned by an element with role radiogroup.
- Each radio button element has role radio.
- If a radio button is checked, the `radio` element has aria-checked set to `true`. If it is not checked, it has aria-checked set to `false`.
- Each `radio` element is labelled by its content, has a visible label referenced by aria-labelledby, or has a label specified with aria-label.
- The `radiogroup` element has a visible label referenced by aria-labelledby or has a label specified with aria-label.
- If elements providing additional information about either the radio group or each radio button are present, those elements are referenced by the `radiogroup` element or `radio` elements with the aria-describedby property.

3.18 Slider §

A slider is an input where the user selects a value from within a given range. Sliders typically have a slider thumb that can be moved along a bar or track to change the value of the slider.

Examples §

- Horizontal Slider Examples: Demonstrates using three horizontally aligned sliders to make a color picker.
- Slider Examples with aria-orientation and aria-valuetext: Three thermostat control sliders that demonstrate using aria-orientation and aria-valuetext.

Keyboard Interaction §

- `Right Arrow`: Increase the value of the slider by one step.
- `Up Arrow`: Increase the value of the slider by one step.
- `Left Arrow`: Decrease the value of the slider by one step.
- `Down Arrow`: Decrease the value of the slider by one step.
- `Home`: Set the slider to the first allowed value in its range.
- `End`: Set the slider to the last allowed value in its range.
- `Page Up` (Optional): Increment the slider by an amount larger than the step change made by `Up Arrow`.
- `Page Down` (Optional): Decrement the slider by an amount larger than the step change made by `Down Arrow`.

> NOTE
>
> 1. Focus is placed on the slider (the visual object that the mouse user would move, also known as the thumb.
> 2. In some circumstances, reversing the direction of the value change for the keys specified above, e.g., having `Up Arrow` decrease the value, could create a more intuitive experience.

WAI-ARIA Roles, States, and Properties §

- The element serving as the focusable slider control has role slider.
- The slider element has the aria-valuenow property set to a decimal value representing the current value of the slider.
- The slider element has the aria-valuemin property set to a decimal value representing the minimum allowed value of the slider.
- The slider element has the aria-valuemax property set to a decimal value representing the maximum allowed value of the slider.
- If the value of `aria-valuenow` is not user-friendly, e.g., the day of the week is represented by a number, the aria-valuetext property is set to a string that makes the slider value understandable, e.g., "Monday".
- If the slider has a visible label, it is referenced by aria-labelledby on the slider element. Otherwise, the slider element has a label provided by aria-label.
- If the slider is vertically oriented, it has aria-orientation set to `vertical`. The default value of `aria-orientation` for a slider is `horizontal`.

3.19 Slider (Multi-Thumb) §

A multi-thumb slider is a slider with two or more thumbs that each set a value in a group of related values. For example, in a product search, a two-thumb slider could be used to enable users to set the minimum and maximum price limits for the search. In many two-thumb sliders, the thumbs are not allowed to pass one another, such as when the slider sets the minimum and maximum values for a range. For example, in a price range selector, the maximum value of the thumb that sets the lower end of the range is limited by the current value of the thumb that sets the upper end of the range. Conversely, the minimum value of the upper end thumb is limited by the current value of the lower end thumb. However, in some multi-thumb sliders, each thumb sets a value that does not depend on the other thumb values.

Example §

Multi-Thumb Slider Examples: Demonstrates two-thumb sliders for picking price ranges for an airline flight and hotel reservation.

Keyboard Interaction §

- Each thumb is in the page tab sequence and has the same keyboard interaction as a single-thumb slider.
- The tab order remains constant regardless of thumb value and visual position within the slider. For example, if the value of a thumb changes such that it moves past one of the other thumbs, the tab order does not change.

WAI-ARIA Roles, States, and Properties §

- Each element serving as a focusable slider thumb has role slider.
- Each slider element has the aria-valuenow property set to a decimal value representing the current value of the slider.
- Each slider element has the aria-valuemin property set to a decimal value representing the minimum allowed value of the slider.

- Each slider element has the aria-valuemax property set to a decimal value representing the maximum allowed value of the slider.
- When the range (e.g. minimum and/or maximum value) of another slider is dependent on the current value of a slider, the values of aria-valuemin or aria-valuemax of the dependent sliders are updated when the value changes.
- If a value of `aria-valuenow` is not user-friendly, e.g., the day of the week is represented by a number, the aria-valuetext property is set to a string that makes the slider value understandable, e.g., "Monday".
- If a slider has a visible label, it is referenced by aria-labelledby on the slider element. Otherwise, the slider element has a label provided by aria-label.
- If a slider is vertically oriented, it has aria-orientation set to `vertical`. The default value of `aria-orientation` for a slider is `horizontal`.

3.20 Spinbutton §

A spinbutton is an input widget that restricts its value to a set or range of discrete values. For example, in a widget that enables users to set an alarm, a spinbutton could allow users to select a number from 0 to 59 for the minute of an hour.

Spinbuttons often have three components, including a text field that displays the current value, an increment button, and a decrement button. The text field is usually the only focusable component because the increment and decrement functions are keyboard accessible via arrow keys. Typically, the text field also allows users to directly edit the value.

If the range is large, a spinbutton may support changing the value in both small and large steps. For instance, in the alarm example, the user may be able to move by 1 minute with `Up Arrow` and `Down Arrow` and by 10 minutes with `Page Up` and `Page Down`.

Example §

Date Picker Spin Button Example: Illustrates a date picker made from thre spin buttons for day, month, and year.

Keyboard Interaction §

- `Up Arrow`: Increases the value.
- `Down Arrow`: Decreases the value.
- `Home`: If the spinbutton has a minimum value, sets the value to its minimum.
- `End`: If the spinbutton has a maximum value, sets the value to its maximum.
- `Page Up` (Optional): Increases the value by a larger step than `Up Arrow`.
- `Page Down` (Optional): Decreases the value by a larger step than `Down Arrow`.
- If the spinbutton text field allows directly editing the value, the following keys are supported:
  - Standard single line text editing keys appropriate for the device platform (see note below).
  - Printable Characters: Type characters in the textbox. Note that many implementations allow only certain characters as part of the value and prevent input of any other characters. For example, an hour-and-minute spinner would allow only integer values from 0 to 59, the colon ':', and the letters 'AM' and 'PM'. Any other character input does not change the contents of the text field nor the value of the spinbutton.

NOTE

1. Focus remains on the text field during operation.
2. Standard single line text editing keys appropriate for the device platform:
    1. include keys for input, cursor movement, selection, and text manipulation.
    2. Standard key assignments for editing functions depend on the device operating system.
    3. The most robust approach for providing text editing functions is to rely on browsers, which supply them for HTML inputs with type text and for elements with the `contenteditable` HTML attribute.
    4. **IMPORTANT:** Be sure that JavaScript does not interfere with browser-provided text editing functions by capturing key events for the keys used to perform them.

WAI-ARIA Roles, States, and Properties §

- The focusable element serving as the spinbutton has role spinbutton. This is typically an element that supports text input.
- The spinbutton element has the aria-valuenow property set to a decimal value representing the current value of the spinbutton.
- The spinbutton element has the aria-valuemin property set to a decimal value representing the minimum allowed value of the spinbutton if it has a known minimum value.
- The spinbutton element has the aria-valuemax property set to a decimal value representing the maximum allowed value of the spinbutton if it has a known maximum value.
- If the value of `aria-valuenow` is not user-friendly, e.g., the day of the week is represented by a number, the aria-valuetext property is set on the spinbutton element to a string that makes the spinbutton value understandable, e.g., "Monday".
- If the spinbutton has a visible label, it is referenced by aria-labelledby on the spinbutton element. Otherwise, the spinbutton element has a label provided by aria-label.
- The spinbutton element has aria-invalid set to `true` if the value is outside the allowed range. Note that most implementations prevent input of invalid values, but in some scenarios, blocking all invalid input may not be practical.

3.21 Table §

Like an HTML `table` element, a WAI-ARIA table is a static tabular structure containing one or more rows that each contain one or more cells; it is not an interactive widget. Thus, its cells are not focusable or selectable. The grid pattern is used to make an interactive widget that has a tabular structure.

However, tables are often used to present a combination of information and interactive widgets. Since a table is not a widget, each widget contained in a table is a separate stop in the page tab sequence. If the number of widgets is large, replacing the table with a grid can dramatically reduce the length of the page tab sequence because a grid is a composite widget that can contain other widgets.

NOTE

As with other WAI-ARIA roles that have a native host language equivalent, authors are strongly encouraged to use a native HTML `table` element whenever possible. This is especially important with role `table` because it is a new feature of WAI-ARIA 1.1. It is thus advisable to test implementations thoroughly with each browser and assistive technology combination that could be used by the target audience.

Examples §

Table Example: ARIA table made using HTML `div` and `span` elements.

Keyboard Interaction §

Not applicable.

WAI-ARIA Roles, States, and Properties §

- The table container has role table.
- Each row container has role row and is either a DOM descendant of or owned by the `table` element or an element with role rowgroup.
- Each cell is either a DOM descendant of or owned by a `row` element and has one of the following roles:
  - columnheader if the cell contains a title or header information for the column.
  - rowheader if the cell contains title or header information for the row.
  - cell if the cell does not contain column or row header information.
- If there is an element in the user interface that serves as a label for the table, aria-labelledby is set on the table element with a value that refers to the labelling element. Otherwise, a label is specified for the table element using aria-label.
- If the table has a caption or description, aria-describedby is set on the table element with a value referring to the element containing the description.
- If the table contains sortable columns or rows, aria-sort is set to an appropriate value on the header cell element for the sorted column or row as described in the section on grid and table properties.
- If there are conditions where some rows or columns are hidden or not present in the DOM, e.g., there are widgets on the page for hiding rows or columns, the following properties are applied as described in the section on grid and table properties.
  - aria-colcount or aria-rowcount is set to the total number of columns or rows, respectively.
  - aria-colindex or aria-rowindex is set to the position of a cell within a row or column, respectively.
- If the table includes cells that span multiple rows or multiple columns, then aria-rowspan or aria-colspan is applied as described in grid and table properties.

NOTE

If rows or cells are included in a table via aria-owns, they will be presented to assistive technologies after the DOM descendants of the `table` element unless the DOM descendants are also included in the `aria-owns` attribute.

3.22 Tabs §

Tabs are a set of layered sections of content, known as tab panels, that display one panel of content at a time. Each tab panel has an associated tab element, that when activated, displays the panel. The list of tab elements is arranged along one edge of the currently displayed panel, most commonly the top edge.

Terms used to describe this design pattern include:

Tabs or Tabbed Interface
: A set of tab elements and their associated tab panels.

Tab List
: A set of tab elements contained in a tablist element.

tab
: An element in the tab list that serves as a label for one of the tab panels and can be activated to display that panel.

tabpanel
: The element that contains the content associated with a tab.

When a tabbed interface is initialized, one tab panel is displayed and its associated tab is styled to indicate that it is active. When the user activates one of the other tab elements, the previously displayed tab panel is hidden, the tab panel associated with the activated tab becomes visible, and the tab is considered "active".

Examples §

- Tabs With Automatic Activation: A tabs widget where tabs are automatically activated and their panel is displayed when they receive focus.
- Tabs With Manual Activation: A tabs widget where users activate a tab and display its panel by pressing `Space` or `Enter`.

Keyboard Interaction §

For the tab list:

- `Tab`: When focus moves into the tab list, places focus on the active `tab` element. When the tab list contains the focus, moves focus to the next element in the page tab sequence outside the tablist, which is typically either the first focusable element inside the tab panel or the tab panel itself.
- When focus is on a tab element in a horizontal tab list:
  - `Left Arrow`: moves focus to the previous tab. If focus is on the first tab, moves focus to the last tab. Optionally, activates the newly focused tab (See note below).
  - `Right Arrow`: Moves focus to the next tab. If focus is on the last tab element, moves focus to the first tab. Optionally, activates the newly focused tab (See note below).
- When focus is on a tab in a tablist with either horizontal or vertical orientation:
  - `Space or Enter`: Activates the tab if it was not activated automatically on focus.
  - `Home` (Optional): Moves focus to the first tab. Optionally, activates the newly focused tab (See note below).
  - `End` (Optional): Moves focus to the last tab. Optionally, activates the newly focused tab (See note below).
  - `Shift + F10`: If the tab has an associated pop-up menu, opens the menu.

- `Delete` (Optional): If deletion is allowed, deletes (closes) the current tab element and its associated tab panel, sets focus on the tab following the tab that was closed, and optionally activates the newly focused tab. If there is not a tab that followed the tab that was deleted, e.g., the deleted tab was the right-most tab in a left-to-right horizontal tab list, sets focus on and optionally activates the tab that preceded the deleted tab. If the application allows all tabs to be deleted, and the user deletes the last remaining tab in the tab list, the application moves focus to another element that provides a logical work flow. As an alternative to `Delete`, or in addition to supporting `Delete`, the delete function is available in a context menu.

NOTE

1. It is recommended that tabs activate automatically when they receive focus as long as their associated tab panels are displayed without noticeable latency. This typically requires tab panel content to be preloaded. Otherwise, automatic activation slows focus movement, which significantly hampers users' ability to navigate efficiently across the tab list. For additional guidance, see § 6.4 Deciding When to Make Selection Automatically Follow Focus.
2. If the tabs in a tab list are arranged vertically:
    1. `Down Arrow` performs as `Right Arrow` is described above.
    2. `Up Arrow` performs as `Left Arrow` is described above.
3. If the tab list is horizontal, it does not listen for `Down Arrow` or `Up Arrow` so those keys can provide their normal browser scrolling functions even when focus is inside the tab list.

WAI-ARIA Roles, States, and Properties §

- The element that serves as the container for the set of tabs has role tablist.
- Each element that serves as a tab has role tab and is contained within the element with role `tablist`.
- Each element that contains the content panel for a `tab` has role tabpanel.
- If the tab list has a visible label, the element with role `tablist` has aria-labelledby set to a value that refers to the labelling element. Otherwise, the `tablist` element has a label provided by aria-label.
- Each element with role `tab` has the property aria-controls referring to its associated `tabpanel` element.
- The active `tab` element has the state aria-selected set to `true` and all other `tab` elements have it set to `false`.
- Each element with role `tabpanel` has the property aria-labelledby referring to its associated `tab` element.
- If a `tab` element has a pop-up menu, it has the property aria-haspopup set to either `menu` or `true`.
- If the `tablist` element is vertically oriented, it has the property aria-orientation set to `vertical`. The default value of `aria-orientation` for a `tablist` element is `horizontal`.

3.23 Toolbar §

A toolbar is a container for grouping a set of controls, such as buttons, menubuttons, or checkboxes.

When a set of controls is visually presented as a group, the `toolbar` role can be used to communicate the presence and purpose of the grouping to screen reader users. Grouping controls into toolbars can also be an effective way of reducing the number of tab stops in the keyboard interface.

To optimize the benefit of toolbar widgets:

- Implement focus management so the keyboard tab sequence includes one stop for the toolbar and arrow keys move focus among the controls in the toolbar.
    - In horizontal toolbars, `Left Arrow` and `Right Arrow` navigate among controls. `Up Arrow` and `Down Arrow` can duplicate `Left Arrow` and `Right Arrow`, respectively, or can be reserved for operating controls, such as spin buttons that require vertical arrow keys to operate.
    - In vertical toolbars, `Up Arrow` and `Down Arrow` navigate among controls. `Left Arrow` and `Right Arrow` can duplicate `Up Arrow` and `Down Arrow`, respectively, or can be reserved for operating controls, such as horizontal sliders that require horizontal arrow keys to operate.
    - In toolbars with multiple rows of controls, `Left Arrow` and `Right Arrow` can provide navigation that wraps from row to row, leaving the option of reserving vertical arrow keys for operating controls.
- Avoid including controls whose operation requires the pair of arrow keys used for toolbar navigation. If unavoidable, include only one such control and make it the last element in the toolbar. For example, in a horizontal toolbar, a textbox could be included as the last element.
- Use toolbar as a grouping element only if the group contains 3 or more controls.

Example §

Toolbar Example: A toolbar that uses roving tabindex to manage focus and contains several types of controls, including toggle buttons, radio buttons, a menu button, a spin button, a checkbox, and a link.

Keyboard Interaction §

- `Tab` and `Shift + Tab`: Move focus into and out of the toolbar. When focus moves into a toolbar:
    - If focus is moving into the toolbar for the first time, focus is set on the first control that is not disabled.
    - If the toolbar has previously contained focus, focus is optionally set on the control that last had focus. Otherwise, it is set on the first control that is not disabled.
- For a horizontal toolbar (the default):
    - `Left Arrow`: Moves focus to the previous control. Optionally, focus movement may wrap from the first element to the last element.
    - `Right Arrow`: Moves focus to the next control. Optionally, focus movement may wrap from the last element to the first element.
- `Home` (Optional): Moves focus to first element.
- `End` (Optional): Moves focus to last element.

NOTE

1. If the items in a toolbar are arranged vertically:
    1. `Down Arrow` performs as `Right Arrow` is described above.
    2. `Up Arrow` performs as `Left Arrow` is described above.
2. Typically, disabled elements are not focusable when navigating with a keyboard. However, in circumstances where discoverability of a function is crucial, it may be helpful if disabled controls are focusable so screen reader users are more likely to be aware of their presence. For additional guidance, see § 6.7 Focusability of disabled controls.
3. In applications where quick access to a toolbar is important, such as accessing an editor's toolbar from its text area, a documented shortcut key for moving focus from the relevant context to its corresponding toolbar is recommended.

WAI-ARIA Roles, States, and Properties §

- The element that serves as the toolbar container has role toolbar.
- If the toolbar has a visible label, it is referenced by aria-labelledby on the toolbar element. Otherwise, the toolbar element has a label provided by aria-label.
- If the controls are arranged vertically, the toolbar element has aria-orientation set to `vertical`. The default orientation is horizontal.

3.24 Tooltip Widget §

NOTE: This design pattern is work in progress; it does not yet have task force consensus. Progress and discussions are captured in issue 128.

A tooltip is a popup that displays information related to an element when the element receives keyboard focus or the mouse hovers over it. It typically appears after a small delay and disappears when `Escape` is pressed or on mouse out.

Tooltip widgets do not receive focus. A hover that contains focusable elements can be made using a non-modal dialog.

Example §

Work to develop a tooltip example is tracked by issue 127.

Keyboard Interaction §

`Escape`: Dismisses the Tooltip.

NOTE

1. Focus stays on the triggering element while the tooltip is displayed.
2. If the tooltip is invoked when the trigger element receives focus, then it is dismissed when it no longer has focus (onBlur). If the tooltip is invoked with mouseIn, then it is dismissed with on mouseOut.

WAI-ARIA Roles, States, and Properties §

- The element that serves as the tooltip container has role tooltip.
- The element that triggers the tooltip references the tooltip element with aria-describedby.

3.25 Tree View §

A tree view widget presents a hierarchical list. Any item in the hierarchy may have child items, and items that have children may be expanded or collapsed to show or hide the children. For example, in a file system navigator that uses a tree view to display folders and files, an item representing a folder can be expanded to reveal the contents of the folder, which may be files, folders, or both.

Terms for understanding tree views include:

Node
An item in a tree.

Root Node
Node at the base of the tree; it may have one or more child nodes but does not have a parent node.

Child Node
Node that has a parent; any node that is not a root node is a child node.

End Node
Node that does not have any child nodes; an end node may be either a root node or a child node.

Parent Node
Node with one or more child nodes. It can be open (expanded) or closed (collapsed).

Open Node
Parent node that is expanded so its child nodes are visible.

Closed Node
Parent node that is collapsed so the child nodes are not visible.

When using a keyboard to navigate a tree, a visual keyboard indicator informs the user which node is focused. If the tree allows the user to choose just one item for an action, then it is known as a single-select tree. In some implementations of single-select tree, the focused item also has a selected state; this is known as selection follows focus. However, in multi-select trees, which enable the user to select more than one item for an action, the selected state is always independent of the focus. For example, in a typical file system navigator, the user can move focus to select any number of files for an action, such as copy or move. It is important that the visual design distinguish between items that are selected and the item that has focus. For more details, see this description of differences between focus and selection and Deciding When to Make Selection Automatically Follow Focus.

Examples §

- File Directory Treeview Example Using Computed Properties: A file selector tree that demonstrates browser support for automatically computing `aria-level`, `aria-posinset` and `aria-setsize` based on DOM structure.
- File Directory Treeview Example Using Declared Properties: A file selector tree that demonstrates how to explicitly define values for `aria-level`, `aria-posinset` and `aria-setsize`.
- Navigation Treeview Example Using Computed Properties: A tree that provides navigation to a set of web pages and demonstrates browser support for automatically computing `aria-level`, `aria-posinset` and `aria-setsize` based on DOM structure.
- Navigation Treeview Example Using Declared Properties: A tree that provides navigation to a set of web pages and demonstrates how to explicitly define values for `aria-level`, `aria-posinset` and `aria-setsize`.

Keyboard Interaction §

For a vertically oriented tree:

- When a single-select tree receives focus:
  - If none of the nodes are selected before the tree receives focus, focus is set on the first node.
  - If a node is selected before the tree receives focus, focus is set on the selected node.
- When a multi-select tree receives focus:
  - If none of the nodes are selected before the tree receives focus, focus is set on the first node.
  - If one or more nodes are selected before the tree receives focus, focus is set on the first selected node.
- `Right arrow`:
  - When focus is on a closed node, opens the node; focus does not move.
  - When focus is on a open node, moves focus to the first child node.
  - When focus is on an end node, does nothing.
- `Left arrow`:
  - When focus is on an open node, closes the node.
  - When focus is on a child node that is also either an end node or a closed node, moves focus to its parent node.
  - When focus is on a root node that is also either an end node or a closed node, does nothing.
- `Down Arrow`: Moves focus to the next node that is focusable without opening or closing a node.
- `Up Arrow`: Moves focus to the previous node that is focusable without opening or closing a node.
- `Home`: Moves focus to the first node in the tree without opening or closing a node.
- `End`: Moves focus to the last node in the tree that is focusable without opening a node.
- `Enter`: activates a node, i.e., performs its default action. For parent nodes, one possible default action is to open or close the node. In single-select trees where selection does not follow focus (see note below), the default action is typically to select the focused node.
- Type-ahead is recommended for all trees, especially for trees with more than 7 root nodes:
  - Type a character: focus moves to the next node with a name that starts with the typed character.
  - Type multiple characters in rapid succession: focus moves to the next node with a name that starts with the string of characters typed.
- `*` (Optional): Expands all siblings that are at the same level as the current node.

- **Selection in multi-select trees:** Authors may implement either of two interaction models to support multiple selection: a recommended model that does not require the user to hold a modifier key, such as `Shift` or `Control`, while navigating the list or an alternative model that does require modifier keys to be held while navigating in order to avoid losing selection states.
  - Recommended selection model -- holding a modifier key while moving focus is not necessary:
    - `Space`: Toggles the selection state of the focused node.
    - `Shift + Down Arrow` (Optional): Moves focus to and toggles the selection state of the next node.
    - `Shift + Up Arrow` (Optional): Moves focus to and toggles the selection state of the previous node.
    - `Shift + Space` (Optional): Selects contiguous nodes from the most recently selected node to the current node.
    - `Control + Shift + Home` (Optional): Selects the node with focus and all nodes up to the first node. Optionally, moves focus to the first node.
    - `Control + Shift + End` (Optional): Selects the node with focus and all nodes down to the last node. Optionally, moves focus to the last node.
    - `Control + A` (Optional): Selects all nodes in the tree. Optionally, if all nodes are selected, it can also unselect all nodes.
  - Alternative selection model -- Moving focus without holding the `Shift` or `Control` modifier unselects all selected nodes except for the focused node:
    - `Shift + Down Arrow`: Moves focus to and toggles the selection state of the next node.
    - `Shift + Up Arrow`: Moves focus to and toggles the selection state of the previous node.
    - `Control + Down Arrow`: Without changing the selection state, moves focus to the next node.
    - `Control + Up Arrow`: Without changing the selection state, moves focus to the previous node.
    - `Control + Space`: Toggles the selection state of the focused node.
    - `Shift + Space` (Optional): Selects contiguous nodes from the most recently selected node to the current node.
    - `Control + Shift + Home` (Optional): Selects the node with focus and all nodes up to the first node. Optionally, moves focus to the first node.
    - `Control + Shift + End` (Optional): Selects the node with focus and all nodes down to the last node. Optionally, moves focus to the last node.
    - `Control + A` (Optional): Selects all nodes in the tree. Optionally, if all nodes are selected, it can also unselect all nodes.

NOTE

1. DOM focus (the active element) is functionally distinct from the selected state. For more details, see this description of differences between focus and selection.
2. The `tree` role supports the aria-activedescendant property, which provides an alternative to moving DOM focus among `treeitem` elements when implementing keyboard navigation. For details, see Managing Focus in Composites Using aria-activedescendant.
3. In a single-select tree, moving focus may optionally unselect the previously selected node and select the newly focused node. This model of selection is known as "selection follows focus". Having selection follow focus can be very helpful in some circumstances and can severely degrade accessibility in others. For additional guidance, see Deciding When to Make Selection Automatically Follow Focus.
4. If selecting or unselecting all nodes is an important function, implementing separate controls for these actions, such as buttons for "Select All" and "Unselect All", significantly improves accessibility.
5. If the nodes in a tree are arranged horizontally:
    1. `Down Arrow` performs as `Right Arrow` is described above, and vice versa.
    2. `Up Arrow` performs as `Left Arrow` is described above, and vice versa.

WAI-ARIA Roles, States, and Properties §

- All tree nodes are contained in or owned by an element with role tree.
- Each element serving as a tree node has role treeitem.
- Each root node is contained in the element with role `tree` or referenced by an aria-owns property set on the `tree` element.
- Each parent node contains or owns an element with role group.
- Each child node is contained in or owned by an element with role group that is contained in or owned by the node that serves as the parent of that child.
- Each element with role `treeitem` that serves as a parent node has aria-expanded set to `false` when the node is in a closed state and set to `true` when the node is in an open state. End nodes do not have the `aria-expanded` attribute because, if they were to have it, they would be incorrectly described to assistive technologies as parent nodes.
- If the tree supports selection of more than one node, the element with role `tree` has aria-multiselectable set to `true`. Otherwise, `aria-multiselectable` is either set to `false` or the default value of `false` is implied.
- If the tree does not support multiple selection, aria-selected is set to `true` for the selected node and it is not present on any other node in the tree.
- if the tree supports multiple selection:
    - All selected nodes have aria-selected set to `true`.
    - All nodes that are selectable but not selected have aria-selected set to `false`.
    - If the tree contains nodes that are not selectable, those nodes do not have the `aria-selected` state.
- The element with role `tree` has either a visible label referenced by aria-labelledby or a value specified for aria-label.
- If the complete set of available nodes is not present in the DOM due to dynamic loading as the user moves focus in or scrolls the tree, each node has aria-level, aria-setsize, and aria-posinset specified.

- If the `tree` element is horizontally oriented, it has aria-orientation set to `horizontal`. The default value of `aria-orientation` for a tree is `vertical`.

> NOTE
>
> If aria-owns is set on the tree container to include elements that are not DOM children of the container, those elements will appear in the reading order in the sequence they are referenced and after any items that are DOM children. Scripts that manage focus need to ensure the visual focus order matches this assistive technology reading order.

3.26 Treegrid §

A treegrid widget presents a hierarchical data grid consisting of tabular information that is editable or interactive. Any row in the hierarchy may have child rows, and rows with children may be expanded or collapsed to show or hide the children. For example, in a `treegrid` used to display messages and message responses for a e-mail discussion list, messages with responses would be in rows that can be expanded to reveal the response messages.

In a treegrid both rows and cells are focusable. Every row and cell contains a focusable element or is itself focusable, regardless of whether individual cell content is editable or interactive. There is one exception: if column header cells do not provide functions, such as sort or filter, they do not need to be focusable. One reason it is important for all cells to be able to receive or contain keyboard focus is that screen readers will typically be in their application reading mode, rather than their document reading mode, when users are interacting with the grid. While in application mode, a screen reader user hears only focusable elements and content that labels focusable elements. So, screen reader users may unknowingly overlook elements contained in a `treegrid` that are either not focusable or not used to label a column or row.

When using a keyboard to navigate a `treegrid`, a visual keyboard indicator informs the user which row or cell is focused. If the `treegrid` allows the user to choose just one item for an action, then it is known as a single-select `treegrid`, and the item with focus also has a selected state. However, in multi-select `treegrid`s, which enable the user to select more than one row or cell for an action, the selected state is independent of the focus. For example, in a hierarchical e-mail discussion grid, the user can move focus to select any number of rows for an action, such as delete or move. It is important that the visual design distinguish between items that are selected and the item that has focus. For more details, see this description of differences between focus and selection.

Examples §

- E-mail Inbox `treegrid` Example: A treegrid for navigating an e-mail inbox that demonstrates three keyboard navigation models -- rows first, cells first, and cells only.

Keyboard Interaction §

The following keys provide `treegrid` navigation by moving focus among rows and cells of the grid. Implementations of `treegrid` make these key commands available when an element in the grid has received focus, e.g., after a user has moved focus to the grid with `Tab`. Moving focus into the grid may result in the first cell or the first row being focused. Whether focus goes to a cell or the row depends on author preferences and whether row focus is supported, since some `treegrid`s may not provide focus to rows.

- `Enter`: If cell-only focus is enabled and focus is on the first cell with the `aria-expanded` property, opens or closes the child rows.,Otherwise, performs the default action for the cell.

- `Tab`: If the row containing focus contains focusable elements (e.g., inputs, buttons, links, etc.), moves focus to the next input in the row. If focus is on the last focusable element in the row, moves focus out of the `treegrid` widget to the next focusable element.
- `Right Arrow`:
  - If focus is on a collapsed row, expands the row.
  - If focus is on an expanded row or is on a row that does not have child rows, moves focus to the first cell in the row.
  - If focus is on the right-most cell in a row, focus does not move.
  - If focus is on any other cell, moves focus one cell to the right.
- `Left Arrow`:
  - If focus is on an expanded row, collapses the row.
  - If focus is on a collapsed row or on a row that does not have child rows, focus does not move.
  - If focus is on the first cell in a row and row focus is supported, moves focus to the row.
  - If focus is on the first cell in a row and row focus is not supported, focus does not move.
  - If focus is on any other cell, moves focus one cell to the left.
- `Down Arrow`:
  - If focus is on a row, moves focus one row down. If focus is on the last row, focus does not move.
  - If focus is on a cell, moves focus one cell down. If focus is on the bottom cell in the column, focus does not move.
- `Up Arrow`:
  - If focus is on a row, moves focus one row up. If focus is on the first row, focus does not move.
  - If focus is on a cell, moves focus one cell up. If focus is on the top cell in the column, focus does not move.
- `Page Down`:
  - If focus is on a row, moves focus down an author-determined number of rows, typically scrolling so the bottom row in the currently visible set of rows becomes one of the first visible rows. If focus is in the last row, focus does not move.
  - If focus is on a cell, moves focus down an author-determined number of cells, typically scrolling so the bottom row in the currently visible set of rows becomes one of the first visible rows. If focus is in the last row, focus does not move.
- `Page Up`:
  - If focus is on a row, moves focus up an author-determined number of rows, typically scrolling so the top row in the currently visible set of rows becomes one of the last visible rows. If focus is in the first row, focus does not move.
  - If focus is on a cell, moves focus up an author-determined number of cells, typically scrolling so the top row in the currently visible set of rows becomes one of the last visible rows. If focus is in the first row, focus does not move.
- `Home`:
  - If focus is on a row, moves focus to the first row. If focus is in the first row, focus does not move.
  - If focus is on a cell, moves focus to the first cell in the row. If focus is in the first cell of the row, focus does not move.
- `End`:
  - If focus is on a row, moves focus to the last row. If focus is in the last row, focus does not move.

    - If focus is on a cell, moves focus to the last cell in the row. If focus is in the last cell of the row, focus does not move.
- `Control + Home`:
    - If focus is on a row, moves focus to the first row. If focus is in the first row, focus does not move.
    - If focus is on a cell, moves focus to the first cell in the column. If focus is in the first row, focus does not move.
- `Control + End`:
    - If focus is on a row, moves focus to the last row. If focus is in the last row, focus does not move.
    - If focus is on a cell, moves focus to the last cell in the column. If focus is in the last row, focus does not move.

> NOTE
>
> - When the above `treegrid` navigation keys move focus, whether the focus is set on an element inside the cell or on the cell depends on cell content. See Whether to Focus on a Cell or an Element Inside It.
> - While navigation keys, such as arrow keys, are moving focus from cell to cell, they are not available to do something like operate a combobox or move an editing caret inside of a cell. If this functionality is needed, see Editing and Navigating Inside a Cell.
> - If navigation functions can dynamically add more rows or columns to the DOM, key events that move focus to the beginning or end of the grid, such as `control + End`, may move focus to the last row in the DOM rather than the last available row in the back-end data.

If a treegrid supports selection of cells, rows, or columns, the following keys are commonly used for these functions.

- `Control + Space`:
    - If focus is on a row, selects all cells.
    - If focus is on a cell, selects the column that contains the focus.
- `Shift + Space`:
    - If focus is on a row, selects the row.
    - If focus is on a cell, selects the row that contains the focus. If the treegrid includes a column with checkboxes for selecting rows, this key can serve as a shortcut for checking the box when focus is not on the checkbox.
- `Control + A`: Selects all cells.
- `Shift + Right Arrow`:
    - If focus is on a row, does not change selection.
    - if focus is on a cell, extends selection one cell to the right.
- `Shift + Left Arrow`:
    - If focus is on a row, does not change selection.
    - if focus is on a cell, extends selection one cell to the left.
- `Shift + Down Arrow`:
    - If focus is on a row, extends selection to all the cells in the next row.
    - If focus is on a cell, extends selection one cell down.
- `Shift + Up Arrow`:
    - If focus is on a row, extends selection to all the cells in the previous row.

- If focus is on a cell, extends selection one cell up.

> NOTE
>
> See § 6.8 Key Assignment Conventions for Common Functions for cut, copy, and paste key assignments.

WAI-ARIA Roles, States, and Properties §

- The treegrid container has role treegrid.
- Each row container has role row and is either a DOM descendant of or owned by the `treegrid` element or an element with role rowgroup.
- Each cell is either a DOM descendant of or owned by a `row` element and has one of the following roles:
  - columnheader if the cell contains a title or header information for the column.
  - rowheader if the cell contains title or header information for the row.
  - gridcell if the cell does not contain column or row header information.
- A `row` that can be expanded or collapsed to show or hide a set of child rows is a parent row. Each parent `row` has the aria-expanded state set on either the `row` element or on a cell contained in the`row`. The `aria-expanded` state is set to `false` when the child rows are not displayed and set to `true` when the child rows are displayed. Rows that do not control display of child rows do not have the `aria-expanded` attribute because, if they were to have it, they would be incorrectly described to assistive technologies as parent rows.
- If the treegrid supports selection of more than one row or cell, it is a multi-select treegrid and the element with role `treegrid` has aria-multiselectable set to `true`. Otherwise, it is a single-select treegrid, and `aria-multiselectable` is either set to `false` or the default value of `false` is implied.
- If the treegrid is a single-select treegrid, aria-selected is set to `true` on the selected row or cell, and it is not present on any other row or cell in the treegrid.
- if the treegrid is a multi-select treegrid:
  - All selected rows or cells have aria-selected set to `true`.
  - All rows and cells that are not selected have aria-selected set to `false`.
- If there is an element in the user interface that serves as a label for the treegrid, aria-labelledby is set on the grid element with a value that refers to the labelling element. Otherwise, a label is specified for the grid element using aria-label.
- If the treegrid has a caption or description, aria-describedby is set on the grid element with a value referring to the element containing the description.
- If the treegrid provides sort functions, aria-sort is set to an appropriate value on the header cell element for the sorted column or row as described in the section on grid and table properties.
- If the treegrid provides content editing functionality and contains cells that may have edit capabilities disabled in certain conditions, aria-readonly is set to `true` on cells where editing is disabled. If edit functions are disabled for all cells, instead of setting `aria-readonly` to `true` on every cell, `aria-readonly` may be set to `true` on the `treegrid` element. Treegrids that do not provide cell content editing functions do not include the `aria-readonly` attribute on any of their elements.
- If there are conditions where some rows or columns are hidden or not present in the DOM, e.g., data is dynamically loaded when scrolling or the grid provides functions for hiding rows or columns, the following properties are applied as described in the section on grid and table properties.

    - aria-colcount or aria-rowcount is set to the total number of columns or rows, respectively.
    - aria-colindex or aria-rowindex is set to the position of a cell within a row or column, respectively.
- If the treegrid includes cells that span multiple rows or multiple columns, and if the `treegrid` role is NOT applied to an HTML `table` element, then aria-rowspan or aria-colspan is applied as described in grid and table properties.

> NOTE
>
> - A `treegrid` built from an HTML `table` that includes cells that span multiple rows or columns must use HTML `rowspan` and `colspan` and must not use `aria-rowspan` or `aria-colspan`.
> - If rows or cells are included in a treegrid via aria-owns, they will be presented to assistive technologies after the DOM descendants of the `treegrid` element unless the DOM descendants are also included in the `aria-owns` attribute.

3.27 Window Splitter §

NOTE: ARIA 1.1 introduced changes to the separator role so it behaves as a widget when focusable. While this pattern has been revised to match the ARIA 1.1 specification, the task force will not complete its review until a functional example that matches the ARIA 1.1 specification is complete. Progress on this pattern is tracked by issue 129.

A window splitter is a moveable separator between two sections, or panes, of a window that enables users to change the relative size of the panes. A Window Splitter can be either variable or fixed. A fixed splitter toggles between two positions whereas a variable splitter can be adjusted to any position within an allowed range.

A window splitter has a value that represents the size of one of the panes, which, in this pattern, is called the primary pane. When the splitter has its minimum value, the primary pane has its smallest size and the secondary pane has its largest size. The splitter also has an accessible name that matches the name of the primary pane.

For example, consider a book reading application with a primary pane for the table of contents and a secondary pane that displays content from a section of the book. The two panes are divided by a vertical splitter labelled "Table of Contents". When the table of contents pane has its maximum size, the splitter has a value of `100`, and when the table of contents is completely collapsed, the splitter has a value of `0`.

Note that the term "primary pane" does not describe the importance or purpose of content inside the pane.

Example §

Work to develop an example window splitter widget is tracked by issue 130.

Keyboard Interaction §

- `Left Arrow`: Moves a vertical splitter to the left.
- `Right Arrow`: Moves a vertical splitter to the right.
- `Up Arrow`: Moves a horizontal splitter up.
- `Down Arrow`: Moves a horizontal splitter down.

- `Enter`: If the primary pane is not collapsed, collapses the pane. If the pane is collapsed, restores the splitter to its previous position.
- `Home` (Optional): Moves splitter to the position that gives the primary pane its smallest allowed size. This may completely collapse the primary pane.
- `End` (Optional): Moves splitter to the position that gives the primary pane its largest allowed size. This may completely collapse the secondary pane.
- `F6` (Optional): Cycle through window panes.

> NOTE
>
> A fixed size splitter omits implementation of the arrow keys.

WAI-ARIA Roles, States, and Properties §

- The element that serves as the focusable splitter has role separator.
- The separator element has the aria-valuenow property set to a decimal value representing the current position of the separator.
- The separator element has the aria-valuemin property set to a decimal value that represents the position where the primary pane has its minimum size. This is typically `0`.
- The separator element has the aria-valuemax property set to a decimal value that represents the position where the primary pane has its maximum size. This is typically `100`.
- If the primary pane has a visible label, it is referenced by aria-labelledby on the separator element. Otherwise, the separator element has a label provided by aria-label.
- The separator element has aria-controls referring to the primary pane.

4. Landmark Regions §

ARIA landmark roles provide a powerful way to identify the organization and structure of a web page. By classifying and labelling sections of a page, they enable structural information that is conveyed visually through layout to be represented programmatically. Screen readers exploit landmark roles to provide keyboard navigation to important sections of a page. Landmark regions can also be used as targets for "skip links" and by browser extensions to enhanced keyboard navigation.

This section explains how HTML sectioning elements and ARIA landmark roles are used to make it easy for assistive technology users to understand the meaning of the layout of a page.

4.1 HTML Sectioning Elements §

Several HTML sectioning elements automatically create ARIA landmark regions. So, in order to provide assistive technology users with a logical view of a page, it is important to understand the effects of using HTML sectioning elements. [HTML-ARIA] contains more information on HTML element role mapping.

Default landmark roles for HTML sectioning elements

HTML Element	Default Landmark Role
`aside`	`complementary`

HTML Element	Default Landmark Role
`footer`	`contentinfo` when in context of the `body` element
`header`	`banner` when in context of the `body` element
`main`	`main`
`nav`	`navigation`
`section`	`region` when it has an accessible name using `aria-labelledby` or `aria-label`

4.2 General Principles of Landmark Design §

Including **all perceivable content** on a page in one of its landmark regions and giving each landmark region a semantically meaningful role is one of the most effective ways of ensuring assistive technology users will not overlook information that is relevant to their needs.

Step 1: Identify the logical structure

- Break the page into perceivable areas of content which designers typically indicate visually using alignment and spacing.
- Areas can be further defined into logical sub-areas as needed.
- An example of a sub-area is a portlet in a portal application.

Step 2: Assign landmark roles to each area

- Assign landmark roles based on the type of content in the area.
- `banner`, `main`, `complementary` and `contentinfo` landmarks should be top level landmarks.
- Landmark roles can be nested to identify parent/child relationships of the information being presented.

Step 3: Label areas

- If a specific landmark role is used more than once on a page, provide each instance of that landmark with a unique label. There is one rare circumstance where providing the same label to multiple instances of a landmark can be beneficial: the content and purpose of each instance is identical. For example, a large search results table has two sets of identical pagination controls -- one above and one below the table, so each set is in a navigation region labelled “Search Results”. In this case, adding extra information to the label that distinguishes the two instances may be more distracting than helpful.
- If a landmark is only used once on the page it may not require a label. See Landmark Roles section below.
- If an area begins with a heading element (e.g. `h1-h6`) it can be used as the label for the area using the `aria-labelledby` attribute.
- If an area requires a label and does not have a heading element, provide a label using the `aria-label` attribute.
- Do not use the landmark role as part of the label. For example, a navigation landmark with a label "Site Navigation" will be announced by a screen reader as "Site Navigation Navigation". The label should simply be "Site".

4.3 Landmark Roles §

4.3.1 Banner §

A `banner` landmark identifies site-oriented content at the beginning of each page within a website. Site-oriented content typically includes things such as the logo or identity of the site sponsor, and site-specific search tool. A banner usually appears at the top of the page and typically spans the full width.

- Each page may have one `banner` landmark.
- The `banner` landmark should be a top-level landmark.
- When a page contains nested `document` and/or `application` roles (e.g. typically through the use of `iframe` and `frame` elements), each `document` or `application` role may have one `banner` landmark.
- If a page includes more than one `banner` landmark, each should have a unique label (see Step 3 above).

HTML Techniques §

- The HTML `header` element defines a `banner` landmark when its context is the `body` element.
- The HTML `header` element is not considered a `banner` landmark when it is descendant of any of following elements (see HTML Accessibility Mappings [HTML-AAM]):
  - `article`
  - `aside`
  - `main`
  - `nav`
  - `section`

ARIA Techniques §

If the HTML `header` element technique is not being used, a `role="banner"` attribute should be used to define a `banner` landmark.

Examples §

Banner Landmark Example

4.3.2 Complementary §

A `complementary` landmark is a supporting section of the document, designed to be complementary to the main content at a similar level in the DOM hierarchy, but remains meaningful when separated from the main content.

- `complementary` landmarks should be top level landmarks (e.g. not contained within any other landmarks).
- If the complementary content is not related to the main content, a more general role should be assigned (e.g. `region`).
- If a page includes more than one `complementary` landmark, each should have a unique label (see Step 3 above).

HTML Technique §

Use the HTML `aside` element to define a `complementary` landmark.

ARIA Technique §

If the HTML `aside` element technique is not being used, use a `role="complementary"` attribute to define a `complementary` landmark.

Examples §

Complementary Landmark Example

4.3.3 Contentinfo §

A `contentinfo` landmark is a way to identify common information at the bottom of each page within a website, typically called the "footer" of the page, including information such as copyrights and links to privacy and accessibility statements.

- Each page may have one `contentinfo` landmark.
- The `contentinfo` landmark should be a top-level landmark.
- When a page contains nested `document` and/or `application` roles (e.g. typically through the use of `iframe` and `frame` elements), each `document` or `application` role may have one `contentinfo` landmark.
- If a page includes more than one `contentinfo` landmark, each should have a unique label (see Step 3 above).

HTML Techniques §

- The HTML `footer` element defines a `contentinfo` landmark when its context is the `body` element.
- The HTML `footer` element is not considered a `contentinfo` landmark when it is descendant of any of following elements (see HTML Accessibility Mappings [HTML-AAM]):
  - `article`
  - `aside`
  - `main`
  - `nav`
  - `section`

ARIA Technique §

If the HTML `footer` element technique is not being used, a `role="contentinfo"` attribute should be used to define a `contentinfo` landmark.

Examples §

Contentinfo Landmark Example

4.3.4 Form §

A `form` landmark identifies a region that contains a collection of items and objects that, as a whole, combine to create a form when no other named landmark is appropriate (e.g. main or search).

- Use the `search` landmark instead of the `form` landmark when the form is used for search functionality.
- A `form` landmark should have a label to help users understand the purpose of the form.
- A label for the `form` landmark should be visible to all users (e.g. an `h1-h6` element).
- If a page includes more than one `form` landmark, each should have a unique label (see Step 3 above).
- Whenever possible, controls contained in a `form` landmark in an HTML document should use native host semantics:
  - `button`
  - `input`
  - `select`
  - `textarea`

HTML Techniques §

The HTML `form` element defines a `form` landmark when it has an accessible name (e.g. `aria-labelledby`, `aria-label` or `title`).

ARIA Technique §

Use the `role="form"` to identify a region of the page; do not use it to identify every form field.

Examples §

Form Landmark Example

4.3.5 Main §

A `main` landmark identifies the primary content of the page.

- Each page should have one `main` landmark.
- The `main` landmark should be a top-level landmark.
- When a page contains nested `document` and/or `application` roles (e.g. typically through the use of `iframe` and `frame` elements), each `document` or `application` role may have one `main` landmark.
- If a page includes more than one `main` landmark, each should have a unique label (see Step 3 above).

HTML Technique §

Use the HTML `main` element to define a `main` landmark.

ARIA Technique §

If the HTML `main` element technique is not being used, use a `role="main"` attribute to define a `main` landmark.

Examples §

Main Landmark Example

4.3.6 Navigation §

`Navigation` landmarks provide a way to identify groups (e.g. lists) of links that are intended to be used for website or page content navigation.

- If a page includes more than one `navigation` landmark, each should have a unique label (see Step 3 above).
- If a `navigation` landmark has an identical set of links as another `navigation` landmark on the page, use the same label for each `navigation` landmark.

HTML Technique §

Use the HTML `nav` element to define a `navigation` landmark.

ARIA Technique §

If the HTML `nav` element technique is not being used, use a `role="navigation"` attribute to define a `navigation` landmark.

Examples §

Navigation Landmark Example

4.3.7 Region §

A `region` landmark is a perceivable section of the page containing content that is sufficiently important for users to be able to navigate to the section.

- A `region` landmark must have a label.
- If a page includes more than one `region` landmark, each should have a unique label (see Step 3 above).
- The `region` landmark can be used identify content that named landmarks do not appropriately describe.

HTML Technique §

The HTML `section` element defines a `region` landmark when it has an accessible name (e.g. `aria-labelledby`, `aria-label` or `title`).

ARIA Technique §

If the HTML `section` element technique is not being used, use a `role="region"` attribute to define a `region` landmark.

Examples §

Region Landmark Example

4.3.8 Search §

A `search` landmark contains a collection of items and objects that, as a whole, combine to create search functionality.

- Use the `search` landmark instead of the `form` landmark when the form is used for search functionality.
- If a page includes more than one `search` landmark, each should have a unique label (see Step 3 above).

HTML Technique §

There is no HTML element that defines a `search` landmark.

ARIA Technique §

The `role="search"` attribute defines a `search` landmark.

Examples §

Search Landmark Example

5. Providing Accessible Names and Descriptions §

Providing elements with accessible names, and where appropriate, accessible descriptions is one of the most important responsibilities authors have when developing accessible web experiences. While doing so is straightforward for most elements, technical mistakes that can completely block users of assistive technologies are easy to make and unfortunately common. To help authors effectively provide accessible names and descriptions, this section explains their purpose, when authors need to provide them, how browsers assemble them, and rules for coding and composing them. It also guides authors in the use of the following naming and describing techniques and WAI-ARIA properties:

- Naming:
  - Naming with child content.
  - Naming with a string attribute via `aria-label`.

    - Naming by referencing content with `aria-labelledby`.
    - Naming form controls with the label element.
    - Naming fieldsets with the legend element.
    - Naming tables and figures with captions.
    - Fallback names derived from titles and placeholders.
- Describing:
    - Describing by referencing content with `aria-describedby`.
    - Describing tables and figures with captions.
    - Descriptions derived from titles.

5.1 What ARE Accessible Names and Descriptions? §

An accessible name is a short string, typically 1 to 3 words, that authors associate with an element to provide users of assistive technologies with a label for the element. For example, an input field might have an accessible name of "User ID" or a button might be named "Submit".

An accessible name serves two primary purposes for users of assistive technologies, such as screen readers:

1. Convey the purpose or intent of the element.
2. Distinguish the element from other elements on the page.

Both the WAI-ARIA specification and WCAG require all focusable, interactive elements to have an accessible name. In addition dialogs and some structural containers, such as tables and regions, are required to have a name. Many other elements can be named, but whether a name will enhance the accessible experience is determined by various characteristics of the surrounding context. Finally, there are some elements where providing an accessible name is technically possible but not advisable. The Accessible Name Guidance by Role section lists naming requirements and guidelines for every ARIA role.

An accessible description is also an author-provided string that is rendered by assistive technologies. Authors supply a description when there is a need to associate additional information with an element, such as instructions or format requirements for an input field.

assistive technologies present names differently from descriptions. For instance, screen readers typically announce the name and role of an element first, e.g., a button named “Mute Conversation”could be spoken as “Mute Conversation button”. If an element has a state, it could be announced either before or after the name and role; after name and role is the typical default. For example, a switch button named “Mute Conversation” in the “off” state could be announced as “Mute Conversation switch button off”. Because descriptions are optional strings that are usually significantly longer than names, they are presented last, sometimes after a slight delay. For example, “Mute Conversation Switch button off, Silences alerts and notifications about activity in this conversation.” To reduce verbosity, some screen readers do not announce descriptions by default but instead inform users of their presence so that users can press a key that will announce the description.

5.2 How Are Name and Description Strings Derived? §

Because there are several elements and attributes for specifying text to include in an accessible name or description string, and because authors can combine them in a practically endless number of ways, browsers implement fairly complex algorithms for assembling the strings. The sections on accessible name calculation and accessible description calculation

explain the algorithms and how they implement precedence. However, most authors do not need such detailed understanding of the algorithms since nearly all circumstances where a name or description is useful are supported by the coding patterns described in the naming techniques and describing techniques sections.

5.3 Accessible Names §

5.3.1 Cardinal Rules of Naming §

5.3.1.1 Rule 1: Heed Warnings and Test Thoroughly §

Several of the naming techniques below include notes that warn against specific coding patterns that are either prohibited by the ARIA specification or fall into gray space that is not yet fully specified. Some of these prohibited or ambiguous patterns may appear logical and even yield desired names in some browsers. However, it is unlikely they will provide consistent results across browsers, especially over time as work to improve the consistency of name calculation across browsers progresses.

In addition to heeding the warnings provided in the naming techniques, it is difficult to over emphasize the importance of testing to ensure that names browsers calculate match expectations.

5.3.1.2 Rule 2: Prefer Visible Text §

When a user interface includes visible text that could be used to provide an appropriate accessible name, using the visible text for the accessible name simplifies maintenance, prevents bugs, and reduces language translation requirements. When names are generated from text that exists only in markup and is never displayed visually, there is a greater likelihood that accessible names will not be updated when the user interface design or content are changed.

If an interactive element, such as an input field or button, does not have a visually persistent text label, consider adjusting the design to include one. In addition to serving as a more robust source for an accessible name, visible text labels enhance accessibility for many people with disabilities who do not use assistive technologies that present invisible accessible names. In most circumstances, visible text labels also make the user interface easier to understand for all users.

5.3.1.3 Rule 3: Prefer Native Techniques §

In HTML documents, whenever possible, rely on HTML naming techniques, such as the HTML `label` element for form elements and `caption` element for tables. While less flexible, their simplicity and reliance on visible text help ensure robust accessible experiences. Several of the naming techniques highlight specific accessibility advantages of using HTML features instead of ARIA attributes.

5.3.1.4 Rule 4: Avoid Browser Fallback §

When authors do not specify an accessible name using an element or attribute that is intended for naming, browsers attempt to help assistive technology users by resorting to fallback methods for generating a name. For example, the HTML `title` and `placeholder` attributes are used as last resort sources of content for accessible names. Because the purpose of these attributes is not naming, their content typically yields low quality accessible names that are not effective.

5.3.1.5 Rule 5: Compose Brief, Useful Names §

Similar to how visually crowded screens and ambiguous icons reduce usability, excessively long, insufficiently distinct, or unclear accessible names can make a user interface very difficult, or even impossible, to use for someone who relies on a non-visual form of the user interface. In other words, for a web experience to be accessible, its accessible names must be effective. The section on Composing Effective and User-friendly Accessible Names provides guidance for balancing brevity and clarity.

5.3.2 Naming Techniques §

5.3.2.1 Naming with Child Content §

Certain elements get their name from the content they contain. For example, the following link is named "Home".

```
<a href="/">Home</a>
```

When assistive technologies render an element that gets its accessible name from its content, such as a link or button, the accessible name is the only content the user can perceive for that element. This is in contrast to other elements, such as text fields or tables, where the accessible name is a label that is presented in addition to the value or content of the element. For instance, the accessible name of a table can be derived from a caption element, and assistive technologies render both the caption and all other content contained inside the table.

Elements having one of the following roles are, by default, named by a string calculated from their descendant content:

- button
- cell
- checkbox
- columnheader
- gridcell
- heading
- link
- menuitem (content contained in a child `menu` element is excluded.)
- menuitemcheckbox
- menuitemradio
- option
- radio
- row
- rowheader
- switch
- tab
- tooltip
- treeitem (content included in a child `group` element is excluded.)

When calculating a name from content for an element, user agents recursively walk through each of its descendant elements, calculate a name string for each descendant, and concatenate the resulting strings. In two special cases, certain descendants are ignored: `group` descendants of `treeitem` elements and `menu` descendants of `menuitem` elements are omitted from the calculation. For example, in the following `tree`, the name of the first tree item is “Fruits”; “Apples”, “Bananas”, and “Oranges” are omitted.

```
<ul role="tree">
  <li role="treeitem">Fruits
    <ul role="group">
      <li role="treeitem">Apples</li>
      <li role="treeitem">Bananas</li>
      <li role="treeitem">Oranges</li>
    </ul>
  </li>
</ul>
```

Warning

If an element with one of the above roles that supports naming from child content is named by using `aria-label` or `aria-labelledby`, content contained in the element and its descendants is hidden from assistive technology users unless the descendant content is referenced by `aria-labelledby`. It is strongly recommended to avoid using either of these attributes to override content of one of the above elements except in rare circumstances where hiding content from assistive technology users is beneficial. In addition, in situations where visible content is hidden from assistive technology users by use of one of these attributes, thorough testing with assistive technologies is particularly important.

5.3.2.2 Naming with a String Attribute Via `aria-label` §

The aria-label property enables authors to name an element with a string that is not visually rendered. For example, the name of the following button is "Close".

```
<button type="button" aria-label="Close">X</button>
```

The `aria-label` property is useful when there is no visible text content that will serve as an appropriate accessible name.

The `aria-label` property affects assistive technology users in one of two different ways, depending on the role of the element to which it is applied. When applied to an element with one of the roles that supports naming from child content, `aria-label` hides descendant content from assistive technology users and replaces it with the value of `aria-label`. However, when applied to nearly any other type of element, assistive technologies will render both the value of `aria-label` and the content of the element. For example, the name of the following navigation region is "Product".

```
<nav aria-label="Product">
  <!-- list of navigation links to product pages -->
</nav>
```

When encountering this navigation region, a screen reader user will hear the name and role of the element, e.g., "Product navigation region", and then be able to read through the links contained in the region.

Warning

1. If `aria-label` is applied to an element with one of the roles that supports naming from child content, content contained in the element and its descendants is hidden from assistive technology users. It is strongly recommended to avoid using `aria-label` to override content of one of these elements except in rare circumstances where hiding content from assistive technology users is beneficial.
2. There are certain types of elements, such as paragraphs and list items, that should not be named with `aria-label`. They are identified in the table in the Accessible Name Guidance by Role section.
3. Because the value of `aria-label` is not rendered visually, testing with assistive technologies to ensure the expected name is presented to users is particularly important.
4. When a user interface is translated into multiple languages, ensure that `aria-label` values are translated.

5.3.2.3 Naming with Referenced Content Via `aria-labelledby` §

The aria-labelledby property enables authors to reference other elements on the page to define an accessible name. For example, the following switch is named by the text content of a previous sibling element.

```
<span id="night-mode-label">Night mode</span>
<span role="switch" aria-checked="false" tabindex="0" aria-labelledby="night-mode-label"></span>
```

Note that while using `aria-labelledby` is similar in this situation to using an HTML `label` element with the `for` attribute, one significant difference is that browsers do not automatically make clicking on the labeling element activate the labeled element; that is an author responsibility. However, HTML `label` cannot be used to label a `span` element. Fortunately, an HTML `input` with `type="checkbox"` allows the ARIA `switch` role, so when feasible, using the following approach creates a more robust solution.

```
<label for="night-mode">Night mode</label>
<input type="checkbox" role="switch" id="night-mode">
```

The `aria-labelledby` property is useful in a wide variety of situations because:

- It has the highest precedence when browsers calculate accessible names, i.e., it overrides names from child content and all other naming attributes, including `aria-label`.
- It can concatenate content from multiple elements into a single name string.
- It incorporates content from elements regardless of their visibility, i.e., it even includes content from elements with the HTML `hidden` attribute, CSS `display: none`, or CSS `visibility: hidden` in the calculated name string.
- It incorporates the value of input elements, i.e., if it references a textbox, the value of the textbox is included in the calculated name string.

An example of referencing a hidden element with `aria-labelledby` could be a label for a night switch control:

```
<span id="night-mode-label" hidden>Night mode</span>
<input type="checkbox" role="switch" aria-labelledby="night-mode-label">
```

In some cases, the most effective name for an element is its own content combined with the content of another element. Because `aria-labelledby` has highest precedence in name calculation, in those situations, it is possible to use `aria-labelledby` to reference both the element itself and the other element. In the following example, the "Read more..." link is

named by the element itself and the article's heading, resulting in a name for the link of "Read more... 7 ways you can help save the bees".

```
<h2 id="bees-heading">7 ways you can help save the bees</h2>
<p>Bees are disappearing rapidly. Here are seven things you can do to help.</p>
<p><a id="bees-read-more" aria-labelledby="bees-read-more bees-heading">Read more...</a></p>
```

When multiple elements are referenced by `aria-labelledby`, text content from each referenced element is concatenated in the order specified in the `aria-labelledby` value. If an element is referenced more than one time, only the first reference is processed. When concatenating content from multiple elements, browsers trim leading and trailing white space and separate content from each element with a single space.

```
<button id="download-button" aria-labelledby="download-button download-details">Download</button>
<span id="download-details">PDF, 2.4 MB</span>
```

In the above example, the accessible name of the button will be "Download PDF, 2.4 MB", with a space between "Download" and "PDF", and not "DownloadPDF, 2.4 MB".

Warning

1. The `aria-labelledby` property cannot be chained, i.e., if an element with `aria-labelledby` references another element that also has `aria-labelledby`, the `aria-labelledby` attribute on the referenced element will be ignored.
2. If an element is referenced by `aria-labelledby` more than one time during a name calculation, the second and any subsequent references will be ignored.
3. There are certain types of elements, such as paragraphs and list items, that should not be named with `aria-labelledby`. They are identified in the table in the Accessible Name Guidance by Role section.
4. If `aria-labelledby` is applied to an element with one of the roles that supports naming from child content, content contained in the element and its descendants is hidden from assistive technology users unless it is also referenced by `aria-labelledby`. It is strongly recommended to avoid using this attribute to override content of one of these elements except in rare circumstances where hiding content from assistive technology users is beneficial.
5. Because calculating the name of an element with `aria-labelledby` can be complex and reference hidden content, testing with assistive technologies to ensure the expected name is presented to users is particularly important.

5.3.2.4 Naming Form Controls with the Label Element §

The HTML `label` element enables authors to identify content that serves as a label and associate it with a form control. When a `label` element is associated with a form control, browsers calculate an accessible name for the form control from the `label` content.

For example, text displayed adjacent to a checkbox may be visually associated with the checkbox, so it is understood as the checkbox label by users who can perceive that visual association. However, unless the text is programmatically associated with the checkbox, assistive technology users will experience a checkbox without a label. Wrapping the checkbox and the labeling text in a `label` element as follows gives the checkbox an accessible name.

```
<label>
  <input type="checkbox" name="subscribe">
  subscribe to our newsletter
```

```
</label>
```

A form control can also be associated with a label by using the `for` attribute on the `label` element. This allows the label and the form control to be siblings or have different parents in the DOM, but requires adding an `id` attribute to the form control, which can be error-prone. When possible, use the above encapsulation technique for association instead of the following `for` attribute technique.

```
<input type="checkbox" name="subscribe" id="subscribe_checkbox">
<label for="subscribe_checkbox">subscribe to our newsletter</label>
```

Using the `label` element is an effective technique for satisfying Rule 2: Prefer Visible Text. It also satisfies Rule 3: Prefer Native Techniques. Native HTML labels offer an important usability and accessibility advantage over ARIA labeling techniques: browsers automatically make clicking the label equivalent to clicking the form control. This increases the hit area of the form control.

5.3.2.5 Naming Fieldsets with the Legend Element §

The HTML `fieldset` element can be used to group form controls, and the `legend` element can be used to give the group a name. For example, a group of radio buttons can be grouped together in a `fieldset`, where the `legend` element labels the group for the radio buttons.

```
<fieldset>
  <legend>Select your starter class</legend>
  <label><input type="radio" name="starter-class" value="green"> Green</label>
  <label><input type="radio" name="starter-class" value="red"> Red</label>
  <label><input type="radio" name="starter-class" value="blue"> Blue</label>
</fieldset>
```

This grouping technique is particularly useful for presenting multiple choice questions. It enables authors to associate a question with a group of answers. If a question is not programmatically associated with its answer options, assistive technology users may access the answers without being aware of the question.

Similar benefits can be gained from grouping and naming other types of related form fields using `fieldset` and `legend`.

```
<fieldset>
  <legend>Shipping address</legend>
  <p><label>Full name <input name="name" required></label></p>
  <p><label>Address line 1 <input name="address-1" required></label></p>
  <p><label>Address line 2 <input name="address-2"></label></p>
  ...
</fieldset>
<fieldset>
  <legend>Billing address</legend>
  ...
</fieldset>
```

Using the `legend` element to name a `fieldset` element satisfies Rule 2: Prefer Visible Text and Rule 3: Prefer Native Techniques.

5.3.2.6 Naming Tables and Figures with Captions §

The accessible name for HTML `table` and `figure` elements can be derived from a child `caption` or `figcaption` element, respectively. Tables and figures often have a caption to explain what they are about, how to read them, and sometimes giving them numbers used to refer to them in surrounding prose. Captions can help all users better understand content, but are especially helpful to users of assistive technologies.

In HTML, the `table` element marks up a data table, and can be provided with a caption using the `caption` element. If the `table` element does not have `aria-label` or `aria-labelledby`, then the `caption` will be used as the accessible name. For example, the accessible name of the following table is “Special opening hours”.

```
<table>
 <caption>Special opening hours</caption>
 <tr><td>30 May <td>Closed
 <tr><td>6 June <td>11:00-16:00
</table>
```

The following example gives the table a number (“Table 1”) so it can be referenced.

```
<table>
 <caption>Table 1. Traditional dietary intake of Okinawans and other Japanese circa 1950</caption>
 <thead>
  <tr>
   <th>
   <th>Okinawa, 1949
   <th>Japan, 1950
 <tbody>
  <tr>
   <th>Total calories
   <td>1785
   <td>2068

  [...]

</table>
```

Note: Above table content is from Caloric restriction, the traditional Okinawan diet, and healthy aging: the diet of the world's longest-lived people and its potential impact on morbidity and life span.

If a `table` is named using `aria-label` or `aria-labelledby`, then a `caption` element, if present, will become an accessible description. For an example, see Describing Tables and Figures with Captions.

Similarly, an HTML `figure` element can be given a caption using the `figcaption` element. The caption can appear before or after the figure, but it is more common for figures to have the caption after.

```
<figure>
 <img alt="Painting of a person walking in a desert." src="Hole_JesusalDesierto.jpg">
 <figcaption>Jesus entering the desert as imagined by William Hole, 1908</figcaption>
</figure>
```

Like with `table` elements, if a `figure` is not named using `aria-label` or `aria-labelledby`, the content of the `figcaption` element will be used as the accessible name. However unlike `table` elements, if the `figcaption` element is not used for the name, it does not become an accessible description unless it is referenced by `aria-describedby`. Nevertheless, assistive technologies will render the content of a `figcaption` regardless of whether it is used as a name, description, or neither.

Using the `caption` element to name a `table` element, or a `figcaption` element to name a `figure` element, satisfies Rule 2: Prefer Visible Text and Rule 3: Prefer Native Techniques.

5.3.2.7 Fallback Names Derived from Titles and Placeholders §

When an accessible name is not provided using one of the primary techniques (e.g., the `aria-label` or `aria-labelledby` attributes), or native markup techniques (e.g., the HTML `label` element, or the `alt` attribute of the HTML `img` element), browsers calculate an accessible name from other attributes as a fallback mechanism. Because the attributes used in fallback name calculation are not intended for naming, they typically yield low quality accessible names that are not effective. So, As advised by Rule 4: Avoid Browser Fallback, prefer the explicit labeling techniques described above over fallback techniques described in this section.

Any HTML element can have a `title` attribute specified. The `title` attribute may be used as the element's fallback accessible name. The `title` attribute is commonly presented visually as a tooltip when the user hovers over the element with a pointing device, which is not particularly discoverable, and is also not accessible to visual users without a pointing device.

For example, a `fieldset` element without a `legend` element child, but with a `title` attribute, gets its accessible name from the `title` attribute.

```
<fieldset title="Select your starter class">
  <label><input type="radio" name="starter-class" value="green"> Green</label>
  <label><input type="radio" name="starter-class" value="red"> Red</label>
  <label><input type="radio" name="starter-class" value="blue"> Blue</label>
</fieldset>
```

For the HTML `input` and `textarea` elements, the `placeholder` attribute is used as a fallback labeling mechanism if nothing else (including the `title` attribute) results in a label. It is better to use a `label` element, since it does not disappear visually when the user focuses the form control.

```
<!-- Using a <label> is recommended -->
<label>Search <input type=search name=q></label>

<!-- A placeholder is used as fallback -->
<input type=search name=q placeholder="Search">
```

5.3.3 Composing Effective and User-friendly Accessible Names §

For assistive technology users, especially screen reader users, the quality of accessible names is one of the most significant contributors to usability. Names that do not provide enough information reduce users' effectiveness while names that are too long reduce efficiency. And, names that are difficult to understand reduce effectiveness, efficiency, and enjoyment.

The following guidelines provide a starting point for crafting user friendly names.

- Convey function or purpose, not form. For example, if an icon that looks like the letter “X” closes a dialog, name it “Close”, not “X”. Similarly, if a set of navigation links in the left side bar navigate among the product pages in a shopping site, name the navigation region “Product”, not “Left”.
- Put the most distinguishing and important words first. Often, for interactive elements that perform an action, this means a verb is the first word. For instance, if a list of contacts displays “Edit”, “Delete”, and “Actions” buttons for each

contact, then “Edit John Doe”, “Delete John Doe”, and “Actions for John Doe” would be better accessible names than “John Doe edit”, “John Doe delete”, and “John Doe actions”. By placing the verb first in the name, screen reader users can more easily and quickly distinguish the buttons from one another as well as from the element that opens the contact card for John Doe.

- Be concise. For many elements, one to three words is sufficient. Only add more words when necessary.
- Do NOT include a WAI-ARIA role name in the accessible name. For example, do not include the word “button” in the name of a button, the word “image” in the name of an image, or the word “navigation” in the name of a navigation region. Doing so would create duplicate screen reader output since screen readers convey the role of an element in addition to its name.
- Create unique names for elements with the same role unless the elements are actually identical. For example, ensure every link on a page has a different name except in cases where multiple links reference the same location. Similarly, give every navigation region on a page a different name unless there are regions with identical content that performs identical navigation functions.
- Start names with a capital letter; it helps some screen readers speak them with appropriate inflection. Do not end names with a period; they are not sentences.

5.3.4 Accessible Name Guidance by Role §

Certain elements always require a name, others may usually or sometimes require a name, and still others should never be named. The table below lists all ARIA roles and provides the following information for each :

Necessity of Naming
Indicates how necessary it is for authors to add a naming attribute or element to supplement or override the content of an element with the specified role. This column may include one of the following values:

- Required **Only If** Content Insufficient: An element with this role is named by its descendant content. If `aria-label` or `aria-labelledby` is applied, content contained in the element and its descendants is hidden from assistive technology users unless it is also referenced by `aria-labelledby`. Avoid hiding descendant content except in the rare circumstances where doing so benefits assistive technology users.
- Required: The ARIA specification requires authors to provide a name; a missing name triggers accessibility validators to flag a violation.
- Recommended: Providing a name is strongly recommended.
- Discretionary: Naming is either optional or, in the circumstances described in the guidance column, is discouraged.
- Do Not Name: Naming is strongly discouraged even if it is technically permitted; often assistive technologies do not render a name even if provided.

Guidance:
Provides information to help determine if providing a name is beneficial, and if so, describes any recommended techniques.

role	Necessity of Naming	Guidance

role	Necessity of Naming	Guidance
`alert`	Discretionary	Some screen readers announce the name of an alert before announcing the content of the alert. Thus, `aria-label` provides a method for prefacing the visible content of an alert with text that is not displayed as part of the alert. Using `aria-label` is functionally equivalent to providing off-screen text in the contents of the alert, except off-screen text would be announced by screen readers that do not support `aria-label` on `alert` elements.
`alertdialog`	Required	Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.
`application`	Required	Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.
`article`	Recommended	• Recommended to distinguish articles from one another; helps users when navigating among articles. • Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.
`banner`	Discretionary	• Necessary in the uncommon circumstance where two banner landmark regions are present on the same page. It is otherwise optional. • Named using `aria-labelledby` if a visible label is present, otherwise with `aria-label`. • See the Banner Landmark section.
`button`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content.
`cell`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content. • Note that a name is not required; assistive technologies expect an empty cell in a table to be represented by an empty name. • Note that associated row or column headers do not name a `cell`; the name of a cell in a table is its content. Headers are complementary information.
`checkbox`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide any descendant content from assistive technologies. • If based on HTML `type="checkbox"`, use a `label` element. • Otherwise, reference visible content via `aria-labelledby`.

role	Necessity of Naming	Guidance
`columnheader`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content. • If the `columnheader` role is implied from an HTML `th`, the HTML `abbr` attribute can be used to specify an abbreviated version of the name that is only announced when screen readers are reading an associated `cell` within the `table`, `grid`, or `treegrid`.
`combobox`	Required	• If the `combobox` role is applied to an HTML `select` or `input` element, can be named with an HTML `label` element. • Otherwise use `aria-labelledby` if a visible label is present. • Use `aria-label` if a visible label is not present.
`complementary`	Recommended	• Naming is necessary when two complementary landmark regions are present on the same page. • Naming is recommended even when one complementary region is present to help users understand the purpose of the region's content when navigating among landmark regions. • Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`. • See the Complementary Landmark section.
`contentinfo`	Discretionary	• Necessary in the uncommon circumstance where two contentinfo landmark regions are present on the same page. It is otherwise optional. • Named using `aria-labelledby` if a visible label is present, otherwise with `aria-label`.
`definition`	Recommended	Reference the term being defined with `role="term"`, using `aria-labelledby`.
`dialog`	Required	Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.
`directory`	Discretionary	• Naming can help users understand the purpose of the directory. • Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.
`document`	Discretionary	Elements with the `document` role are contained within an element with the `application` role, which is required to have a name. Typically, the name of the `application` element will provide sufficient context and identity for the `document` element. Because the `application` element is used only to create unusual, custom widgets, careful assessment is necessary to determine whether or not adding an accessible name is beneficial.

role	Necessity of Naming	Guidance
`feed`	Recommended	• Helps screen reader users understand the context and purpose of the feed. • Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`. • See the Feed Design Pattern.
`figure`	Recommended	• For HTML, use the `figure` and `figcaption` elements. The `figcaption` will serve as the accessible name for the `figure`. See the Naming Tables and Figures with Captions section. • When not using HTML, or when retrofitting legacy HTML, use the `aria-labelledby` on the figure, pointing to the figure's caption. • If there is no visible caption, `aria-label` can be used.
`form`	Recommended	• Helps screen reader users understand the context and purpose of the form landmark. • Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`. • See the Form Landmark section.
`grid`	Required	• If the `grid` is applied to an HTML `table` element, then the accessible name can be derived from the table's `caption` element. • Otherwise, use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.
`gridcell`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content. • Note that a name is not required; assistive technologies expect an empty cell in a grid to be represented by an empty name. • Note that associated row or column headers do not name a `gridcell`; the name of a cell in a grid is its content. Headers are complementary information.
`group`	Discretionary	• When using the HTML `fieldset` element, the accessible name can be derived from the `legend` element. • When using the HTML `details` element, do not provide an accessible name for this element. The user interacts with the `summary` element, and that can derive its accessible name from its contents. • When using the HTML `optgroup` element, use the `label` attribute. • Otherwise, use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.

role	Necessity of Naming	Guidance
`heading`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content.
`img`	Required	For the HTML `img` element, use the `alt` attribute. For other elements with the `img` role, use `aria-labelledby` or `aria-label`.
`link`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content.
`list`	Discretionary	• Potentially beneficial for users of screen readers that support both list names and navigation among lists on a page. • Potentially a source of distracting or undesirable screen reader verbosity, especially if nested within a named container, such as a navigation region. • Can be named using `aria-labelledby` if a visible label is present, otherwise with `aria-label`.
`listbox`	Required	• If the `listbox` role is applied to an HTML `select` element (with the `multiple` attribute or a `size` attribute having a value greater than 1), can be named with an HTML `label` element. • Otherwise use `aria-labelledby` if a visible label is present. • Use `aria-label` if a visible label is not present. • See the Listbox Design Pattern.
`listitem`	Do Not Name	Not supported by assistive technologies; it is necessary to include relevant content within the list item.
`log`	Required	Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.
`main`	Discretionary	• Potentially helpful for orienting assistive technology users, especially in single-page applications where main content changes happen without generating a page load event. • Can be named using `aria-labelledby` if a visible label is present, otherwise with `aria-label`. • See the Main Landmark section.
`marquee`	Required	Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.

role	Necessity of Naming	Guidance
`math`	Recommended	• If the `math` element has only presentational children and the accessible name is intended to convey the mathematical expression, use `aria-label` to provide a string that represents the expression. • If the `math` element contains navigable content that conveys the mathematical expression and a visible label for the expression is present, use `aria-labelledby`. • Otherwise, use a`aria-label` to name the expression, e.g., `aria-label="Pythagorean Theorem"`.
`menu`	Recommended	• Use `aria-labelledby` to refer to the menuitem or button that controls this element's display. • Otherwise, use `aria-label`. • See the Menu or Menu bar Design Pattern.
`menubar`	Recommended	• Helps screen reader users understand the context and purpose of `menuitem` elements in a `menubar`. Naming a `menubar` is comparable to naming a menu button. The name of a `button` that opens a `menu` conveys the purpose of the menu it opens. Since a `menubar` element is displayed persistently, a name on the `menubar` can serve that same purpose. • Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`. • See the Menu or Menu bar Design Pattern.
`menuitem`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide any descendant content from assistive technologies. • Ideally named by visible, descendant content. • Note: content contained within a child `menu` is automatically excluded from the accessible name calculation. • See the Menu or Menu bar Design Pattern.
`menuitemcheckbox`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide any descendant content from assistive technologies. • Ideally named by visible, descendant content. • See the Menu or Menu bar Design Pattern.
`menuitemradio`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide any descendant content from assistive technologies. • Ideally named by visible, descendant content. • See the Menu or Menu bar Design Pattern.

role	Necessity of Naming	Guidance
navigation	Recommended	• Helps screen reader users understand the context and purpose of the navigation landmark. • Use aria-labelledby if a visible label is present, otherwise use aria-label. • See the Navigation Landmark section.
none	Do Not Name	An element with role="none" is not part of the accessibility tree (except in error cases). Do not use aria-labelledby or aria-label.
note	Discretionary	• Naming is optional, but can help screen reader users understand the context and purpose of the note. • Named using aria-labelledby if a visible label is present, otherwise with aria-label.
option	Required **Only If** Content Insufficient	• Warning! Using aria-label or aria-labelledby will hide any descendant content from assistive technologies. • Ideally named by visible, descendant content. • See the Combo Box Design Pattern.
presentation	Do Not Name	An element with role="presentation" is not part of the accessibility tree (except in error cases). Do not use aria-labelledby or aria-label.
progressbar	Required	• If the progressbar role is applied to an HTML progress element, can be named with an HTML label element. • Otherwise use aria-labelledby if a visible label is present. • Use aria-label if a visible label is not present.
radio	Required **Only If** Content Insufficient	• Warning! Using aria-label or aria-labelledby will hide any descendant content from assistive technologies. • If based on HTML type="checkbox", use a label element. • Otherwise, reference visible content via aria-labelledby.
radiogroup	Required	• Recommended to help assistive technology users understand the purpose of the group of radio buttons. • Use aria-labelledby if a visible label is present, otherwise use aria-label. • See the Radio Group Design Pattern.
region	Required	• Helps screen reader users understand the context and purpose of the landmark. • Use aria-labelledby if a visible label is present, otherwise use aria-label. • See the Region Landmark section.

role	Necessity of Naming	Guidance
row	Required **Only If** Content Insufficient **AND** descendant of a `treegrid` **AND** the row is focusable	When `row` elements are focusable in a treegrid, screen readers announce the entire contents of a row when navigating by row. This is typically the most appropriate behavior. However, in some circumstances, it could be beneficial to change the order in which cells are announced or exclude announcement of certain cells by using `aria-labelledby` to specify which cells to announce.
rowgroup	Do Not Name	Not supported by assistive technologies.
rowheader	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content. • If the `rowheader` role is implied from an HTML `th`, the HTML `abbr` attribute can be used to specify an abbreviated version of the name that is only announced when screen readers are reading an associated `cell` within the `table`, `grid`, or `treegrid`.
scrollbar	Discretionary	• Naming is optional, but can potentially help screen reader users understand the purpose of the scrollbar. The purpose is also conveyed using the `aria-controls` attribute, which is required for `scrollbar`. • Named using `aria-labelledby` if a visible label is present, otherwise with `aria-label`.
search	Recommended	• Helps screen reader users understand the context and purpose of the search landmark. • Named using `aria-labelledby` if a visible label is present, otherwise with `aria-label`. • See the Search Landmark section.
searchbox	Required	• If the `searchbox` role is applied to an HTML `input` element, can be named with an HTML `label` element. • Otherwise use `aria-labelledby` if a visible label is present. • Use `aria-label` if a visible label is not present.
separator	Discretionary	• Recommended if there is more than one focusable `separator` element on the page. • Can help assistive technology users understand the purpose of the separator. • Named using `aria-labelledby` if a visible label is present, otherwise with `aria-label`.

role	Necessity of Naming	Guidance
`slider`	Required	• If the `slider` role is applied to an HTML `input` element, can be named with an HTML `label` element. • Otherwise use `aria-labelledby` if a visible label is present. • Use `aria-label` if a visible label is not present. • See the Slider Design Pattern and the Slider (Multi-Thumb) Design Pattern.
`spinbutton`	Required	• If the `textbox` role is applied to an HTML `input` element, can be named with an HTML `label` element. • Otherwise use `aria-labelledby` if a visible label is present. • Use `aria-label` if a visible label is not present. • See the Spinbutton Design Pattern.
`status`	Discretionary	Some screen readers announce the name of a status element before announcing the content of the status element. Thus, `aria-label` provides a method for prefacing the visible content of a status element with text that is not displayed as part of the status element. Using `aria-label` is functionally equivalent to providing off-screen text in the contents of the status element, except off-screen text would be announced by screen readers that do not support `aria-label` on `status` elements.
`switch`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide any descendant content from assistive technologies. • If based on HTML `type="checkbox"`, use a `label` element. • Otherwise, reference visible content via `aria-labelledby`.
`tab`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content.
`table`	Required	• If using HTML `table` element, use the `caption` element. • Otherwise use `aria-labelledby` if a visible label is present. • Use `aria-label` if a visible label is not present. • See the Table Design Pattern.
`tablist`	Recommended	• Helps screen reader users understand the context and purpose of the tablist. • Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`. • See the Carousel Design Pattern and Tabs Design Pattern.

role	Necessity of Naming	Guidance
`tabpanel`	Required	• Use `aria-labelledby` pointing to the `tab` element that controls the `tabpanel`. • See the Carousel Design Pattern and Tabs Design Pattern.
`term`	Do Not Name	Since a term is usually the name for the `role="definition"` element, it could be confusing if the term itself also has a name.
`textbox`	Required	• If the `textbox` role is applied to an HTML `input` or `textarea` element, can be named with an HTML `label` element. • Otherwise use `aria-labelledby` if a visible label is present. • Use `aria-label` if a visible label is not present.
`timer`	Required	Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`.
`toolbar`	Recommended	• If there is more than one `toolbar` element on the page, naming is required. • Helps assistive technology users to understand the purpose of the toolbar, even when there is only one toolbar on the page. • Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`. • See the Toolbar Pattern.
`tooltip`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide descendant content from assistive technologies. • Ideally named by visible, descendant content.
`tree`	Required	• Use `aria-labelledby` if a visible label is present, otherwise use `aria-label`. • See the Tree View Design Pattern.
`treegrid`	Required	• If the `treegrid` is applied to an HTML `table` element, then the accessible name can be derived from the table's `caption` element. • Otherwise, use `aria-labelledby` if a visible label is present, otherwise use `aria-label`. • See the Treegrid Design Pattern.
`treeitem`	Required **Only If** Content Insufficient	• Warning! Using `aria-label` or `aria-labelledby` will hide any descendant content from assistive technologies. • Ideally named by visible, descendant content. • Note: content contained within a child `group` is automatically excluded from the accessible name calculation. • See the Tree View Design Pattern.

5.3.5 Accessible name calculation §

User agents construct an accessible name string for an element by walking through a list of potential naming methods and using the first that generates a name. The algorithm they follow is defined in the accessible name specification. It is roughly like the following:

1. The `aria-labelledby` property is used if present.

2. If the name is still empty, the `aria-label` property is used if present.

3. If the name is still empty, then host-language-specific attributes or elements are used if present. For HTML, these are, depending on the element:

   ↪ **`input` whose `type` attribute is in the Button, Submit Button, or Reset Button state**
   The `value` attribute.

   ↪ **`input` whose `type` attribute is in the Image Button state**
   ↪ **`img`**
   ↪ **`area`**
   The `alt` attribute.

   ↪ **`fieldset`**
   The first child `legend` element.

   ↪ **Other form elements**
   The associated `label` element(s).

   ↪ **`figure`**
   The first child `figcaption` element.

   ↪ **`table`**
   The first child `caption` element.

4. If the name is still empty, then for elements with a role that supports naming from child content, the content of the element is used.

5. Finally, if the name is still empty, then other fallback host-language-specific attributes or elements are used if present. For HTML, these are, depending on the element:

   ↪ **`input` whose `type` attribute is in the Text, Password, Search, Telephone, or URL states**
   ↪ **`textarea`**
   The `title` attribute. Otherwise, the `placeholder` attribute.

   ↪ **`input` whose `type` attribute is in the Submit Button state**
   A localized string of the word "submit".

   ↪ **`input` whose `type` attribute is in the Reset Button state**
   A localized string of the word "reset".

   ↪ **`input` whose `type` attribute is in the Image Button state**
   The `title` attribute. Otherwise, a localized string of the phrase "Submit Query".

   ↪ **`summary`**
   The word "Details".

   ↪ **Other elements**
   The `title` attribute.

The final step is a fallback mechanism. Generally when labeling an element, use one of the non-fallback mechanisms.

When calculating a name from content, the user agent walks through all descendant nodes except in the cases of `treeitem` and `menuitem` as described below. And, when following references in an `aria-labelledby` attribute, it similarly walks the tree of each referenced element. Thus, the naming algorithm is recursive. The following two sections explain non-recursive and recursive examples of how the algorithm works.

When calculating a name from content for the `treeitem` role, descendant content of child `group` elements are not included. For example, in the following `tree`, the name of the first tree item is “Fruits”; “Apples”, “Bananas”, and “Oranges” are automatically omitted.

```
<ul role="tree">
  <li role="treeitem">Fruits
    <ul role="group">
      <li role="treeitem">Apples</li>
      <li role="treeitem">Bananas</li>
      <li role="treeitem">Oranges</li>
    </ul>
  </li>
</ul>
```

Similarly, when calculating a name from content for the `menuitem` role, descendant content of child `menu` elements are not included. So, the name of the first parent `menuitem` in the following `menu` is “Fruits”.

```
<ul role="menu">
  <li role="menuitem">Fruits
    <ul role="menu">
      <li role="menuitem">Apples</li>
      <li role="menuitem">Bananas</li>
      <li role="menuitem">Oranges</li>
    </ul>
  </li>
</ul>
```

5.3.5.1 Examples of non-recursive accessible name calculation §

Consider an `input` element that has no associated `label` element and only a `name` attribute and so does not have an accessible name (do not do this):

```
<input name="code">
```

If there is a `placeholder` attribute, then it serves as a naming fallback mechanism (avoid doing this):

```
<input name="code"
       placeholder="One-time code">
```

If there is also a `title` attribute, then it is used as the accessible name instead of `placeholder`, but it is still a fallback (avoid doing this):

```
<input name="code"
       placeholder="123456"
       title="One-time code">
```

If there is also a `label` element (recommended), then that is used as the accessible name, and the `title` attribute is instead used as the accessible description:

```
<label>One-time code
 <input name="code"
        placeholder="123456"
        title="Get your code from the app.">
</label>
```

If there is also an `aria-label` attribute (not recommended unless it adds clarity for assistive technology users), then that becomes the accessible name, overriding the `label` element:

```
<label>Code
 <input name="code"
        aria-label="One-time code"
        placeholder="123456"
        title="Get your code from the app.">
</label>
```

If there is also an `aria-labelledby` attribute, that wins over the other elements and attributes (the `aria-label` attribute ought to be removed if it is not used):

```
<p>Please fill in your <span id="code-label">one-time code</span> to log in.</p>
<p>
 <label>Code
  <input name="code"
         aria-labelledby="code-label"
         aria-label="This is ignored"
         placeholder="123456"
         title="Get your code from the app.">
 </label>
</p>
```

5.3.5.2 Examples of recursive accessible name calculation §

The accessible name calculation algorithm will be invoked recursively when necessary. An `aria-labelledby` reference causes the algorithm to be invoked recursively, and when computing an accessible name from content the algorithm is invoked recursively for each child node.

In this example, the label for the button is computed by recursing into each child node, resulting in “Move to trash”.

```
<button>Move to <img src="bin.svg" alt="trash"></button>
```

When following an `aria-labelledby` reference, the algorithm avoids following the same reference twice to avoid infinite loops.

In this example, the label for the button is computed by first following the `aria-labelledby` reference to the parent element, and then computing the label for that element from the child nodes, first visiting the `button` element again but ignoring the `aria-labelledby` reference and instead using the `aria-label`, and then visiting the next child (the text node). The resulting label is “Remove meeting: Daily status report”.

```
<div id="meeting-1">
  <button aria-labelledby="meeting-1" aria-label="Remove meeting:">X</button>
  Daily status report
</div>
```

5.4 Accessible Descriptions §

5.4.1 Describing Techniques §

5.4.1.1 Describing by referencing content with `aria-describedby` §

The `aria-describedby` property works similarly to the `aria-labelledby` property. For example, a button could be described by a sibling paragraph.

```
<button aria-describedby="trash-desc">Move to trash</button>
...
<p id="trash-desc">Items in the trash will be permanently removed after 30 days.</p>
```

Descriptions are reduced to text strings. For example, if the description contains an HTML `img` element, a text equivalent of the image is computed.

```
<button aria-describedby="trash-desc"> Move to <img src="bin.svg" alt="trash"></button>
...
<p id="trash-desc">Items in <img src="bin.svg" alt="the trash"> will be permanently removed after 30
```

As with `aria-labelledby`, it is possible to reference an element using `aria-describedby` even if that element is hidden. For example, a text field in a form could have a description that is hidden by default, but can be revealed on request using a disclosure widget. The description could also be referenced from the text field directly with `aria-describedby`. In the following example, the accessible description for the `input` element is “Your username is the name that you use to log in to this service.”

```
<label for="username">Username</label>
<input id="username" name="username" aria-describedby="username-desc">
<button aria-expanded="false" aria-controls="username-desc" aria-label="Help about username">?</butt
<p id="username-desc" hidden>
  Your username is the name that you use to log in to this service.
</p>
```

5.4.1.2 Describing Tables and Figures with Captions §

In HTML, if the `table` is named using `aria-label` or `aria-labelledby`, a child `caption` element becomes an accessible description. For example, a preceding heading might serve as an appropriate accessible name, and the `caption` element might contain a longer description. In such a situation, `aria-labelledby` could be used on the `table` to set the accessible name to the heading content and the `caption` would become the accessible description.

```
<h2 id="events-heading">Upcoming events</h2>
<table aria-labelledby="events-heading">
 <caption>
  Calendar of upcoming events, weeks 27 through 31, with each week starting with
  Monday. The first column is the week number.
 </caption>
 <tr><th>Week<th>Monday<th>Tuesday<th>Wednesday<th>Thursday<th>Friday<th>Saturday<th>Sunday
 <tr><td>27<td><td><td><td><td><td><td>
 <tr><td>28<td><td><td><td><td><td><td><a href="/events/9856">Crown Princess's birthday</a>
 <tr><td>29<td><td><td><td><td><td><td>
 <tr><td>30<td><td><td><td><td><td><td>
 <tr><td>31<td><td><td><td><td><td><td>
</table>
```

The HTML `figure` element can get its accessible *name* from its `figcaption` element, but it will not be used as the accessible *description*, even if it was not used as the accessible name. If the `figcaption` element is appropriate as an accessible description, and the accessible name is set using `aria-labelledby` or `aria-label`, then the `figcaption` can be explicitly set as the accessible description using the `aria-describedby` attribute.

```
<h2 id="neutron">Neutron</h2>
<figure aria-labelledby="neutron" aria-describedby="neutron-caption">
 <img src="neutron.svg" alt="Within the neutron are three quarks (blue 'u',
 red 'd', green 'd') that are interconnected.">
 <figcaption id="neutron-caption">
  The quark content of the neutron. The color assignment of individual quarks is
  arbitrary, but all three colors must be present. Forces between quarks are
  mediated by gluons.
 </figcaption>
</figure>
```

5.4.1.3 Descriptions Derived from Titles §

If an accessible description was not provided using the `aria-describedby` attribute or one of the primary host-language-specific attributes or elements (e.g., the `caption` element for `table`), then, for HTML, if the element has a `title` attribute, that is used as the accessible description.

A visible description together with `aria-describedby` is generally recommended. If a description that is not visible is desired, then the `title` attribute can be used, for any HTML element that can have an accessible description.

Note that the `title` attribute might not be accessible to some users, in particular sighted users not using a screen reader and not using a pointing device that supports hover (e.g., a mouse).

For example, an `input` element with input constrained using the `pattern` attribute can use the `title` attribute to describe what the expected input is.

```
<label> Part number:
 <input pattern="[0-9][A-Z]{3}" name="part"
        title="A part number is a digit followed by three uppercase letters."/>
</label>
```

The `title` attribute in this case can be shown to the user as a tooltip when the user hovers or focuses the control, but also as part of the error message when the user agent validates the form, if the `input` element's value doesn't match the `pattern`.

As another example, a link can use the `title` attribute to describe the link in more detail.

```
<a href="http://twitter.com/W3C"
   title="Follow W3C on Twitter">
   <img src="/2008/site/images/Twitter_bird_logo_2012.svg"
        alt="Twitter" class="social-icon" height="40" />
</a>
```

5.4.2 Accessible description calculation §

Like the accessible name calculation, the accessible description calculation produces a text string.

The accessible description calculation algorithm is the same as the accessible name calculation algorithm except for a few branch points that depend on whether a name or description is being calculated. In particular, when accumulating text for an accessible description, the algorithm uses `aria-describedby` instead of `aria-labelledby`.

User agents construct an accessible description string for an element by walking through a list of potential description methods and using the first that generates a description. The algorithm they follow is defined in the accessible name specification. It is roughly like the following:

1. The `aria-describedby` property is used if present.
2. If the description is still empty, then host-language-specific attributes or elements are used if present, if it wasn't already used as the accessible name. For HTML, these are, depending on the element:

   ↪ **`input` whose `type` attribute is in the Button, Submit Button, or Reset Button state**
   The `value` attribute.

   ↪ **`summary`**
   The element's subtree.

   ↪ **`table`**
   The first child `caption` element.

3. Finally, if the description is still empty, then other host-language-specific attributes or elements are used if present, if it wasn't already used for the accessible name. For HTML, this is the `title` attribute.

6. Developing a Keyboard Interface §

Unlike native HTML form elements, browsers do not provide keyboard support for graphical user interface (GUI) components that are made accessible with ARIA; authors have to provide the keyboard support in their code. This section describes the principles and methods for making the functionality of a web page that includes ARIA widgets, such as menus and grids, as well as interactive components, such as toolbars and dialogs, operable with a keyboard. Along with the basics

of focus management, this section offers guidance toward the objective of providing experiences to people who rely on a keyboard that are as efficient and enjoyable as the experiences available to others.

This section covers:

1. Understanding fundamental principles of focus movement conventions used in ARIA design patterns.
2. Maintaining visible focus, predictable focus movement, and distinguishing between keyboard focus and the selected state.
3. Managing movement of keyboard focus between components, e.g., how the focus moves when the `Tab` and `Shift+Tab` keys are pressed.
4. Managing movement of keyboard focus inside components that contain multiple focusable elements, e.g., two different methods for programmatically exposing focus inside widgets like radio groups, menus, listboxes, trees, and grids.
5. Determining when to make disabled interactive elements focusable.
6. Assigning and revealing keyboard shortcuts, including guidance on how to avoid problematic conflicts with keyboard commands of assistive technologies, browsers, and operating systems.

6.1 Fundamental Keyboard Navigation Conventions §

ARIA roles, states, and properties model accessibility behaviors and features shared among GUI components of popular desktop GUIs, including Microsoft Windows, macOS, and GNOME. Similarly, ARIA design patterns borrow user expectations and keyboard conventions from those platforms, consistently incorporating common conventions with the aim of facilitating easy learning and efficient operation of keyboard interfaces across the web.

For a web page to be accessible, all interactive elements must be operable via the keyboard. In addition, consistent application of the common GUI keyboard interface conventions described in the ARIA design patterns is important, especially for assistive technology users. Consider, for example, a screen reader user operating a tree. Just as familiar visual styling helps users discover how to expand a tree branch with a mouse, ARIA attributes give the tree the sound and feel of a tree in a desktop application. So, screen reader users will commonly expect that pressing the right arrow key will expand a collapsed node. Because the screen reader knows the element is a tree, it also has the ability to instruct a novice user how to operate it. Similarly, voice recognition software can implement commands for expanding and collapsing branches because it recognizes the element as a tree and can execute appropriate keyboard commands. All this is only possible if the tree implements the GUI keyboard conventions as described in the ARIA tree pattern.

A primary keyboard navigation convention common across all platforms is that the `tab` and `shift+tab` keys move focus from one UI component to another while other keys, primarily the arrow keys, move focus inside of components that include multiple focusable elements. The path that the focus follows when pressing the `tab` key is known as the tab sequence or tab ring.

Common examples of UI components that contain multiple focusable elements are radio groups, tablists, menus, and grids. A radio group, for example, contains multiple radio buttons, each of which is focusable. However, only one of the radio buttons is included in the tab sequence. After pressing the `Tab` key moves focus to a radio button in the group, pressing arrow keys moves focus among the radio buttons in the group, and pressing the `Tab` key moves focus out of the radio group to the next element in the tab sequence.

The ARIA specification refers to a discrete UI component that contains multiple focusable elements as a composite widget. The process of controlling focus movement inside a composite is called managing focus. Following are some ARIA design patterns with example implementations that demonstrate focus management:

- Combobox
- Grid
- Listbox
- Menu or menu bar
- Radiogroup
- Tabs
- Toolbar
- Tree Grid
- Tree View

6.2 Discernible and Predictable Keyboard Focus §

Work to complete this section is tracked by issue 217.

When operating with a keyboard, two essentials of a good experience are the abilities to easily discern the location of the keyboard focus and to discover where focus landed after a navigation key has been pressed. The following factors affect to what extent a web page affords users these capabilities.

1. Visibility of the focus indicator: Users need to be able to easily distinguish the keyboard focus indicator from other features of the visual design. Just as a mouse user may move the mouse to help find the mouse pointer, a keyboard user may press a navigation key to watch for movement. If visual changes in response to focus movement are subtle, many users will lose track of focus and be unable to operate. Authors are advised to rely on the default focus indicators provided by browsers. If overriding the default, consider:
    - something about ... Colors and gradients can disappear in high contrast modes.
    - Users need to be able to easily distinguish between focus and selection as described in § 6.3 Focus VS Selection and the Perception of Dual Focus, especially when a component that contains selected elements does not contain the focus.
    - ... other considerations to be added ...
2. Persistence of focus: It is essential that there is always a component within the user interface that is active (document.activeElement is not null or is not the body element) and that the active element has a visual focus indicator. Authors need to manage events that effect the currently active element so focus remains visible and moves logically. For example, if the user closes a dialog or performs a destructive operation like deleting an item from a list, the active element may be hidden or removed from the DOM. If such events are not managed to set focus on the button that triggered the dialog or on the list item following the deleted item, browsers move focus to the body element, effectively causing a loss of focus within the user interface.
3. Predictability of movement: Usability of a keyboard interface is heavily influenced by how readily users can guess where focus will land after a navigation key is pressed. Some possible approaches to optimizing predictability include:
    - Move focus in a pattern that matches the reading order of the page's language. In left to right languages, for example, create a tab sequence that moves focus left to right and then top to bottom.
    - Incorporate all elements of a section of the page in the tab sequence before moving focus to another section. For instance, in a page with multiple columns that has content in a left side bar, center region, and right side bar, build a tab sequence that covers all elements in the left sidebar before focus moves to the first focusable element in the center column.

- When the distance between two consecutive elements in the tab sequence is significant, avoid movement that would be perceived as backward. For example, on a page with a left to right language, a jump from the last element in the bottom right of the main content to the top element in a left-hand sidebar is likely to be less predictable and more difficult to follow, especially for users with a narrow field of view.
- Follow consistent patterns across a site. The keyboard experience is more predictable when similar pages have similar focus movement patterns.
- Do not set initial focus when the page loads except in cases where:
  - The page offers a single, primary function that nearly all users employ immediately after page load.
  - Any given user is likely to use the page often.

6.3 Focus VS Selection and the Perception of Dual Focus §

Occasionally, it may appear as if two elements on the page have focus at the same time. For example, in a multi-select list box, when an option is selected it may be greyed. Yet, the focus indicator can still be moved to other options, which may also be selected. Similarly, when a user activates a tab in a tablist, the selected state is set on the tab and its visual appearance changes. However, the user can still navigate, moving the focus indicator elsewhere on the page while the tab retains its selected appearance and state.

Focus and selection are quite different. From the keyboard user's perspective, focus is a pointer, like a mouse pointer; it tracks the path of navigation. There is only one point of focus at any time and all operations take place at the point of focus. On the other hand, selection is an operation that can be performed in some widgets, such as list boxes, trees, and tablists. If a widget supports only single selection, then only one item can be selected and very often the selected state will simply follow the focus when focus is moved inside of the widget. That is, in some widgets, moving focus may also perform the select operation. However, if the widget supports multiple selection, then more than one item can be in a selected state, and keys for moving focus do not perform selection. Some multi-select widgets do support key commands that both move focus and change selection, but those keys are different from the normal navigation keys. Finally, when focus leaves a widget that includes a selected element, the selected state persists.

From the developer's perspective, the difference is simple -- the focused element is the active element (document.activeElement). Selected elements are elements that have `aria-selected="true"`.

With respect to focus and the selected state, the most important considerations for designers and developers are:

- The visual focus indicator must always be visible.
- The selected state must be visually distinct from the focus indicator.

6.4 Deciding When to Make Selection Automatically Follow Focus §

in composite widgets where only one element may be selected, such as a tablist or single-select listbox, moving the focus may also cause the focused element to become the selected element. This is called having selection follow focus. Having selection follow focus is often beneficial to users, but in some circumstances, it is extremely detrimental to accessibility.

For example, in a tablist, the selected state is used to indicate which panel is displayed. So, when selection follows focus in a tablist, moving focus from one tab to another automatically changes which panel is displayed. If the content of panels is present in the DOM, then displaying a new panel is nearly instantaneous. A keyboard user who wishes to display the fourth of six tabs can do so with 3 quick presses of the right arrow. And, a screen reader user who perceives the labels on tabs by navigating through them may efficiently read through the complete list without any latency.

However, if displaying a new panel causes a network request and possibly a page refresh, the effect of having selection automatically focus can be devastating to the experience for keyboard and screen reader users. In this case, displaying the fourth tab or reading through the list becomes a tedious and time-consuming task as the user experiences significant latency with each movement of focus. Further, if displaying a new tab refreshes the page, then the user not only has to wait for the new page to load but also return focus to the tab list.

When selection does not follow focus, the user changes which element is selected by pressing the `Enter` or `Space` key.

6.5 Keyboard Navigation Between Components (The Tab Sequence) §

As explained in section § 6.1 Fundamental Keyboard Navigation Conventions, all interactive UI components need to be reachable via the keyboard. This is best achieved by either including them in the tab sequence or by making them accessible from a component that is in the tab sequence, e.g., as part of a composite component. This section addresses building and managing the tab sequence, and subsequent sections cover making focusable elements that are contained within components keyboard accessible.

The [HTML] tabindex and [SVG2] tabindex attributes can be used to add and remove elements from the tab sequence. The value of tabindex can also influence the order of the tab sequence, although authors are strongly advised not to use tabindex for that purpose.

In HTML, the default tab sequence of a web page includes only links and HTML form elements, except In macOS, where it includes only form elements. macOS system preferences include a keyboard setting that enables the tab key to move focus to all focusable elements.

The default order of elements in the tab sequence is the order of elements in the DOM. The DOM order also determines screen reader reading order. It is important to keep the keyboard tab sequence and the screen reader reading order aligned, logical, and predictable as described in § 6.2 Discernible and Predictable Keyboard Focus. The most robust method of manipulating the order of the tab sequence while also maintaining alignment with the reading order that is currently available in all browsers is rearranging elements in the DOM.

The values of the tabindex attribute have the following effects.

tabindex is not present or does not have a valid value
: The element has its default focus behavior. In HTML, only form controls and anchors with an HREF attribute are included in the tab sequence.

tabindex="0"
: The element is included in the tab sequence based on its position in the DOM.

tabindex="-1"
: The element is not included in the tab sequence but is focusable with element.focus().

tabindex="X" where X is an integer in the range 1 <= X <= 32767
: Authors are strongly advised NOT to use these values. The element is placed in the tab sequence based on the value of tabindex. Elements with a tabindex value of 0 and elements that are focusable by default will be in the sequence after elements with a tabindex value of 1 or greater.

6.6 Keyboard Navigation Inside Components §

As described in section § 6.1 Fundamental Keyboard Navigation Conventions, the tab sequence should include only one focusable element of a composite UI component. Once a composite contains focus, keys other than `Tab` and `Shift + Tab` enable the user to move focus among its focusable elements. Authors are free to choose which keys move focus inside of a

composite, but they are strongly advised to use the same key bindings as similar components in common GUI operating systems as demonstrated in § 3. Design Patterns and Widgets.

The convention for where focus lands in a composite when it receives focus as a result of a `Tab` key event depends on the type of composite. It is typically one of the following.

- The element that had focus the last time the composite contained focus. Or, if the composite has not yet contained the focus, the first element. Widgets that usually employ this pattern include grid and tree grid.
- The selected element. Or, if there is no selected element, the first element. Widgets where this pattern is commonly implemented include radio groups, tabs, list boxes, and trees. Note: For radio groups, this pattern is referring to the checked radio button; the selected state is not supported for radio buttons.
- The first element. Components that typically follow this pattern include menubars and toolbars.

The following sections explain two strategies for managing focus inside composite elements: creating a roving tabindex and using the aria-activedescendant property.

6.6.1 Managing Focus Within Components Using a Roving tabindex §

When using roving tabindex to manage focus in a composite UI component, the element that is to be included in the tab sequence has tabindex of "0" and all other focusable elements contained in the composite have tabindex of "-1". The algorithm for the roving tabindex strategy is as follows.

- When the component container is loaded or created, set `tabindex="0"` on the element that will initially be included in the tab sequence and set `tabindex="-1"` on all other focusable elements it contains.
- When the component contains focus and the user presses a navigation key that moves focus within the component, such as an arrow key:
  - set `tabindex="-1"` on the element that has `tabindex="0"`.
  - Set `tabindex="0"` on the element that will become focused as a result of the key event.
  - Set focus, `element.focus()`, on the element that has `tabindex="0"`.
- If the design calls for a specific element to be focused the next time the user moves focus into the composite with `Tab` or `Shift+Tab`, check if that target element has `tabindex="0"` when the composite loses focus. If it does not, set `tabindex="0"` on the target element and set `tabindex="-1"` on the element that previously had `tabindex="0"`.

One benefit of using roving tabindex rather than aria-activedescendant to manage focus is that the user agent will scroll the newly focused element into view.

6.6.2 Managing Focus in Composites Using aria-activedescendant §

If a component container has an ARIA role that supports the aria-activedescendant property, it is not necessary to manipulate the tabindex attribute and move DOM focus among focusable elements within the container. Instead, only the container element needs to be included in the tab sequence. When the container has DOM focus, the value of aria-activedescendant on the container tells assistive technologies which element is active within the widget. Assistive technologies will consider the element referred to as active to be the focused element even though DOM focus is on the element that has the aria-activedescendant property. And, when the value of aria-activedescendant is changed, assistive technologies will receive focus change events equivalent to those received when DOM focus actually moves.

The steps for using the aria-activedescendant method of managing focus are as follows.

- When the container element that has a role that supports aria-activedescendant is loaded or created, ensure that:
  - The container element is included in the tab sequence as described in § 6.5 Keyboard Navigation Between Components (The Tab Sequence) or is a focusable element of a composite that implements a roving tabindex.
  - It has `aria-activedescendant="IDREF"` where IDREF is the ID of the element within the container that should be identified as active when the widget receives focus. The referenced element needs to meet the DOM relationship requirements described below.
- When the container element receives DOM focus, draw a visual focus indicator on the active element and ensure the active element is scrolled into view.
- When the composite widget contains focus and the user presses a navigation key that moves focus within the widget, such as an arrow key:
  - Change the value of aria-activedescendant on the container to refer to the element that should be reported to assistive technologies as active.
  - Move the visual focus indicator and, if necessary, scrolled the active element into view.
- If the design calls for a specific element to be focused the next time a user moves focus into the composite with `Tab` or `Shift+Tab`, check if aria-activedescendant is referring to that target element when the container loses focus. If it is not, set aria-activedescendant to refer to the target element.

The specification for aria-activedescendant places important restrictions on the DOM relationship between the focused element that has the aria-activedescendant attribute and the element referenced as active by the value of the attribute. One of the following three conditions must be met.

1. The element referenced as active is a DOM descendant of the focused referencing element.
2. The focused referencing element has a value specified for the aria-owns property that includes the ID of the element referenced as active.
3. The focused referencing element has role of textbox and has aria-controls property referring to an element with a role that supports aria-activedescendant and either:
   1. The element referenced as active is a descendant of the controlled element.
   2. The controlled element has a value specified for the aria-owns property that includes the ID of the element referenced as active.

6.7 Focusability of disabled controls §

By default, disabled HTML input elements are removed from the tab sequence. In most contexts, the normal expectation is that disabled interactive elements are not focusable. However, there are some contexts where it is common for disabled elements to be focusable, especially inside of composite widgets. For example, as demonstrated in the § 3.15 Menu or Menu bar pattern, disabled items are focusable when navigating through a menu with the arrow keys.

Removing focusability from disabled elements can offer users both advantages and disadvantages. Allowing keyboard users to skip disabled elements usually reduces the number of key presses required to complete a task. However, preventing focus from moving to disabled elements can hide their presence from screen reader users who "see" by moving the focus.

Authors are encouraged to adopt a consistent set of conventions for the focusability of disabled elements. The examples in this guide adopt the following conventions, which both reflect common practice and attempt to balance competing concerns.

1. For elements that are in the tab sequence when enabled, remove them from the tab sequence when disabled.
2. For the following composite widget elements, keep them focusable when disabled:

- Options in a Listbox
- Menu items in a Menu or menu bar
- Tab elements in a set of Tabs
- Tree items in a Tree View

3. For elements contained in a toolbar, make them focusable if discoverability is a concern. Here are two examples to aid with this judgment.
    1. A toolbar with buttons for moving, removing, and adding items in a list includes buttons for "Up", "Down", "Add", and "Remove". The "Up" button is disabled and its focusability is removed when the first item in the list is selected. Given the presence of the "Down" button, discoverability of the "Up" button is not a concern.
    2. A toolbar in an editor contains a set of special smart paste functions that are disabled when the clipboard is empty or when the function is not applicable to the current content of the clipboard. It could be helpful to keep the disabled buttons focusable if the ability to discover their functionality is primarily via their presence on the toolbar.

One design technique for mitigating the impact of including disabled elements in the path of keyboard focus is employing appropriate keyboard shortcuts as described in § 6.9 Keyboard Shortcuts.

6.8 Key Assignment Conventions for Common Functions §

The following key assignments can be used in any context where their conventionally associated functions are appropriate. While the assignments associated with Windows and Linux platforms can be implemented and used in browsers running in macOS, replacing them with macOS assignments in browsers running on a macOS device can make the keyboard interface more discoverable and intuitive for those users. In some cases, it may also help avoid system or browser keyboard conflicts.

Function	Windows/Linux Key	macOS Key
open context menu	`Shift + F10`	
Copy to clipboard	`Control + C`	`Command + C`
Paste from clipboard	`Control + V`	`Command + V`
Cut to clipboard	`Control + X`	`Command + X`
undo last action	`Control + Z`	`Command + Z`
Redo action	`Control + Y`	Command + Shift + Z

6.9 Keyboard Shortcuts §

When effectively designed, keyboard shortcuts that focus an element, activate a widget, or both can dramatically enhance usability of frequently used features of a page or site. This section addresses some of the keyboard shortcut design and implementation factors that most impact their effectiveness, including:

1. Understanding how keyboard shortcuts augment a keyboard interface and whether to make a particular shortcut move focus, perform a function, or both.
2. Making key assignments and avoiding assignment conflicts with assistive technologies, browsers, and operating systems.
3. Exposing and documenting key assignments.

6.9.1 Designing the Scope and Behavior of Keyboard Shortcuts §

This section explains the following factors when determining which elements and features to assign keyboard shortcuts and what behavior to give each shortcut:

1. Ensuring discovery through navigation; keyboard shortcuts enhance, not replace, standard keyboard access.
2. Effectively choosing from among the following behaviors:
    1. Navigation: Moving focus to an element.
    2. Activation: Performing an operation associated with an element that does not have focus and might not be visible.
    3. Navigation and activation: Both moving focus to an element and activating it.
3. Balancing efficiency and cognitive load: lack of a shortcut can reduce efficiency while too many shortcuts can increase cognitive load and clutter the experience.

6.9.1.1 Ensure Basic Access Via Navigation §

Before assigning keyboard shortcuts, it is essential to ensure the features and functions to which shortcuts may be assigned are keyboard accessible without a keyboard shortcut. In other words, all elements that could be targets for keyboard shortcuts need to be focusable via the keyboard using the methods described in:

- § 6.5 Keyboard Navigation Between Components (The Tab Sequence)
- § 6.6 Keyboard Navigation Inside Components

Do not use keyboard shortcuts as a substitute for access via navigation. This is essential to full keyboard access because:

1. The primary means of making functions and their shortcuts discoverable is by making the target elements focusable and revealing key assignments on the element itself.
2. If people who rely on the keyboard have to read documentation to learn which keys are required to use an interface, the interface may technically meet some accessibility standards but in practice is only accessible to the small subset of them who have the knowledge that such documentation exists, have the extra time available, and the ability to retain the necessary information.
3. Not all devices that depend on keyboard interfaces can support keyboard shortcuts.

6.9.1.2 Choose Appropriate Shortcut Behavior §

The following conventions may help identify the most advantageous behavior for a keyboard shortcut.

- Move focus when the primary objective is to make navigation more efficient, e.g., reduce the number of times the user must press `Tab` or the arrow keys. This behavior is commonly expected when assigning a shortcut to a text box, toolbar, or composite, such as a listbox, tree, grid, or menubar. This behavior is also useful for moving focus to a section of a page, such as the main content or a complementary landmark section.
- Activate an element without moving focus when the target context of the function is the context that contains the focus. This behavior is most common for command buttons and for functions associated with elements that are not visible, such as a "Save" option that is accessible via a menu. For example, if the focus is on an option in a listbox and a toolbar contains buttons for moving and removing options, it is most beneficial to keep focus in the listbox when the user presses a key shortcut for one of the buttons in the toolbar. This behavior can be particularly important for screen reader users because it provides confirmation of the action performed and makes performing multiple commands more

efficient. For instance, when a screen reader user presses the shortcut for the "Up" button, the user will be able to hear the new position of the option in the list since it still has the focus. Similarly, when the user presses the shortcut for deleting an option, the user can hear the next option in the list and immediately decide whether to press the delete shortcut again.

- Move focus and activate when the target of the shortcut has a single function and the context of that function is the same as the target. This behavior is typical when a shortcut is assigned to a button that opens a menu or dialog, to a checkbox, or to a navigation link or button.

6.9.1.3 Choose Where to Add Shortcuts §

Work to draft content for this section is tracked in issue 219.

The first goal when designing a keyboard interface is simple, efficient, and intuitive operation with only basic keyboard navigation support. If basic operation of a keyboard interface is inefficient, attempting to compensate for fundamental design issues, such as suboptimal layout or command structure, by implementing keyboard shortcuts will not likely reduce user frustration. The practical implication of this is that, in most well-designed user interfaces, the percentage of functionality that needs to be accessible via a keyboard shortcut in order to create optimal usability is not very high. In many simple user interfaces, keyboard shortcuts can be entirely superfluous. And, in user interfaces with too many keyboard shortcuts, the excess shortcuts create cognitive load that make the most useful ones more difficult to remember.

Consider the following when deciding where to assign keyboard shortcuts:

1. To be written.

6.9.2 Assigning Keyboard Shortcuts §

When choosing the keys to assign to a shortcut, there are many factors to consider.

- Making the shortcut easy to learn and remember by using a mnemonic (e.g., `Control + S` for "Save") or following a logical or spacial pattern.
- Localizing the interface, including for differences in which keys are available and how they behave and for language considerations that could impact mnemonics.
- Avoiding and managing conflicts with key assignments used by an assistive technology, the browser, or the operating system.

Methods for designing a key shortcut scheme that supports learning and memory is beyond the scope of this guide. Unless the key shortcut scheme is extensive, it is likely sufficient to mimic concepts that are familiar from common desktop software, such as browsers. Similarly, while localization is important, describing how to address it is left to other resources that specialize in that topic.

The remainder of this section provides guidance balancing requirements and concerns related to key assignment conflicts. It is typically ideal if key assignments do not conflict with keys that are assigned to functions in the user's operating system, browser, or assistive technology. Conflicts can block efficient access to functions that are essential to the user, and a perfect storm of conflicts can trap a user. At the same time, there are some circumstances where intentional conflicts are useful. And, given the vast array of operating system, browser, and assistive technology keys, it is almost impossible to be certain conflicts do not exist. So it is also important to employ strategies that mitigate the impact of conflicts whether they are intentional or unknown.

NOTE

In the following sections, `meta` key refers to the `Windows` key on Windows-compatible keyboards and the `Command` key on MacOS-compatible keyboards.

6.9.2.1 Operating System Key Conflicts §

It is essential to avoid conflicts with keys that perform system level functions, such as application and window management and display and sound control. In general, this can be achieved by refraining from the following types of assignments.

1. Any modifier keys + any of `Tab`, `Enter`, `Space`, or `Escape`.
2. `Meta` key + any other single key (there are exceptions, but they can be risky as these keys can change across versions of operating systems).
3. `Alt` + a function key.

In addition, there are some important application level features that most applications, including browsers, generally support. These include:

1. Zoom
2. Copy/Paste
3. ... to be continued ...

6.9.2.2 Assistive Technology Key Conflicts §

Even though assistive technologies have collectively taken thousands of key assignments, avoiding conflicts is relatively easy. This is because assistive technologies have had to develop key assignment schemes that avoid conflicts with both operating systems and applications. They do this by hijacking specific keys as modifiers that uniquely define their key commands. For example, many assistive technologies use the `Caps Lock` key as a modifier.

Deflect assistive technology key conflicts by steering clear of the following types of assignments.

1. `Caps Lock` + any other combination of keys.
2. `Insert` + any combination of other keys.
3. `Scroll Lock` + any combination of other keys.
4. macOS only: `Control+Option` + any combination of other keys.

6.9.2.3 Browser Key Conflicts §

While there is considerable similarity among browser keyboard schemes, the patterns within the schemes are less homogenous. Consequently, it is more difficult to avoid conflicts with browser key assignments. While the impact of conflicts is sometimes mitigated by the availability of two paths to nearly every function -- keyboard accessible menus and keyboard shortcuts, avoiding conflicts with shortcuts to heavily used functions is nonetheless important. Pay special attention to avoiding conflicts with shortcuts to:

1. Address or location bar

2. Notification bar
3. Page refresh
4. Bookmark and history functions
5. Find functions

6.9.2.4 Intentional Key Conflicts §

While avoiding key conflicts is usually desirable, there are circumstances where intentionally conflicting with a browser function is acceptable or even desirable. This can occur when the following combination of conditions arises:

- A web application has a frequently used function that is similar to a browser function.
- Users will often want to execute the web application function.
- Users will rarely execute the browser function.
- There is an efficient, alternative path to the browser function.

For example, consider a save function that is available when the focus is in an editor. Most browsers use ... to be continued ...

7. Grid and Table Properties §

To fully present and describe a grid or table, in addition to parsing the headers, rows, and cells using the roles described in the grid pattern or table pattern, assistive technologies need to be able to determine:

- The number of rows and columns.
- Whether any columns or rows are hidden, e.g., columns 1 through 3 and 5 through 8 are visible but column 4 is hidden.
- Whether a cell spans multiple rows or columns.
- Whether and how data is sorted.

Browsers automatically populate their accessibility tree with the number of rows and columns in a grid or table based on the rendered DOM. However, there are many situations where the DOM does not contain the whole grid or table, such as when the data set is too large to fully render. Additionally, some of this information, like skipped columns or rows and how data is sorted, cannot be derived from the DOM structure.

The below sections explain how to use the following properties that ARIA provides for grid and table accessibility.

Grid and Table Property Definitions

Property	Definition
`aria-colcount`	Defines the total number of columns in a `table`, `grid`, or `treegrid`.
`aria-rowcount`	Defines the total number of rows in a `table`, `grid`, or `treegrid`.

Property	Definition
`aria-colindex`	• Defines a cell's position with respect to the total number of columns within a `table`, `grid`, or `treegrid`. • **Note:** Numbering starts with 1, not 0.
`aria-rowindex`	• Defines a cell's position with respect to the total number of rows within a `table`, `grid`, or `treegrid`. • **Note:** Numbering starts with 1, not 0.
`aria-colspan`	Defines the number of columns spanned by a cell or gridcell within a `table`, `grid`, or `treegrid`.
`aria-rowspan`	Defines the number of rows spanned by a cell or gridcell within a `table`, `grid`, or `treegrid`.
`aria-sort`	Indicates if items in a row or column are sorted in ascending or descending order.

7.1 Using `aria-rowcount` and `aria-rowindex` §

When the number of rows represented by the DOM structure is not the total number of rows available for a table, grid, or treegrid, the `aria-rowcount` property is used to communicate the total number of rows available, and it is accompanied by the `aria-rowindex` property to identify the row indices of the rows that are present in the DOM.

The `aria-rowcount` is specified on the element with the `table`, `grid`, or `treegrid` role. Its value is an integer equal to the total number of rows available, including header rows. If the total number of rows is unknown, a value of `-1` may be specified. Using a value of `-1` indicates that more rows are available to include in the DOM without specifying the size of the available supply.

When `aria-rowcount` is used on a `table`, `grid`, or `treegrid`, a value for `aria-rowindex` property is specified on each of its descendant rows, including any header rows. The value of `aria-rowindex` is an integer that is:

1. Greater than or equal to 1.
2. Greater than the value of `aria-rowindex` on any previous rows.
3. Set to the index of the first row in the span if cells span multiple rows.
4. Less than or equal to the total number of rows.

WARNING! Missing or inconsistent values of `aria-rowindex` could have devastating effects on assistive technology behavior. For example, specifying an invalid value for `aria-rowindex` or setting it on some but not all rows in a table, could cause screen reader table reading functions to skip rows or simply stop functioning.

The following code demonstrates the use of `aria-rowcount` and `aria-rowindex` properties on a table containing a hypothetical class list.

```
<!--
  aria-rowcount tells assistive technologies the actual size of the grid
  is 463 rows even though only 4 rows are present in the markup.
-->
<table role="grid" aria-rowcount="463">
  aria-label="Student roster for history 101"
  <thead>
    <tr aria-rowindex="1">
```

```
        <th>Last Name</th>
        <th>First Name</th>
        <th>E-mail</th>
        <th>Major</th>
        <th>Minor</th>
        <th>Standing</th>
      </tr>
    </thead>
    <tbody>
        <!--
          aria-rowindex tells assistive technologies that this
          row is row 51 in the grid of 463 rows.
        -->
      <tr aria-rowindex="51">
        <td>Henderson</td>
        <td>Alan</td>
        <td>ahederson56@myuniveristy.edu</td>
        <td>Business</td>
        <td>Spanish</td>
        <td>Junior</td>
      </tr>
        <!--
          aria-rowindex tells assistive technologies that this
          row is row 52 in the grid of 463 rows.
        -->
      <tr aria-rowindex="52">
        <td>Henderson</td>
        <td>Alice</td>
        <td>ahederson345@myuniveristy.edu</td>
        <td>Engineering</td>
        <td>none</td>
        <td>Sophomore</td>
      </tr>
        <!--
          aria-rowindex tells assistive technologies that this
          row is row 53 in the grid of 463 rows.
        -->
      <tr aria-rowindex="53">
        <td>Henderson</td>
        <td>Andrew</td>
        <td>ahederson75@myuniveristy.edu</td>
        <td>General Studies</td>
        <td>none</td>
        <td>Freshman</td>
      </tr>
    </tbody>
  </table>
```

7.2 Using `aria-colcount` and `aria-colindex` §

When the number of columns represented by the DOM structure is not the total number of columns available for a table, grid, or treegrid, the `aria-colcount` property is used to communicate the total number of columns available, and it is accompanied by the `aria-colindex` property to identify the column indices of the columns that are present in the DOM.

The `aria-colcount` is specified on the element with the `table`, `grid`, or `treegrid` role. Its value is an integer equal to the total number of columns available. If the total number of columns is unknown, a value of `-1` may be specified. Using a value of `-1` indicates that more columns are available to include in the DOM without specifying the size of the available supply.

When `aria-colcount` is used on a `table`, `grid`, or `treegrid`, a value for `aria-colindex` property is either specified on each of its descendant rows or on every cell in each descendant row, depending on whether the columns are contiguous as described below. The value of `aria-colindex` is an integer that is:

1. Greater than or equal to 1.
2. When set on a cell, greater than the value set on any previous cell within the same row.
3. Set to the index of the first column in the span if a cell spans multiple columns.
4. Less than or equal to the total number of columns.

WARNING! Missing or inconsistent values of `aria-colindex` could have devastating effects on assistive technology behavior. For example, specifying an invalid value for `aria-colindex` or setting it on some but not all cells in a row, could cause screen reader table reading functions to skip cells or simply stop functioning.

7.2.1 Using `aria-colindex` When Column Indices Are Contiguous §

When all the cells in a row have column index numbers that are consecutive integers, `aria-colindex` can be set on the row element with a value equal to the index number of the first column in the set. Browsers will then compute a column number for each cell in the row.

The following code shows a grid with 16 columns, of which columns 2 through 5 are displayed to the user. Because the set of columns is contiguous, `aria-colindex` can be placed on each row.

```
<div role="grid" aria-colcount="16">
  <div role="rowgroup">
    <div role="row" aria-colindex="2">
      <span role="columnheader">First Name</span>
      <span role="columnheader">Last Name</span>
      <span role="columnheader">Company</span>
      <span role="columnheader">Address</span>
    </div>
  </div>
  <div role="rowgroup">
    <div role="row" aria-colindex="2">
      <span role="gridcell">Fred</span>
      <span role="gridcell">Jackson</span>
      <span role="gridcell">Acme, Inc.</span>
      <span role="gridcell">123 Broad St.</span>
    </div>
    <div role="row" aria-colindex="2">
      <span role="gridcell">Sara</span>
      <span role="gridcell">James</span>
      <span role="gridcell">Acme, Inc.</span>
      <span role="gridcell">123 Broad St.</span>
    </div>
```

```
    …
  </div>
</div>
```

7.2.2 Using `aria-colindex` When Column Indices Are Not Contiguous §

When the cells in a row have column index numbers that are not consecutive integers, `aria-colindex` needs to be set on each cell in the row. The following example shows a grid for an online grade book where the first two columns contain a student name and subsequent columns contain scores. In this example, the first two columns with the student name are shown, but the score columns have been scrolled to show columns 10 through 13. Columns 3 through 9 are not visible so are not in the DOM.

```
<table role="grid" aria-rowcount="463" aria-colcount="13">
  aria-label="Student grades for history 101"
  <!--
    aria-rowcount and aria-colcount tell assistive technologies
    the actual size of the grid is 463 rows by 13 columns,
    which is not the number rows and columns found in the markup.
  -->
  <thead>
    <tr aria-rowindex="1">
      <!--
        aria-colindex tells assistive technologies that the
        following columns represent columns 1 and 2 of the total data set.
      -->
      <th aria-colindex="1">Last Name</th>
      <th aria-colindex="2">First Name</th>
      <!--
        aria-colindex tells users of assistive technologies that the
        following columns represent columns 10, 11, 12, and 13 of
        the overall data set of grades.
      -->
      <th aria-colindex="10">Homework 4</th>
      <th aria-colindex="11">Quiz 2</th>
      <th aria-colindex="12">Homework 5</th>
      <th aria-colindex="13">Homework 6</th>
    </tr>
  </thead>
  <tbody>
    <tr aria-rowindex="50">
      <!--
        every cell needs to define the aria-colindex attribute
      -->
      <td aria-colindex="1">Henderson</td>
      <td aria-colindex="2">Alan</td>
      <td aria-colindex="10">8</td>
      <td aria-colindex="11">25</td>
      <td aria-colindex="12">9</td>
      <td aria-colindex="13">9</td>
    </tr>
    <tr aria-rowindex="51">
      <td aria-colindex="1">Henderson</td>
```

```
      <td aria-colindex="2">Alice</td>
      <td aria-colindex="10">10</td>
      <td aria-colindex="11">27</td>
      <td aria-colindex="12">10</td>
      <td aria-colindex="13">8</td>
    </tr>
    <tr aria-rowindex="52">
      <td aria-colindex="1">Henderson</td>
      <td aria-colindex="2">Andrew</td>
      <td aria-colindex="10">9</td>
      <td aria-colindex="11">0</td>
      <td aria-colindex="12">29</td>
      <td aria-colindex="13">8</td>
    </tr>
  </tbody>
</table>
```

7.3 Defining cell spans using `aria-colspan` and `aria-rowspan` §

For tables, grids, and treegrids created using elements other than HTML `table` elements, row and column spans are defined with the `aria-rowspan` and `aria-colspan` properties.

The value of `aria-colspan` is an integer that is:

1. Greater than or equal to 1.
2. less than the value that would cause the cell to overlap the next cell in the same row.

The value of `aria-rowspan` is an integer that is:

1. Greater than or equal to 0.
2. 0 means the cell spans all the remaining rows in its row group.
3. less than the value that would cause the cell to overlap the next cell in the same column.

The following example grid has a two row header. The first two columns have headers that span both rows of the header. The subsequent 6 columns are grouped into 3 pairs with headers in the first row that each span two columns.

```
<div role="grid" aria-rowcount="463">
  aria-label="Student grades for history 101"
  <div role="rowgroup">
    <div role="row" aria-rowindex="1">
        <!--
          aria-rowspan and aria-colspan provide
          assistive technologies with the correct data cell header information
          when header cells span more than one row or column.
        -->
        <span role="columnheader" aria-rowspan="2">Last Name</span>
        <span role="columnheader" aria-rowspan="2">First Name</span>
        <span role="columnheader" aria-colspan="2">Test 1</span>
        <span role="columnheader" aria-colspan="2">Test 2</span>
        <span role="columnheader" aria-colspan="2">Final</span>
    </div>
```

```
      <div role="row" aria-rowindex="2">
          <span role="columnheader">Score</span>
          <span role="columnheader">Grade</span>
          <span role="columnheader">Score</span>
          <span role="columnheader">Grade</span>
          <span role="columnheader">Total</span>
          <span role="columnheader">Grade</span>
      </div>
    </div>
    <div role="rowgroup">
      <div role="row" aria-rowindex="50">
        <span role="cell">Henderson</span>
        <span role="cell">Alan</span>
        <span role="cell">89</span>
        <span role="cell">B+</span>
        <span role="cell">72</span>
        <span role="cell">C</span>
        <span role="cell">161</span>
        <span role="cell">B-</span>
      </div>
      <div role="row"  aria-rowindex="51">
        <span role="cell">Henderson</span>
        <span role="cell">Alice</span>
        <span role="cell">94</span>
        <span role="cell">A</span>
        <span role="cell">86</span>
        <span role="cell">B</span>
        <span role="cell">180</span>
        <span role="cell">A-</span>
      </div>
      <div role="row"  aria-rowindex="52">
        <span role="cell">Henderson</span>
        <span role="cell">Andrew</span>
        <span role="cell">82</span>
        <span role="cell">B-</span>
        <span role="cell">95</span>
        <span role="cell">A</span>
        <span role="cell">177</span>
        <span role="cell">B+</span>
      </div>
    </div>
  </div>
```

Note: When using HTML `table` elements, use the native semantics of the `th` and `td` elements to define row and column spans by using the `rowspan` and `colspan` attributes.

7.4 Indicating sort order with `aria-sort` §

When rows or columns are sorted, the `aria-sort` property can be applied to a column or row header to indicate the sorting method. The following table describes allowed values for `aria-sort`.

Description of values for `aria-sort`

Value	Description

Value	Description
`ascending`	Data are sorted in ascending order.
`descending`	Data are sorted in descending order.
`other`	Data are sorted by an algorithm other than ascending or descending.
`none`	Default (no sort applied).

It is important to note that ARIA does not provide a way to indicate levels of sort for data sets that have multiple sort keys. Thus, there is limited value to applying `aria-sort` with a value other than `none` to more than one column or row.

The following example grid uses `aria-sort` to indicate the rows are sorted from the highest "Quiz 2" score to the lowest "Quiz 2" score.

```
<table role="grid" aria-rowcount="463" aria-colcount="13"
  aria-label="Student grades for history 101">
  <thead>
    <tr aria-colindex="10" aria-rowindex="1">
      <th>Homework 4</th>
      <!--
        aria-sort indicates the column with the heading
        "Quiz 2" has been used to sort the rows of the grid.
      -->
      <th aria-sort="descending">Quiz 2</th>
      <th>Homework 5</th>
      <th>Homework 6</th>
    </tr>
  </thead>
  <tbody>
    <tr aria-colindex="10" aria-rowindex="50">
      <td>8</td>
      <td>30</td>
      <td>9</td>
      <td>9</td>
    </tr>
    <tr aria-colindex="10"  aria-rowindex="51">
      <td>10</td>
      <td>29</td>
      <td>10</td>
      <td>8</td>
    </tr>
    <tr aria-colindex="10"  aria-rowindex="52">
      <td>9</td>
      <td>9</td>
      <td>27</td>
      <td>6</td>
    </tr>
    <tr aria-colindex="10"  aria-rowindex="53">
      <td>9</td>
      <td>10</td>
      <td>26</td>
      <td>8</td>
    </tr>
    <tr aria-colindex="10"  aria-rowindex="54">
```

```
        <td>9</td>
        <td>7</td>
        <td>24</td>
        <td>6</td>
      </tr>
    </tbody>
  </table>
```

8. Intentionally Hiding Semantics with the `presentation` Role §

While ARIA is primarily used to express semantics, there are some situations where hiding an element's semantics from assistive technologies is helpful. This is done with the presentation role, which declares that an element is being used only for presentation and therefore does not have any accessibility semantics. The ARIA 1.1 specification also includes role none, which serves as a synonym for `presentation`.

For example, consider a tabs widget built using an HTML `ul` element.

```
<ul role="tablist">
  <li role="presentation">
    <a role="tab" href="#">Tab 1</a>
  </li>
  <li role="presentation">
    <a role="tab" href="#">Tab 2</a>
  </li>
  <li role="presentation">
    <a role="tab" href="#">Tab 3</a>
  </li>
</ul>
```

Because the list is declared to be a tablist, the list items are not in a list context. It could confuse users if an assistive technology were to render those list items. Applying role `presentation` to the `li` elements tells browsers to leave those elements out of their accessibility tree. Assistive technologies will thus be unaware of the list item elements and see the tab elements as immediate children of the tablist.

Three common uses of role `presentation` are:

1. Hiding a decorative image; it is equivalent to giving the image null alt text.
2. Suppressing table semantics of tables used for layout in circumstances where the table semantics do not convey meaningful relationships.
3. Eliminating semantics of intervening orphan elements in the structure of a composite widget, such as a tablist, menu, or tree as demonstrated in the example above.

8.1 Effects of Role `presentation` §

When `role="presentation"` is specified on an element, if a condition that requires a browser to ignore the `presentation` role does not exist, it has the following three effects.

1. The element's implied ARIA role and any ARIA states and properties associated with that role are hidden from assistive technologies.

2. Text contained by the element, i.e., inner text, as well as inner text of all its descendant elements remains visible to assistive technologies except, of course, when the text is explicitly hidden, e.g., styled with `display: none` or has `aria-hidden="true"`.
3. The roles, states, and properties of each descendant element remain visible to assistive technologies unless the descendant requires the context of the presentational element. For example:
    - If `presentation` is applied to a `ul` or `ol` element, each child `li` element inherits the `presentation` role because ARIA requires the `listitem` elements to have the parent `list` element. So, the `li` elements are not exposed to assistive technologies, but elements contained inside of those `li` elements, including nested lists, are visible to assistive technologies.
    - Similarly, if `presentation` is applied to a `table` element, the descendant `caption`, `thead`, `tbody`, `tfoot`, `tr`, `th`, and `td` elements inherit role `presentation` and are thus not exposed to assistive technologies. But, elements inside of the `th` and `td` elements, including nested tables, are exposed to assistive technologies.

8.2 Conditions That Cause Role `presentation` to be Ignored §

Browsers ignore `role="presentation"`, and it therefore has no effect, if either of the following are true about the element to which it is applied:

- The element is focusable, e.g. it is natively focusable like an HTML link or input, or it has a `tabindex` attribute.
- The element has any of the twenty-one global ARIA states and properties, e.g., `aria-label`.

8.3 Example Demonstrating Effects of the `presentation` Role §

This code:

```
<ul role="presentation">
  <li>Date of birth:</li>
  <li>January 1, 3456</li>
</ul>
```

when parsed by a browser, is equivalent to the following from the perspective of a screen reader or other assistive technology that relies on the browser's accessibility tree:

```
<div>Date of birth:</div>
  <div>January 1, 3456</div>
```

9. Roles That Automatically Hide Semantics by Making Their Descendants Presentational §

There are some types of user interface components that, when represented in a platform accessibility API, can only contain text. For example, accessibility APIs do not have a way of representing semantic elements contained in a button. To deal with this limitation, WAI-ARIA requires browsers to automatically apply role `presentation` to all descendant elements of any element with a role that cannot support semantic children.

The roles that require all children to be presentational are:

- button
- checkbox
- img
- math
- menuitemcheckbox
- menuitemradio
- option
- progressbar
- radio
- scrollbar
- separator
- slider
- switch
- tab

For instance, consider the following tab element, which contains a heading.

```
<li role="tab"><h3>Title of My Tab</h3></li>
```

Because WAI-ARIA requires descendants of tab to be presentational, the following code is equivalent.

```
<li role="tab"><h3 role="presentation">Title of My Tab</h3></li>
```

And, from the perspective of anyone using a technology that relies on an accessibility API, such as a screen reader, the heading does not exist since the previous code is equivalent to the following.

```
<li role="tab">Title of My Tab</li>
```

See the section about role `presentation` for a detailed explanation of what it does.

A. Indexes §

- Design Pattern Examples by Role
- Design Pattern Examples by Properties and States

B. Change History §

B.1 Changes in July 2019 Publication of Note Release 4 §

- Major additions and revisions:
  - Added section providing guidance on coding and composing accessible names and descriptions. This section comprehensively covers ARIA and HTML naming and describing techniques as well as specific guidance for

every ARIA role.

  - Added example of a site navigation bar (“menu system”) with dropdown lists of links coded using the disclosure pattern.
  - Added an example of a date picker based on the dialog and grid patterns. A modal dialog contains a calendar grid that presents buttons for each day of a month.
  - Added an example of a date picker that uses three spin buttons for day, month, and year.
  - Image Carousel Example: Updated the example so that the rotation control is always visible instead of only being visible for keyboard users who move focus into the carousel region. Also changed the controls so they are implemented with HTML button elements instead of links with the `button` role.
  - Toolbar Example: Added popup text labels for icons that are displayed on both hover and focus.
- Minor revisions of guidance or implementation were made to the following design patterns and examples:
  - Editor Menubar Example: Updated example so that meta data used by the scripts is specified using `data-` attributes instead of with `rel` attributes.
  - Radio Group with Active Descendant Example: Removed unused keycodes and fixed linting issues.

Also see:

- The APG 1.1 Release 4 Milestone: which lists the GitHub issues that document discussions and reviews of changes included in the July 2019 publication.
- Detailed change log with links to all commits that are new in the July 2019 publication since the January 2019 publication.

B.2 Changes in January 2019 Publication of Note Release 3 §

- Major additions and revisions:
  - Added indexes that allow lookup of examples by role, state, or property.
  - Added a carousel design pattern and example implementation.
  - Revised radio group pattern to accommodate radio groups nested in toolbars: left and right arrow keys do not change the checked state and can move focus outside the group.
  - Redesigned toolbar example to demonstrate an editor toolbar that includes a nested radio group, toggle buttons, menu button, spin button, checkbox, and link. Also fixed several bugs.
  - Restructured the accordion example to use HTML heading elements containing HTML button elements.
  - Added a regression test framework and suite of regression tests that test all documented keyboard behaviors, roles, states, and properties for each example implementation of a pattern.
- Minor revisions of guidance or implementation were made to the following design patterns and examples:
  - Alert dialog example: Replaced lorem ipsum placeholder with text that describes actions that trigger the alert dialog and alert message.
  - Button Example: Button activation now happens on keyup for space key and aria-hidden was removed from child svg.
  - Legacy Combobox examples: Fixed enter key documentation, made Up Arrow set activedescendant to last match in list, corrected up arrow documentation for the textbox.
  - Dialog Examples: Fixed IE11 incompatibility in role validity check.

- Single-select listbox example: Fix a bug with aria-selected.
- Grid Examples: Fixed how event listeners were cleaned up.
- Menubar Examples: Applied role="none" where missing from elements in menubar and changed visual design to better distinguish between menuitemradio and meuitemcheckbox elements.
- Radio Group Example: Fixed documentation of elements used in attributes table.
- Spin Button Pattern: Added aria-invalid guidance for values outside allowed range.
- Toolbar Pattern: Clarified keyboard guidance for Tab and Shift+Tab and revised description to remove guidance that conflicted with updated radio group pattern.

Also see:

- The APG 1.1 Release 3 Milestone: which lists the GitHub issues that document discussions and reviews of changes included in the January 2019 publication.
- Detailed change log with links to all commits that are new in the January 2019 publication since the July 2018 publication.

B.3 Changes in July 2018 Publication of Note Release 2 §

- Added the following:
  - Treegrid design pattern and example
  - Alert dialog example
- Significant revisions of guidance or implementation were made to the following design patterns and examples:
  - Accordion pattern and example, including removing two optional key commands from the pattern.
  - Checkbox example bug fixes
  - ARIA 1.0 Combobox example, including escape key behavior
  - ARIA 1.1 Combobox example labeling
  - Modal dialog example, primarily code refactoring when adding alert dialog
  - Grid pattern, including guidance on column selection
  - Listbox examples
  - Menu pattern, including guidance on menu/submenu structure
  - editor menubar example, including significant improvement to mouse behaviors.
  - Menu button example bug fixes
  - Navigation menubar example bug fixes
  - Radio group example styling
  - Tabs pattern, including labeling guidance
  - Tree view pattern, including multi-select guidance
  - Tree view example bug fixes

Also see:

- The APG 1.1 Release 2 Milestone: which lists the GitHub issues that document discussions and reviews of changes included in the July 2018 publication.
- Detailed change log with links to all commits that are new in the July 2018 publication since the December 2017 publication of the Note.

B.4 Changes in December 2017 Publication as Note §

- Added the following:
  - Read Me First section
  - Combobox pattern
  - Combobox examples: 3 ARIA 1.0 style and 4 ARIA 1.1 style
  - Disclosure pattern
  - Feed example display page
  - Grid and table properties guidance section
  - Collapsible dropdown listbox example
  - Multi-thumb slider examples
  - Table pattern and example
- The top of each example page now includes a set of four related links to:
  - “Browser and Assistive Technology Support” section of “Read Me First”
  - “Report Issue” page in the Github repository
  - “Related Issues” listed in the Github project for the example
  - “Design Pattern” section that applies to the example
- All Javascript and CSS files used by the examples include the correct license and copyright statements.
- Significant revisions of guidance and implementation were made to the following sections, design patterns, and examples:
  - Introduction
  - Keyboard guidance for Mac OS
  - Accordion example
  - Mixed checkbox examples
  - Scrollable layout grid example
  - Listbox examples
  - Modal dialog example
  - editor menubar example
  - Navigation menubar example
  - Menu button examples
  - Spinbutton pattern
  - Toolbar example
  - Tree view examples

Also see:

- 1.1 APG Release 1 milestone, which lists the GitHub issues that document discussions and reviews of changes included in the December 2017 publication.
- Detailed change log with links to all commits that are new in the December 2017 publication as a Note since the June 2017 working draft.

B.5 Changes in June 2017 Working Draft §

- Added the following:
  - Modal dialog example
  - Disclosure design pattern
  - Example disclosure for FAQ
  - Example disclosure for image description
  - Draft of feed pattern
  - Example feed implementation
  - Example of menu button using aria-activedescendant
  - Example of tabs with manual activation
- Design pattern section: moved examples subsection of each design pattern to be the first subsection.
- Across all example pages:
  - Improved visual design of tables that document keyboard implementation, roles, states, and properties.
  - Improved consistency of editorial style.
- Significant revisions of guidance and implementation were made to the following design patterns and example pages:
  - Accordion example
  - Alert example
  - Breadcrumb example
  - Button example
  - Checkbox examples
  - Dialog (modal) design pattern
  - Grid examples
  - Landmark examples
  - Link example
  - Listbox example
  - Menubar examples
  - Menu button examples
  - Radio group example
  - Slider design pattern
  - Slider example

- Example of tabs with automatic activation
- Tree view examples

Also see:

- January 2017 Clean Up Milestone, which lists the GitHub issues that document discussions and reviews of changes included in the June 2017 working draft.
- Detailed change log with links to all commits that are new in the June 2017 working draft.

C. Acknowledgements §

C.1 Major Contributors to Version 1.1 §

While WAI-ARIA Authoring Practices 1.1 is the work of the entire Authoring Practices Task Force and also benefits from many people throughout the open source community who both contribute significant work and provide valuable feedback, special thanks goes to the following people who provided distinctly large portions of the content and code in version 1.1.

- Jon Gunderson and Nicholas Hoyt of the Division of Disability Resources and Education Services at the University of Illinois Urbana/Champaign and the students Max Foltz, Sulaiman Sanaullah, Mark McCarthy, and Jinyuan Zhou for their contributions to the development of many of the design pattern examples.
- Valerie Young of Bocoup and her sponsor, Facebook, for development of the example test framework and regressions tests for more than 50 examples.
- Simon Pieters of Bocoup and his sponsor, Facebook, for authoring of significant guidance sections, including comprehensive treatement of the topic of accessible names and descriptions.

C.2 Participants active in the ARIA Authoring Practices Task Force §

- Ann Abbott (Invited Expert)
- Shirisha Balusani (Microsoft Corporation)
- Dorothy Bass (Wells Fargo Bank N.A.)
- Curt Bellew (Oracle)
- Zoë Bijl (Invited Expert)
- Michael Cooper (W3C)
- Bryan Garaventa (Level Access)
- Jon Gunderson (University of Illinois at Urbana-Champaign)
- Jesse Hausler(Salesforce)
- Sarah Higley (Microsoft Corporation)
- Hans Hillen (The Paciello Group, LLC)
- Matt King (Facebook)
- Jaeun Ku (University of Illinois at Urbana-Champaign)
- Aaron Leventhal (Google)

- Carolyn MacLeod (IBM Corporation)
- Mark McCarthy (University of Illinois at Urbana-Champaign)
- James Nurthen (Adobe)
- Scott O'Hara (The Paciello Group, LLC)
- Simon Pieters (Bocoup)
- Scott Vinkle (Shopify)
- Evan Yamanishi (W. W. Norton)
- Valerie Young (Bocoup)

C.3 Other commenters and contributors to Version 1.1 §

- Vyacheslav Aristov
- J. Renée Beach
- Kasper Christensen
- Gerard K. Cohen
- Anne-Gaelle Colom
- Kevin Coughlin
- Cameron Cundiff
- Manish Dahamiwal
- Gilmore Davidson
- Boris Dušek
- Michael Fairchild
- Jeremy Felt
- Rob Fentress
- Geppy
- Tatiana Iskandar
- Patrick Lauke
- Marek Lewandowski
- Dan Matthew
- Shane McCarron
- Victor Meyer
- Jonathan Neal
- Philipp Rudloff
- Joseph Scheuhammer
- Nick Schonning
- thomascorthals
- Christopher Tryens

C.4 Enabling funders §

This publication has been funded in part with U.S. Federal funds from the Department of Education, National Institute on Disability, Independent Living, and Rehabilitation Research (NIDILRR), initially under contract number ED-OSE-10-C-0067 and currently under contract number HHSP23301500054C. The content of this publication does not necessarily reflect the views or policies of the U.S. Department of Education, nor does mention of trade names, commercial products, or organizations imply endorsement by the U.S. Government.

D. References §

D.1 Informative references §

[HTML]

HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[HTML-AAM]

HTML Accessibility API Mappings 1.0. Steve Faulkner; Alexander Surkov; Scott O'Hara; Bogdan Brinza; Jason Kiss; Cynthia Shelly. W3C. 10 July 2019. W3C Working Draft. URL: https://www.w3.org/TR/html-aam-1.0/

[HTML-ARIA]

ARIA in HTML. Steve Faulkner. W3C. 5 July 2019. W3C Working Draft. URL: https://www.w3.org/TR/html-aria/

[SVG2]

Scalable Vector Graphics (SVG) 2. Amelia Bellamy-Royds; Bogdan Brinza; Chris Lilley; Dirk Schulze; David Storey; Eric Willigers. W3C. 4 October 2018. W3C Candidate Recommendation. URL: https://www.w3.org/TR/SVG2/

[WAI-ARIA]

Accessible Rich Internet Applications (WAI-ARIA) 1.1. Joanmarie Diggs; Shane McCarron; Michael Cooper; Richard Schwerdtfeger; James Craig. W3C. 14 December 2017. W3C Recommendation. URL: https://www.w3.org/TR/wai-aria-1.1/

↑