Session 5

Building accessible widgets

Slide instructions

`SPACEBAR` to **move forward** through slides.

`SHIFT` & `SPACEBAR` to **move backwards** through slides.

`LEFT ARROW` & `RIGHT ARROW` to **move through sections**.

`ESC` to **see overview** and `ESC` again to exit.

`F` to **enter presentation mode** and `ESC` to exit.

Introduction

Forms are great for communication with our users.
Widgets are uses **for other types of interactivity**.

“Widgets” can be anything from a checkbox to a drop-down, a progress loader or a date picker - any self-contained UI component.

Widgets are often created using a combination of **semantic and non-semantic elements**.

For this reason, we need to **make sure we provide all information to the accessibility API**.

Today we'll use a **widget accessibility checklist** and work through some common examples.

What are non-native widgets?

“Non-native” means that the widget has been built using one or more elements in a different way than the intended purpose.

1. A widget that has been built using elements **in a different way than the intended purpose**.
e.g. a button being used for a dropdown

2. A widget where there are **no native elements available** that could be used to build the widget.
e.g. date pickers, accordions, carousels

Let’s look at a native vs non-native example and **see some of the differences in their accessibility**.

A native example

If we use a simple drop-down input as an example, a **native solution** would involve using the `<label>`, `<select>` and `<option>` elements.

```
<label for="fruit">Favourite fruit</label>
<select id="fruit">
  <option>Choose an option</option>
  <option>Apple</option>
  <option>Apricot</option>
  <option>Avocado</option>
</select>
```

1. Role

The `<label>`, `<select>` and `<option>` elements each have a specific semantic meaning that is **understood by accessibility APIs**.

Element	Role	Description
`<label>`	`LabelText`	A label for form controls
`<select>`	`combobox`	A selection list within a form
`<option>`	`menuitem`	An option in a selection list

2. Name

If the `<label>` and `<select>` elements are given matching `for` and `id` values, the `<select>` will then **have an accessible name**.

```
<label for="fruit">Favourite fruit</label>
<select id="fruit">
  <option>Choose an option</option>
  <option>Apple</option>
  <option>Apricot</option>
</select>
```

More importantly, the visible text label and accessible name (the name in the accessibility tree) **should match**.

3. Properties

The `<select>` element is **defined with a property** of `hasPopup: menu` in the accessibility tree.

Interactive components that **appear on top of other content when triggered** to appear are considered "popups".

The presence of `haspopup` indicates the element **can trigger a popup**.

Value	**Description**
`false` (default)	The element does not have a popup
`true`	The popup is a menu
`menu`	The popup is a menu
`listbox`	The popup is a listbox
`tree`	The popup is a tree
`grid`	The popup is a grid
`dialog`	The popup is a dialog

4. Current state

The `<select>` element will have a state of `expanded: false` **until it is expanded by the user**.

5. Current value

If an `<option>` is selected it will be **defined in the accessibility tree as the value** - i.e. “Apple”.

6. Keyboard accessible

There are also a range of **pre-defined keystrokes that can be used to interact with** the `<select>` and `<option>` elements.

Scorecard?

- **Role**: `Combobox`.
- **Name**: “Choose your favourite fruit”.
- **Properties**: `hasPopup: menu`.
- **Current state**: `Expanded: false/true`.
- **Current value**: “Apple”.
- **Keyboard accessible**: Yes.

Any questions or comments?

A non-native example

An example of **a non-native component** would be to use the `<button>` and `<ul>` elements to create a dropdown.

```
<span>Choose your favourite fruit</span>
<button>Apple</button>
<ul>
  <li>Apple</li>
  <li>Apricot</li>
  <li>Avocado</li>
</ul>
```

1. Role

The `<button>` element will be defined in the accessibility tree as `button` which is **incorrect in this case.**

2. Name

This component **will have an accessible name** of "Choose your favourite fruit" in the accessibility tree, which is acceptable.

3. Properties

There no **no additional properties** assigned to the elements to provide additional context.

4. Current state

There is **no native way** to inform users about the dropdown’s current state.

4. Current value

There is **no native way** to inform users of the currently selected value.

5. Keyboard accessible

The component **will have no native keystrokes defined**. So, it is not keyboard accessible.

Scorecard?

- **Role**: `button` (Incorrect).
- **Name**: “Choose your favourite fruit”.
- **Properties**: Not available.
- **Current state**: Not available.
- **Current value**: Not available.
- **Keyboard accessible**: No.

All of these problems can fixed using a combination of ARIA and JavaScript. **But it takes work**.

Any questions or comments?

Criteria for accessible widgets

1. Are all **roles** defined?
2. Does it have an **accessible name**?
3. Are all relevant **properties** defined?
4. Are all **states** defined?
5. Is the current **value** defined?
6. Does the component work using **keyboard-only**?
7. Are all visible **states** clearly defined? (focus, hover, active, checked)
8. Is **focus order** managed correctly?
9. Does the component have any **dynamical content**?

Any questions or comments?

Exercise: Clickatron

Accessing the exercise:

https://codepen.io/intopia/pen/WNdEqJw

Your goal: make a fully accessible button from a `<div>` element. When time's up, we'll score your work out of 12.

1 point: your `<div>` looks like a button.

1 point: your `<div>` has a role of button in the accessibility API.

1 point: your `<div>` increments the second counter when clicked with a mouse.

1 point: your `<div>` can be focused when navigating the page with the `Tab` key.

1 point: your `<div>` increments the counter with the `Enter` key.

1 point: your `<div>` increments the counter with the `Space` key.

1 point: your `<div>` does not increment the second counter at all while the checkbox is checked.

1 point: your `<div>` switches between enabled and disabled in the accessibility API when the checkbox is toggled.

1 point: your `<div>` has visually distinct hover, focus and active styles.

1 point: your `<div>` cannot have its text selected.

One solution:

https://codepen.io/stringyland/pen/qBpjqmP

Did anyone reach the stretch goal?

If so, listen to the difference between how the div and the button are announced when disabled. The `disabled` attribute doesn't work on `<div>` elements even with `role="button"`.

Whenever possible, use native HTML form elements as the foundation of your widgets. They give you free accessibility support, and more time for custom behaviours.

Any questions or comments?

Meaningful sequencing for widgets

When we're pulling elements together to make a widget, we need to make sure the **keyboard focus path is logical**.

The keyboard focus path will follow the **order the elements are placed in the DOM**.

Any questions or comments?

Focus management for widgets

Focus management is **important to consider when building any widget**. Let's look at some examples.

When we're adding or revealing content, we usually **send focus to the newly injected content**.

When we're removing or hiding content, we usually **return focus to the element that triggered the removal**.

If the content is being rearranged, we may need to **keep focus in place** but include a status update.

Any questions or comments?

Accessible modals

Trigger element

Modals should be triggered using the `<button>` element rather than the `<a>` element, as **users are performing an action**, not going to a new page.

Focus

When the modal is triggered, **focus should be sent to either**:

- The first heading inside the modal. This is preferred. OR
- The first focusable element inside the modal.

When the modal is closed, **focus should shift to either**:

- The element that triggered the modal, OR
- Content that has changed after actions within the modal.

Keyboard

Users should **not be able to** `TAB` or `SHIFT` + `TAB` outside the modal. So, these keystrokes should be trapped inside the modal.

Users should be able to use the `ESC` key to **close the modal and return to the page below**.

Users must have the ability to **close the modal** using a `<button>` element.

Ideally, this should be the **first focusable element inside the modal**, even if there is a close function at the bottom of a form.

Screen readers

When the modal is triggered, **three things should happen**:

1. The `role` is announced.
2. An accessible name is announced.
3. An accessible description is announced (optional).

Any questions or comments?

Exercise: Creating an accessible modal

Accessing the exercise:

DEVELOPER EXAMPLE: Creating an accessible modal

Step 1:

Apply `role="dialog"` to parent container.

```
<div
  id="myDialog"
  tabindex="0"
  onkeydown="escapeMe(event)"
  role="dialog"
>
  <div id="modal-content">
    <div>
      <h2>Contact details</h2>
    </div>
    <p>Make sure to...</p>
    <button>Close</button>
  </div>
</div>
```

Step 2:

Apply `aria-labelledby="heading"` to parent.

```
<div
  id="myDialog"
  tabindex="0"
  onkeydown="escapeMe(event)"
  role="dialog"
  aria-labelledby="heading"
>
  <div id="modal-content">
    <div>
      <h2>Contact details</h2>
    </div>
    <p>Make sure to...</p>
    <button>Close</button>
  </div>
</div>
```

Step 3:

Apply `id="heading"` to `<div>` around the heading.

```
<div
  id="myDialog"
  tabindex="0"
  onkeydown="escapeMe(event)"
  role="dialog"
  aria-labelledby="heading"
>
  <div id="modal-content">
    <div id="heading">
      <h2>Contact details</h2>
    </div>
    <p>Make sure to...</p>
    <button>Close</button>
  </div>
</div>
```

Step 4:

Apply `aria-describedby="intro"` to parent.

```
<div
  id="myDialog"
  tabindex="0"
  onkeydown="escapeMe(event)"
  role="dialog"
  aria-labelledby="heading"
  aria-describedby="intro"
>
  <div id="modal-content">
    <div id="heading">
      <h2>Contact details</h2>
    </div>
    <p>Make sure to...</p>
    <button>Close</button>
  </div>
</div>
```

Step 5:

Apply `id="intro"` to paragraph.

```
<div
  id="myDialog"
  tabindex="0"
  onkeydown="escapeMe(event)"
  role="dialog"
  aria-labelledby="heading"
  aria-describedby="intro"
>
  <div id="modal-content">
    <div id="heading">
      <h2>Contact details</h2>
    </div>
    <p id="intro">Make sure to...</p>
    <button>Close</button>
  </div>
</div>
```

Now **trigger the modal and listen to how it is announced** using a screen reader.

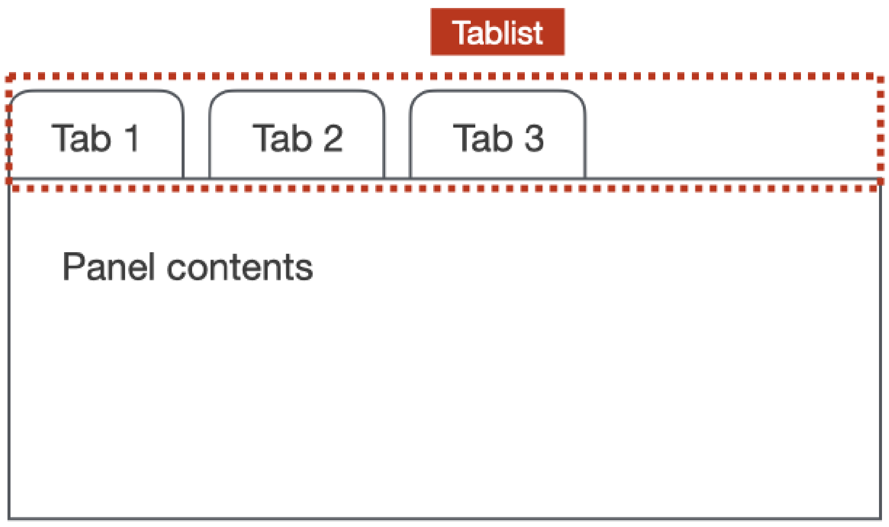
Accessible in-page tabs

Three roles work together when **defining in-page tabs**:

- `tablist`.
- `tab`.
- `tabpanel`.

tablist

The `tablist` role **defines the parent element** for a list of tabs.

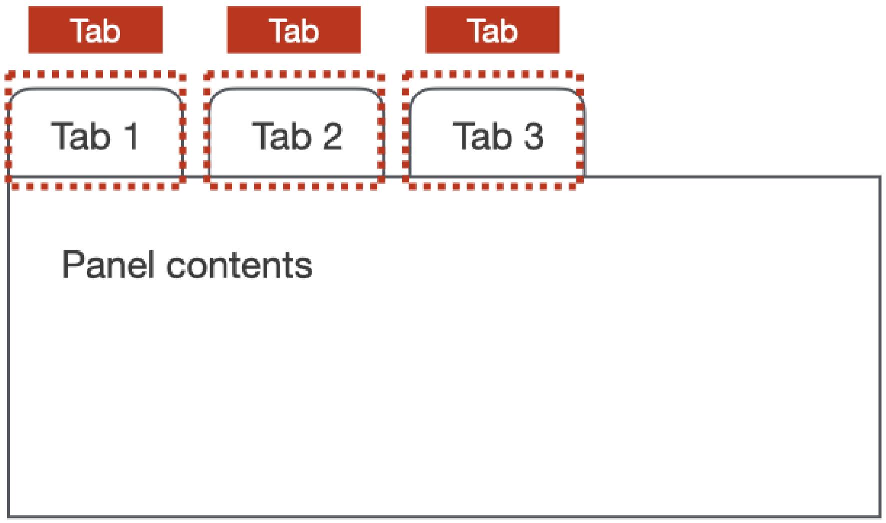


```html
<div role="tablist">
  <button role="tab">One</button>
  <button role="tab">Two</button>
</div>

<div role="tabpanel"></div>
<div role="tabpanel"></div>
```

tab

The `tab` role **define elements that act as tabs**, used to control the visibility of individual tabpanel elements. Each tab must be contained in a tablist.

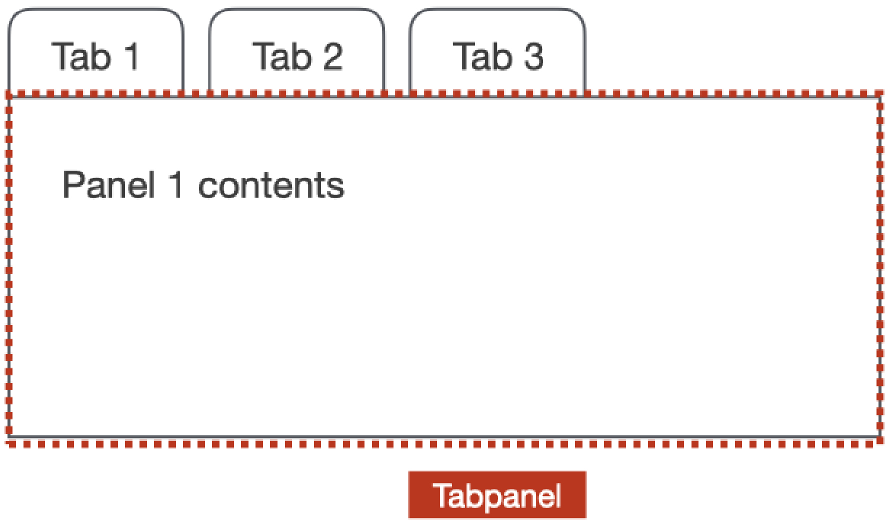


```
<div role="tablist">
  <button role="tab">One</button>
  <button role="tab">Two</button>
</div>

<div role="tabpanel"></div>
<div role="tabpanel"></div>
```

tabpanel

The `tabpanel` is a **container for the content associated with a tab**. These containers are made visible when tab elements are activated.



```
<div role="tablist">
  <button role="tab">One</button>
  <button role="tab">Two</button>
</div>

<div role="tabpanel"></div>
<div role="tabpanel"></div>
```

aria-controls

The `aria-controls` attribute identifies the element(s) that are **controlled by the current element**, when that relationship isn't represented in the DOM.

This attribute is **applied to the controlling element**, and is literally saying:
"This element currently controls the [ID] element".

An example we will see soon is where the `tab` elements are **used to control** a series of `tabpanel` elements.

```
<div role="tablist">
  <button role="tab" aria-controls="p1">One</butto
  <button role="tab" aria-controls="p2">Two</butto
</div>

<div role="tabpanel" id="p1"></div>
<div role="tabpanel" id="p2"></div>
```

Any questions or comments?

Exercise: Making accessible in-page tabs

Accessing the exercise:

DEVELOPER EXERCISE: Making accessible in-page tabs

Step 1: The `tablist` element needs to be given an accessible name.

```
<div role="tablist" aria-label="Animal types">
</div>
```

Step 2: Set `aria-selected="true"` on the first `<button>`.

```
<button
  role="tab"
  aria-selected="true"
>
  Mammals
</button>
```

Step 3: Set `aria-selected="false"` on other `<button>` elements.

```
<button
  role="tab"
  aria-selected="false"
>
  Birds
</button>
```

```
<button
  role="tab"
  aria-selected="false"
>
  Fish
</button>
```

Note: These states will need to be **changed dynamically using JavaScript** as each `<button>` is selected.

Step 4: Set `aria-controls` on all `<button>` elements.

```
<button
  role="tab"
  aria-selected="true"
  aria-controls="tabpanel1"
>
  Mammals
</button>
```

```
<button
  role="tab"
  aria-selected="false"
  aria-controls="tabpanel2"
>
  Birds
</button>
```

```
<button
  role="tab"
  aria-selected="false"
  aria-controls="tabpanel3"
>
  Fish
</button>
```

Step 5: Set matching `id` values on all `tabpanel` elements.

```
<div
  role="tabpanel"
  id="tabpanel1"
>
</div>
```

```
<div
  role="tabpanel"
  id="tabpanel2"
>
</div>
```

```
<div
  role="tabpanel"
  id="tabpanel3"
>
</div>
```

Step 6: Set all `tabpanel` elements with `tabindex="0"`.

```
<div
  role="tabpanel"
  id="tabpanel1"
  tabindex="0"
>
</div>
```

```
<div
  role="tabpanel"
  id="tabpanel2"
  tabindex="0"
>
</div>
```

```
<div
  role="tabpanel"
  id="tabpanel3"
  tabindex="0"
>
</div>
```

Step 7: Set `class=""` on the first `tabpanel` and `class="is-hidden"` on the second and third.

```html
<div
  role="tabpanel"
  id="tabpanel1"
  tabindex="0"
  class=""
>
</div>
```

```html
<div
  role="tabpanel"
  id="tabpanel2"
  tabindex="0"
  class="is-hidden"
>
</div>
```

```
<div
  role="tabpanel"
  id="tabpanel3"
  tabindex="0"
  class="is-hidden"
>
</div>
```

Note: These values will need to be **changed dynamically using JavaScript** as each panel becomes visible.

Step 8: Set `aria-labelledby` on all `tabpanel` elements.

```
<div
  role="tabpanel"
  id="tabpanel1"
  tabindex="0"
  aria-labelledby="tab1"
>
</div>
```

```
<div
  role="tabpanel"
  id="tabpanel2"
  tabindex="0"
  hidden
  aria-labelledby="tab2"
>
</div>
```

```
<div
  role="tabpanel"
  id="tabpanel3"
  tabindex="0"
  hidden
  aria-labelledby="tab3"
>
</div>
```

Step 9: Set matching `id` values on the all `tab` elements.

```
<button
  role="tab"
  aria-selected="true"
  aria-controls="tabpanel1"
  id="tab1"
>
  Mammals
</button>
```

```
<button
  role="tab"
  aria-selected="false"
  aria-controls="tabpanel2"
  id="tab2"
>
  Birds
</button>
```

```
<button
  role="tab"
  aria-selected="false"
  aria-controls="tabpanel3"
  id="tab3"
>
  Fish
</button>
```

Step 10: Set `tabindex="-1"` on the second and third `tab` elements.

```
<button
  role="tab"
  aria-selected="false"
  aria-controls="tabpanel2"
  id="tab2"
  tabindex="-1"
>
  Birds
</button>
```

```
<button
  role="tab"
  aria-selected="false"
  aria-controls="tabpanel3"
  id="tab3"
  tabindex="-1"
>
  Fish
</button>
```

Exercise: Making an accessible autocomplete

Accessing the exercise:

DEVELOPER EXERCISE: Making an accessible autocomplete

Note: **this example is not "operational"**, we are just focusing on the markup.

Review of elements used for the widget

First off, **let’s look at the basic markup** already in place.

The widget is **placed inside** a `<form>` component.

```
<form action="#">
  <label for="search">Search towns in Australia</l
  <input type="text" id="search">
  <button type="button">Clear</button>
  <button type="submit">Search</button>
  <ul id="results">
    <li>Aarons Pass</li>
  </ul>
  <div id="instructions"></div>
  <div></div>
</form>
```

There is a `<label>` to **provide an accessible name** for the `<input>`.

```
<form action="#">
  <label for="search">Search towns in Australia</l
  <input type="text" id="search">
  <button type="button">Clear</button>
  <button type="submit">Search</button>
  <ul id="results">
    <li>Aarons Pass</li>
  </ul>
  <div id="instructions"></div>
  <div></div>
</form>
```

The `<input>` is **programatically associated with the** `<label>` via matching `for` and `id` values.

```
<form action="#">
  <label for="search">Search towns in Australia</l
  <input type="text" id="search">
  <button type="button">Clear</button>
  <button type="submit">Search</button>
  <ul id="results">
    <li>Aarons Pass</li>
  </ul>
  <div id="instructions"></div>
  <div></div>
</form>
```

The first `<button>` is **used to clear the** `<input>` field.

```
<form action="#">
  <label for="search">Search towns in Australia</l
  <input type="text" id="search">
  <button type="button">Clear</button>
  <button type="submit">Search</button>
  <ul id="results">
    <li>Aarons Pass</li>
  </ul>
  <div id="instructions"></div>
  <div></div>
</form>
```

The second `<button>` is **used to clear submit**.

```
<form action="#">
  <label for="search">Search towns in Australia</l
  <input type="text" id="search">
  <button type="button">Clear</button>
  <button type="submit">Search</button>
  <ul id="results">
    <li>Aarons Pass</li>
  </ul>
  <div id="instructions"></div>
  <div></div>
</form>
```

The `<ul>` **displays possible results** when triggered.

```
<form action="#">
  <label for="search">Search towns in Australia</l
  <input type="text" id="search">
  <button type="button">Clear</button>
  <button type="submit">Search</button>
  <ul id="results">
    <li>Aarons Pass</li>
  </ul>
  <div id="instructions"></div>
  <div></div>
</form>
```

The first `<div>` is **used to provide instructions in using the widget**.

```
<form action="#">
  <label for="search">Search towns in Australia</l
  <input type="text" id="search">
  <button type="button">Clear</button>
  <button type="submit">Search</button>
  <ul id="results">
    <li>Aarons Pass</li>
  </ul>
  <div id="instructions"></div>
  <div></div>
</form>
```

The second `<div>` is **used to provide live updates** associated with how many results are displayed in the dropdown at any given time.

```
<form action="#">
  <label for="search">Search towns in Australia</l
  <input type="text" id="search">
  <button type="button">Clear</button>
  <button type="submit">Search</button>
  <ul id="results">
    <li>Aarons Pass</li>
  </ul>
  <div id="instructions"></div>
  <div></div>
</form>
```

Adding accessibility

Step 1: Add `role="searchbox"` to `<input>`.

This will change the input element's `role` from `textbox` to `searchbox`.

```
<input
  type="text"
  id="search"
  role="searchbox"
>
```

Step 2: Add `aria-describedby="instructions"` to `<input>`.

This will be used to link up with the instructions content to provide an accessible description for the `<input>`.

```
<input
  type="text"
  id="search"
  role="searchbox"
  aria-describedby="instructions"
>
```

Step 3: Add `aria-owns="results"` to `<input>`.

This will be used to provide a relationship between the `<input>` and `<ul>` element.

```
<input
  type="text"
  id="search"
  role="searchbox"
  aria-describedby="instructions"
  aria-owns="results"
>
```

Step 4: Add `aria-expanded="false"` to `<input>`. This will change to `aria-expanded="true"` when the `<ul>` is triggered and becomes visible.

```
<input
  type="text"
  id="search"
  role="searchbox"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="false"
>
```

Step 5: Add `id="results"` to `<ul>`.

This will relate to the `aria-owns` value associated with the `<input>`.

```
<ul
  id="results"
>
  <li>Aarons Pass</li>
</ul>
```

Step 6: Add `role="listbox"` to `<ul>`.

This will define the element as a parent for a list of options.

```
<ul
  id="results"
  role="listbox"
>
  <li>Aarons Pass</li>
</ul>
```

Step 7: Add `tabindex="-1"` to `<ul>`.

This element will initially be hidden, and should not receive focus until it becomes visible.

```
<ul
  id="results"
  role="listbox"
  tabindex="-1"
>
  <li>Aarons Pass</li>
</ul>
```

Step 8: Add `role="option"` to each `<li>`.

This will define the elements as options.

```
<ul
  id="results"
  role="listbox"
  tabindex="-1"
>
  <li role="option">Aarons Pass</li>
</ul>
```

Step 9: Add `aria-selected="false"` to each `<li>`. This will need to change to `aria-selected="true"` when an individual option is selected.

```
<ul
  id="results"
  role="listbox"
  tabindex="-1"
>
  <li role="option" aria-selected="false">Aarons P
</ul>
```

Step 10: Add `class="sr-only"` to the first `<div>` element.
This will hide the element off-screen but still make it available to assistive technologies.

```
<div id="instructions" class="sr-only">
</div>
```

Step 11: Add `aria-live="assertive"` to the second `<div>` element.

This will allow dynamic changes to be announced to assistive technologies.

```
<div aria-live="assertive"></div>
```

Step 12: Add `class="sr-only"` to the second `<div>` element.
This will hide the element off-screen but still make it available to assistive technologies.

```html
<div aria-live="assertive" class="sr-only"></div>
```

The **dynamically announced information** could be something like:
[6] options available.

Recap

Today's session covered **accessible widgets** - things to keep in mind as well as a deep dive into some complex widgets.

More widget patterns can be found at the ARIA authoring practices guide. These are just guidelines, not final products.

Accessible widgets should be built with semantic HTML, using CSS and JavaScript to extend the functionality. Then use ARIA to inform the accessibility API of each role, state and property.

Thankyou ✨

Feedback welcome!