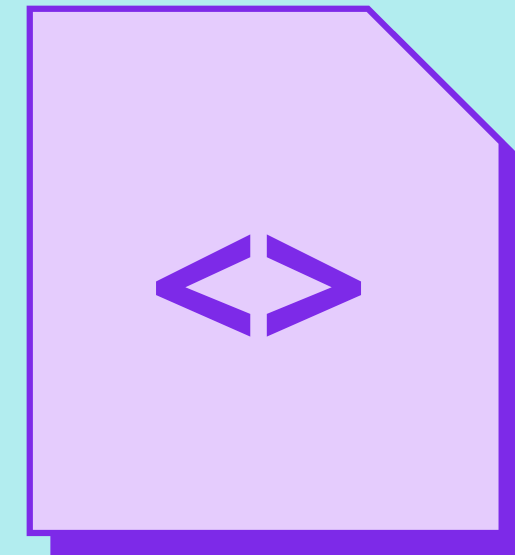
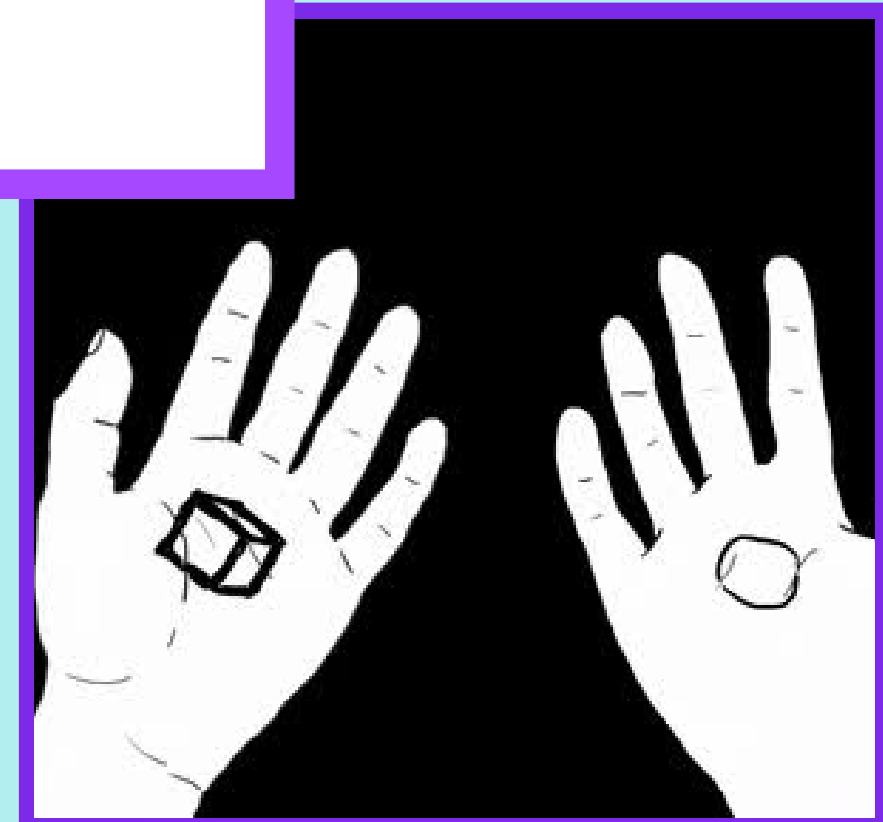




Accessibility Part 2



Presented by
Mike Nam-Lee



2. Operable

Components and navigation must be operable (the interface cannot require interaction that a user cannot perform).

Page Navigation

Not everyone who interacts with a webpage will prefer to use a mouse or a touch device. It's important that all page actions support keyboard interactions:

- Arrow keys and Space to jump down the page
- Tab and Shift+Tab to jump between interactive elements
- Enter and Arrow keys to interact with the element


Buttons need to support click and tap as well as the Enter and Space key.

Select dropdowns need to support arrow keys to change the selected item and support Enter and Escape.

Don't set keyboard traps

Tabbing from the top of the page should hit every interactive element and after the last interactive element, it should cycle to the top. This also means showing the focus outline on every element, at least for keyboard users.

```
<a onKeyDown={e => {  
    e.preventDefault()  
}}>Trap</a>  
<a>Unreachable</a>
```



**Let's look at some
interactive elements**

Forms and form elements should be marked up correctly

Use standard HTML elements (buttons, inputs, textareas) over messing around with div's to achieve a particular style. Using standard HTML markup makes it a lot easier for browsers to support operability and for people to understand. It's also important to add appropriate attributes. This will be expanded upon in the next lecture or skip ahead and read the spec on MDN (e.g. [inputs](#)).

```
<form onSubmit={onSubmit}>
  <!-- Adding the type, required, and label to inputs is super
  important. The name has no accessibility features; it's only for
  the form request. -->
  <label for="email">Email</label>
  <input id="email" type="email" name="email" required />
  <!-- Make sure you add text and type to buttons. -->
  <button type="submit">Submit</button>
</form>
```

* The one exception is select elements as CSS customisation of the HTML element is fairly limited. I'll elaborate more later in the lecture

**Use headers,
sections,
asides and
footers.**

**These improve
operability.**

```
<header>
  <nav>{...}</nav>
  <form role="search">{...}</form>
</header>
<main>
  <section>{...}</section>
  <section>{...}</section>
</main>
<aside>{...}</aside>
<footer>{...}</footer>
```

**If there's
repeated
content, use a
skip link to
bypass it.**

// CSS

```
.skipLink {  
  position: absolute;  
  z-index: 100;  
  display: block;  
  background: white;  
  overflow: hidden;  
  white-space: nowrap;  
  clip: rect(1px, 1px, 1px, 1px);  
  height: 1px;  
  width: 1px;  
  margin: -1px;  
}
```

```
.skipLink:focus, .skipLink:active {  
  overflow: initial;  
  white-space: initial;  
  clip: initial;  
  height: initial;  
  width: initial;  
  margin: initial;  
}
```

// HTML

```
<a className="skipLink" href="#afterCarousel">Skip over  
carousel</a>  
<div className="carousel">{...}</div>  
<span id="afterCarousel"></span>
```


Introducing ARIA

HTML is quite limited as it was never designed with web applications in mind and so, was never built for more complex UIs.

ARIA is a series of additional attributes to supplement HTML. It is not intended to replace HTML and it should be used sparingly and thoughtfully on top of HTML tags and roles.

Read more about ARIA attributes [here](#).



With ARIA, you can add labels for icon-only buttons

```
<button  
  aria-label="Show menu"  
>  
    
</button>
```

You can add state for custom components that have no native HTML equivalent.

Many attributes are only valid on particular roles.

```
<button  
  role="switch"  
  aria-checked={true}  
>  
  Accept cookies  
</button>
```

You can define modal takeovers.
This only adds markup: in many
cases, you need to add JS to
implement it.

```
<div
  role="alertdialog"
  aria-modal={true}
>
  Sign in to perform this action
</div>
```

You can define control relationships between elements.

```
<button  
  aria-controls="menuElem"  
  aria-expanded={true}  
  aria-haspopup="menu"  
>Show menu</button>  
<aside id="menuElem">{...}</aside>
```

Define whether the element is currently expanded with `aria-expanded` for pop-up or pop-out elements.

```
<button
  aria-controls="menuElem"
  aria-expanded={true}
  aria-haspopup="menu"
>Show menu</button>
<aside id="menuElem">{...}</aside>
```

You can also define the type of popup. Valid options include "menu", "listbox" and "dialog".

```
<button
  aria-controls="menuElem"
  aria-expanded={true}
  aria-haspopup="menu"
>Show menu</button>
<aside id="menuElem">{...}</aside>
```

Combining that altogether gives us the ability to mark up our own select component.

Read more on [W3](#).

```
<!-- If we want to build our own
select, we make a list element with
role="listbox" -->
<button
  aria-controls="listboxElem"
  aria-expanded={true}
  aria-haspopup="listbox"
>Travel destination</button>
<ul role="listbox" id="listboxElem">
  <li role="option" aria-selected=
    {false}>Canberra</li>
  {...}
</ul>
```


Extra Considerations

Do not design content in a way that is known to cause seizures. This includes flashing content very quickly and you need to be extra careful with flashing red content even moderately quickly.

It is also important to give sufficient time for users to complete tasks unless it is impossible to do so (say, an online auction site). For example, error messages should stay visible for as long as it's relevant and any tooltips you may want to use should stay visible if you hover over the tooltip.

**See you
next time!**