Accessible

- index
- Accessible responsive design
- Accessible tap targets
- Color and contrast accessibility
- Content reordering
- Control focus with tabindex
- Headings and landmarks
- Keyboard access fundamentals
- Labels and text alternatives
- Semantics and screen readers
- Style focus
- Use semantic HTML for easy keyboard wins
- What is accessibility?

index

layout: collection override:tags: [] date: 2018-11-05 pathName: accessible

Accessible responsive design

We know that it's a good idea to design responsively to provide the best multi-device experience, but responsive design also yields a win for accessibility.

Consider a site like Udacity:

> {% Img src="image/admin/5q9dDtEubSM23SzokQNu.jpg", alt="The Udacity site", width="800", height="393" %}

A low-vision user who has difficulty reading small print might zoom in the page, perhaps as much as 400%. Because the site is designed responsively, the UI will rearrange itself for the "smaller viewport" (actually for the larger page), which is great for desktop users who require screen magnification and for mobile screen reader users as well. It's a win-win. Here's the same page magnified to 400%:

> {% Img src="image/admin/WKHO21uWQz5lJ7Aqej1E.jpg", alt="The Udacity site zoomed to 400%", width="800", height="393" %}

In fact, just by designing responsively, we're meeting rule 1.4.4 of the WebAIM checklist, which states that a page "... should be readable and functional when the text size is doubled."

Going over all of responsive design is outside the scope of this guide, but here are a few important takeaways that will benefit your responsive experience and give your users better access to your content.

Use the viewport meta tag

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Setting `width=device-width` will match the screen's width in device-independent pixels, and setting `initial-scale=1` establishes a 1:1 relationship between CSS pixels and device-independent pixels. Doing this instructs the browser to fit your content to the screen size, so users don't just see a bunch of scrunched-up text.

See Size content to viewport to learn more.

Allow users to zoom

It is possible to use the viewport meta tag to prevent zooming, by setting `maximum-scale=1` or `user-scaleable=no`. Avoid doing this, and let your users zoom in if they need to.

Design with flexibility

Avoid targetting specific screen sizes and instead use a flexible grid, making changes to the layout when the content dictates. As we saw with the Udacity example above, this approach ensures that the design responds whether the reduced space is due to a smaller screen or a higher zoom level.

You can read more about these techniques in the Responsive web design basics article.

Use relative units for text

To get the best out of your flexible grid use relative units like em or rem for things like text size, instead of pixel values. Some browsers support resizing text only in user preferences, and if you're using a pixel value for text, this setting will not affect your copy. If, however, you've used relative units throughout, then the site copy will update to reflect the user's preference.

This will enable the whole site to reflow as the user zooms, creating the reading experience they need to use your site.

Avoid disconnecting the visual view from the source order

A visitor who is tabbing through your site with the keyboard will be following the order of the content in the HTML document. When using modern layout methods such as Flexbox and Grid, it is easy to make the visual rendering not match the source order. This can lead to disconcerting jumps around the page for someone using the keyboard to move around.

Make sure to test your design at each breakpoint by tabbing through the content, does the flow through the page still make sense?

Read more about the source and visual display disconnect.

Take care with spatial clues

When writing microcopy avoid using language which indicates the location of an element on the page. For example, referring to navigation "on your left" makes no sense in a mobile version when the navigation is at the top of the screen.

Ensure tap targets are large enough on touchscreen devices

On touchscreen devices make sure your tap targets are large enough to make them easy to activate without hitting other links. A good size for any tappable element is 48px, read more guidance on tap targets.

Accessible tap targets

When your design is displayed on a mobile device, you should ensure that interactive elements like buttons or links are large enough, and have enough space around them, to make them easy to press without accidentally overlapping onto other elements. This benefits all users, but is especially helpful for anyone with a motor impairment.

A minimum recommended touch target size is around 48 device independent pixels on a site with a properly set mobile viewport. For example, while an icon may only have a width and height of 24px, you can use additional padding to bring the tap target size up to 48px. The 48x48 pixel area corresponds to around 9mm, which is about the size of a person's finger pad area.

In the demo, I have added padding to all of the links in order to make sure they meet the minimum size.

{% Glitch { id: 'tap-targets', path: 'index.html' } %}

Touch targets should also be spaced about 8 pixels apart, both horizontally and vertically, so that a user's finger pressing on one tap target does not inadvertently touch another tap target.

Testing your touch targets

If your target is text and you have used relative values such as `em` or `rem` to size the text and any padding, you can use DevTools to check that the computed value of that area is large enough. In the example below I am using `em` for my text and padding.

{% Glitch { id: 'tap-targets-2', path: 'style.css' } %}

Inspect the `a` of the link, and in Chrome DevTools switch to the Computed pane where you can inspect the various parts of the box and see what pixel size they resolve to. In Firefox DevTools there is a Layout Panel. In that Panel you get the actual size of the inspected element.

> {% Img src="image/admin/vmFzREveRttHVDfLqqCx.jpg", alt="The Layout Panel in Firefox DevTools showing the size of the a element", width="800", height="565" %}

Using media queries to detect touchscreen use

Instead of simply testing for viewport dimensions, and then guessing that small dimensions are likely to be phones or tablets, which in turn use a touchscreen, there are now more robust ways to adapt your design based on actual device capabilities.

One of the media features we can now test for with media queries is whether the user's *primary input* is a touchscreen (`pointer`) and whether *any* of the currently detected inputs is a touchscreen (`any-pointer`). The `pointer` and `any-pointer` features will return `fine` or `coarse`. A fine pointer will be someone using a mouse or trackpad, even if that mouse is connected via Bluetooth to a phone. A `coarse` pointer indicates a touchscreen, which could be a mobile device but may also be a laptop screen or large tablet.

If you are adjusting your CSS within a media query to increase the touch target, testing for a coarse pointer allows you to increase the tap targets for all touchscreen users. This gives the larger tap area whether the device is a phone or larger device.

```
.container a {
  padding: .2em;
}

@media (any-pointer: coarse) {
  .container a {
    padding: .8em;
  }
}
```

You can find out more about interaction media features such as pointer in the Responsive web design basics article.

Color and contrast accessibility

If you have good vision, it's easy to assume that everyone perceives colors, or text legibility, the same way you do—but of course that's not the case.

As you might imagine, some color combinations that are easy for some people to read are difficult or impossible for others. This usually comes down to color contrast, the relationship between the foreground and background colors' luminance. When the colors are similar, the contrast ratio is low; when they are different, the contrast ratio is high.

The WebAIM guidelines recommend an AA (minimum) contrast ratio of 4.5:1 for all text. An exception is made for very large text (120-150% larger than the default body text), for which the ratio can go down to 3:1. Notice the difference in the contrast ratios shown below.

> {% Img src="image/admin/DcYclKelVqhQ2CWIIG97.jpg", alt="An image showing the different contrast ratios", width="800", height="328" %}

The contrast ratio of 4.5:1 was chosen for level AA because it compensates for the loss in contrast sensitivity usually experienced by users with vision loss equivalent to approximately 20/40 vision. 20/40 is commonly reported as typical visual acuity of people at about age 80. For users with low vision impairments or color deficiencies, we can increase the contrast up to 7:1 for body text.

You can use the Accessibility Audit in Lighthouse to check your color contrast. Open DevTools, click Audits, and select Accessibility to run the report.

> {% Img src="image/admin/vSFzNOurQO6z2xV6qWuW.png", alt="A screenshot of the output of an audit for color contrast.", width="800", height="218" %}

Chrome also includes an experimental feature to help you detect all the low contrast texts of your page. You can also use the accessible color suggestion to fix the low contrast text.

> {% Img src="image/admin/VYZeK2l2vs6pIoWhH2hO.png", alt="A screenshot of the output of the Chrome low contrast text experimental feature.", width="800", height="521" %}

For a more complete report, install the Accessibility Insights Extension. One of the checks in the Fastpass report, is color contrast. You will get a detailed report of any failing elements.

{% Img src="image/admin/CR21TFMZw8gWsSTWOGIF.jpg", alt="The report in Accessibility Insights", width="800", height="473" %}

Advanced Perceptual Contrast Algorithm (APCA)

The Advanced Perceptual Contrast Algorithm (APCA) is a new way to compute contrast based on modern research on color perception.

Compared to AA/AAA guidelines, APCA is more context-dependent.

The contrast is calculated based on the following features:

- Spatial properties (font weight and text size)
- Text color (perceived lightness difference between text and background)
- Context (ambient light, surroundings, and intended purpose of the text)

Chrome includes an experimental feature to replace the AA/AAA contrast ratio guidelines with APCA.

{% Img src="image/admin/YhGKRLYvt37j3ldlwiXE.png", alt="A screenshot of the output of the APCA feature in Chrome.", width="800", height="543" %}

Don't convey information with color alone

There are roughly 320 million people worldwide with color vision deficiency. About 1 in 12 men and 1 in 200 women have some form of "color blindness"; that means about 1/20th, or 5%, of your users will not experience your site the way you intended. When we rely on color to convey information, we push that number to unacceptable levels.

{% Aside %} Note: The term "color blindness" is often used to describe a visual condition where a person has trouble distinguishing colors, but in fact very few people are truly color blind. Most people with color deficiencies can see some or most colors, but have difficulty differentiating between certain colors such as reds and greens (most common), browns and oranges, and blues and purples. {% endAside %}

For example, in an input form, a telephone number might be underlined in red to show that it is invalid. But to a color deficient or screen reader user, that information is not conveyed well, if at all. Thus, you should always try to provide multiple avenues for the user to access critical information.

{% Img
src="image/admin/MKmlhejyjNpk7XE9R2KV.png",
alt="A image of an input form with an incorrect phone number highlighted only with a red color.",
width="293", height="323"
%}

The WebAIM checklist states in section 1.4.1 that "color should not be used as the sole method of conveying content or distinguishing visual elements." It also notes that "color alone should not be used to distinguish links from surrounding text" unless they meet certain contrast requirements. Instead, the checklist recommends adding an additional indicator such as an underscore (using the CSS `text-decoration` property) to indicate when the link is active.

An easy way to fix the previous example is to add an additional message to the field, announcing that it is invalid and why.

{% Img
src="image/admin/FLQPcG16akNRoElx3pnz.png",
alt="The same input form as in the last example, this time with a text label indicating the problem with the field.", width="292",
height="343" %}

When you're building an app, keep these sorts of things in mind and watch out for areas where you may be relying too heavily on color to convey important information.

If you're curious about how your site looks to different people, or if you rely heavily on the use of color in your UI, you can use DevTools to simulate various forms of visual impairment, including different types of color blindness. Chrome includes an Emulate Vision Deficiencies feature. To access it open DevTools and then open the **Rendering** tab in the Drawer, you can then emulate the following color deficiencies.

- Protanopia: the inability to perceive any red light.
- Deuteranopia: the inability to perceive any green light.
- Tritanopia: the inability to perceive any blue light.

- Achromatopsia: the inability to perceive any color except for shades of grey (extremely rare).

    {% Img src="image/admin/VAnFxYhzFcpovdTCToPl.jpg", alt="Emulating the vision of a person with Achromatopsia shows our page in greyscale.", width="800", height="393" %}

High contrast mode

High-contrast mode allows a user to invert foreground and background colors, which often helps text stand out better. For someone with a low vision impairment, high-contrast mode can make it much easier to navigate the content on the page. There are a few ways to get a high-contrast setup on your machine.

Operating systems like Mac OSX and Windows offer high-contrast modes that can be enabled for everything at the system level.

A useful exercise is to turn on high-contrast settings and verify that all of the UI in your application is still visible and usable.

For example, a navigation bar might use a subtle background color to indicate which page is currently selected. If you view it in a high-contrast extension, that subtlety completely disappears, and with it goes the reader's understanding of which page is active.

{% Img src="image/admin/dgmA4W1Qu8JmcgsH80HD.png", alt="Screenshot of a navigation bar in high contrast mode where the acvtive tab is hard to read", width="640", height="57" %}

Similarly, if you consider the example from the previous lesson, the red underline on the invalid phone number field might be displayed in a hard-to-distinguish blue-green color.

{% Img src="image/admin/HtlXwmHQHBcAO4LYSfAA.jpg", alt="Screenshot of the address form used earlier, this time in high contrast mode. The color change of the invalid element is hard to read.", width="700", height="328" %}

If you are meeting the contrast ratios covered earlier you should be fine when it comes to supporting high-contrast mode. But for added peace of mind, consider installing the High Contrast Chrome Extension and giving your page a once-over just to check that everything works, and looks, as expected.

Content reordering

The order of content in your document is important for the accessibility of your site. A screen reader will read out content based on the document order, using the HTML elements that you have selected to give meaning to that content. A person navigating the site using a keyboard, rather than a touchscreen or mouse, will tab around the document. This means that they will jump from active element to active element, tabbing between links and form fields, once again in the order they exist in the document.

Starting with a well-structured document and using all of the right HTML elements is therefore a key part of creating an accessible site. However, it is possible to undo some of that good work when you start using CSS. Let's take a look at why.

Source vs. visual order

Website navigation is often marked up as a list of links. You can then use Flexbox to turn these into a horizontal bar. In the Glitch example below, I have created this commonly used pattern. Click into the example, and tab between the links. The focus will move in a logical direction from left to right, the order that we read in English.

{% Glitch { id: 'flex-nav-source-ordered', path: 'index.html' } %}

If you have created this sort of pattern and then were asked to move **Contact Us**, which is second in the source, to the end. You could use the `order` property which works in Flexbox. Try tabbing through the items in the example below, which has used the `order` property to rearrange the items.

{% Glitch { id: 'flex-nav-ordered', path: 'style.css' } %}

The focus jumps to the final item and then back again. As far as the tab order is concerned that item is the second item. Visually however, it's the last item.

The above example highlights the problem that we face if we rearrange and reorder content using CSS. If you were dealing with this issue then the right thing to do would be to change the order in the source, rather than using CSS.

Which CSS properties can cause reordering?

Any layout method that allows you to move elements around can cause this problem. The following CSS properties commonly cause content reordering problems:

- Using `position: absolute` and taking an item out of flow visually.
- The `order` property in Flexbox and Grid layout.
- The `row-reverse` and `column-reverse` values for `flex-direction` in Flexbox.
- The `dense` value for `grid-auto-flow` in Grid Layout.
- Any positioning by line name or number, or with `grid-template-areas` in Grid Layout.

In this next example, I have created a layout using CSS Grid and positioned the items using line numbers, without considering where they are in the source.

{% Glitch { id: 'grid-mixed-up-order', path: 'index.html' } %}

Try tabbing around this example, and see how the focus jumps about. This makes for a very confusing experience, especially if this is a long page.

Testing for the problem

A very simple test is to keyboard navigate through your page. Can you get to everything? Are there any strange jumps as you do so?

For a visual demonstration of content reordering, try the Tab Stop checker in the Accessibility Insights extension for Chrome. The image below shows the CSS Grid example in that tool. You can see how the focus has to jump around the layout.

> {% Img src="image/admin/n0i0TJf3OdZYvwswrHDV.jpg", alt="A screenshot of the Accessibility Insights Tool demonstrating the confusing order of items.", width="800", height="919" %}

Content reordering and responsive web design

If you only have one presentation of your content, then having your source in a logical order, and reflecting that in layout is not usually difficult. It can become harder, when you consider the layout at different breakpoints. It might make sense to have an element moved to the bottom of the layout on smaller screens for example.

There is not at this time a good solution for this problem. In most situations developing "mobile first", will help you keep your source and layout in order. The choices you make about priority on mobile, are often solid ones for the content in general. The key is to be aware when there is a possibility of this type of content reordering, and to test that the end experience is not too jarring at each breakpoint.

Control focus with tabindex

Standard HTML elements such as `<button>` or `<input>` have keyboard accessibility built in for free. If you're building *custom* interactive components, however, use the `tabindex` attribute to ensure that they're keyboard accessible.

{% Aside %} Whenever possible, use a built-in HTML element rather than building your own custom version. `<button>`, for example, is very easy to style and already has full keyboard support. This will save you from needing to manage `tabindex` or add semantics with ARIA. {% endAside %}

Check if your controls are keyboard accessible

A tool like Lighthouse is great at detecting certain accessibility issues, but some things can only be tested by a human.

Try pressing the `Tab` key to navigate through your site. Are you able to reach all the interactive controls on the page? If not, you may need to use `tabindex` to improve the focusability of those controls.

{% Aside 'warning' %} If you don't see a focus indicator at all, it may be hidden by your CSS. Check for any styles that mention `:focus { outline: none; }`. You can learn how to fix this in our guide on styling focus. {% endAside %}

Insert an element into the tab order

Insert an element into the natural tab order using `tabindex="0"`. For example:

```
<div tabindex="0">Focus me with the TAB key</div>
```

To focus an element, press the `Tab` key or call the element's `focus()` method.

{% Codepen { user: 'web-dot-dev', id: 'XWzzMqp', height: 360, theme: 'dark', tab: 'result' } %}

Remove an element from the tab order

Remove an element using `tabindex="-1"`. For example:

```
<button tabindex="-1">Can't reach me with the TAB key!</button>
```

This removes an element from the natural tab order, but the element can still be focused by calling its `focus()` method.

{% Codepen { user: 'web-dot-dev', id: 'BammWGg', height: 360, theme: 'dark', tab: 'result' } %}

Note that applying `tabindex="-1"` to an element doesn't affect its children; if they're in the tab order naturally or because of a `tabindex` value, they'll remain in the tab order. To remove an element and all its children from the tab order, consider using the WICG's `inert` polyfill. The polyfill emulates the behavior of a proposed `inert` attribute, which prevents elements from being selected or read by assistive technologies.

{% Aside 'caution' %} The `inert` polyfill is experimental and may not work as expected in all cases. Test carefully before using in production. {% endAside %}

Avoid `tabindex > 0`

Any `tabindex` greater than 0 jumps the element to the front of the natural tab order. If there are multiple elements with a `tabindex` greater than 0, the tab order starts from the lowest value greater than zero and works its way up.

Using a `tabindex` greater than 0 is considered an **anti-pattern** because screen readers navigate the page in DOM order, not tab order. If you need an element to come sooner in the tab order, it should be moved to an earlier spot in the DOM.

Lighthouse makes it easy to identify elements with a `tabindex` > 0. Run the Accessibility Audit (Lighthouse > Options > Accessibility) and look for the results of the "No element has a [tabindex] value greater than 0" audit.

Create accessible components with "roving `tabindex`"

If you're building a complex component, you may need to add additional keyboard support beyond focus. Consider the built-in `select` element. It is focusable and you can use the arrow keys to expose additional functionality (the selectable options).

To implement similar functionality in your own components, use a technique known as "roving `tabindex`". Roving tabindex works by setting `tabindex` to -1 for all children except the currently-active one. The component then uses a keyboard event listener to determine which key the user has pressed.

When this happens, the component sets the previously focused child's `tabindex` to -1, sets the to-be-focused child's `tabindex` to 0, and calls the `focus()` method on it.

Before

```
<div role="toolbar">
  <button tabindex="-1">Undo</button>
  <button tabindex="0">Redo</button>
  <button tabindex="-1">Cut</button>
</div>
```

After

```
<div role="toolbar">
  <button tabindex="-1">Undo</button>
  <button tabindex="-1">Redo</button>
  <button tabindex="0">Cut</button>
</div>
```

{% Codepen { user: 'web-dot-dev', id: 'LYOOWam', height: 360, theme: 'dark', tab: 'result' } %}

{% Aside %} Curious what those `role=""` attributes are for? They let you change the semantics of an element so it will be announced properly by a screen reader. You can learn more about them in our guide on screen reader basics. {% endAside %}

{% Assessment 'self-assessment' %}

Keyboard access recipes

If you're unsure what level of keyboard support your custom components might need, you can refer to the ARIA Authoring Practices 1.1. This handy guide lists common UI patterns and identifies which keys your components should support.

Headings and landmarks

{% include 'content/why-headings.njk' %}

Use headings to outline the page

Use `h1`-`h6` elements to create a *structural* outline for your page. The goal is to create a skeleton or scaffold of the page such that anyone navigating by headings can form a mental picture.

A common practice is to use a single `h1` for the primary headline or logo on a page, `h2` elements to designate major sections, and `h3` elements in supporting subsections:

```
<h1>Company name</h1>
<section>
  <h2>Section Heading</h2>
  …
  <h3>Sub-section Heading</h3>
</section>
```

Don't skip heading levels

Developers often skip heading levels to use the browser's default styles that closely match their design. This is considered an anti-pattern because it breaks the outline model.

Instead of relying on the browser's default font-sizing for headings, use your own CSS, and don't skip levels.

For example, this site has a section called "IN THE NEWS", followed by two headlines:

{% Img src="image/admin/CdBjBuUo2yVVHWVFnQzx.png", alt="A news site with a headline, hero image, and subsections.", width="800", height="414" %}

The section heading, "IN THE NEWS", could be an `h2`, and the supporting headlines could both be `h3` elements.

Because the `font-size` for "IN THE NEWS" is *smaller* than the headline, it may be tempting to make the headline for the first story an `h2` and to make "IN THE NEWS" an `h3`. While that may match

the browser's default styling, it would break the outline conveyed to a screen reader user!

{% Aside %} Though it may seem counterintuitive, it does not matter if *visually* `h3` and `h4` elements are larger than their `h2` or `h1` counterparts. What matters is the outline conveyed by the elements and elements' ordering. {% endAside %}

You can use Lighthouse to check if your page skips any heading levels. Run the Accessibility audit (**Lighthouse > Options > Accessibility**) and look for the results of the **Headings don't skip levels** audit.

Use landmarks to aid navigation

HTML5 elements such as `main`, `nav`, and `aside` act as **landmarks**, or special regions on the page to which a screen reader can jump.

Use landmark tags to define major sections of your page, instead of relying on `div`s. Be careful not to go overboard because having *too many* landmarks can be overwhelming. For example, stick to just one `main` element instead of 3 or 4.

Lighthouse recommends manually auditing your site to check that "HTML5 landmark elements are used to improve navigation." You can use this list of landmark elements to check your page.

Bypass repetitive content with skip links

Many sites contain repetitive navigation in their headers, which can be annoying to navigate with assistive technology. Use a **skip link** to let users bypass this content.

A skip link is an offscreen anchor that is always the first focusable item in the DOM. Typically, it contains an in-page link to the main content of the page. Because it is the first element in the DOM, it only takes a single action from assistive technology to focus it and bypass repetitive navigation.

```
<!-- index.html -->
<a class="skip-link" href="#main">Skip to main</a>
…
<main id="main">
  [Main content]
</main>
```

```
/* style.css */
.skip-link {
  position: absolute;
  top: -40px;
  left: 0;
  background: #000000;
  color: white;
  padding: 8px;
  z-index: 100;
}

.skip-link:focus {
  top: 0;
}
```

{% Aside 'codelab' %} Live skip link example. {% endAside %}

Many popular sites such as GitHub, The NY Times, and Wikipedia all contain skip links. Try visiting them and pressing the `TAB` key on your keyboard a few times.

Lighthouse can help you check if your page contains a skip link. Run the Accessibility audit again and look for the results of the **The page contains a heading, skip link, or landmark region** audit.

{% Aside %} Technically, this test will also pass if your site contains any `h1`-`h6` elements or any of the HTML5 landmark elements. But although the test is vague in its requirements, it's still nice to pass it if you can! {% endAside %}

Keyboard access fundamentals

Many different users rely on the keyboard to navigate applications—from users with temporary and permanent motor impairments to users who use keyboard shortcuts to be more efficient and productive. Having a good keyboard navigation strategy for your application creates a better experience for everyone.

Focus and the tab order

At a given moment, **focus** refers to what element in your application (such as a field, checkbox, button, or link) currently receives input from the keyboard. In addition to receiving keyboard events, the focused element also gets content that is pasted from the clipboard.

To move the focus on a page, use `TAB` to navigate "forward" and `SHIFT + TAB` to navigate "backward." The currently focused element is often indicated by a **focus ring**, and various browsers style their focus rings differently. The order in which focus proceeds forward and backward through interactive elements is called the **tab order**.

Interactive HTML elements like text fields, buttons, and select lists are **implicitly focusable**: they are automatically inserted into the tab order based on their position in the DOM. These interactive elements also have built-in keyboard event handling. Elements such as paragraphs and divs are not implicitly focusable because users typically do not need to interact with them.

Implementing a logical tab order is an important part of providing your users with a smooth keyboard navigation experience. There are two main ideas to keep in mind when assessing and adjusting your tab order:

1. Arrange elements in the DOM to be in logical order
2. Correctly set the visibility of offscreen content that should not receive focus

Arrange elements in the DOM to be in logical order

To check if your application's tab order is logical, try tabbing through your page. In general, focus should follow reading order, moving from left to right, from the top to the bottom of your page.

If the focus order seems wrong, you should rearrange the elements in the DOM to make the tab order more natural. **If you want something to appear visually earlier on the screen, move it earlier in the DOM**.

Try tabbing through the two sets of buttons below to experience a logical tab order versus an illogical tab order:

Logical tab order

{% Glitch { id: 'logical-tab-order', path: 'index.html', height: 346 } %}

Illogical tab order

{% Glitch { id: 'illogical-tab-order', path: 'index.html', height: 346 } %}

The code for these two examples is compared below:

Logical tab order

```
<button>Kiwi</button>
<button>Peach</button>
<button>Coconut</button>
```

Illogical tab order

```
<button style="float: right">Kiwi</button>
<button>Peach</button>
<button>Coconut</button>
```

Be careful when changing the visual position of elements using CSS to avoid creating an illogical tab order. To fix the illogical tab order above, move the floating "Kiwi" button so it comes after the "Coconut" button in the DOM, and remove the inline style.

Correctly set the visibility of offscreen content

Sometimes offscreen interactive elements need to be in the DOM but should not be in your tab order. For example, if you have a responsive side-nav that opens when you click a button, the user

should not be able to focus on the side-nav when it's closed.

To prevent a particular interactive element from receiving focus, you should give the element either of the following CSS properties:

- `display: none`
- `visibility: hidden`

To add the element back into the tab order, for example when the side-nav is opened, replace the above CSS properties respectively with:

- `display: block`
- `visibility: visible`

{% Aside %} If you can't figure out where the focus on your page is as you're tabbing, open the console and type: `document.activeElement`. This property will return the element that currently has focus. {% endAside %}

Next steps

For users who operate their computer almost entirely with the keyboard or another input device, a logical tab order is essential for making your application accessible and usable. As a good habit for checking your tab order, try **tabbing through your application before each publish**.

Labels and text alternatives

In order for a screen reader to present a spoken UI to the user, meaningful elements must have proper labels or text alternatives. A label or text alternative gives an element its accessible **name**, one of the key properties for expressing element semantics in the accessibility tree.

When an element's name is combined with the element's **role**, it gives the user context so they can understand what type of element they're interacting with and how it is represented on the page. If a name is not present, then a screen reader only announces the element's role. Imagine trying to navigate a page and hearing, "button," "checkbox," "image" without any additional context. This is why labeling and text alternatives are crucial to a good, accessible experience.

Inspect an element's name

It's easy to check an element's accessible name using Chrome's DevTools:

1. Right-click on an element and choose **Inspect**. This opens the DevTools Elements panel.
2. In the Elements panel, look for the **Accessibility** pane. It may be hidden behind a `»` symbol.
3. In the **Computed Properties** dropdown, look for the **Name** property.

   {% Img src="image/admin/38c68DmamTCqt2LFxTmu.png", alt="", width="800", height="471" %}
   DevTools accessibility pane showing the computed name for a button.

{% Aside %} To learn more, check out the DevTools Accessibility Reference. {% endAside %}

Whether you're looking at an `img` with `alt` text or an `input` with a `label`, all of these scenarios result in the same outcome: giving an element its accessible name.

Check for missing names

There are different ways to add an accessible name to an element, depending on its type. The following table lists the most common element types that need accessible names and links to explanations for how to add them.

Element type	How to add a name

HTML document	Label documents and frames
`<frame>` or `<iframe>` elements	Label documents and frames
Image elements	Include text alternatives for images and objects
`<input type="image">` elements	Include text alternatives for images and objects
`<object>` elements	Include text alternatives for images and objects
Buttons	Label buttons and links
Links	Label buttons and links
Form elements	Label form elements

Label documents and frames

Every page should have a `title` element that briefly explains what the page is about. The `title` element gives the page its accessible name. When a screen reader enters the page, this is the first text that is announced.

For example, the page below has the title "Mary's Maple Bar Fast-Baking Recipe":

```
<!doctype html>
  <html lang="en">
    <head>
      <title>Mary's Maple Bar Fast-Baking Recipe</title>
    </head>
  <body>
    …
  </body>
</html>
```

{% Aside %} For tips on writing effective titles, see the Write descriptive titles guide. {% endAside %}

Similarly, any `frame` or `iframe` elements should have `title` attributes:

```
<iframe title="An interactive map of San Francisco" src="…"></iframe>
```

While an `iframe`'s contents may contain their own internal `title` element, a screen reader usually stops at the frame boundary and announces the element's role—"frame"—and its accessible name,

provided by the `title` attribute. This lets the user decide if they wish to enter the frame or bypass it.

Include text alternatives for images and objects

An `img` should always be accompanied by an `alt` attribute to give the image its accessible name. If the image fails to load, the `alt` text is used as a placeholder so users have a sense of what the image was trying to convey.

Writing good `alt` text is a bit of an art, but there are a couple of guidelines you can follow:

1. Determine if the image provides content that would otherwise be difficult to attain from reading the surrounding text.
2. If so, convey the content as succinctly as possible.

If the image acts as decoration and does not provide any useful content, you can give it an empty `alt=""` attribute to remove it from the accessibility tree.

{% Aside %} Learn more about writing effective `alt` text by checking out WebAIM's guide to Alternative Text. {% endAside %}

Images as links and inputs

An image wrapped in a link should use the `img`'s `alt` attribute to describe where the user will navigate to if they click on the link:

```
<a href="https://en.wikipedia.org/wiki/Google">
  <img alt="Google's wikipedia page" src="google-logo.jpg">
</a>
```

Similarly, if an `<input type="image">` element is used to create an image button, it should contain `alt` text that describes the action that occurs when the user clicks on the button:

```
<form>
  <label>
    Username:
    <input type="text">
```

```
  </label>
  <input type="image" alt="Sign in" src="./sign-in-button.png">
</form>
```

Embedded objects

`<object>` elements, which are typically used for embeds like Flash, PDFs, or ActiveX, should also contain alternative text. Similar to images, this text is displayed if the element fails to render. The alternative text goes inside the `object` element as regular text, like "Annual report" below:

```
<object type="application/pdf" data="/report.pdf">
Annual report.
</object>
```

Label buttons and links

Buttons and links are often crucial to the experience of a site, and it is important that both have good accessible names.

Buttons

A `button` element always attempts to compute its accessible name using its text content. For buttons that are not part of a `form`, writing a clear action as the text content may be all you need to create a good accessible name.

```
<button>Book Room</button>
```

{% Img src="image/admin/tcIDzNpCHS9AlfwflQjI.png", alt="A mobile form with a 'Book Room' button.", width="800", height="269" %}

One common exception to this rule is icon buttons. An icon button may use an image or an icon font to provide the text content for the button. For example, the buttons used in a What You See Is What You Get (WYSIWYG) editor to format text are typically just graphic symbols:

{% Img src="image/admin/ZmQ77kLPbqd5iFOmn4SU.png", alt="A left align icon button.", width="800", height="269" %}

When working with icon buttons, it can be helpful to give them an explicit accessible name using the `aria-label` attribute. `aria-label` overrides any text content inside the button, letting you clearly describe the action to anyone using a screen reader.

```
<button aria-label="Left align"></button>
```

Links

Similar to buttons, links primarily get their accessible name from their text content. A nice trick when creating a link is to put the most meaningful piece of text into the link itself, rather than filler words like "Here" or "Read More."

{% Compare 'worse', 'Not descriptive enough' %}

```
Check out our guide to web performance <a href="/guide">here</a>.
```

{% endCompare %}

{% Compare 'better', 'Useful content!' %}

```
Check out <a href="/guide">our guide to web performance</a>.
```

{% endCompare %}

This is especially helpful for screen readers that offer shortcuts to list all of the links on the page. If links are full of repetitive filler text, these shortcuts become much less useful:

> {% Img src="image/admin/IPxS2dwHMyGRvGxGi5n2.jpg", alt="VoiceOver's links menu filled with the word 'here'.", width="519", height="469" %}
> Example of VoiceOver, a screen reader for macOS, showing the navigate by links menu.

Label form elements

There are two ways to associate a label with a form element such as a checkbox. Either of the methods causes the label text to also become a click target for the checkbox, which is also helpful for mouse or touchscreen users. To associate a label with an element, either:

- Place the input element inside of a label element

```
<label>
  <input type="checkbox">Receive promotional offers?</input>
</label>
```

- Or use the label's `for` attribute and refer to the element's `id`

```
<input id="promo" type="checkbox"></input>
<label for="promo">Receive promotional offers?</label>
```

When the checkbox has been labeled correctly, the screen reader can report that the element has a role of checkbox, is in a checked state, and is named "Receive promotional offers?" like in the VoiceOver example below:

> {% Img src="image/admin/WklT2ymrCmceyrGUNizF.png", alt="VoiceOver text output showing 'Receive promotional offers?'", width="640", height="174" %}

{% Assessment 'self-assessment' %}

Semantics and screen readers

Have you ever stopped to wonder *how* assistive technology, such as a screen reader, knows what to announce to users? The answer is that these technologies rely on developers marking up their pages with **semantic** HTML. But what are semantics, and how do screen readers use them?

Affordances and semantics

Before diving into semantics, it's helpful to understand another term: **affordances**. An affordance is any object that offers, or affords, its user the opportunity to perform an action. A classic example is the teapot:

> {% Img src="image/admin/EXqR31Kq3mosH6jv6jiG.png", alt="", width="640", height="446" %}
> A teapot's handle is a natural affordance.

This teapot doesn't need an instruction manual; instead, its physical design conveys to the user how it should be operated. It has a handle, and because you've seen other objects in the world with similar handles, you can infer how you should pick it up and operate it.

When we build graphical user interfaces, we use things like CSS to add **visual affordances** to our UI. For instance, you might give a button a drop shadow and border so that it resembles an actual button in the real world.

But if a user is unable to see the screen, then these visual affordances will not be conveyed to them. Therefore, you need to make sure that your UI is constructed in a way that can convey these same affordances to assistive technology. This non-visual exposure of a UI element's affordances is called its **semantics**.

Use semantic HTML

The easiest way of conveying proper semantics is to use semantically rich HTML elements.

Using CSS, it's possible to style the `<div>` and `<button>` elements so they convey the same visual affordances, but the two experiences are very different when using a screen reader. A `<div>` is just

a generic grouping element, so a screen reader only announces the text content of the `<div>`. The `<button>` is announced as a "button," a much stronger signal to the user that it's something they can interact with.

The simplest and often best solution to this problem is to avoid custom interactive controls altogether. For example, replace a `<div>` that's acting like a button with an actual `<button>`.

Semantic properties and the accessibility tree

Generally speaking, every HTML element will have some of the following semantic properties:

- A **role** or type
- A **name**
- A **value** (optional)
- A **state** (optional)

An element's **role** describes its type, for example, "button," "input," or even just "group" for things like `div` and `span` elements.

An element's **name** is its computed label. Screen readers typically announce an element's name followed by its role, e.g. "Sign Up, button." The algorithm that determines an element's name factors in things like if there is any text content inside the element, whether or not it has attributes such as `title` or `placeholder`, whether or not the element is associated with an actual `<label>` element, and if the element has any ARIA attributes such as `aria-label` and `aria-labelledby`.

Some elements *may* have a **value**. For instance, `<input type="text">` may have a value that reflects whatever the user has typed into the text field.

Some elements *may* also have a **state**, which conveys their current status. For instance, a `<select>` element can be in either an *expanded* or a *collapsed* state, depending on if it's open or closed.

The accessibility tree

For each node in the DOM, the browser determines if the node is semantically "interesting" and adds it to the accessibility tree. When assistive technology, like a screen reader, is providing an alternative UI to the user, it is often doing so by walking this accessibility tree.

{% Aside %} Browsers will often remove semantically uninteresting nodes like `div` and `span` from the accessibility tree, especially if they're just being used to position their children with CSS. For

instance, if you have a `button` nested inside of 5 `div` elements, the browser may prune out some of the `div` elements in the middle to cut down on noise. {% endAside %}

Using Chrome's DevTools you can inspect an element's semantic properties and explore its position in the accessibility tree.

Next steps

Once you know a bit about semantics and how they aid screen reader navigation, you can't help but look at the pages you build differently. In the next section, we'll take a step back and consider how the entire outline of a page can be conveyed using effective headings and landmarks.

Style focus

The focus indicator (often signified by a "focus ring") identifies the currently focused element on your page. For users who are unable to use a mouse, this indicator is *extremely important* because it acts as a stand-in for their mouse-pointer.

If the browser's default focus indicator clashes with your design, you can use CSS to restyle it. Just remember to keep your keyboard users in mind!

Use `:focus` to always show a focus indicator

The `:focus` pseudo-class is applied any time an element is focused, regardless of the input device (mouse, keyboard, stylus, etc.) or method used to focus it. For example, the `<div>` below has a `tabindex` which makes it focusable. It also has a custom style for its `:focus` state:

```
div[tabindex="0"]:focus {
  outline: 4px dashed orange;
}
```

Regardless of whether you use a mouse to click on it or a keyboard to tab to it, the `<div>` will *always* look the same.

{% Glitch { id: 'focus-style', path: 'index.html', height: 346 } %}

Unfortunately browsers can be inconsistent with how they apply focus. Whether or not an element receives focus may depend on the browser and the operating system.

For example, the `<button>` below also has a custom style for its `:focus` state.

```
button:focus {
  outline: 4px dashed orange;
}
```

If you click on the `<button>` with a mouse in Chrome on macOS you should see its custom focus style. However, you will not see the custom focus style if you click on the `<button>` in Safari on macOS. This is because in Safari the element does not receive focus when you click on it.

{% Glitch { id: 'focus-style2', path: 'index.html', height: 346 } %}

{% Aside %} See the MDN reference for `<button>` focus behavior for a summary of which browsers and operating systems will apply focus to `<button>` elements. {% endAside %}

Because the behavior of focus is inconsistent, it may require a bit of testing on different devices to ensure your focus styles are acceptable to your users.

Use `:focus-visible` to selectively show a focus indicator

The new `:focus-visible` pseudo-class is applied any time that an element receives focus and the browser determines via heuristics that displaying a focus indicator would be beneficial to the user. In particular, if the most recent user interaction was via the keyboard and the key press did not include a meta, `ALT` / `OPTION`, or `CONTROL` key, then `:focus-visible` will match.

The button in the example below will *selectively* show a focus indicator. If you use a mouse to click on it, the results are different than if you first use a keyboard to tab to it.

```
button:focus-visible {
  outline: 4px dashed orange;
}
```

{% Glitch { id: 'focus-visible-style', path: 'index.html', height: 346 } %}

Use `:focus-within` to style the parent of a focused element

The `:focus-within` pseudo-class is applied to an element either when the element itself receives focus or when another element inside that element receives focus.

It can be used to highlight a region of the page to draw the user's attention to that area. For example, the form below receives focus both when the form itself is selected and also when any of its radio buttons are selected.

```
form:focus-within {
  background: #ffecb3;
```

```
}
```

{% Glitch { id: 'focus-within-style', path: 'index.html', height: 346 } %}

When to display a focus indicator

A good rule of thumb is to ask yourself, "If you clicked on this control while using a mobile device, would you expect it to display a keyboard?"

If the answer is "yes," then the control should probably *always* show a focus indicator, regardless of the input device used to focus it. A good example is the `<input type="text">` element. The user will need to send input to the element via the keyboard regardless of how the input element originally received focus, so it's helpful to always display a focus indicator.

If the answer is "no," then the control may choose to selectively show a focus indicator. A good example is the `<button>` element. If a user clicks on it with a mouse or touch screen, the action is complete, and a focus indicator may not be necessary. However, if the user is *navigating* with a keyboard, it's useful to show a focus indicator so the user can decide whether or not they want to click the control using the `ENTER` or `SPACE` keys.

Avoid `outline: none`

The way browsers decide when to draw a focus indicator is, frankly, very confusing. Changing the appearance of a `<button>` element with CSS or giving an element a `tabindex` will cause the browser's default focus ring behavior to kick-in.

A very common anti-pattern is to remove the focus indicator using CSS such as:

```
/* Don't do this!!! */
:focus {
  outline: none;
}
```

A better way to work around this issue is to use a combination of `:focus` and the `:focus-visible` polyfill. The first block of code below demonstrates how the polyfill works, and the sample app beneath it provides an example of using the polyfill to change the focus indicator on a button.

```
/*
  This will hide the focus indicator if the element receives focus via the
  mouse, but it will still show up on keyboard focus.
```

```
*/
.js-focus-visible :focus:not(.focus-visible) {
  outline: none;
}

/*
  Optionally: Define a strong focus indicator for keyboard focus.
  If you choose to skip this step, then the browser's default focus
  indicator will be displayed instead.
*/
.js-focus-visible .focus-visible {
  …
}
```

{% Glitch { id: 'focus-visible', path: 'index.html', height: 346 } %}

Use semantic HTML for easy keyboard wins

By using the correct semantic HTML elements you may be able to meet most or all of your keyboard access needs. That means less time fiddling with `tabindex`, and more happy users!

Keyboard support for free (and better mobile experiences)

There are a number of built-in interactive elements with proper semantics and keyboard support. The ones most developers use are:

- `<a>`
- `<button>`
- `<input>` (and its many types)
- `<select>`
- `<textarea>`

In addition, elements with the `contenteditable` attribute are sometimes used for freeform text entry.

It's easy to overlook the built-in keyboard support that these elements offer. Below are some example elements to explore. Instead of using your mouse, try using your keyboard to operate them. You can use `TAB` (or `SHIFT + TAB`) to move between controls, and you can use the arrow keys and keys like `ENTER` and `SPACE` to manipulate their values.

{% Glitch { id: 'interactive-elements', path: 'index.html', height: 450 } %}

If you have a phone handy, you can see that many times these built-in elements have unique interactions on mobile. Attempting to reproduce these mobile interactions yourself is a lot of work! It's another good reason to stick to built-in elements whenever possible.

Use `button` instead of `div`

A common accessibility anti-pattern is to treat a non-interactive element, like a `div` or a `span`, as a button by adding a click handler to it.

But to be considered accessible, a button should:

- Be focusable via the keyboard
- Support being disabled
- Support the `ENTER` or `SPACE` keys to perform an action
- Be announced properly by a screen reader

A `div` button has none of these things. That means you'll need to write additional code to replicate what the `button` element gives you for free!

For example, `button` elements have a neat trick called **synthetic click activation**. If you add a "click" handler to a `button`, it will run when the user presses `ENTER` or `SPACE`. A `div` button doesn't have this feature, so you'll need to write additional code to listen for the `keydown` event, check that the keycode is `ENTER` or `SPACE`, and then run your click handler. Ouch! That's a lot of extra work!

Compare the difference in this example. `TAB` to either control, and use `ENTER` and `SPACE` to attempt to click on them.

{% Glitch { id: 'synthetic-click', path: 'index.html', height: 346 } %}

If you have `div` buttons in your existing site or application, consider swapping them out for `button` elements. `button` is easy to style and full of accessibility wins!

Links versus buttons

Another common anti-pattern is to treat links as buttons by attaching JavaScript behavior to them.

```
<a href="#" onclick="// perform some action">
```

Both buttons and links support some form of synthetic click activation. So which should you choose?

- If clicking on the element will perform an *action* on the page, use `<button>`.
- If clicking on the element will *navigate* the user to a new page then use `<a>`. This includes single page web apps that load new content and update the URL using the History API.

The reason for this is that buttons and links are announced differently by screen readers. Using the correct element helps screen reader users know which outcome to expect.

{% Assessment 'self-assessment' %}

Styling

Some built-in elements, in particular `<input>`, can be difficult to style. With a bit of clever CSS you may be able to work around some of these limitations. The (hilariously named) WTFForms project contains an example stylesheet that demonstrates a number of techniques for styling some of the tougher built-in elements.

Next steps

Using built-in HTML elements can greatly improve the accessibility of your site, and significantly cut down on your workload. Try tabbing through your site and look for any controls which lack keyboard support. If possible, switch them out for standardized HTML alternatives.

Sometimes you may find an element that doesn't have a counterpart in HTML. That's okay! Read on to learn how to add keyboard support to custom interactive controls using `tabindex`.

What is accessibility?

An accessible site is one whose content can be accessed regardless of any user's impairments and whose functionality can also be operated by the most diverse range of users possible.

As developers, it's easy to assume that all users can see and use a keyboard, mouse, or touch screen to interact with your page. This can lead to an experience that works well for some people but creates issues for others that range from simple annoyances to complete blockers.

Understanding the diverse needs of your users

When learning about accessibility, it helps to have an understanding of the diverse range of users in the world and the kinds of accessibility topics that affect them. To explain further, here's an informative video from Victor Tsaran, a Technical Program Manager at Google.

{% YouTube 'RHIVx4m8V4M' %}

Generally speaking, accessibility concerns can be split into four broad categories:

- Vision
- Motor/dexterity
- Auditory
- Cognitive

Planning for accessibility means thinking about users who are experiencing some type of impairment or disability in one or more of these categories. Bear in mind that that experience might be non-physical or temporary—for instance, trying to read a screen outside during a bright, sunny day or operating a device one-handed while carrying a cup of coffee.

When you plan for these situations upfront, you end up with an experience that is more robust and works for more users regardless of their ability or context.

Vision

Vision impairments range from limited or low vision to complete blindness. Users with low vision may use a combination of screen magnification, high contrast themes, and text-to-speech to

access content. Some users may rely on a screen reader or braille display to navigate a page, perform actions, and read descriptions of content and controls.

Motor/dexterity

Motor and dexterity impairments may affect a user's ability to use a mouse, touchscreen, or other pointing device. Some users may rely on alternative input devices to access content. These devices might include a keyboard, head- or eye-tracking software, switch devices, sip-and-puff devices, or voice access.

Auditory

Auditory impairments range from difficulty hearing certain frequencies, to speech processing issues, to a total inability to hear sound. Users experiencing an auditory impairment may rely on captions or transcripts to provide an alternative to sound in an interface.

Cognitive

Cognitive impairment is a broad category, encompassing topics such as ADHD, dyslexia, and autism, just to name a few. The accommodations for these users are quite diverse, but generally speaking, users may seek to minimize distractions, flashing, heavy animations, and anything which shifts the user's context around the page in an unexpected way. Users may also use custom colors and styles to improve readability or prevent headaches.

Next steps

Now that you have a high level understanding of accessibility, it's time to dive in to more specific details, starting with keyboard access.