The most common link pseudo-classes are as follows:

- `:link` – A link the user has never visited.
- `:visited` – A link the user has visited before.
- `:hover` – A link with the user's mouse over it.
- `:active` – A link that's being pressed down by a mouse (or finger).

Let's take a look at all of these by adding the following rules to our CSS stylesheet (also note our use of keyword colors instead of our usual hex codes):

```
a:link {
  color: blue;
  text-decoration: none;
}
a:visited {
  color: purple;
}
a:hover {
  color: aqua;
  text-decoration: underline;
}
a:active {
  color: red;
}
```

If you've never been to the InternetingIsHard.com home page, you should see a blue link. Otherwise, you'll see a purple link. When you hover over the link, it will turn aqua, and when you push down on it, it'll turn red.

VISITED HOVER STATE

The above snippet is just fine for most websites, but take a closer look at the `a:visited` behavior by changing the `href` attribute to a URL that you've been to before. Our `a:hover` style is applied to both visited and unvisited links. We can refine our links even more by stringing pseudo-classes together. Add this below the previous snippet:

```
a:visited:hover {
  color: orange;
}
```

This creates a dedicated hover style for visited links. Hovering over an unvisited link changes it to aqua, while hovering over a visited link will turn it orange. Fantastic! Except for the fact that this breaks our `a:active` style due to some complicated CSS internals that you'll never want to read about. When you click down, our link won't turn red anymore.

VISITED ACTIVE STATE

We can fix that with `a:visited:active`. Add the following to the end of our stylesheet. Note that, as with our `.call-to-action` class, the order in which these are defined in `styles.css` matters:

```
a:visited:active {
  color: red;
}
```

These last two sections let you style visited links entirely separately from unvisited ones. It's a nice option to have, but again, you're welcome to stop at basic link styles if that's all you need.

PSEUDO-CLASSES FOR BUTTONS

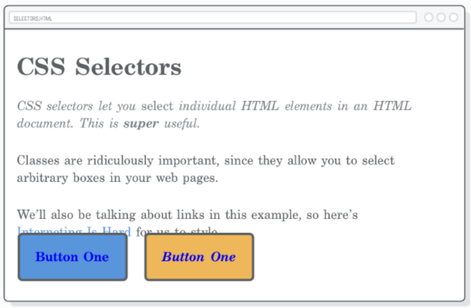
Pseudo-classes aren't just for styling text links—they can be applied to any kind of selector (not just type selectors). Instead of styling `a:link` and friends, we'll be modifying our `.button` class with pseudo-classes in this section. This will let us create buttons that actually go somewhere.

LINK ELEMENTS, NOT DIV ELEMENTS

First, we need change our buttons to be `<a href>` elements instead of generic `<div>` elements, as shown below:

```
<a class='button' href='nowhere.html'>Button One</a>
<a class='button call-to-action' href='nowhere.html'>Button Two</a>
```

If you reload this in your browser, you'll see that we lost some of our styles even though we're using the same classes. This is because `<a>` is an inline element by default and also has a default `color` value.


SELECTORS.HTML

CSS Selectors

CSS selectors let you select *individual HTML elements in an HTML document. This is* ***super*** *useful.*

Classes are ridiculously important, since they allow you to select arbitrary boxes in your web pages.

We'll also be talking about links in this example, so here's Interneting Is Hard for us to style.

We need to change it to a block element and remove some of the default link styling.

BUTTON STYLES

Let's start with `:link` and `:visited` variants. We're using a similar pattern as in the previous section, but since these are buttons, we want to keep both the unvisited and visited color the same. Change the existing `.button` rules to match the following:

```
.button:link,                 /* Change this */
.button:visited {             /* Change this */
  display: block;             /* Add this */
  text-decoration: none;      /* Add this */

  color: #FFF;                /* The rest is the same */
  background-color: #5995DA;
  font-weight: bold;
  padding: 20px;
  text-align: center;
  border: 2px solid #5D6063;
  border-radius: 5px;

  width: 200px;
  margin: 20px auto;
}
```

Notice the new `:link` and `:visited` pseudo-classes in the selector. Without it, our `color` would not override the browser's default `a:link` style. CSS specificity explains why this is the case in greater detail. Next, let's do the hover states:

```
.button:hover,
.button:visited:hover {
  color: #FFF;
  background-color: #76AEED;  /* Light blue */
}
```

Both our buttons will be a lighter blue on hover. Finally, let's make it a little darker when the user presses the mouse down with the `:active` pseudo-class:

```
.button:active,
.button:visited:active {
  color: #FFF;
  background-color: #5995DA;  /* Blue */
}
```

The great part about this is that all the styles we just defined are entirely reusable. Stick a `.button` class on *any* HTML element, and you'll turn it into an interactive button.

THE OTHER BUTTON

Now, what about that second button? It's supposed to have a yellow background, but we broke it with the code from the previous section. Our `.button:link` selector was more "specific" than our current `.call-to-action` rule, so it took precedence. Again, we'll explore this further at the end of the chapter.

For now, let's fix it by applying some pseudo-classes to our `.call-to-action` rule. Replace the existing rule with the following (make sure this appears *after* the new `.button` styles from the previous section):

```
.call-to-action:link,
.call-to-action:visited {
  font-style: italic;
  background-color: #EEB75A;     /* Yellow */
}

.call-to-action:hover,
.call-to-action:visited:hover {
  background-color: #F5CF8E;     /* Light yellow */
}

.call-to-action:active,
.call-to-action:visited:active {
  background-color: #EEB75A;     /* Yellow */
}
```

Since we only added the `.call-to-action` class to our second button, that's the only one that'll turn yellow. Of course, we still need the `.button` class on both `<a>` elements because it defines shared styles like the padding, border radius, and font weight.

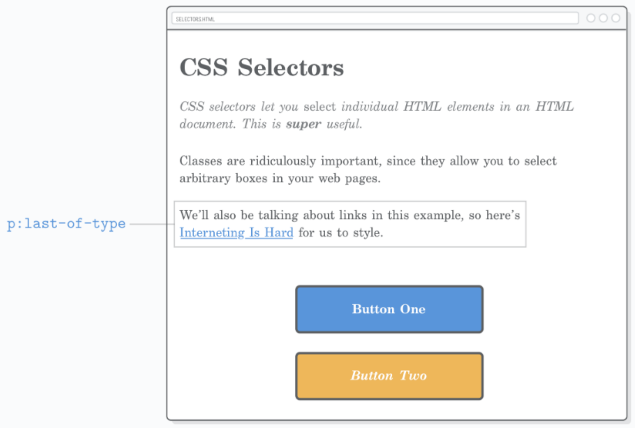
PSEUDO-CLASSES FOR STRUCTURE

Link states are just one aspect of pseudo-classes. There's also a bunch of other pseudo-classes that provide extra information about an element's surroundings. For example, the `:last-of-type` pseudo-class selects the final element of a particular type in its parent element. This gives us an alternative to class selectors for selecting specific elements.

For instance, we could use `:last-of-type` to add some space after the last paragraph of our example page:

```
p:last-of-type {
  margin-bottom: 50px;
}
```

This avoids selecting the first two `<p>` elements *without* requiring a new `class` attribute on the last paragraph:



CSS Selectors

*CSS selectors let you select individual HTML elements in an HTML document. This is **super** useful.*

Classes are ridiculously important, since they allow you to select arbitrary boxes in your web pages.

We'll also be talking about links in this example, so here's Interneting Is Hard for us to style.

We could even use a `:first-of-type` pseudo-class in place of our `.synopsis` class. Replacing the existing `.synopsis` rule with the following snippet should result in the exact same page.

```
p:first-of-type {
  color: #7E8184;
  font-style: italic;
}
```

There's pros and cons to using this method over plain old classes. For instance, this *only* works if our synopsis is a `<p>` element. If we ever wanted to create a multi-paragraph synopsis by wrapping a bunch of `<p>` elements in a `<div class='synopsis'>`, we'd have to rewrite our CSS accordingly. On the other hand, the pseudo-class method lets us style specific elements without having to alter the HTML at all. This gives us a very clean separation of content from presentation.

CAVEATS

Ok, so actually the pseudo-class method is a little more complicated. They're still a useful tool—as long as you know their ins-and-outs. The `:first-of-type` and `:last-of-type` selectors only operate inside their parent element. That is to say, `p:first-of-type` selects the first `<p>` in *every* container element.

We have a single generic `<div>` wrapping our content (`.page`), so this isn't a problem for us. However, consider what happens when we add this to the bottom of our `.page` element:

```
<div class='sidebar'>
  <p>If this page had a sidebar...</p>
  <p>We'd have some problems with pseudo-classes.</p>
</div>
```

We won't be able to make a real sidebar until the next chapter, but this does highlight the complications of pseudo-classes for structure. The first `<p>` element here will also match `p:first-of-type` because the pseudo-class's scope is limited to the parent element.

If you wanted to avoid the sidebar paragraphs and select only the first `<p>` in our `<div class='page'>`, you would need to limit its scope using a child selector, like so:

```
.page > p:first-of-type {
  color: #7E8184;
  font-style: italic;
}
```

All of this is yet another example of how there are many ways to do the same thing in the wonderful world of HTML and CSS. Different developers adhere to different schools of thought. Some like the semantic nature of pseudo-classes, while others go to the far extreme with explicit `class` attributes on *every* HTML element.

ID SELECTORS

"ID selectors" are a more stringent alternative to class selectors. They work pretty much the same way, except you can only have *one* element with the same ID per page, which means you can't reuse styles at all. Instead of a `class` attribute, they require an `id` attribute on whatever HTML element you're trying to select. Try adding one to our second button:

```
<a id='button-2' class='button' href='nowhere.html'>Button Two</a>
```

The corresponding CSS selector must begin with a hash sign (#) opposed to a dot. Adding the following to `styles.css` will change the text color of our yellow button:

```
#button-2 {
  color: #5D6063;  /* Dark gray */
}
```

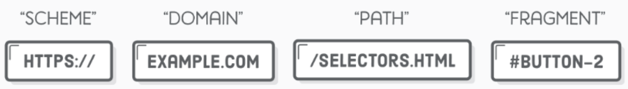
The problem is, if we wanted to share this style with another button, we'd have to give it another unique `id` attribute. Pretty soon, our CSS would start to look pretty gnarly:

```
/* (This is painful to maintain) */
#button-2,
#button-3,
#checkout-button,
#menu-bar-call-to-action {
  color: #5D6063;
}
```

For this reason, ID selectors are generally frowned upon. Use class selectors instead.

URL FRAGMENTS

`id` attributes need to be unique because they serve as the target for "URL fragments", which we sort of glossed over in our discussion of URLs. Fragments are how you point the user to a specific part of a web page. They look like an ID selector stuck on the end of a URL.



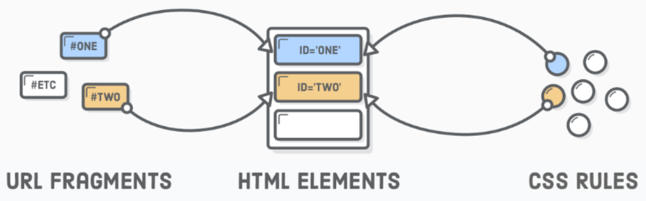
For example, if we wanted to point the user to our second button, we could use the following. Note that we can omit the URL entirely if we're linking to a different section on the same page:

```
<!-- From the same page -->
<a href='#button-2'>Go to Button Two</a>
```

```html
<a href='#button-2'>Go to Button Two</a>

<!-- From a different page -->
<a href='selectors.html#button-2'>Go to Button Two</a>
```

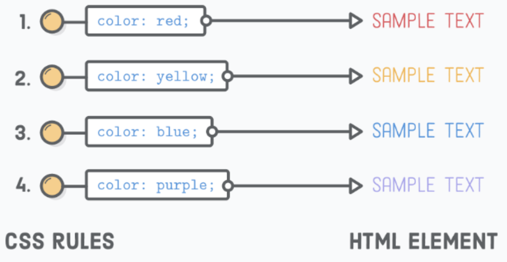
If you add the first option to our `selectors.html` page and click it, you'll see the URL in the browser change. To actually see it jump down to the second button, you'll need to add some more dummy content to the page or make the window height very short, as the browser will limit scrolling to the visible page.



This overlapping functionality is more reason to avoid ID selectors. They create a dependency between your website's URLs and your CSS styles. Imagine using a bunch of `id` attributes on your headings as both URL fragments and ID selectors. If you forgot to update your stylesheet every time you edited the URL of a section, you would actually break your website.

CSS SPECIFICITY

Earlier in this chapter, we talked about how order matters when it comes to CSS rules in an external stylesheet. All else being equal, rules are applied from top-to-bottom. This allowed us to override rules in a predictable manner.


1. color: red; SAMPLE TEXT
2. color: yellow; SAMPLE TEXT
3. color: blue; SAMPLE TEXT
4. color: purple; SAMPLE TEXT
CSS RULES
HTML ELEMENT

Unfortunately, not all CSS selectors are created equal. "CSS specificity" is the weight given to different categories of selectors. This means that certain selectors will *always* override other ones, regardless of where they appear in the stylesheet.

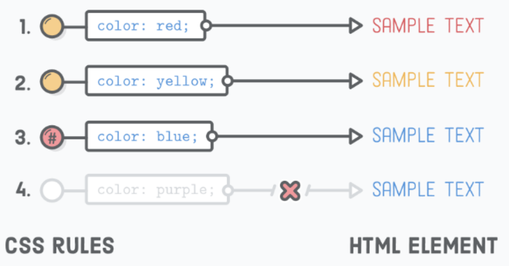
Let's start by seeing where this *doesn't* break. If you add the following after our existing `.call-to-action` rules, it will override the previous `background-color`. If you stick it at the top of the file, it'll get overridden later on, so our button won't turn red. This is expected behavior.

```css
.call-to-action:link,
.call-to-action:visited {
  background-color: #D55C5F;    /* Red */
}
```

Now, watch what happens when we try to do the same thing with an ID selector. First, be sure to delete the previous snippet, then try adding this *before* our existing `.call-to-action` rules:

```
#button-2 {
  background-color: #D55C5F;    /* Red */
}
```

ID selectors have higher specificity than class selectors, so this *will* turn our second button red even though we try to set the `background-color` with `.call-to-action:link` later in our stylesheet. The whole "order matters" concept only works when all your rules have the same specificity.



1. color: red; SAMPLE TEXT
2. color: yellow; SAMPLE TEXT
3. color: blue; SAMPLE TEXT
4. color: purple; SAMPLE TEXT
CSS RULES
HTML ELEMENT

The specificity of selectors we've seen in this chapter are show below, from greatest to least:

- `#button-2`
- `.button:link`
- `a:link` and `.synopsis em` (they're equal)
- `.button`
- `a`

This can get very confusing. It's such a big problem that an entire methodology called "BEM" has evolved. BEM attempts to make CSS rules more reusable by making *everything* a class selector. This completely eliminates the potential for specificity issues.

BEM is outside the scope of this tutorial. The takeaway here is that CSS rules are not necessarily applied in sequential order, but you should try to make the browser do so by writing CSS that uses the same specificity.

SUMMARY

In this chapter, we got some hands-on experience with class selectors, descendant selectors, pseudo-classes, link styling, and ID selectors. The goal of all this was to be able to target a specific HTML element from your CSS. Class selectors are by far the most versatile and come with the least amount of drawbacks. As a result, they'll become part of your daily life as a web developer.

Like it or not, things got a lot more complicated this chapter. We're now able to make our CSS interact with an HTML document in half a dozen different ways. Furthermore, over the next few chapters, we'll begin to see a dependency between our HTML's structure and the layout of a web page. With all this interplay between CSS and HTML, it can be hard to know where to start building a new web page.

The separation of content from presentation helps guide this process. You need content before you can present it, so your first step is usually to mark up your raw content with HTML tags. Once that's prepared, you're ready to add `class` attributes to your elements and style them one-by-one. When you discover a need for some extra structure to create a desired layout (e.g., turn a group of elements into a sidebar), that's when you start wrapping your content in container <div>'s.

This chapter covered almost all the CSS selectors that power real websites. We've got the tools we need to dive much deeper into complex CSS layouts. In the next installment of *HTML and CSS Is Hard*, we'll learn how to create columns and sidebars using CSS floats.

NEXT CHAPTER >