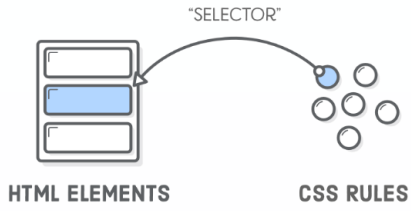
CSS SELECTORS

Nº 6. OF HTML & CSS IS HARD

A friendly web development tutorial for plucking out HTML elements

Way back in the Links and Images chapter, we learned how to connect an HTML document to other files in our project. "CSS selectors" are similar, except instead of navigating between whole files, they let us map a single CSS rule to a specific HTML element. This makes it possible to *selectively* style individual elements while ignoring others.



Unless you want every section of your website to look exactly the same, this is a crucial bit of functionality for us. It's how we say things like "I want this paragraph to be blue and that other paragraph to be yellow." Until now, we've only been able to turn *all* our paragraphs blue (or yellow).

The only CSS selector we've seen so far is called the "type selector", which targets all the matching elements on a page. In this chapter, we'll explore more granular ways to style a web page with class selectors, descendant selectors, pseudo-classes, and ID selectors.

SETUP

We'll only need one HTML file and a CSS stylesheet for our example this chapter. Create a new folder called `css-selectors` and new web page called `selectors.html` with the following markup:

```
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='UTF-8'/>
    <title>CSS Selectors</title>
    <link rel='stylesheet' href='styles.css'/>
  </head>
  <body>
    <h1>CSS Selectors</h1>

    <p>CSS selectors let you <em>select</em> individual HTML elements in an HTM
       document. This is <strong>super</strong> useful.</p>

    <p>Classes are ridiculously important, since they allow you to select
       arbitrary boxes in your web pages.</p>
```

```
    <p>We'll also be talking about links in this example, so here's
       <a href='https://internetingishard.com'>Interneting Is Hard</a> for us t
       style.</p>

    <div>Button One</div>

  </body>
</html>
```

Go ahead and create that `styles.css` stylesheet in the same folder, too. This gives us everything we need to explore CSS selectors.

If you're just diving into this tutorial series, be sure to have a quick read through the Introduction to get set up with the Atom text editor.

CLASS SELECTORS

"Class selectors" let you apply CSS styles to a specific HTML element. They let you differentiate between HTML elements of the same type, like when we had two `<div>` elements in the previous chapter, but only wanted to style one of them. Class selectors require two things:

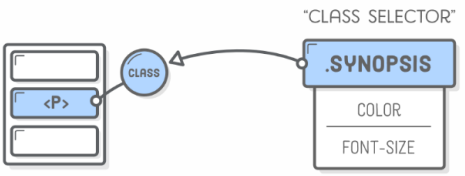
- A `class` attribute on the HTML element in question.

If you're just diving into this tutorial series, be sure to have a quick read through the Introduction to get set up with the Atom text editor.

CLASS SELECTORS

"Class selectors" let you apply CSS styles to a specific HTML element. They let you differentiate between HTML elements of the same type, like when we had two `<div>` elements in the previous chapter, but only wanted to style one of them. Class selectors require two things:

- A `class` attribute on the HTML element in question.
- A matching CSS class selector in your stylesheet.



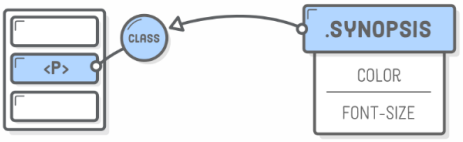
We can use a class selector to style the first paragraph of our example page differently than the rest of them. This could be, for instance, the synopsis of a newspaper article. First, let's add a `class` attribute to the desired we had two `<div>` elements in the previous chapter, but only wanted to style one of them. Class selectors require two things:

- A `class` attribute on the HTML element in question.
- A matching CSS class selector in your stylesheet.

We can use a class selector to style the first paragraph of our example page differently than the rest of them. This could be, for instance, the synopsis of a newspaper article. First, let's add a `class` attribute to the desired paragraph:

```
<p class='synopsis'>CSS selectors let you <em>select</em> individual HTML
   elements in an HTML document. This is <strong>super</strong> useful.</p>
```

Now, we can pluck out that `<p class='synopsis'>` element in our CSS with the following (add this to `styles.css`):



We can use a class selector to style the first paragraph of our example page differently than the rest of them. This could be, for instance, the synopsis of a newspaper article. First, let's add a `class` attribute to the desired paragraph:

```
<p class='synopsis'>CSS selectors let you <em>select</em> individual HTML
   elements in an HTML document. This is <strong>super</strong> useful.</p>
```

Now, we can pluck out that `<p class='synopsis'>` element in our CSS with the following (add this to `styles.css`):

```
.synopsis {
  color: #7E8184;        /* Light gray */
  font-style: italic;
}
```

We can use a class selector to style the first paragraph of our example page differently than the rest of them. This could be, for instance, the synopsis of a newspaper article. First, let's add a `class` attribute to the desired paragraph:

```
<p class='synopsis'>CSS selectors let you <em>select</em> individual HTML
   elements in an HTML document. This is <strong>super</strong> useful.</p>
```

Now, we can pluck out that `<p class='synopsis'>` element in our CSS with the following (add this to `styles.css`):

```
.synopsis {
  color: #7E8184;        /* Light gray */
  font-style: italic;
}
```

This rule is *only* applied to elements with the corresponding `class` attribute. Notice the dot (`.`) prefixing the class name. This distinguishes class selectors from the type selectors that we've been working with before this chapter.

differently than the rest of them. This could be, for instance, the synopsis of a newspaper article. First, let's add a `class` attribute to the desired paragraph:

```
<p class='synopsis'>CSS selectors let you <em>select</em> individual HTML
   elements in an HTML document. This is <strong>super</strong> useful.</p>
```

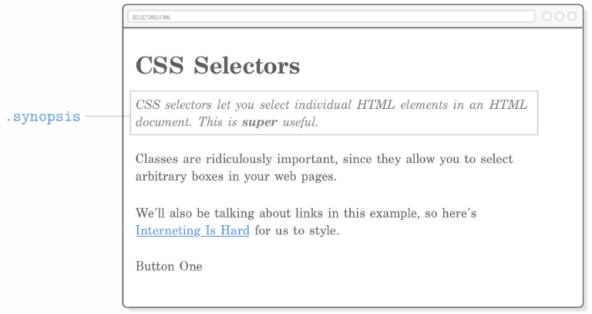
Now, we can pluck out that `<p class='synopsis'>` element in our CSS

with the following (add this to `styles.css`):

```
.synopsis {
  color: #7E8184;        /* Light gray */
  font-style: italic;
}
```

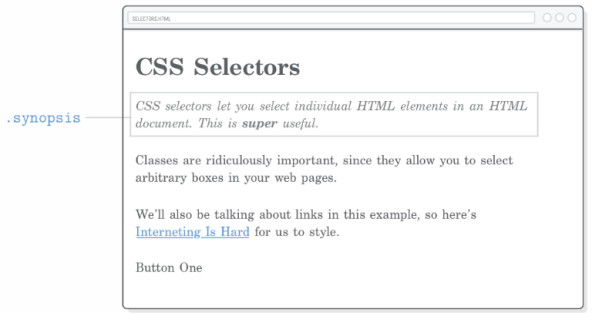
This rule is *only* applied to elements with the corresponding `class` attribute. Notice the dot (`.`) prefixing the class name. This distinguishes class selectors from the type selectors that we've been working with before this chapter.

This rule is *only* applied to elements with the corresponding `class` attribute. Notice the dot (`.`) prefixing the class name. This distinguishes class selectors from the type selectors that we've been working with before this chapter.



CLASS NAMING CONVENTIONS

selectors from the type selectors that we've been working with before this chapter.



CLASS NAMING CONVENTIONS

The value of the HTML `class` attribute can be (almost) anything you want as long as it matches the selector in your CSS. The standard naming convention for classes is to use all lowercase and hyphens for spaces, just like file and folder names.

Adding a `class` attribute doesn't alter the semantic meaning of your HTML document at all—it's purely for hooking into your CSS stylesheet. However,

it's still usually a good idea to avoid naming classes based on their appearance. If we chose to name our class `.italic`, we wouldn't be able to do much besides make it italic in our CSS without leading to a confusing situation. Using something semantic like `.synopsis` gives us more freedom for our CSS to customize how that synopsis is displayed.

CLASS NAMING CONVENTIONS

The value of the HTML `class` attribute can be (almost) anything you want as long as it matches the selector in your CSS. The standard naming convention for classes is to use all lowercase and hyphens for spaces, just like file and folder names.

Adding a `class` attribute doesn't alter the semantic meaning of your HTML document at all—it's purely for hooking into your CSS stylesheet. However, it's still usually a good idea to avoid naming classes based on their appearance. If we chose to name our class `.italic`, we wouldn't be able to do much besides make it italic in our CSS without leading to a confusing situation. Using something semantic like `.synopsis` gives us more freedom for our CSS to customize how that synopsis is displayed.

CLASS NAMING CONVENTIONS

The value of the HTML `class` attribute can be (almost) anything you want as long as it matches the selector in your CSS. The standard naming convention for classes is to use all lowercase and hyphens for spaces, just like file and folder names.

Adding a `class` attribute doesn't alter the semantic meaning of your HTML document at all—it's purely for hooking into your CSS stylesheet. However, it's still usually a good idea to avoid naming classes based on their appearance. If we chose to name our class `.italic`, we wouldn't be able to do much besides make it italic in our CSS without leading to a confusing situation. Using something semantic like `.synopsis` gives us more freedom for our CSS to customize how that synopsis is displayed.

CLASS NAMING CONVENTIONS

The value of the HTML `class` attribute can be (almost) anything you want as long as it matches the selector in your CSS. The standard naming convention for classes is to use all lowercase and hyphens for spaces, just like file and folder names.

Adding a `class` attribute doesn't alter the semantic meaning of your HTML document at all—it's purely for hooking into your CSS stylesheet. However, it's still usually a good idea to avoid naming classes based on their appearance. If we chose to name our class `.italic`, we wouldn't be able to do much besides make it italic in our CSS without leading to a confusing situation. Using something semantic like `.synopsis` gives us more freedom for our CSS to customize how that synopsis is displayed.

MORE USEFUL DIVS

The `class` attribute isn't limited to `<p>` elements—it can be defined on *any* HTML element. So, armed with CSS class selectors, our generic `<div>` and `<span>` boxes from the previous chapter become much, much more useful. We can use them to style both individual elements as well as arbitrary sections of our web page.

Let's start with individual elements by recreating our button from the previous chapter. This time, we'll use a class instead of a `div` selector. Add the following to `styles.css`:

```
.button {
  color: #FFF;
  background-color: #5995DA;    /* Blue */
  font-weight: bold;
  padding: 20px;
  text-align: center;
  border: 2px solid #5D6063;    /* Dark gray */
  border-radius: 5px;

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

Of course, we need a corresponding `class` attribute for this to work. Change the `<div>` in `selectors.html` to match the following:

```
<div class='button'>Button One</div>
```

Unlike the previous chapter that styled *all* the `<div>` elements, this lets us use it for other things besides buttons.

CONTAINER DIVS

Remember that `<div>` doesn't alter the semantic structure of a page. This makes it a great tool for defining the *presentational* structure of a web page. Change the `<div>` in `selectors.html` to match the following:

```
<div class='button'>Button One</div>
```

Unlike the previous chapter that styled *all* the `<div>` elements, this lets us use it for other things besides buttons.

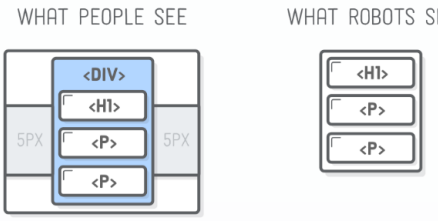
CONTAINER DIVS

Remember that doesn't alter the semantic structure of a page. This makes it a great tool for defining the *presentational* structure of a web page. By wrapping other HTML elements in tags, we can organize our site into larger layout-oriented chunks without messing up how search engines view our content.

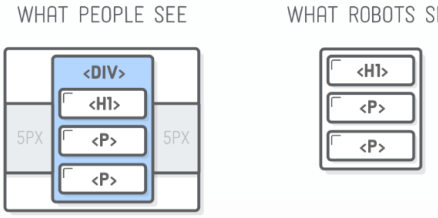
WHAT PEOPLE SEE WHAT ROBOTS SEE

CONTAINER DIVS

Remember that doesn't alter the semantic structure of a page. This makes it a great tool for defining the *presentational* structure of a web page. By wrapping other HTML elements in tags, we can organize our site into larger layout-oriented chunks without messing up how search engines view our content.



Remember that doesn't alter the semantic structure of a page. This makes it a great tool for defining the *presentational* structure of a web page. By wrapping other HTML elements in tags, we can organize our site into larger layout-oriented chunks without messing up how search engines view our content.



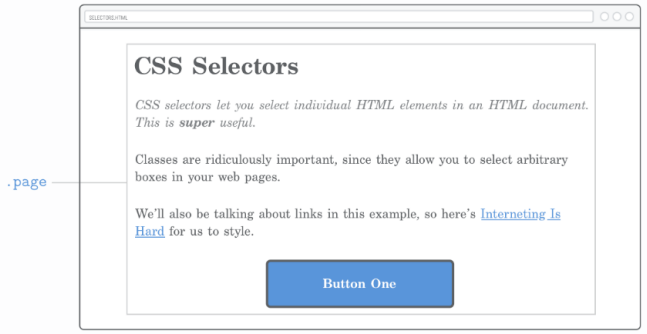
For example, let's try to create a fixed-width layout using the auto-margin technique that we learned in the previous chapter. First, wrap our entire document in a generic and give it a unique class:

```
<body>
  <div class='page'>  <!-- Add this -->
    <h1>CSS Selectors</h1>

    <p  class='synopsis'>CSS selectors let you <em>select</em> individual HTML
    elements in an HTML document. This is <strong>super</strong> useful.</p>

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

    <p>We'll also be talking about links in this example, so here's
    <a href='https://internetingishard.com'>Interneting Is Hard</a> for us to
    style.</p>

    <div class='button'>Button One</div>
  </div>  <!-- And this -->
</body>
```

Then, add the following to `styles.css`:

```
.page {
  width: 600px;
  margin: 0 auto;
}
```

No matter how you resize the browser window, our web page will always be 600 pixels wide and centered in the available space. Note that this was the exact same way we centered our button, but now we're doing it to multiple elements at the same time by nesting them in a generic container.



This is how layouts are defined in more complex web pages. For instance, if our page had a sidebar, we would nest all the sidebar elements in *another* `<div>` with a `.sidebar` class. We'll see this in action in the next chapter. For now, the key takeaway is that without class selectors to differentiate our `<div>` elements, none of this would be possible.

REUSING CLASS STYLES

The same class can be applied to multiple elements in a single HTML document. This means that we can now reuse arbitrary CSS declarations wherever we want. To create another button, all we have to do is add another HTML element with the same class:

```
<div class='button'>Button One</div>
<div class='button'>Button Two</div>
```

`<div>` elements, none of this would be possible.

REUSING CLASS STYLES

The same class can be applied to multiple elements in a single HTML document. This means that we can now reuse arbitrary CSS declarations wherever we want. To create another button, all we have to do is add another HTML element with the same class:

```
<div class='button'>Button One</div>
<div class='button'>Button Two</div>
```

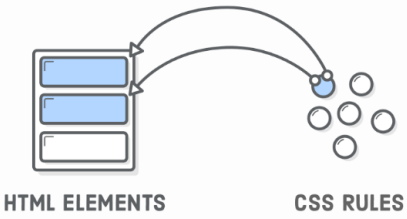
This gives us a second button that looks just like the first one—without writing a single line of CSS! Organizing similar graphical elements into reusable CSS rules like this makes life much easier as a web developer. If we ever wanted to, say, change the button color, we would only have to do it in one place and all our buttons would automatically update.

REUSING CLASS STYLES

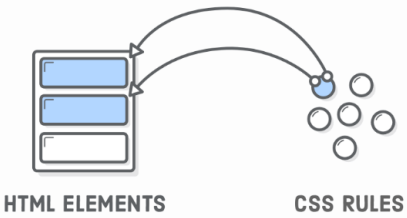
The same class can be applied to multiple elements in a single HTML document. This means that we can now reuse arbitrary CSS declarations wherever we want. To create another button, all we have to do is add another HTML element with the same class:

```
<div class='button'>Button One</div>
<div class='button'>Button Two</div>
```

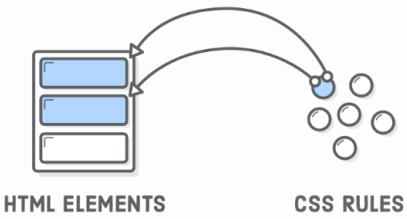
This gives us a second button that looks just like the first one—without writing a single line of CSS! Organizing similar graphical elements into reusable CSS rules like this makes life much easier as a web developer. If we ever wanted to, say, change the button color, we would only have to do it in one place and all our buttons would automatically update.



This gives us a second button that looks just like the first one—without writing a single line of CSS! Organizing similar graphical elements into reusable CSS rules like this makes life much easier as a web developer. If we ever wanted to, say, change the button color, we would only have to do it in one place and all our buttons would automatically update.



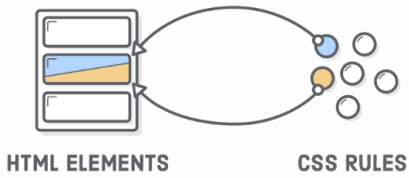
reusable CSS rules like this makes life much easier as a web developer. If we ever wanted to, say, change the button color, we would only have to do it in one place and all our buttons would automatically update.



MODIFYING CLASS STYLES

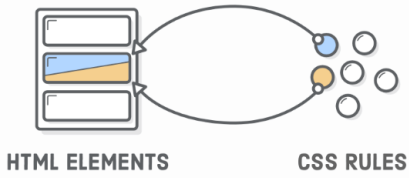
What if we want to alter our second button a little bit? Fortunately, we can

What if we want to alter our second button a little bit? Fortunately, we can apply multiple classes to the *same* HTML element, too. The styles from each class will be applied to the element, giving us the opportunity to both reuse styles from `.button` and override some of them with a new class.



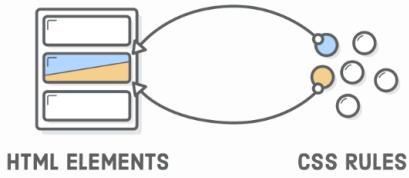
MODIFYING CLASS STYLES

What if we want to alter our second button a little bit? Fortunately, we can apply multiple classes to the *same* HTML element, too. The styles from each class will be applied to the element, giving us the opportunity to both reuse styles from `.button` and override some of them with a new class.



Go ahead and add another class to our second button with the following

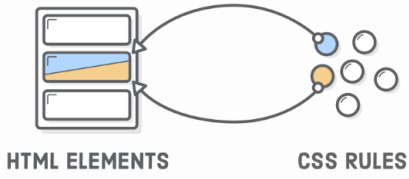
What if we want to alter our second button a little bit? Fortunately, we can apply multiple classes to the *same* HTML element, too. The styles from each class will be applied to the element, giving us the opportunity to both reuse styles from `.button` and override some of them with a new class.



Go ahead and add another class to our second button with the following markup. Notice how multiple classes live in the same `class` attribute, separated by spaces:

```
<div class='button call-to-action'>Button Two</div>
```

This element now has two separate classes, and we can use either of them to style it. This opens up some options. Styles shared by both buttons can



Go ahead and add another class to our second button with the following markup. Notice how multiple classes live in the same `class` attribute, separated by spaces:

```
<div class='button call-to-action'>Button Two</div>
```

This element now has two separate classes, and we can use either of them to style it. This opens up some options. Styles shared by both buttons can live in the `.button` class (as they already do), and styles specific to the second button reside in the `.call-to-action` class (be sure to add this *after* the `.button` rule):

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

ORDER MATTERS

There's a couple of important things going on with our second button now:

```
<div class='button call-to-action'>Button Two</div>
```

This element now has two separate classes, and we can use either of them to style it. This opens up some options. Styles shared by both buttons can live in the `.button` class (as they already do), and styles specific to the second button reside in the `.call-to-action` class (be sure to add this *after* the `.button` rule):

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

ORDER MATTERS

There's a couple of important things going on with our second button now:

- It's adding a *new* `font-style` declaration to the original `.button` rule.
- It's *overriding* an existing `background-color` style from `.button`.

Overriding occurs because of the order of `.call-to-action` and `.button` in our stylesheet. When there's two conflicting properties in a CSS file, the last one is always the one that gets applied. So, if you moved `.call-to-`

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

ORDER MATTERS

There's a couple of important things going on with our second button now:

- It's adding a *new* `font-style` declaration to the original `.button` rule.
- It's *overriding* an existing `background-color` style from `.button`.

Overriding occurs because of the order of `.call-to-action` and `.button` in our stylesheet. When there's two conflicting properties in a CSS file, the last one is always the one that gets applied. So, if you moved `.call-to-action` to the top of `styles.css`, `.button` would have the final word on the value of `background-color`, and it would remain blue.

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

ORDER MATTERS

There's a couple of important things going on with our second button now:

- It's adding a *new* `font-style` declaration to the original `.button` rule.
- It's *overriding* an existing `background-color` style from `.button`.

Overriding occurs because of the order of `.call-to-action` and `.button` in our stylesheet. When there's two conflicting properties in a CSS file, the last one is always the one that gets applied. So, if you moved `.call-to-action` to the top of `styles.css`, `.button` would have the final word on the value of `background-color`, and it would remain blue.

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

ORDER MATTERS

There's a couple of important things going on with our second button now:

- It's adding a *new* `font-style` declaration to the original `.button` rule.
- It's *overriding* an existing `background-color` style from `.button`.

Overriding occurs because of the order of `.call-to-action` and `.button` in our stylesheet. When there's two conflicting properties in a CSS file, the last one is always the one that gets applied. So, if you moved `.call-to-action` to the top of `styles.css`, `.button` would have the final word on the value of `background-color`, and it would remain blue.

This means that the order of the `class` attribute in our HTML element has no effect on override behavior. Multiple classes on a single element are applied "equally" (for lack of a better term), so the precedence is determined solely by the order of the rules in `styles.css`. In other words, the following elements are effectively equivalent:

```
<!-- These result in the same rendered page -->
<div class='button call-to-action'>Button Two</div>
<div class='call-to-action button'>Button Two</div>
```

Overriding occurs because of the order of `.call-to-action` and `.button` in our stylesheet. When there's two conflicting properties in a CSS file, the last one is always the one that gets applied. So, if you moved `.call-to-action` to the top of `styles.css`, `.button` would have the final word on the value of `background-color`, and it would remain blue.

This means that the order of the `class` attribute in our HTML element has no effect on override behavior. Multiple classes on a single element are applied "equally" (for lack of a better term), so the precedence is determined solely by the order of the rules in `styles.css`. In other words, the following elements are effectively equivalent:

```
<!-- These result in the same rendered page -->
<div class='button call-to-action'>Button Two</div>
<div class='call-to-action button'>Button Two</div>
```

This does, however, get more complicated when CSS specificity is involved, which we'll discuss at the end of this chapter.

This means that the order of the `class` attribute in our HTML element has no effect on override behavior. Multiple classes on a single element are applied "equally" (for lack of a better term), so the precedence is determined solely by the order of the rules in `styles.css`. In other words, the following elements are effectively equivalent:

```
<!-- These result in the same rendered page -->
```

```
<div class='button call-to-action'>Button Two</div>
<div class='call-to-action button'>Button Two</div>
```

This does, however, get more complicated when CSS specificity is involved, which we'll discuss at the end of this chapter.

DESCENDANT SELECTORS

You may have noticed that the `<em>` in our first paragraph is no longer distinguishable from its surround text, since our `.synopsis` rule made everything italic.

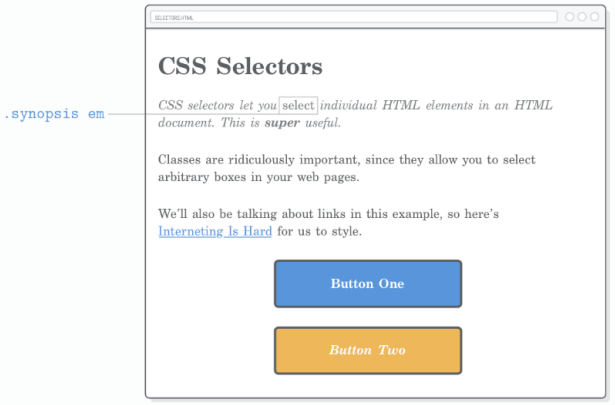
To alter that `<em>` element, we could add another class directly to it, but that won't result in very maintainable code. We want to treat `.synopsis` as

This does, however, get more complicated when CSS specificity is involved, which we'll discuss at the end of this chapter.

DESCENDANT SELECTORS

You may have noticed that the `<em>` in our first paragraph is no longer distinguishable from its surround text, since our `.synopsis` rule made everything italic.

To alter that `<em>` element, we could add another class directly to it, but that won't result in very maintainable code. We want to treat `.synopsis` as its own independent component that we can style entirely from CSS (i.e., without requiring alterations to our HTML just for the sake of styling something.)

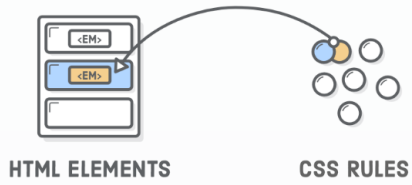


This is what "descendant selectors" are for. They let you target only those elements that are *inside* of another element. For example, we can pull out that `<em>` in the `.synopsis` paragraph with the following:

```
.synopsis em {
  font-style: normal;
}
```

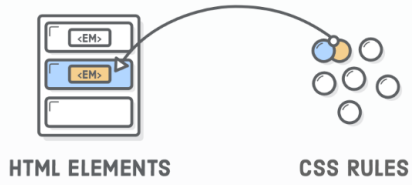
Adding this rule to `styles.css` will make the `<em>` display as upright (roman) characters, thus differentiating it from the italics we put on the entire `<p>` text. The rest of the `<em>` elements on the page will be

unaffected.



Descendant selectors aren't limited to class selectors—you can combine any other group of selectors this way. For instance, if we wanted to select only `<em>` elements inside of headings, we might use something like this:

Adding this rule to `styles.css` will make the `<em>` display as upright (roman) characters, thus differentiating it from the italics we put on the entire `<p>` text. The rest of the `<em>` elements on the page will be unaffected.



Descendant selectors aren't limited to class selectors—you can combine any other group of selectors this way. For instance, if we wanted to select only `<em>` elements inside of headings, we might use something like this:

```
h1 em {
  /* Some other styles */
}
```

Again, the goal of this chapter is to let you apply styles to exactly the element you want. Descendant selectors are a great tool towards this end. You may also want to check out the related "child selector" over at MDN if you've still got room in your toolbox.

DON'T OVERDO IT

You can nest descendant selectors as deep as you want, but don't get carried away. Life gets confusing and terrible when you start writing rules that look like this:

```
/* Try to avoid this */
.article h2 .subheading em {
  /* Special styles */
}
```

Again, the goal of this chapter is to let you apply styles to exactly the element you want. Descendant selectors are a great tool towards this end. You may also want to check out the related "child selector" over at MDN if you've still got room in your toolbox.

DON'T OVERDO IT

You can nest descendant selectors as deep as you want, but don't get carried away. Life gets confusing and terrible when you start writing rules that look like this:

```
/* Try to avoid this */
.article h2 .subheading em {
  /* Special styles */
}
```

This isn't the least bit reusable because it matches *only* the following HTML structure:

```
<div class='article'>
  <h2>
    <span class='.subheading'>This is <em>really</em> special text</span>
  </h2>
</div>
```

carried away. Life gets confusing and terrible when you start writing rules that look like this:

```
/* Try to avoid this */
.article h2 .subheading em {
  /* Special styles */
}
```

This isn't the least bit reusable because it matches *only* the following HTML structure:

```
<div class='article'>
  <h2>
    <span class='.subheading'>This is <em>really</em> special text</span>
  </h2>
</div>
```

If you ever wanted to apply these styles to an `<h2>` heading that's not wrapped in `<div class='article'>` tags, you're kind of screwed. Same deal if you want to apply them to an `<h3>` heading anywhere on the page. This kind of CSS also leads to a specificity nightmare.

This isn't the least bit reusable because it matches *only* the following HTML structure:

```
<div class='article'>
  <h2>
    <span class='.subheading'>This is <em>really</em> special text</span>
  </h2>
</div>
```

If you ever wanted to apply these styles to an `<h2>` heading that's not wrapped in `<div class='article'>` tags, you're kind of screwed. Same deal if you want to apply them to an `<h3>` heading anywhere on the page. This kind of CSS also leads to a specificity nightmare.

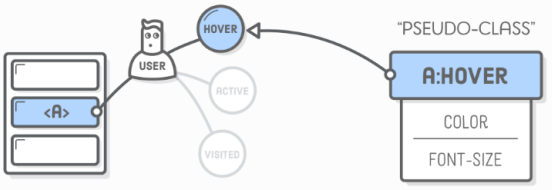
```
<div class='article'>
  <h2>
    <span class='.subheading'>This is <em>really</em> special text</span>
  </h2>
</div>
```

If you ever wanted to apply these styles to an `<h2>` heading that's not wrapped in `<div class='article'>` tags, you're kind of screwed. Same deal if you want to apply them to an `<h3>` heading anywhere on the page. This kind of CSS also leads to a specificity nightmare.

PSEUDO-CLASSES FOR LINKS

So far, all the CSS selectors we've seen map directly to a piece of HTML markup that we wrote. However, there's more going on in a rendered web page than just our HTML content. There's "stateful" information about what the user is doing (opposed to the content we've authored).

The classic example is a link. As a web developer, you create an `<a href>` element. After the browser renders it, the user can interact with that link. They can hover over it, click it, and visit the URL.



CSS "pseudo-classes" provide a mechanism for hooking into this kind of temporary user information. At any given time, an `<a href>` element can be in a number of different states, and you can use pseudo-classes to style each one of them individually. Think of them as class selectors that you don't have to write on your own because they're built into the browser.

BASIC LINK STYLES

Pseudo-classes begin with a colon followed by the name of the desired class. 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 {
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