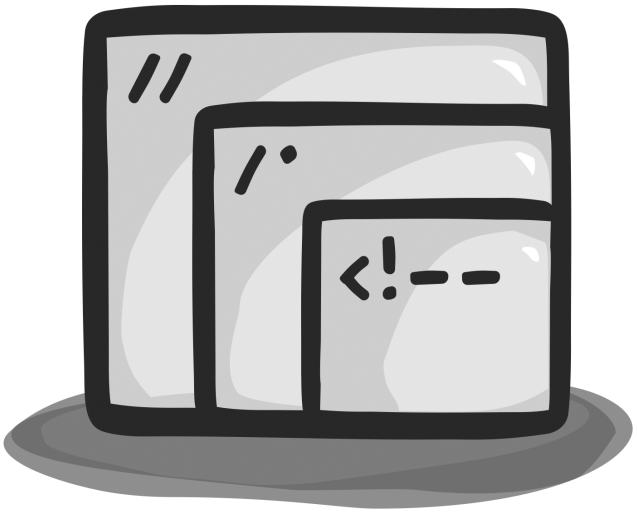


GYMNASIUM

INTRODUCTION TO MODERN WEB DESIGN

Lesson 4 Handout

Getting Started with CSS

ABOUT THIS HANDOUT

This handout includes the following:

- A list of the core concepts covered in this lesson
- The assignment(s) for this lesson
- A list of readings and resources for this lesson including books, articles, and websites mentioned in the videos by the instructor, plus bonus readings and resources hand-picked by the instructor
- A transcript of the lecture videos for this lesson

CORE CONCEPTS

1. Cascading style sheets (CSS) contain one or more rule sets. A rule set has two parts—a selector and a declaration block. The selector is the bit that comes before the opening curly brace (in the following example it is the word “header”), and the declaration block starts with the opening curly brace { and finishes with the closing curly brace }.

```
header {
      color:#663399;
      }
```

2. CSS typically goes into the <head> element and can be included either by using a <link> element, which is for a linked stylesheet, or you could use a <style> element to embed your style sheet into the page. Using the <style> element only affects that page, whereas a linked stylesheet can be linked to from multiple web pages, and those styles can be shared by all of those pages.
3. The concept of CSS proximity is important to understand because it will help manage the styling of your pages, especially as you begin to work with larger and larger websites. Proximity means that if we have two selectors that are equal to one another but are located in different places (one selector might be inline and the other might be located in an external stylesheet) then the selector closest to the element that it’s defining styles for typically “wins”, meaning that its style will be applied.
4. CSS specificity is related to proximity and is the means by which browsers decide which CSS property values are the most relevant to an element and, therefore, will be applied.
5. There are a number of different ways to style the text on your web pages, but everything begins with font size. The recommendation is to use the em value for font-sizing and not pixels—one of the main reasons being that ems provide greater flexibility, especially when designing for multiple screens. Unless specified elsewhere, the default value of 1em is typically 16 pixels.
6. There are a number of different ways to choose as well as specify color in CSS—from color keywords to hexadecimal values and more. A relatively recent addition to CSS is the use of RGBA in which the “A” stands for alpha or transparency.

7. When it comes to controlling the dimensions and appearance of an HTML element using CSS, we generally rely on a combination of changing the width and/or height dimensions as well as margins and padding. In addition, we can choose from two models in order to help us calculate the final size of any element: the border-box model and the content-box model.

ASSIGNMENTS

1. Quiz
2. Explore the box model and the different display modes covered in this lesson. I want you to add borders and padding to each of the elements on your pages and play with the different display modes, especially inline and inline block.
    - How do manipulating these different display modes affect the way the elements lay out with respect to one another?
    - How does adjusting the element size help add weight to it or diminish its visibility compared to other elements?
3. Publish your work on GitHub and post a link to the forum along with your observations.

RESOURCES

- A comprehensive list of the standard CSS properties
  https://developer.mozilla.org/en-US/docs/Web/CSS/Reference
- Understanding the box model (margin, padding, borders)
  https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Introduction_to_the_CSS_box_model
- Working with text styles
  https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_Started/Text_styles
- CSS color
  https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_Started/Color

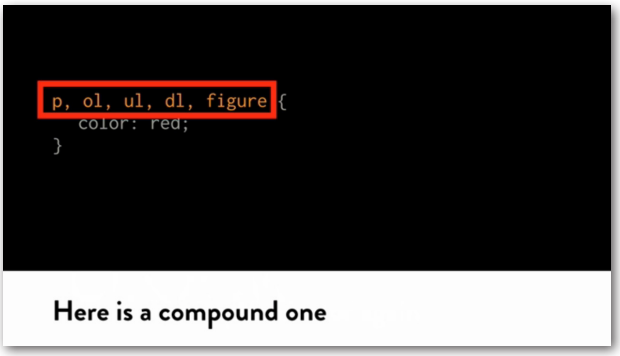
INTRODUCTION

(Note: This is an edited transcript of the Modern Web Design lecture videos. Some students work better with written material than by watching videos alone, so we're offering this to you as an optional, helpful resource. Some elements of the instruction, like live coding, can't be recreated in a document like this one.)

In this lesson, we're going to be talking about how to get started with CSS—in other words, how we can start designing our documents. And in this first chapter, we're going to talk specifically about CSS structures and nomenclature just so you have a general understanding of the bits and pieces that make up a style sheet.

Now first of all, cascading style sheets contain one or more what are called rule sets. Here's an example of three rule sets in a very simple style sheet. The first rule set is highlighted here in orange. Now, a rule set has two parts, a selector and a declaration block. The selector is the bit that comes before the opening curly brace, here highlighted in orange, the p. The declaration block is highlighted here in orange and starts with the opening curly brace and finishes with the closing curly brace.

Now, a selector can propose more than one match. So in this simple example, the selector is for a p, which as you probably guessed selects a p element or a paragraph. But let's say I wanted to apply a color of red to more than just paragraphs. Well, I could use a compound selector, as you see here. This would select paragraphs, ordered lists, unordered lists, definition lists, and figure elements, and set all of them red.



Now, I find this a little bit difficult to read, so I tend to prefer to separate out my compound declarations onto their own lines. It's really important that you remember the commas between each of the different selectors you are trying to put into that compound selector.

Now, a declaration block contains one or more declarations. This rule set example has only a single declaration, color red. This one has two, color red and font weight bold. Again, it's important to note the punctuation here. Semicolons are used to separate each declaration.

Now, each individual declaration within the declaration has two parts itself, a property and a value. Here you see the property, color, highlighted orange, and here you see the value being assigned to that property, red. The property and value of the declaration are always separated by a colon.

As this simple style sheet has shown you, style rules should best be kept in stylesheets, not inline. Now, what does inline mean? Well, here's an example. You could assign a style attribute to any element on the page and assign—let's say, a color red, like I do here. This will only apply this style within this element in this specific context, which is not terribly efficient. It's much more efficient for me to take that color red, which I probably want to apply to all links on the page, and simply say A color red as part of a style sheet. Once I've created my style sheet, I can put that in the head of my document.

So here, coming back to the HTML skeleton you'll probably remember from the previous lessons, you see the CSS should go in the head element. Now, you can include your CSS either using a link element, which is for a linked stylesheet, which is a separate file, or you could use a style element to embed your style sheet into the page.

Either of these options is completely valid, but just remember that a style element in the page is only going to affect that page, whereas a linked stylesheet can be linked to from multiple web pages, and all of those styles can be shared by all of those pages from one single resource.

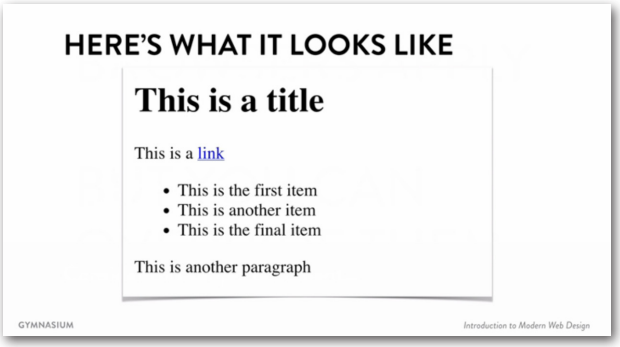
It's worth noting that one style sheet can import another style sheet. It does this using the @import syntax. Here, you see @import another.css file. This is completely legitimate and does work in any browser, but there's a bit of a performance hit in that that separate file then needs to be requested by the browser, which means it's requesting two different CSS files, or maybe three different CSS files depending on how many you have linked versus how many you have importing other files. So that has overhead. It requires the browser to look up that file, make the request for the file, download the file, all of which reduces the speed with which your page is rendered. So generally, avoid @import.

So in this very brief chapter, we've covered how CSS is organized, simple and compound selectors, declaration blocks, declarations, properties and values, and how to include your style sheets. In the next chapter, we're going to tuck into some really important concepts in CSS—proximity and specificity. I'll catch you in a few.

CORE CONCEPTS IN CSS, PART 1

Welcome back. In this chapter, we're going to be discussing core concepts and CSS — or at least starting to.

The first core concept that you should know about is that browsers apply default styles, but you can override them. So if we consider the simple HTML document here, this is what it would look like standardly in a browser like Chrome.



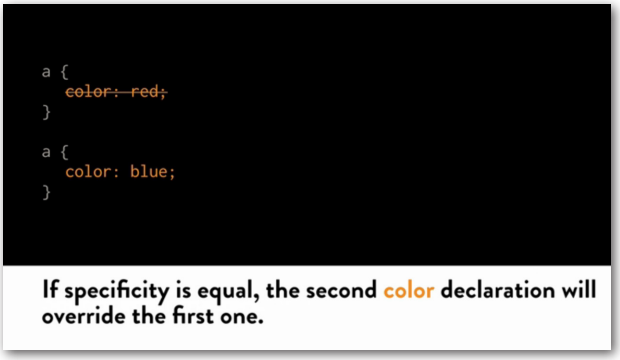
As you can see, the browser has picked out that it wants it to be in a serif font. It wants links to be blue and underlined. It has a certain amount of margin between paragraphs and unordered lists. And so on and so forth.

But we can change all of this by overriding those default styles using our own stylesheets. So the first factor that needs to be considered when we're applying styles is proximity. So what does proximity mean?

Proximity means that we have two selectors that are equal to one another, and we'll talk about quality of specificity in a little bit. But here we have two identical selectors in two separate rule sets.

Generally you wouldn't see these right after each other in a style sheet. But it might be possible that there are other rule sets in between these. Maybe we forgot that we already assigned the style to anchor elements.

So here we have one defining a color of red and one defining a color of blue. When this is the case, when that specificity is equal, the second declaration will override the first one. So the color assigned to all of the anchor elements would be blue.



The same thing is true if you had a single declaration block containing these two declarations. Now again, you wouldn't see this very commonly. But there are specific instances where this can be really handy.

Now, proximity also applies in how your styles are included. (So if you had two linked stylesheets.) Here we have one stylesheet that's including, let's just say, red anchors and one that's including blue anchors. In this case, because the blue anchor comes second, it would override the red anchor CSS.

It doesn't mean it kills the whole CSS file. But even though those two rules are in two separate stylesheets, the blue anchor one is defined second. So the red anchor one is thrown out.

The same logic applies in the context of style elements or embedded stylesheets. So here we have two different style elements. The first one is defining anchors to be red; the second one defines them to be blue. The first rule gets overwritten by the second one because of proximity.

Now, here's where it gets a little bit tricky. Here we have a style element and a link stylesheet. And in this case, the style is coming first. And it's got a color of red for anchors. And then the link stylesheet has blue for anchors. Which one do you think would win?

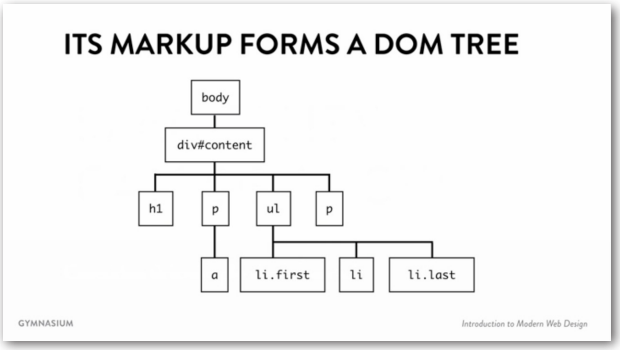
Probably not the one you thought. If you think about it, this make sense. The blue anchor style sheet is an external file. Therefore, it is further away from the element that it's defining styles for.

The embedded stylesheet is closer in proximity. Even though it comes earlier in the document, it is closer to the element because it's only on this page. Therefore, any declarations applied within a rule set (and an embedded stylesheet that could override declaration in a linked stylesheet) will do so.

Now, I mentioned that proximity has an effect when specificity is equal. I kind of dropped that word on you.

So what is specificity? Specificity is another factor in style application. And it can be really, really tricky. This will be the thing that trips you up the most in CSS.

So consider the simple document. Here we have a body with a div element. Inside the div element, we have an h1, a paragraph, an unordered list with list items in it, and another paragraph. All of this markup forms what's called a DOM tree.



So here you can see laid out that the body has a child that is div with an ID of content. The div has four children. An h1, a paragraph, an unordered list, and another paragraph.

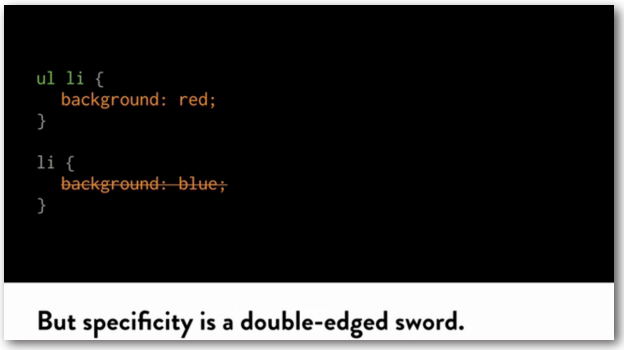
The first paragraph has an anchor element. And the unordered list has three list items. The first one having a class of first assigned to it. And the last one having a class of last assigned to it. So let's focus on those list items for a moment.

If we applied these style rules to the page, li background red, li background blue, what color would the list items turn? Blue. We saw that earlier. Blue would trump red.

Now, if we were to adjust the selector of the first rule set to be any list item that is contained within a ul—and we'll get into these different selector types in a later chapter—but if we were to adjust that selector, now we're selecting list items in the context of an unordered list.

So this has greater specificity. We have the list items, and they are contained within an unordered list.

So what color do you think the list items would turn now? Red. That's because specificity is higher on that first rule set and the order doesn't matter. It's worth noting, however, that specificity is a double-edged sword.



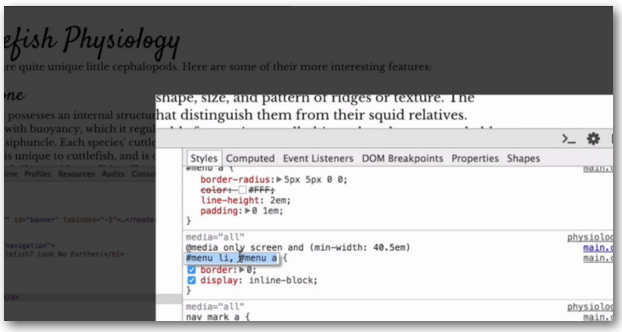
We set a selector of ul li for the color red. But what if all of a sudden we changed that list from being an unordered list to being an ordered list? Now what happens to those list items? They turn blue because the selector no longer matches to turn them red.

Now let's talk a little bit about debugging, proximity, and specificity. The first thing to do to avoid issues with specificity is to keep your specificity as low as possible. And we'll talk about the different selector types and the effect they have on specificity momentarily. If you think you might be having a problem with specificity, the bang important keyword can be used to verify that.

So if we wanted to see the blue apply across the board, we could add this bang important, or !important, to the end of our declaration. And when we do that, that will trump anything having to do with specificity. It says this rule is very, very, very important.

Now, I should note, you do not want to be using important in your production style sheets. Because once you start making one thing important, if you then have an issue with specificity again, you're going to have to make that declaration important. And then if you want to override that, you have to make another declaration important, and you end up in this ever-escalating arms race of importance. And when everything is important, nothing is important. So use this only for debugging purposes. Don't include in your stylesheets when you actually launch your site.

Now, another thing that can help in the world of debugging specificity is DevTools. So here we see the fascinating cuttlefish site. And if I click on Inspect Element, I can go through and I can see all the styles that are being applied to a given element. And then I can see actually which ones are being overwritten.



So here you see I'm selecting based on menu mark a. Then below that, we've got one with less specificity, menu a. And you can see that the color is being overwritten. Below that, you see another menu li, menu a. And then below that, nav mark a.

All of these are being ranked in order of specificity, and then after specificity, being ranked in order of proximity. So this is a really, really helpful diagnostic tool for when you're struggling with why a particular declaration is not being applied to your element.

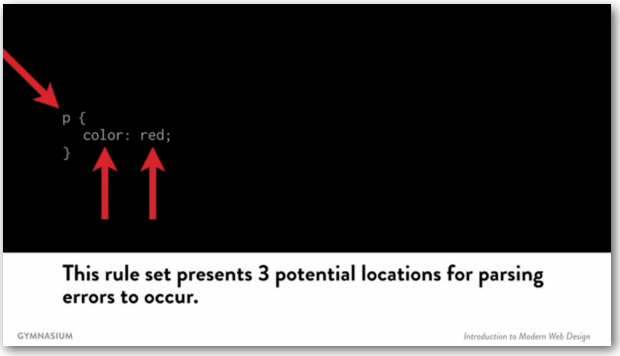
So that wraps it up for this chapter. We've talked about proximity and specificity. And in the next chapter, we're going to tuck into two other core concepts within CSS—fault tolerance and inheritance.

I'll see you in a few minutes.

CORE CONCEPTS IN CSS, PART 2

Welcome back. In this chapter, we're going to go a little bit further and talk about a couple of other core concepts in CSS. The first of which is fault tolerance. Now, fault tolerance has to do with how browsers handle errors. Errors happen.

So here we have our rule set. We've seen this before with our paragraph selector and our color red declaration. And this rule set presents three potential locations for parsing errors, assuming we have the curly braces and our colon and our semicolon in the right spots, which we do. Those three places are the selector, the property, and the value.



If a browser comes along and doesn't understand the selector, it will ignore the entire rule set. If it understands the selector, it proceeds to parse the declaration block. And then within the declaration block, it goes declaration by declaration to determine which ones it understands.

If it understands the property, it progresses on to the value and assigns the value to the property. But if it doesn't understand either the property or the value assigned to it, it would ignore that declaration and move on to the next one. So in that instance, here we have a rule set with two declarations in the declaration block. This would include five places for potential errors—So the selector and then each of the properties and values in the two different declarations.

Now, why does this matter? Well, it matters because knowing how parsing errors work let us use modern CSS and provide fallbacks for older browsers that don't understand those properties or values.

So for example, here we have a background assigned of a light gray color. That's assigned for a browser that doesn't understand RGBA values. Don't worry if you don't understand the colors right now. We'll talk about the colors a little bit later in this lesson. But bear with me here.

```
p {
  background: #ccc;                /* no RGBa */
  background: rgba( 0, 0, 0, .5 ); /* RGBa */
}
```

Knowing how parsing errors work let us use modern CSS and provide fallbacks for older browsers.

So in orange, we have a fallback. And then in green, we have the CSS3 value, which uses an RGBA color. RGBA is for Red, Green, and Blue with an Alpha channel, which sets it to 50% black. Don't worry about, again, understanding how this works, but just know that that's what it is.

So in this instance, any browser that understands RGBA will apply the RGBA value, overriding the proximiity, the light gray value that was applied in the orange declaration. But browsers that don't understand RGBA or the green declaration would ignore that declaration and apply the gray color from the orange declaration. That's how fault tolerance works in the realm of declarations. And that's what allows us to provide fallback colors for older browsers like IE6, for instance.

Moving up a level, we can hide an entire rule set by using a more advanced selector. So in this case, don't worry about understanding the selector, but I'll tell you what it selects. It selects any paragraph that is a descendant of an HTML element with a lang attribute on it. Every browser from IE7 on understands attribute selection, so it would apply these declarations. But older browsers, like IE6 for instance, would see this selector, wouldn't know what to do with it, and would throw away the entire rule set.

```
html[lang] p {
  background: #ccc;                /* no RGBa */
  background: rgba( 0, 0, 0, .5 ); /* RGBa */
}
```

You can also hide groups of declarations. This rule set is thrown away by browsers that don't support attribute selectors.

Moving up another level, we can use @ rules to selectively deliver only certain rule sets to the browser. In this case, in orange, I've got a media query, which we'll talk about media queries in the next lesson. But I have a media query, and browsers that don't understand media queries would ignore this entire @media block and throw away all of the rule sets inside it.

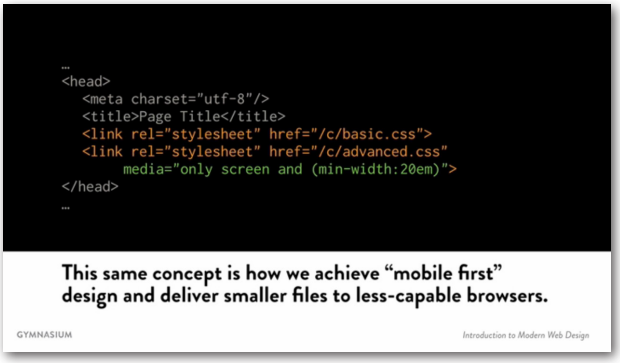
```
@media only screen and (min-width:20em) {

  p {
    background: rgba( 0, 0, 0, .5 );
  }
  ul {
    background: rgba( 255, 255, 255, .5 );
  }

}
```

You can also hide groups of rule sets. These would be ignored by browsers without media query support.

So why would you want to do this? Why would you want to take advantage of fault tolerance? Well, it allows you to create different experiences for different browsers. Here's a rather famous example from Egor Kloos that was in the CSS Zen Garden. And in this case, he was using fault tolerance to deliver the design on the left to advanced browsers, and the design on the right to IE6 and below. And he did this using attribute selectors to weed out older browsers.

So he had his basic layout using more standard selectors, like an ID selector here for intro. And then he used an attribute selector, which you see here in orange, to filter out those older browsers and only apply the advanced layout within a modern browser. Now, it's worth noting that including body and then the square brackets id=css-zen-garden increases the specificity of his selector. Because that is prepended to #intro, which is an ID selector.

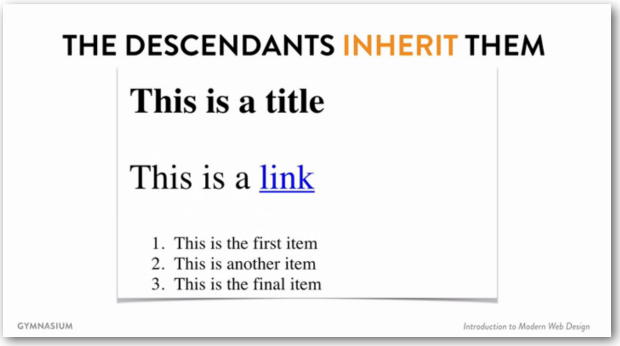
This same concept of fault tolerance is how we achieve mobile first design and deliver smaller files to less capable devices. So here we have two linked style sheets, basic and advanced. Basic is just included. Straight up. No media query assigned to it or anything like that. The second style sheet is included using a media query, and would only apply in the screen medium when there is a min width of 20 ems.



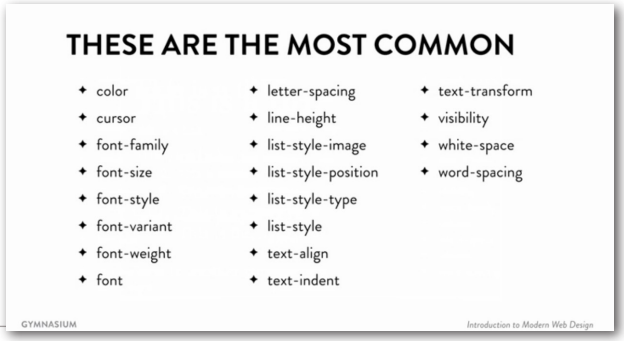
Don't worry about understanding a media query yet. We'll talk about that in the next lesson. But what happens is, an older browser comes and it sees this, and it doesn't understand the keyword only, so it ignores that entire link element. And it only downloads the basic CSS file. A more modern browser that understands the only keyword and media queries would see this and download both stylesheets. So it gets the basic styles and the advanced styles. This is a great way to deliver a selective experience to older devices.

All right. Moving on, the next concept that I want to talk to you about is inheritance. So just like we inherit certain properties from our parents, maybe hair color or eye color, some CSS properties are inherited by an element's descendants. So again, looking at our example page here, we have "This is a title." We've got our link. We've got our ordered list. And then we've got our paragraph.

If we were to assign some font sizes here—and don't worry about understanding ems. We'll talk about ems a little bit later. But here we've got two different font sizes assigned. And what will happen is those will be inherited by the content inside. So you can see with the paragraph being bumped up two ems that the link inside of it also inherited that size increase and became two ems.



Now, if we tweak something further up in the DOM, let's say the div overall would be set to a font size of 0.75, which is like saying "I want this to be 3/4 the size that it normally would," then all of a sudden we go from looking like this to looking like this. Everything is shrunk to 75 percent. Now, there are a lot of properties that inherit from a parent element to its children, but the most common ones that you'll see are things like color,

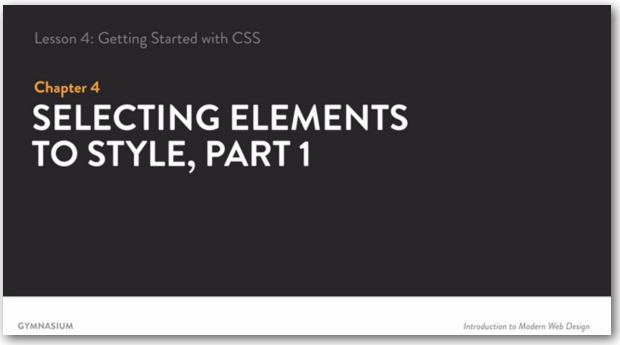


properties that affect a font's display, list related properties, text alignment, visibility, and other things that affect how the text content is laid out.

Now, this was a pretty dense chapter, but we covered two really important core concepts in CSS—fault tolerance and inheritance. In the next chapter, we're going to tuck into selectors and look at different ways to find elements that we want to apply styles to. I'll see you in a few.

SELECTING ELEMENTS TO STYLE, PART 1

Now that we've gotten a lot of the preliminary stuff out of the way, let's talk about how to select elements to style. So the first selector I want to talk about is the universal selector, and the universal selector selects everything. The universal selector is the asterisk seen here in orange. And if we were to say asterisk background red, all elements on the page would be given a red background.



Now, the second type of selector I want to talk about is type selectors. A type selector selects an element based on that element's name. We've seen these a bunch—p selects paragraphs, ul selects unordered lists, and so on. So if we say p's turn red, unordered lists turn green, we see that applied here.

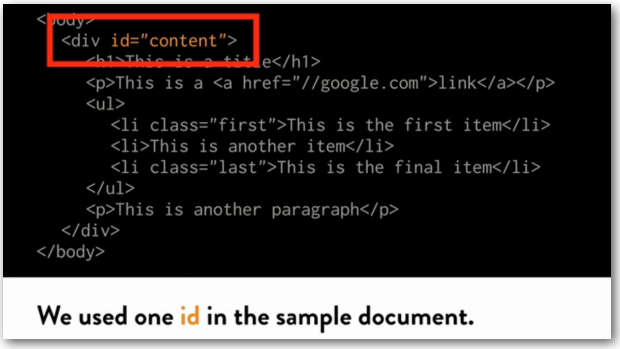
Type selectors—or as they're sometimes referred to, element selectors—are fairly simple and very common. Now, class selectors select based on the class attribute, specifically on the value for the class attribute. If you remember, we discussed back in the HTML chapters that class is for classification, where we classify an element in order to extend it semantics. So here we have our sample document, and we've got a list item with a class of first and another list item with a class of last.

So in our style sheet, we could say all list items should be blue, and then the class selector syntax is dot and then the class value—so .first background green, .last background red. And here we see that applied to the document. All the list items would be blue by default, but then green would be overridden on the first one, because the class selector is

more specific than the element selector or type selector. So if we were to rearrange the styles, it wouldn't matter, because first as a class selector is more specific than li. Now, in the case of the last one, same thing applies. The background of red would be applied because the list item has a class of last.

Now, the class selector will match any class. So if we were to adjust the markup so that each list item had a class of item, and the first one had a class of first and the second one had a class of last, if we were to say .item and assign the color green, that would match all three list items. Class selectors are probably the most or second most common selector type that you will see on the web.

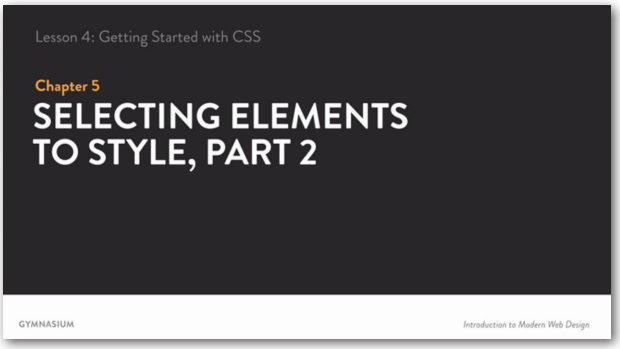
Now, id selectors are the big mac daddy of selectors, and they select based on the id attribute. Now, if you remember back from our HTML chapters, id is for identification. The id attribute, or a specific id value, can only be used once per page. So here we have a div with an id of content. And here we have the selector to select that div, hash, content, or ocothorpe content, and that would select that div specifically and apply a background of red.



So in this chapter, we covered the four basic kinds of selectors, but there are a ton more selectors to go over. So in the next chapter, we're actually going to tuck into attribute selectors and pseudo-element selectors.

SELECTING ELEMENTS TO STYLE, PART 2

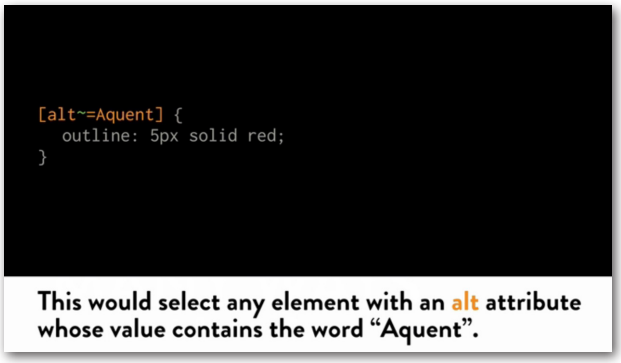
All right, welcome back. So in this chapter, we're going to continue on the selector front and look at other ways to select elements. So we've talked a little bit about attribute selectors already. I gave a couple examples of that. They're pretty self-explanatory. And any attribute is fair game.



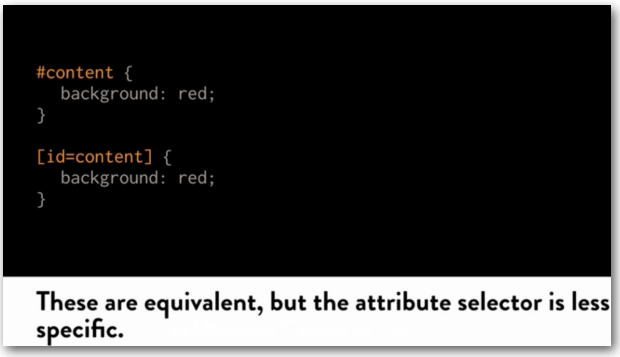
You can test for the existence of an attribute, which you simply do like this. Here I have square brackets [alt]. So this would select any element that has an alt attribute. You can also test for a specific value. If I wanted to make sure that the alt value was exactly Aquent, I could use this syntax, square brackets, alt, Aquent, square brackets, [Aquent]. Now, the match would have to be identical. It would have to have a capital A and then lower case for the rest of the letters.

You can also test for portions of a value. And you can do this in a number of ways. So if I was to say alt tilde equals Aquent [alt~=Aquent], this would select any element with an alt attribute whose value contains the word "Aquent." If I were to say pipe equals Aquent [alt|=Aquent], it would select any alt attribute whose value is Aquent, or Aquent hyphen something.

If I were to say caret equals Aquent [alt^=Aquent], this would select any element with an alt attribute whose value starts with Aquent. And in this case, Aquent could be followed by other letters, spaces, or anything. If I were to say dollar sign equals Aquent [alt$=Aquent], this would select any element with an alt attribute whose value ends with Aquent [alt*=Aquent]. And then finally, if I were to say asterisk equals Aquent, this would select any element with an alt attribute whose value contains Aquent, even as part of another word.



So since we know we have attribute selection, we also can select id's and classes in other ways. And when we do this, we create selectors that are equivalent, but not equal. Now, what do I mean by that? Here we have #content, which is an id selector for the value of content. And then in the second example, we have square brackets id=content close square brackets. Each of these are equivalent in terms of selecting an element. They would select the exact same element. But the second selector is less specific.



Now, once I've gone through the majority of the selectors, I'm actually going to help you understand how specificity calculation is done. For right now, just know that these are equivalent, but that the attribute selector is less specific than the ID selector.

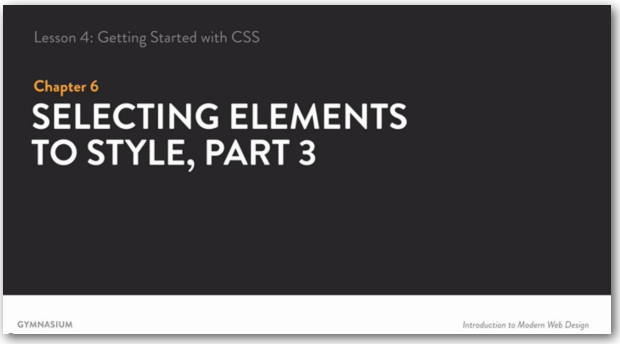
Now, the next kind of selectors I want to touch on are pseudo-element selectors. And these are used for parts of the document. And they're actually dynamic as well. So you could select the first letter, or the first line of a paragraph, or you could select what somebody has selected while they're actually selecting text on a page.

So let's look at an example. We could take the first letter of a paragraph and enlarge it. So then we end up with these big initial caps. Now, you could take this a step further, using some of the other properties that I'll talk about later in this lesson and in the next lesson, to create things like a drop cap.

So in this chapter, we have covered attribute selectors and pseudo-element selectors. In the next chapter, we're going to go through a bunch of pseudo-class selectors. I'll see you in a few minutes.

SELECTING ELEMENTS TO STYLE, PART 3

OK, welcome back for Chapter 6. This is the third of five chapters talking about how to select elements. And in this chapter, we're going to be focusing specifically on pseudo-classes. Now, pseudo-classes target via a context or certain properties that are assigned to a given element. So what do I mean by that?



Well, I mean things like hover and focus and active. These apply to elements based on their interaction state. So hovering over a link, focusing on an element, or activating it by clicking on it. So let's look in the browser so that you can see what this works like.

```
:hover {
  background: red;
}

:focus {
  background: blue;
}

:active {
  background: green;
}
```

These selectors apply to elements based on their interaction state.

Here, when I hover over the links, they turn gray. And when I activate the link, they turn green. If I inspect these elements, I can actually adjust the different states directly in the dev tools in Chrome so that I can make sure the right styles are applying in the right context. Now, in most cases, you want to apply the exact same styles on focus and hover. So that covers people whether they're using a mouse or whether they're navigating with a keyboard.

The next pseudo-class I want to talk about is target. Now, this selects an element that's been targeted using an anchor link. So what does that mean? Well, take a look at this example. We're going to start off, and we're going to come down to the cuttlebone. If I click on that, you see that it goes to the physiology page and then anchors to the cuttlebone content.

So if I click on that, it goes over to that page, and you see the #cuttlebone in the URL. Now, that #cuttlebone matches the id of this section here. So that is being targeted. Now, I don't have any styles applied to it, but I could apply styles to a section that's targeted and change its background to yellow.

So if I were to change the target, change it to skin, now skin has a background of yellow. But cuttlebone no longer does. So that can be pretty useful for highlighting what somebody has anchored to within a larger page. If I could type correctly and spell camouflage, then you could see me changing the target to camouflage. Pretty neat stuff.

So other examples of pseudo-classes that we have available to us are things like first child, last child, and only child. And these select based on source order with respect to a group of siblings. So if I were to just assign this basic style sheet to the sample document we've seen before, we end up with

```
:first-child {
  background: red;
}

:last-child {
  background: blue;
}

:only-child {
  background: green;
}
```

These select based on source order with respect to their siblings.

a rainbow of colors. How did we end up with this rainbow of colors? Well, let's look at the markup.

```
<body>
  <div id="content">
    <h1>This is a title</h1>
    <p>This is a <a href="//google.com">link</a></p>
    <ul>
      <li class="first">This is the first item</li>
      <li>This is another item</li>
      <li class="last">This is the final item</li>
    </ul>
    <p>This is another paragraph</p>
  </div>
</body>
```

Here's a breakdown reflected in the HTML.

First we have the body element. The body element is the last child within the HTML element. Within the body element is a div which is an only child of that body. Within the div, the h1 is a first child. Within the ul, the li is a first child. And the last li is a last child. And then the paragraph that's in blue is the last child of that div. So that's why those colors apply to each of those individual elements.

So what if we were removed the only child? What would happen then? Well, in that case, anything that was previously an only child would be both a first child and a last child. Therefore, anything that was an only child previously, would now turn blue because it's a last child. Because when the selectors are equal, proximity wins. Crazy stuff, I know.

Now, in addition to being able to pick specific children based on their position in the document, whether they're only children, first children, last children, we can choose them based on whether they are the odd children or the even children. So you can use this for zebra striping a page or a table. We can even select a specific entry. So we only want to affect the first child, or we only want the 15th child.

Or we can get really complicated and select based on a formula. In this case 3n+3. Now, that may look really, really weird. It's some strange math, right? Well, let's look at exactly what that does. What that says is that for every group of 3, that's the 3n bit, pick the third one. So it's always the third one. So it'll be the third one, the sixth one, the ninth one, and so on.

```
:nth-child(7n+1) { color: red;    }
:nth-child(7n+2) { color: orange; }
:nth-child(7n+3) { color: yellow; }
:nth-child(7n+4) { color: green;  }
:nth-child(7n+5) { color: blue;   }
:nth-child(7n+6) { color: indigo; }
:nth-child(7n+7) { color: violet; }
```

For every group of seven siblings, assign a different color to each one. Taste the rainbow!

GYMNASIUM Introduction to Modern Web Design

So to illustrate that a little bit more clearly for you, you could create a rainbow by doing nth-child 7n, because there are seven colors in the rainbow, and then 1, 2, 3, 4, 5, 6, 7 for each of the individual colors. So for every group of seven siblings, assign a different color to each one. And each one gets the rainbow. I'll let you play with that one in a browser.

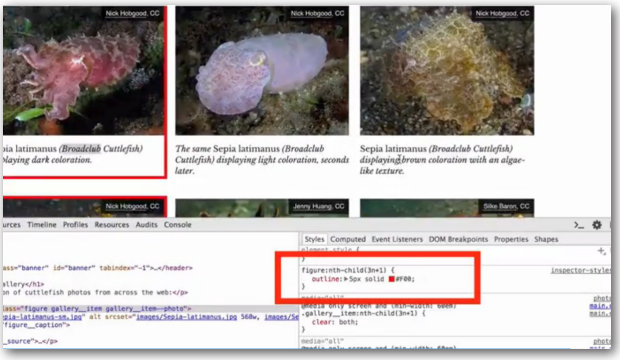
Now, we can also use minus. Can you guess what this selects? You got it. For every group of 10 siblings, select the ninth one. So 9, 19, 29, et cetera. So let's tuck into nth-child selection really quickly looking at the gallery on my cuttlefish site.

So here we have a bunch of different figures. If we inspect one of them, we can see that we have a gallery with multiple figures in it. This being the first one. OK. And they all have a class of figure. So if I add a new rule and I say, figure nth-child odd, and then give it an outline of 5 pixels solid, and red, you can see that the odd figures are now outlined in red.

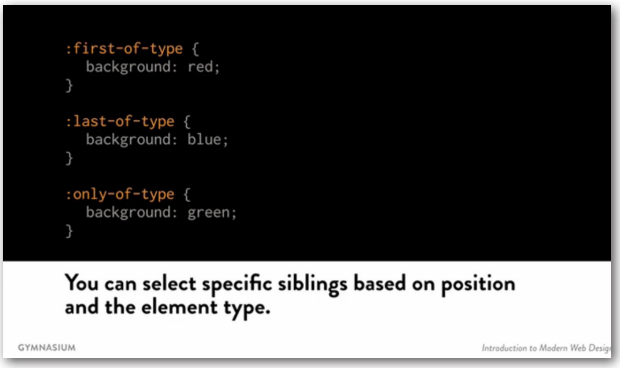
If I change it to even, it switches to the even members. If I set it to 1, it will select the first one. If I set it to 2, it selects the second one, and so on. If I decided to use a formula and do 2n, you see that that is identical to even.

Now, if I do 2n+1, you'll see for every group of two, it selects the first one. If I set it to 2n+2, for every group of two, it selects the second one. If I do 2n+3, for every group of two, it selects the next one.

So you'll notice that it's actually the third, and then the fifth, because the second group of two starts on the third one. If I do 3n+3, then every third one gets it. If I do 3n+1, the first of every group of 3 gets it. Pretty cool stuff.



Now, you can also select specific siblings based on the position and the element type. So first of type, last of type, only of type, even nth of type. But chances are you won't use those very often. One you might find useful is empty, which can select based on its contents, or its lack of contents. So this would select any element who has no children or text content. And then there's a special pseudo-class that's used to invert selection. So if you wanted to select all images that do not have an alt attribute, you could do something like this.

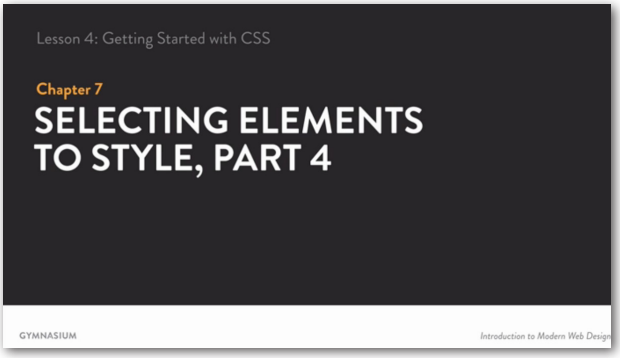


So that's it for pseudo-class selectors. In the next chapter, we're going to talk about specificity calculation, descendant selectors, and child selectors. I'll see you in a few.

SELECTING ELEMENTS TO STYLE, PART 4

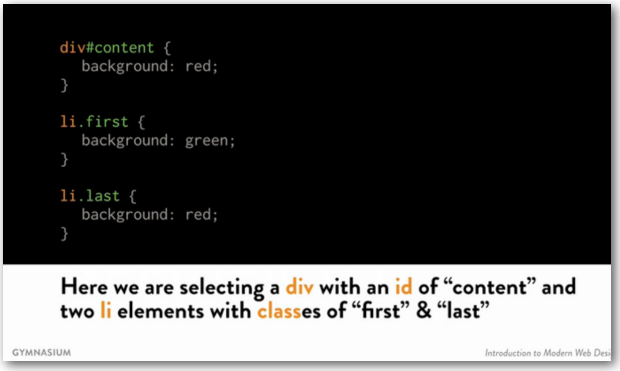
All right, welcome back. We're in the penultimate selection chapter right now, Selecting Elements to Style, Part 4.

So I want to talk now about specificity. So selectors can be combined to achieve greater specificity. We've seen this in a couple examples. And trust me, you need to learn this.



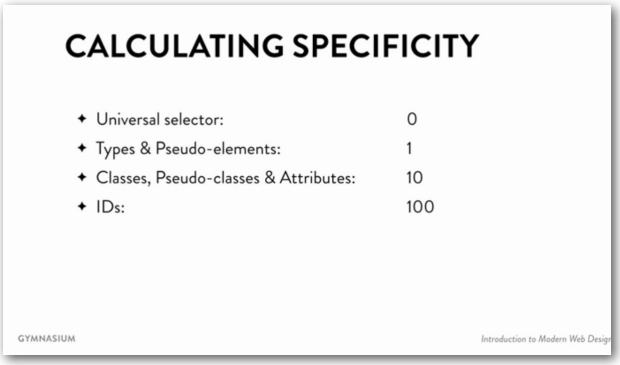
So here we have a couple of combined selectors. The first one combines a type selector of div with an id selector of #content. So it selects any div with an id equal to content.

And then we have two list items, li.first and li.last, which select list items with a class value of first or last, respectively. It's just important to know that the specificity of a selector is cumulative.



So we can calculate specificity by saying the universal selector has a specificity of zero. Types and pseudo-elements have a specificity of one. Classes, pseudo-classes, and attributes have a specificity of 10. And ids have a specificity of 100.

Now, I'm throwing out these numbers because they make it relatively easy to calculate the specificity of one selector versus another. In reality, those are actually infinity. So types and pseudo-elements are infinitely more specific than the universal selector. Classes, pseudo-classes, and attributes are infinitely more specific than types and pseudo-elements. And ids are infinitely more specific than classes, pseudo-classes, and attributes.



Of course, doing math with infinity is a little bit of a challenge. So this is a good rule of thumb. The only thing that you need to keep in mind is if you were to chain 10 classes together, that would not be equal in specificity to an id. Even though 10 plus 10, plus 10, plus 10, and so on is 100.

So which of these selectors is the most specific? Without a doubt, it's the one that includes the id selector. So let's do a couple of examples calculating specificity.

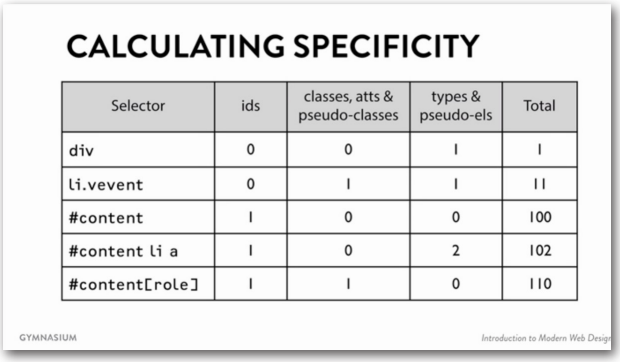
We'll start with the div. It has zero ids, zero classes, attributes, and pseudo-class selectors. One type or pseudo-element because it has a type selector for div. So its specificity would be 1.

Now, li.vevent has zero ids, one class, and one type. Therefore, it has a total of 11. Because as you remember, classes have a specificity of 10 in this calculation. So 10 plus 1 is 11.

#content has one id, zero classes, attributes, or pseudo-classes, and zero types or pseudo-elements. Therefore, it has a specificity of 100. So it's way more specific than either of the previous two selectors.

In the third example, #content li a, we have one id selector, no classes, and two types, for a value of 102.

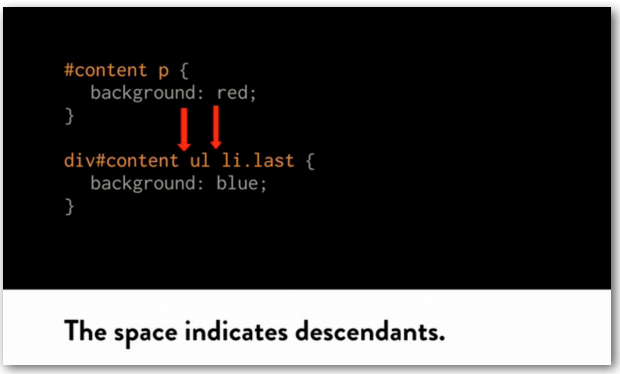
And then finally, #content role has one id, one attribute selector, and no type selectors, for total specificity of 110.



So let's talk about what these combinations mean. The first type of combination, apart from actually selecting the same exact element, is descendant selectors. This is selecting an element based on its context within an ancestor.

So we've seen this where one selector is separated from another with a space. So #content p, select any p element that is a descendant of an element with an id of content.

The second example, div#content ul li.last, this selects any list item with a class of last, that is a descendant of an unordered list, that is a descendant of a div with an id of content. It's a very specific selector. And here we see those style rules applied.



Now, child selectors select based on a parent context as opposed to just a general ancestor context. So if I wish to say, #content greater than p, that says that it wants to select any paragraph that is a child of something with an id of content. And it's going to turn it red.

Now, in the second example, it's going to select any list item with a class of last, that is a child of a div, with an id of content. So when we go over to our page, you see that the paragraphs are red but nothing is blue. So why didn't this apply?

Well, the reason that it didn't apply is that the div with an id of content has a ul within it that contains the list item. The list item is not a child of the div. The list item is a child of the unordered list. And it is a descendant of the div with an id of content.

So we could change the selector to say we want to select any list item with a class of last, this is a child of an unordered list, that is a descendant of content. And then our styles apply.

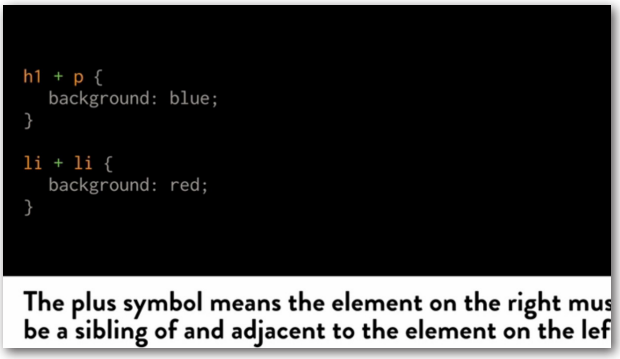
So in this chapter, we talked a little bit about specificity calculation, descendant selectors, and child selectors. In the next and final selection-based chapter, we're going to look at adjacent sibling selectors, general sibling selectors, and specificity calculation with commentators like these. We'll see you in a few.

SELECTING ELEMENTS TO STYLE, PART 5

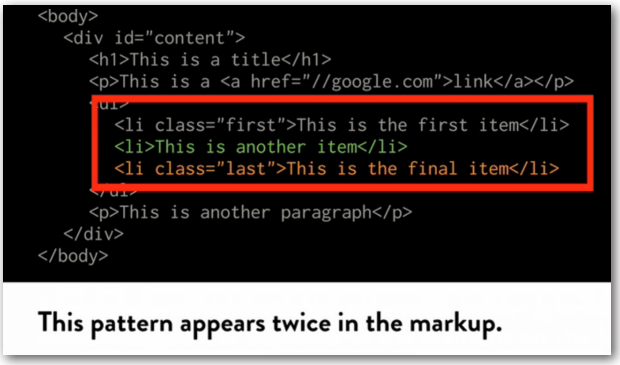
All right, welcome back. This is the final chapter discussing selectors. And we're going to kick it off by discussing the adjacent sibling selector. So the adjacent sibling selector selects an element based on the previous sibling. And we do that using the plus symbol. So here we have h1 + p. So this would select any paragraph that comes immediately after an h1, where they are both inside of the same parent.

In the second example, we have any list item that is adjacent to another list item. So in other words, a list item that comes after another list item. So here that is applied to the page, but why only the second and third list items? Well, the bit in green in the adjacent sibling selector is the context for selecting the thing in orange.

So in the markup, we have a paragraph directly after an h1. In the case of the list item, the list item is in the context of another list item. The context would be the first list item for the second one. And the second one for the third one. But there is no list item to provide context for that first list item. And therefore, only two of them are turned red.



Now, you'll notice that the last paragraph is also not colored blue. If we wanted to do that, we would need to use general sibling selection. General sibling selectors are based on a previous sibling. So "a" instead of "the." Now, what that looks like is it uses the tilde symbol (~). So h1 ~ p selects any paragraph that is a general sibling of the h1. So it follows it in the source order somewhere, but there could be any number of elements in between them.



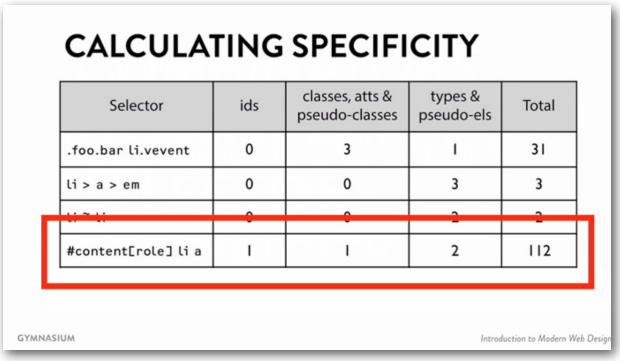
In the case of the second one, we're selecting any list item that is a general sibling of another list item. So now, both paragraphs are blue. But still, those list items are only red on the second and third ones. Again, because of the context. So here the context is in green and the thing selected is in orange. And if we move over to the markup, you see the h1 is there, and then it has two paragraphs that follow it as general siblings. So those are highlighted in orange.

But again, in the case of the list item, the first list item is simply providing context for the second one and the third one. So it's worth noting that none of the symbols used to create these different types of selectors affect the specificity of the selector. So no plus, no tilde, no greater than symbol. Those have no effect on the specificity of the selector. It only comes down to the number of id's, the number of classes, attributes, or pseudo-class selectors, or the types and pseudo-elements selectors.

So here we have a bunch of selectors to calculate the specificity for again. So let's see how you do. I'm going to give you a second and then I'll give you the answer. So in the case of .foo.bar space li.vevent, so this selects any list item with a class of vevent that is a descendant of an element with a class of foo and a class of bar. This would have a specificity of 31.

In the second example, li greater than a greater than em. In this case, we're selecting any emphasis element that is a child of an anchor that is a child of a list item. In this case, we've got three type selectors. So a specificity of 3.

In the third example, li tilde li, we're selecting any list item that is a general sibling of another list item. A specificity of 2, because we have two type selectors. #content square brackets role space li space a. This selects any anchor element that is a descendant of a list item that is a descendant of an element with an id of content and a role attribute assigned to it. This has a specificity of 112. It's got one id selector, one attribute selector, and two type selectors.



And that's it for this chapter. We've covered adjacent sibling selectors, general sibling selectors, and specificity calculation with these various combinators. In the next chapter, we're going to look at font display, typographic controls, and content alignment and flow.

CONTROLLING TYPOGRAPHIC STYLES

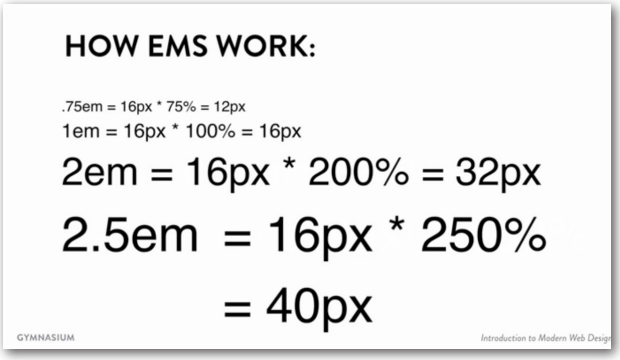
All right, welcome back. Now we get to tuck into some of the actual design decisions that we can make in CSS now that we've gotten all that selector business out of the way. So I want to start by talking a little bit about typographic styles. So CSS offers a variety of ways to control the display of text on a page.

The first way we can control font display is using font-family. Now, font-family accepts what is called a font stack, which is a comma separated list of acceptable font names in order of preference, from your first choice to your last choice. So in this example, I'm choosing Helvetica Neue, Arial, and sans-serif. Now, it moves from left to right, so Helvetica Neue is my first choice for font rendering. But if that's not available on the user's device, it falls back to Arial. And then if Arial is not available, it falls back to whatever the system's sans-serif font is.

Now, this font stack actually gives you three different examples of how fonts need to be defined within our font-family stack. The first example shows a proper name that has a space in it—Helvetica Neue, so that has to be quoted. Arial is only a single word, so it needs to be capitalized, but it doesn't need to be quoted. And then sans-serif is simply a keyword referring to a system font, therefore it can be in lowercase. And there are other keywords, such as serif as well.

Now, once we've chosen our font-family, we can also define the size that we want. Font sizes can be set in a variety of different values, from pixels to ems to inches to picas. Some are flexible and some are fixed. An em-based unit, which you saw there, is always sized according to its context. So let's look at how ems work here, because ems are something that you see people use frequently on the web.

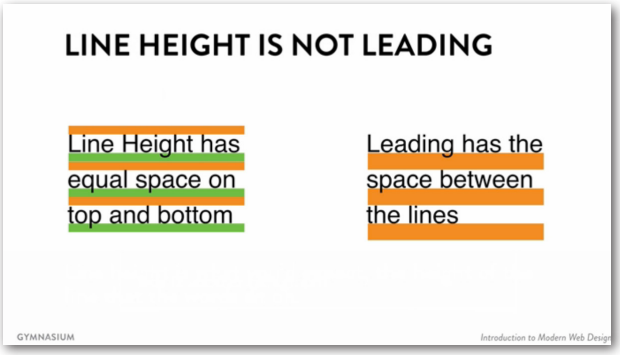
So 1em is equal to whatever the context is times 100 percent. So you can imagine, the em simply being replaced by 100 percentage. Now, 0.75 ems is equivalent to 75 percent of. So context time 75 percent. 2ems is equal to the context times 200 percent, so doubled. 2.5 is the context times 250 percent.



So if we were to establish what the context is, let's say 16 pixels, which is the browser default, then you would see that 1em is 16 pixels. 0.75ems is 12 pixels, because that's 75% of 16 pixels. 2ems would be 16 pixels times 200%, or 32 pixels. And then finally, 2.5ems would be 16 pixels times 250%, or 40 pixels.

So you may remember seeing this in an earlier chapter—paragraph set to 2ems, and unordered lists set to 0.75ems. Here was the original, and once those styles were applied, the list shrunk and the text grew bigger in the paragraph. So the descendants inherit the size of the parent and then create their own context for the elements within them. So again, if we tweaked the font size further up in the DOM, say changing the div to a font size of 0.75ems, we would go from this to everything being shrunk by 0.75. So that's how ems work.

Now, in addition to setting the font-size, we can adjust the line height of the font. So the line height is what you'd expect the height of the line that the words sit on. But it's important to note that line height is not leading. Leading comes from the world of movable type, where you used to put pieces of lead in between the letters that were being used to set up a printed document. That space was added below the line, whereas with line height, the space is equally divided both above and below the line.

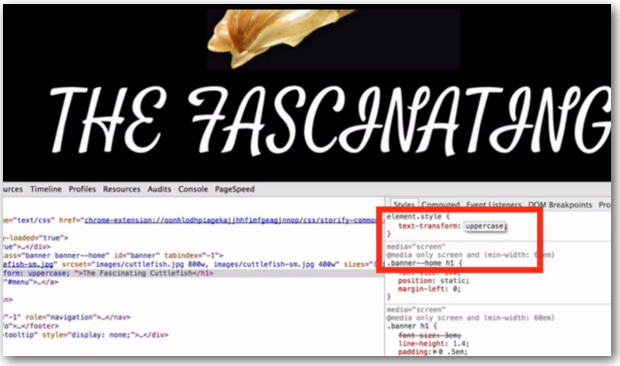


So if you set your line height to 1.5, then the overall height of the line is 1 and 1/2 times the font size. Generally, unitless line heights are the preferred way to go. And they simply act as a multiplier. So 1.5 on a 2em font would effectively be 3em. We can adjust the heaviness of our glyphs using font-weight. Font-weight accepts bold to bold the text, normal to convert it from bold to normal weight, and some fonts even support numeric values for weights for things like light, heavy, black, and so on.

We also have font-style, which allows us to control italics and obliques, or to reset it from italic back to normal again. And finally, we have font-variant, which allows us to set things like small caps, if the font supports it. Now, that's a lot of different CSS properties to define how a given element looks in terms of its typography. You can roll all of those up into a single property simply called font.

It begins with the style, which is optional, the weight, which is optional, and the variant, which is optional. And then you need to set a font-size, and then optionally, a line height after a slash, and then the font stack that you want. So if I remove all of the comments here for brevity, and now all of a sudden you see italic bold small-caps 2em with a line height of 1 and 1/2, and I want the font stack of Helvetica Neue, Arial, sans-serif. There are a bunch of other typographic features that you can control in CSS which are not part of that shorthand. And those include text-transform, which allows you to adjust the case to uppercase, and so on.

So let's take a look at that in the browser. Here I've got the title for The Fascinating Cuttlefish. If I go in there and I want to go ahead and set text-transform, and I can set it to uppercase. And then you see it goes in all caps. I can also set it to all lowercase, or I could set it to capitalize, which will capitalize each word.



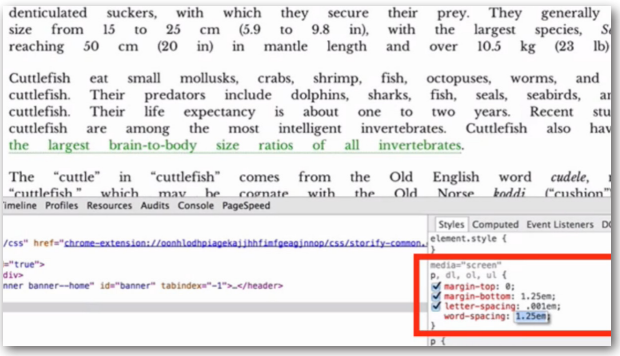
It's worth noting that there isn't a title case, because true title case would have to lowercase words like "the," "and," and so on in the English language. And browsers would need to take into account all of those words in every language in order to properly do title case. So we're left simply with capitalize. There you see I've changed the text to CSS Text-Transform Is Not Very Smart. Is Not would normally be lowercased, but capitalize just makes it capitalized.

We have the ability to control whether there is any sort of text decoration applied to the content. So typically, you see this in the context of anchors having an underline. Here we have The Cuttlebone which is underlined. We inspect that element. We can see that it's got a text decoration none and a border bottom set to 1 pixel dotted. So it's not actually a true underline.

If I turn off the border bottom and turn off text decoration none, you can see that there is an underline there. If I turn back on text decoration, I can show you a couple of different text decoration options. So there's underline, of course, which is the default on links. We could also change the text decoration to overline, which puts a line above the text. Or there is even the option of line-through, which draws a line through the element.

We can also use features like letter-spacing and word-spacing to spread things out or potentially tighten them up. Let's jump over to the browser and take a look at these. So if I take paragraphs and I set the letter spacing to 1 pixel, you see things start to spread out as I increase it to the point where they become illegible. Let's decrease that back down to 0. I can shrink it up and really tighten up those words until you can't make anything out. You can even set them to a relative width using ems.

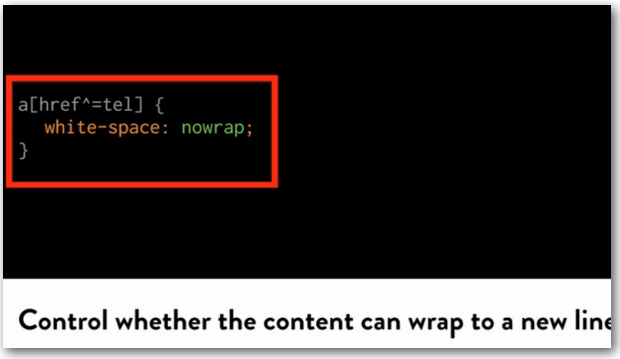
Letter spacing is most useful when you're doing things like all caps for a title and you want to give them a little bit of breathing room so it's easier for somebody to read them. And you can see that I can even get super, super granular doing 0.007, or what have you, of an em, and it will make certain tweaks as the rounding takes place. Word spacing works in very much the same way, but instead of spacing the letters, it's spacing words.



These are not properties that you'll use all that often, but they're good to have in your back pocket just in case you need them. We can also tweak the letter forms for a crisper appearance using font-smoothing antialiased. Unfortunately, this is one property that requires vendor prefixes to be used in certain browsers.

And last but not least, you can control how the content aligns and flows using things like white-space. So white-space: nowrap says that this element is not allowed to wrap to multiple lines. So this is a really useful property to apply in the context of a telephone number, because a telephone number typically has hyphens in it, and a browser will try and wrap that if it needs to.

So saying a square brackets href starts with tel, so the tel pseudo protocol, white-space: nowrap, would make sure that phone numbers don't wrap to multiple lines. The same thing could be used for mailto for email addresses. You can also control how the text is aligning. You can make it justify and fill the container equally on both sides. If you do justify, you could say text-indent 1em so that you get a little tuck in at the first line. But generally, justified text looks pretty bad on the web. But you can align left, you can align right, and so on.

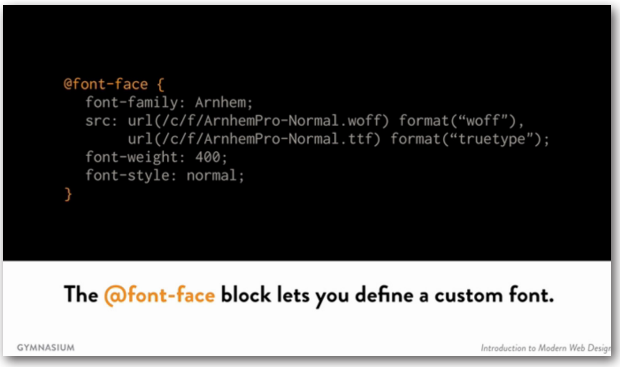


So in this chapter, we've discussed font display, typographic controls, and content alignment and flow. In the next chapter, we're going to talk a lot about custom fonts and how to load those. I'll see you in a few.

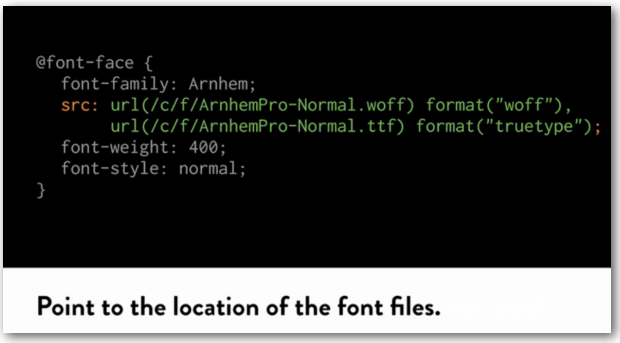
CUSTOM FONTS

All right, welcome back. Hopefully, you've familiarized yourself a little bit more with some of the typographic controls that we talked about in the last chapter. And now we're going to tuck into custom fonts. So when I talk about custom fonts—or you'll sometimes hear the word referred to as web fonts—it basically means you can define custom typefaces. And you do this using an at rule called @font-face. And this block allows you to define what your custom font is.

So it has multiple properties as part of it, the first of which is the font name. So here we say font family is equal to Arnhem. This will be the name of the font family that we will use elsewhere in our style sheets to refer to that font. So here you see font-family is Arnhem, and then Georgia as a fallback, and then Serif as a fallback. So moving back over to the @ font-face rule, we then supply the sources for where that font can be downloaded. In this case, it can be downloaded from the /c/f directory, C for CSS, F for fonts.

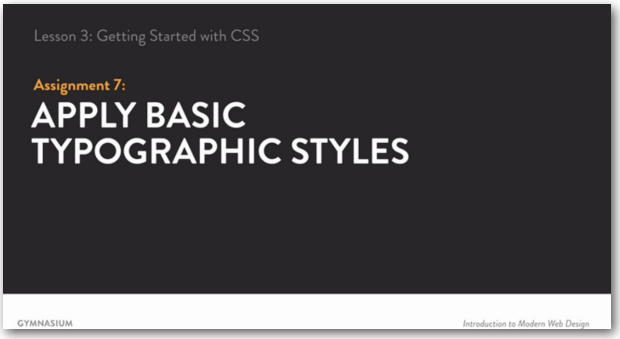


And then there's a woff-formatted font, W-O-F-F, and then there's a truetype font, TTF font. So if the browser understands woff, which tends to be a little bit smaller wen-based format for fonts, it will download that one. If it doesn't know how to handle a woff font, it will fall back to the truetype and download that one instead.

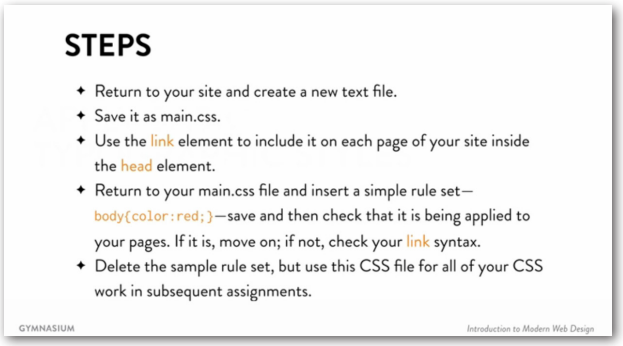


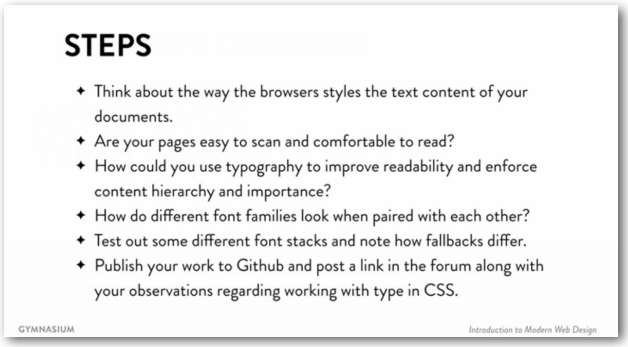
Once we've defined where the font files can be found, we can define what weight this is. So in this case, we're defining that the weight of this particular font file is 400, and it is in the normal style. Whenever the style sheet refers to

Arnhem in normal style and a 400 weight, it will apply this font. You would create corresponding at font face rules for bold, italic, or other weights with the respective individual font files for those fonts, and you would keep the font family the same.



So that was a brief overview of custom fonts. Now I want to go ahead and have you do another assignment. So in this assignment, Assignment 7, I want you to apply some basic typographic styles to your site. So return to your site and create a new text file and save it as main.css. Use a link element to include it on each page of your site inside of the head, like we discussed earlier in this lesson. Return to the main.css file and insert a simple rule set, something like body{color:red;}. Save it and make sure those styles are being applied to your page. If it is, great. If not, check your link syntax and make sure you've gotten that right before you move on.





Assuming all is well, go ahead and delete that sample diagnostic rule set there and use that CSS file for any CSS work you want to do in this and subsequent assignments. So when you're thinking about typographic styles for your page, think about the way the browser styles the text content of your documents by default. Are they easy to read, easy to scan? How could you use typography to improve the readability—maybe enforcing some content hierarchy and bringing some attention to certain important elements on the page?

Try a couple of different font families. There are some great resources online for web-safe fonts and great services out there, like Typekit, that can let you play around with custom web fonts. Test out some different font stacks and note how each fallback differs. Publish your work to GitHub, and post a link in the forum along with your observations regarding working with type in CSS.

That does it for this chapter. In the next chapter, we're going to look at color—specifically color keywords—hex colors, RBG and RGBA, and HSL and HSLA. I'll see you in a few.

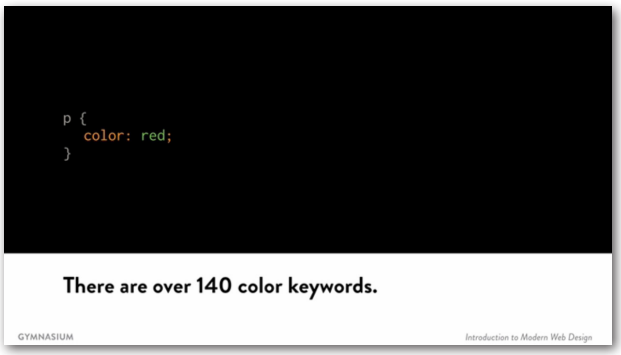
COLOR IN CSS

All right. In this chapter, Chapter 11, we're going to talk about color in CSS.

So color can actually take many forms in style sheets. The most common one we've seen is using color keywords. That's the most simplest and straightforward.

And there are over 140 different color keywords. You probably will never remember them all. But they can be useful when you're just trying to comp something up pretty quickly and you want red, green, what have you.

For most projects however, those color keywords are only going to get you so far. You might be able to use them for white and black. But chances are you're going to want to customize the colors a little bit more specifically for your project.



So to do that, you'll use one of the other color types. The first and most common color type that you'll run into is hexadecimal format. That looks a little something like this. Where you see a hash followed by Fs, and zeros, and numbers, and so on and so forth. Basically A through F and zero through nine.

So hexadecimal is almost always six digits. Or in some cases three, which I'll show you in a minute. Each of those is broken down into pairs. So in this case ff, 00, and 00. And each of those assigned to a different color channel, which I'll show in a moment.

If you're familiar with Photoshop, you often see hexadecimal value in the Color Picker down here in the bottom middle, minus the hash.

Now, you'll sometimes see a hexadecimal color value that is only three letters and numbers long. In this case, it's using hexadecimal shorthand, which is when you've got a matching pair, such as ff, 00, 00, that can be shortened to simply f00, which is red if you're wondering.

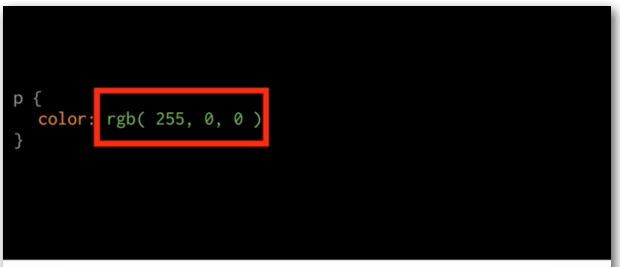
So as I mentioned, the first pair corresponds to the R channel or red channel. The second pair corresponds to the G or green channel. And the third pair corresponds to the B or blue channel.

And speaking of RGB, there's actually RGB color values. We saw these when we saw RGBA a little bit ago.

RGB colors let you to define the value for each channel in a number ranging from 0 to 255. This corresponds to the RGB piece of the Color Picker in Photoshop.

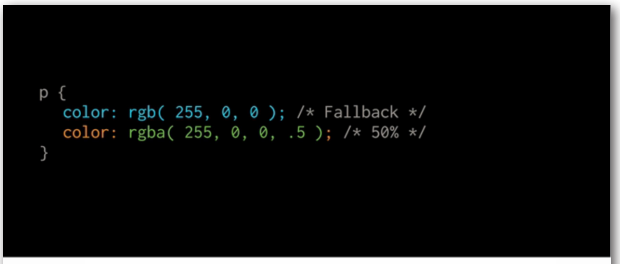
And corresponding to that, we have RGBA, which has a red channel, a green channel, and a blue channel, plus the alpha channel, which is the last bit there. So in this case we have red, but it's 50 percent. And the value of the alpha channel ranges from 0 to 1 in fractions of 1.

So you'll remember back to our fault tolerance discussion. That this would actually allow us to have a fallback for older browsers that don't understand RGBA. They would get the RGB value, with the RGBA value being applied only in browsers and understand it.



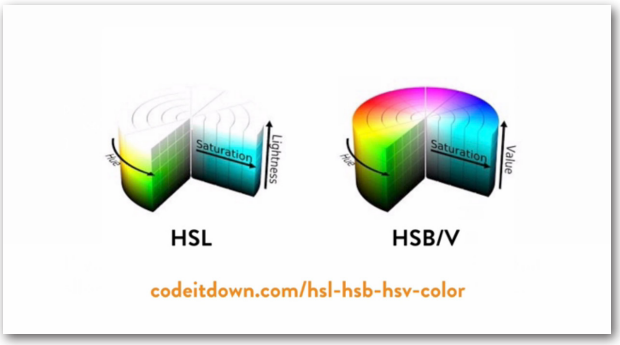
RGB colors let you define each channel in numbers from 0 to 255.

If you're a designer who tends to think more in hues, HSL is another color option that lets you define the color based on hue, saturation, and lightness. Now, that may sound familiar to you if you're user of Photoshop because there's this HSB thing, which stands for hue, saturation, and brightness. But that's not quite the same.



Older browsers should be given a fallback using CSS's fault tolerance mechanism.

With hue, saturation, and lightness, what we're dealing with is the movement from black to white along the L, the lightness channel. With HSB, we're adjusting the brightness. So it's a little bit different. If you go all the way up in the lightness, you're going to hit white. If you go all the way in the brightness, you end up with the most true representation of that particular color. So slightly different.



But HSL can be really useful if you just want to tweak the saturation in order to create a family of colors. Or you want to pick up a bunch of different colors on the wheel that all have the same amount of saturation. You can do that by adjusting only one of the bits of the HSL value.

Now, similar to RGB, HSL has an HSLA equivalent. So you can assign an alpha channel as well. And in the same way as RGBA, you want to provide that fallback for older browsers that don't understand HSLA.

So in this chapter we covered color keywords, hex colors, RGB and RGBA, and HSL and HSLA. In the next chapter, we'll look into using these different colors.

WORKING WITH COLOR

Now that you've seen the different color options, let's look at how we can use color in our websites. First of all, the color property we've seen a bunch already, that governs the foreground color of an element. Foreground refers to both the text color and the border color of that element. And foreground is one of those inherited values.

Background color is another color that you're familiar with that applies to the background of an element. We've seen this a lot in earlier examples. Background color, however, is not inherited. Let's jump over to the browser and take a look at this.

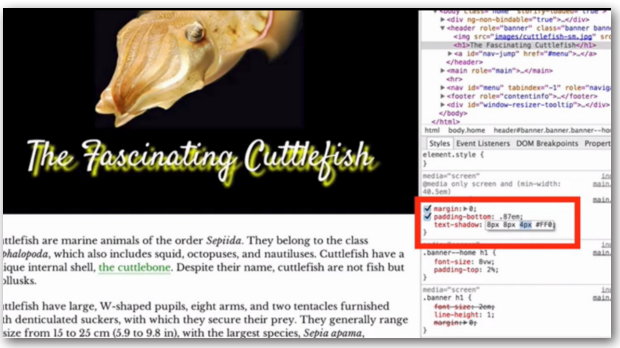
So here we have the body. We see fonts moving [? into your ?] list does not apply. But if we assign a background of red to the document, you see that everything turns red. Right? If I tree down into the div, however, you see that there is no background color inherited by that div. If, however, I change the background to being a foreground color of red, now all of a sudden, its color is red in the diagnostic tools there. But if I inspect the heading level, you'll see that it has a color of red as well. Because that is inherited. Foreground color is inherited.

So color comes up a lot in CSS. Here we have background, which is a shorthand for background color, background image. We'll look at all of those properties later. But background, you could simply say, in this case, is a light gray; border, 1 pixel solid. That's going to inherit the foreground color. We have a box shadow here that's got a color. We have the foreground color. And we have a text shadow, also with a color.

So as I mentioned, the foreground color is applying both to the border and to the text color as well. But we could also specifically say that we want to use a particular color for the border if we don't want to inherit that foreground color.

Now, text shadow can be used to give you a nice drop-shadowy effect on text. So here you see text shadow has four properties—the horizontal offset; the vertical offset; how much of a blur you want, which is optional; and then the color that you want that text shadow to be. Here is that without the comments on it. So it's not necessarily quite as long and scary looking.

So let's look at this in the browser as well. Let's look at the title. So here we have the banner h1. I'm going to set a text shadow on it. Set it to 1 pixel, 1 pixel, 1 pixel. And I don't know—let's set it to yellow. Should stand out a little bit. You can't really see that, so maybe I'll give it a little bit of distance. Ah, there we go. Now you're starting to see it.



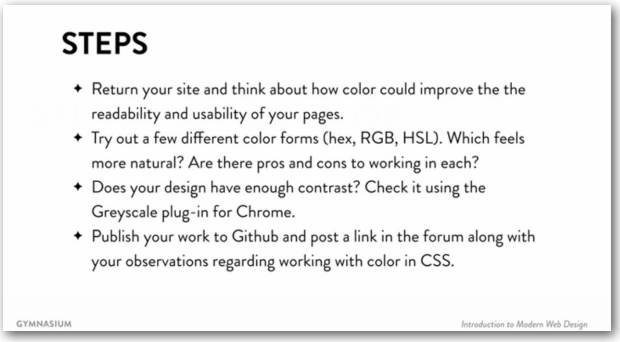
So here it's being shifted by 8 pixels. And now by 9, no, 8 pixels. And then we can increase the blur radius. And you see it's getting blurrier and it's sitting behind the words. So that's pretty awesome. If I set it down to 0 blur radius, then that makes it so that it's sharp.

Now, I can combine multiple text shadows by putting a comma in the value. So here I'm saying that I want the yellow, and then I want a negative 5 pixels, negative 5 pixels, 5 pixels, and let's make this one purple. So now I've got one shadow that's coming down and to the right, and one that's coming up and to the left. And I can tweak the distance. Kind of give myself this weird, I don't know, psychedelic effect. It's a little trippy.

Or I can make myself a rainbow drop shadow. If you remember, we had the rainbow colors with an nth child earlier. I can apply those in the text shadow as well and make something really silly.

So in this chapter, we've covered using colors in both the foreground, the background, and in text shadows. And now I want you to go ahead and add some color to your pages. So return to your site and think about how color could improve the readability and usability of your pages. Try out a couple of the different forms using hex, using RGB, HSL. What feels more natural to you? Do you find pros and cons working in each of them?

Now look at your site. Does it have enough contrast? There is a grayscale plugin for Chrome that you can download to actually see if the colors you've chosen give you enough contrast when you view it in grayscale. This can be helpful for mimicking what it looks like if you have a form of color blindness.

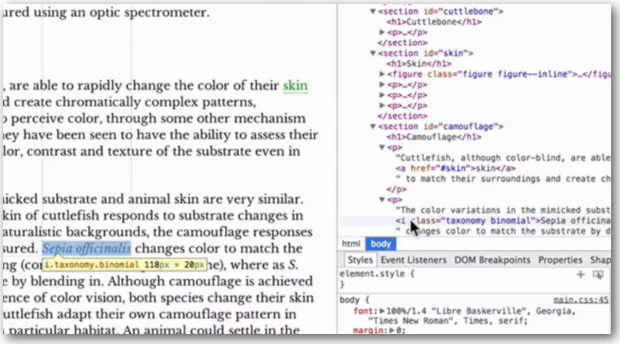


Publish your work on GitHub, and post a link in the forum, along with your observations regarding working with color in CSS. In the next chapter, we're going to look at the box model. You're going to see lots and lots of boxes. And we're going to look at element dimensions and padding. I'll see you in a few minutes.

CONTROLLING AN ELEMENT'S DIMENSIONS, PART 1

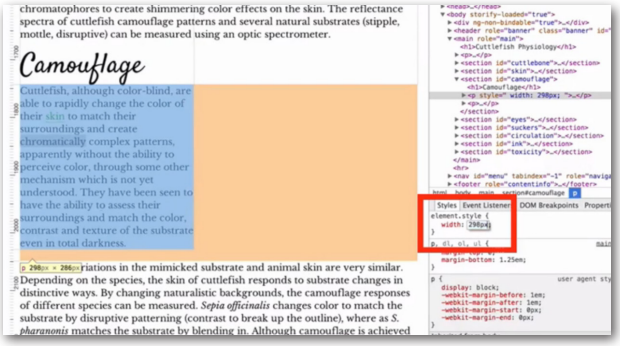
Welcome back. Now we're starting to get into the nitty gritty of designing pages with CSS. We're going to talk about controlling an element's dimensions. This is a little bit of an in-depth topic, so we're going to do it in two parts. The first and most important thing to recognize is that every element on the page creates a box. Block elements stack on top of one another, and inline elements sit next to one another.

So let's jump over to the browser really quickly so you can see this in action. Here you see the header and the main elements stacked on top of one another. If I move into the main, you see the h1 stacks on top of the paragraphs, the sections stack on top of one another, and the elements within the sections stack on top of one another. These are all block-level elements.



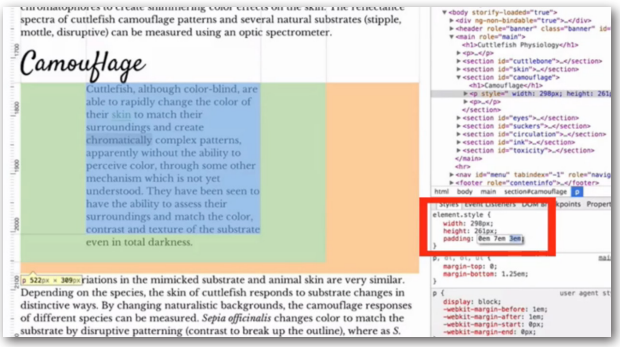
If I go down here to this paragraph in camouflage, and I expand that, drill down into that one, and choose the paragraph that has the skin link in there, you'll see that the skin link, the anchor there, is an inline element. Same thing with this italic Sepia officinalis. These are inline elements.

Now, we can control the size of block elements. So again, jumping into the browser, let's take this first paragraph under camouflage. I'm going to inspect it. And I want to go ahead and set a size on it. So let's start by setting the width. I'm going to set a width of 200 pixels.



If I mouse over it, you see the orange? The orange is actually the margin. The margin is automatically expanding to ensure that the next element after it, the next block-level element, appears on the next line. OK? So we can adjust the width value.

We can also set a height value. And when we set the height, if we set the height smaller than the actual text content of the element, you'll see that the text content actually overflows out of the element, which looks pretty fugly. We can control that using the Overflow property. By setting overflow hidden, it makes any content that flows outside of the element disappear.

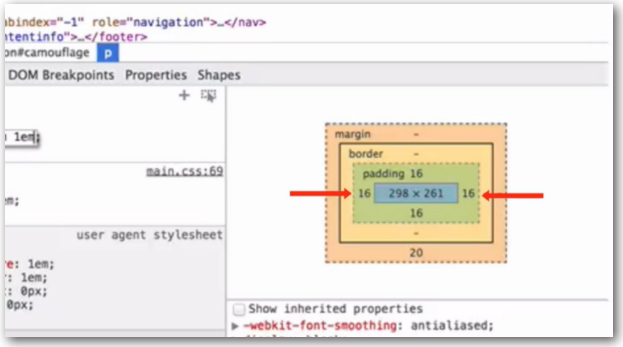


Let's turn that off. Now, let's play a little bit with padding. Padding is going to show up in green here. If I set a padding of 60 ems, you see that there is six times the font size around

it. And it mirrors all the way around. If I set a padding of 1 em, 2 em, or 1 em, 7 em, you see that it actually mirrors in two directions. The 0 mirrors from the top to the bottom, and the 7 mirrors from the right to the left.

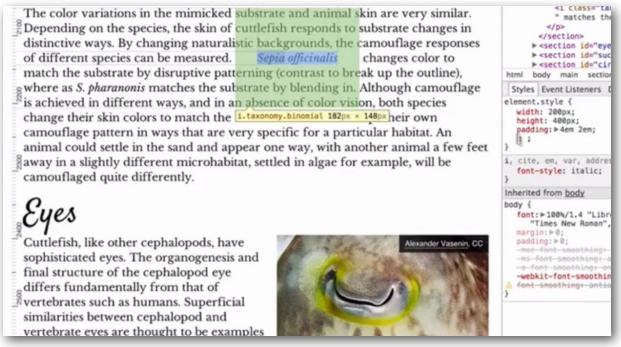
If I add a third value, now all of a sudden, I've got the top set, the right being mirrored to the left at 13, and the bottom having 3. And if I add a fourth, then it's actually assigning all four sides. So we've got the top, right, bottom, and left. So it moves clockwise from the top around the entire element.

So you can set the padding on each side individually, or you can use mirroring to assign values to multiple sides at once. So a single padding value will apply to all sides. Two padding values will mirror the first value to the top and bottom, and the second value to the right and left. Three values will have the top value assigned. The second value will be assigned to both the right and the left. And the third value will be assigned to the bottom.



And then finally, four values would assign each of them, top, right, bottom, left. A good way to remember that is "Trouble"—Top, Right, Bottom, Left—TRBL. That can be quite helpful if you're trying to remember which side you're assigning a padding or margin value to.

So the content box is the default box model. There are two different box models in CSS currently. The border-box is an alternative. Now, let's compare the two. So if we have an element with a width of 200 pixels, padding of 10 pixels, and a border of 10 pixels, the border-box box-sizing method would ensure that that element is never larger than 200 pixels.



The content-box box model, however, which is the default, is additive and would make the content width 200 pixels, and then add 10 pixels to either side for the padding, and add another 10 pixels to either side for the border. So in other words, it keeps the content size fixed at 200, and then adds padding and borders to it. Whereas the border-box box sizing model takes the full width of 200, and then subtracts from that the padding and the border in order to come up with the amount of space that it's left for the actual content.

This is a little bit tricky, so I highly recommend opening up a browser and playing with an element by setting different widths and paddings and borders, and adjusting the box sizing between the default content-box and the border-box model. In many cases, if you're coming from a print design background, where you're used to having a maximum amount of space to work in, the box-sizing border box method will be the most familiar to you.

Now so far, all of this has had to do with sizing for a block-level element. Inline elements are a little bit different in that sizing on them only affects the horizontal layout. So let's look at a quick example.

Here we have another paragraph. And inside of that, we have our i for Sepia officinalis. If we went ahead and set a width on that element, you'll see that it actually doesn't grow. If I set a height, similarly, the element does not get any bigger. If I were to set some padding on that element, you see that it only affects along the horizontal. Doesn't matter how much I grow the vertical, it will never affect an inline element.

And the same thing is true of margins. If I were to set a margin on there, it would only apply in one direction and not the other. Now, I can use the Display property to actually set the inline element to display as though it's own block. And you'll notice when I did that, all of a sudden, it popped down to its own line. And all of a sudden, the dimensioning of the width and the height are applied.

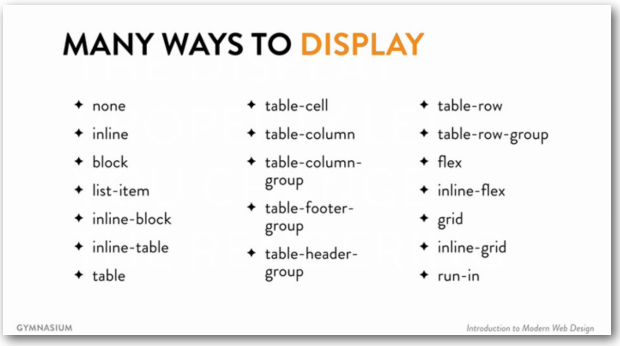
There's another display property called Display Inline Block, which is kind of like the best of both worlds. It lets you give the dimensioning sizes, but keeps the element inline. But by default, the i element is inline. So we'll leave it like that.

So in this chapter, we talked about boxes and the box model. We talked about element dimensioning. And we talked about padding. In the next chapter, we're going to talk about different display types, like we just looked at, with display block and display inline block. And we're going to look at box shadows. See you in a minute.

CONTROLLING AN ELEMENT'S DIMENSIONS, PART 2

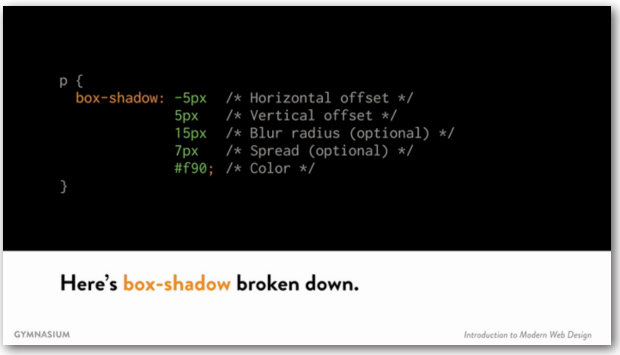
All right, welcome back. We have reached the final chapter in this lesson, Getting Started with CSS. And we're going to talk a little bit more about controlling an element's dimensions. So I showed display as a property, and being able to change something from being an inline element to being a block-level element.

Now, there are many ways that you can adjust the display of an element. You can set it to display inline, to be a block, to display like a list item, to display like a table, a table cell. You could display it flex, which is what's called flexbox, and so on and so forth. You can even do display none, which makes an element disappear from the page. I do not recommend doing that, because once you do that, the content of that element becomes completely unavailable to anyone who's using assistive technology, like a screen reader.

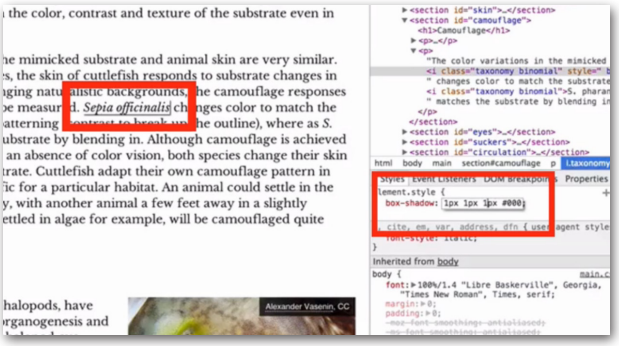


So remember this? We looked at the element that was an inline element by default. And when we set it to block, it jumped to its own line. We also could change it to inline block, and then we saw that it actually had the dimensions and it had the padding, but it was sitting in line as though it was an inline element, even though it had that dimensioning. So that's what display does.

Another way we can manipulate the box is actually to apply shadows to it. So this works in much the same way as the text shadow. We have a horizontal offset and a vertical offset. We have a blur radius. But then we have this other optional property called “spread,” which has to do with how much of the shadow spreads into the blur radius. And then finally, the color of the box shadow. And when we remove all of the comments in there, this is what we end up with.



So let’s take a look at that one in the browser as well. If I jump over to this Sepia officinalis, go ahead and apply a shadow, I can see it starting to peek in there as I fill it in. And there we go. I decided to leave out the spread. And if I increase the horizontal and then increase the vertical, you can see the box being added behind it.



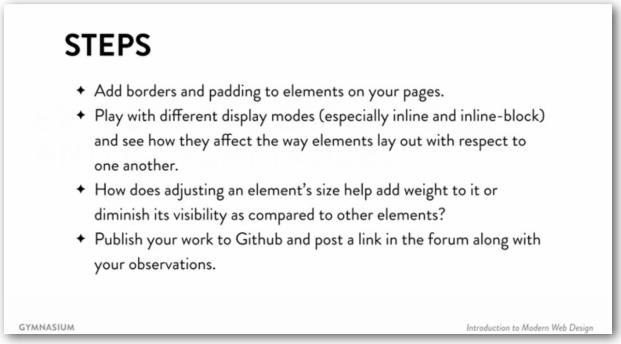
So it’s as though the box is actually white, even though it has no background color. So it’s not a true shadow. It’s a rendering of a shadow around the element. It’s important distinction to make.

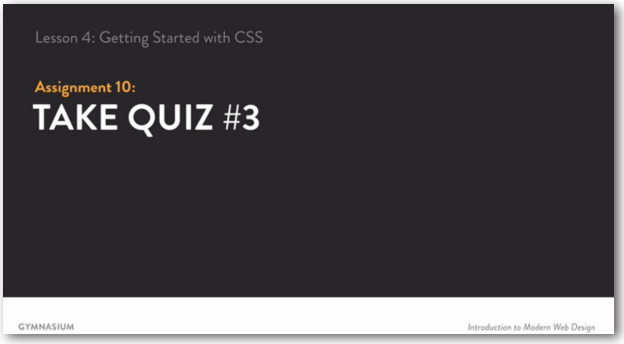
I can go ahead and add that same shadow to the paragraph. Blur it out a bit. And you can see if I let it peek up a bit that it actually gives you a good sense of the size of the box, and where the box for this paragraph element ends. Could also add multiple shadows and start to make this look really gaudy.

And that kind of gives you a sense of what box shadow can do. So box shadow also has another keyword that you can use in it, which is called inset. And with inset, you can actually create shadows within the element.

Now Web Standards Sherpa used this to great effect. If you inspect the main article on the page, what you’ll see is that it has both an external box shadow and an internal box shadow. Now, it’s worth noting they also use border radius to curve the corners, and that the box shadow actually hugs to that.

So in this chapter, we discussed the different display types and box shadows. So finally, in Assignment 9, I want you to explore the box model and the different display modes. I want you to add borders and padding to each of the elements on your pages and play with the different display modes, especially inline and inline block. How do manipulating these different display modes affect the way the elements layout with respect to one another? How does adjusting the element size help add weight to it or diminish its visibility compared to other elements?





Publish your work on GitHub and post a link to the forum along with your observations. And finally, since we're at the end of this lesson, I want you to go ahead and take Quiz #3. I'll see you in the next lesson where we tuck into quite a bit more CSS.