# HTML Basics

## Intro to HTML

**HTML and CSS**

HTML stands for **HyperText Markup Language**. **Hypertext** means "text with links in it." Any time you click on a word that brings you to a new webpage, you've clicked on hypertext!

A **markup language** is a programming language used to make text do more than just sit on a page: it can turn text into images, links, tables, lists, and much more. HTML is the markup language we'll be learning.

What makes webpages pretty? That's CSS—**Cascading Style Sheets**. Think of it like skin and makeup that covers the bones of HTML. We'll learn HTML first, then worry about CSS in later courses.

The first thing we should do is set up the skeleton of the page.

a. Always put <!DOCTYPE html> on the first line. This tells the browser what language it's reading (in this case, HTML).  
b. Always put <html> on the next line. This starts the HTML document.  
c. Always put </html> on the last line. This ends the HTML document.

**Basic terminology**

To learn more HTML, we should learn how to talk about HTML. Already you have seen we use <>s a lot.

1. Things inside <>s are called **tags**.
2. Tags nearly always come in pairs: an opening tag and a closing tag.
3. Example of opening tag: <html>
4. Example of closing tag: </html>

You can think of tags as being like parentheses: whenever you open one, you should close it. Tags also **nest**, so you should close them in the right order: the most recently opened tag should be the first one closed, like in the example below.

<first tag><second tag>Some text!</second tag></first tag>

The last exercise taught us how to set up our HTML file. Everything we do now will go between <html> and</html>.

Practice makes perfect! One more time:

**Make the head**

Everything in our HTML file will go between the opening <html> and closing </html> tags.

There are always two parts to an HTML file: the head and the body. Let's start with the head.

The head contains information about your HTML file, like its title. The title is what we see in the browser's title bar or page tab. For example the title of this page is "HTML Basics | Codecademy".

**Paragraphs in the body**

Great job! To review, an HTML file has both a head and a body. The head is where you put information about your HTML file, like its title.

The body is where you put your content, such as text, images, and links. The content in the body is what will be visible on the actual page.

The body goes inside the <html> tags, right after the <head> tags, like this:

<html>

<head>

<title>My Webpage</title>

</head>

<body>

</body>

</html>

**Paragraphs and headings**

We're definitely making good progress! We've learned when and why we use HTML. We've also learned how to:

a. Set up an HTML file with tags  
b. Title the webpage (in the <head>)  
c. Create paragraphs (in the <body>with <p> tags)

The next step is to give our paragraphs headings using **heading tags**. Let's start with the <h1> tag. The content between this tag will be the biggest!

**More about headings!**

HTML actually lets us have more than one heading size. There are six heading sizes, where <h1> is the boss and <h6> is puny!

* <h1> - The CEO
* <h2> - VP
* <h3> - Director
* <h4> - Middle management
* <h5> - Lowly assistant
* <h6> - Gets coffee for everyone

Below we'll ask you to add headings of various sizes. Feel free to write whatever you like for the headings!

**Mid-lesson breather**

You've done an awesome job! Here's a quick summary of things we've learned:

1. HTML is used to give websites structure.
2. We open HTML files using a browser, and the browser **renders**(shows us) the file.
3. HTML files have a <head> and a<body> (just like you!)
4. In the head, we have the <title>tags, and we use these to specify the webpage's name.
5. How to make headings and paragraphs.

**You're going places!**

What if you wanted to send the user to another part of your website, or another website altogether? You use hyperlinks, or links for short!

<a href="https://www.codecademy.com">My Favorite Site!</a>

1. First, there's an opening <a> tag and that tag has an attribute calledhref. The href value tells your link where you want it to go, in this case[https://www.codecademy.com](https://www.codecademy.com/).
2. Then you have a description of your link between your opening <a>and your closing </a> tags. This is what you will be able to click on.
3. Finally, you have your closing </a>tag.

**Adding images**

You can add images to your websites to make them look fancy.

We use an image tag, like so: <img>. This tag is a bit different from the others. Instead of putting the content between the tags, you tell the tag where to get the picture using src. It's also different because there is no ending tag. It has / in the tag to close it: <img src="url" />.

Check out the tag to the right—it adds a picture of a rubber duck to the page! (You can see it by clicking on the Preview button.)

See the web address (or **URL**) aftersrc=? It's"https://s3.amazonaws.com/codecademy-blog/assets/f3a16fb6.jpg". That tells the <img> tag where to get the picture from!

Every image on the web has its own image URL. Simply right-click on an image and choose "Copy image URL." Paste that URL in quotes after src= to insert with your <img> tag.

**Click that image**

Good work! Now you know how to add links and images to your website. Why not make that image a link? For example:

<a href="https://www.codecademy.com/">

<img src="https://s3.amazonaws.com/codecademy-blog/assets/f3a16fb6.jpg"/>

</a>

1. First we open our <a> tag and point the href to<https://www.codecademy.com/> again.
2. But this time, instead of using text inside the <a> tag, we use an <img>tag.
3. Finally, we have our closing </a>tag.

Now when you click on the yellow duck, you will be taken to[https://www.codecademy.com](https://www.codecademy.com/)!

Placing one HTML tag inside of another is called **nesting**.

## HTML: Using Lists

**Ordered lists**

Good! Now let's learn how to make **ordered lists**. An ordered list is simply a list that is numbered, like the one below.

1. On [line 8](javascript:void(0)), we begin the ordered list with the opening tag <ol>.
2. On [lines 9](javascript:void(0)) – 11, we **wrap** (i.e.surround) each individual item with<li> and </li> tags.
3. Because each listed item is only on one line, we put the entire **element** on one line.
4. On [line 13](javascript:void(0)), we finish the ordered list with the closing tag </ol>.

How cool is this? We can now add ordered lists to headings and paragraphs as things we can use in our HTML body.

**Unordered lists**

We just learned how to make ordered lists, but what if the order doesn't matter, what if we just want bullet points?

<h2>Taco Ingredients</h2>

<ul>

<li>Cheese</li>

<li>Sour Cream</li>

</ul>

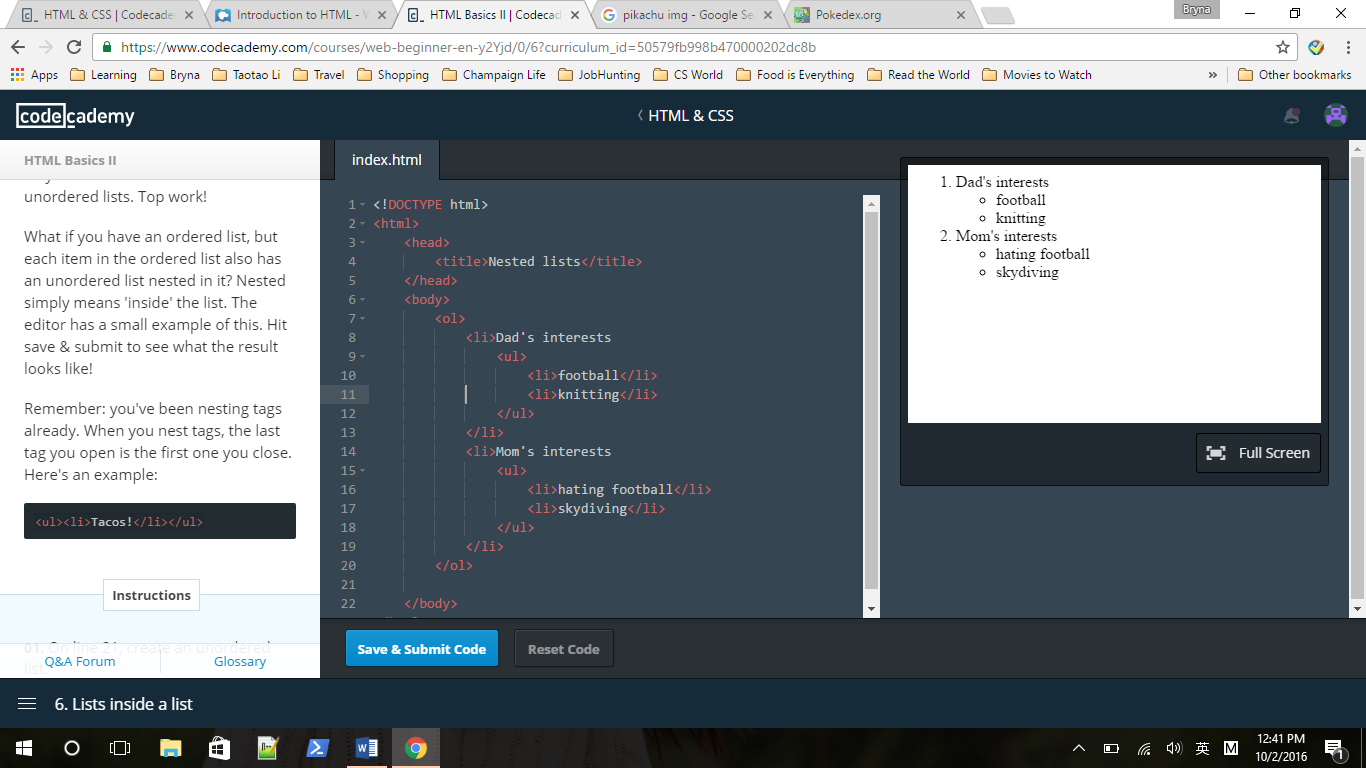
Does this look familiar?

1. First, we open our list with an unordered list <ul> tag
2. For each item we wish to add to the list, we use a list item tag <li>with text in between
3. We then tell the browser we are done with our list by calling our closing</ul> tag

**Lists inside a list**

So you've made ordered lists and unordered lists. Top work!

What if you have an ordered list, but each item in the ordered list also has an unordered list nested in it? Nested simply means 'inside' the list. The editor has a small example of this. Hit save & submit to see what the result looks like!



Remember: you've been nesting tags already. When you nest tags, the last tag you open is the first one you close. Here's an example:

<ul><li>Tacos!</li></ul>

**Making comments**

We have covered a lot about lists. Let's change gears and do some styling. If you recall, HTML is the skeleton of the webpage, and CSS lets you give the skeleton some skin and makeup.

But it is possible to do some **inline CSS**. This simply means we can do some styling in our HTML file without worrying about a separate CSS file! We'll learn this first because it will make learning CSS a lot easier later.

Before we dive into fonts, it's important to learn about making comments. You can include little notes in your HTML code that the browser won't display. But it will be in the code to help you remember why you did certain things.

**Font size**

Recall that <p> and </p> are opening and closing **tags**.

We can give tags more instructions by including **attributes** in the opening tag. An attribute is simply a characteristic or some description for the content in the element. You saw this with src in <img> and href in<a>.

Let's change the size of the text. How? We use the **style** attribute. We make it equal to font-size(Note font-size not font size), followed by a colon, the size you want, and end it with px(short for "pixels"). For example:

<p style="font-size: 12px">

**Font color**

What is awesome about the **style**attribute is that we use it a lot! And we can use it with many different tags, not just <p>. Let's now change the colors of our text in a heading.

To change the color of text, simply add the style attribute in the opening tag, then make the style equal to "color:blue" (or whatever color you like). For example:

<h2 style="color:red">

What if you want to change the colorand the size of the text? Simple! Just add a semi-colon between each bit. For example:

<h2 style="color: green; font-size:12px">

A full list of available colors can be found [here.](http://www.w3.org/TR/css3-color/#svg-color)

**Note**: do not type something like:

<h2 <h2 style="color: green; font-size:12px">

If you incorrectly nest your HTML tags like that, your code **will not pass**.

**Font family**

We've covered font colors and font sizes. But we want more power! We want to decide what font type to use. We can do this using font-family, like this:

<h1 style="font-family: Arial">Title</h1>

1. First we wrote <h1>Big title</h1>
2. Then inside the opening <h1> tag, we added a style attribute, and set it equal to "font-family: Arial".

This styles the <h1> tag with Arial font.

We can do the same for other tags. So we could have a li:

<li style="font-family: Arial">Hello!</li>

[Here's a list](http://www.w3.org/TR/CSS21/fonts.html#generic-font-families) of available fonts.

**Recap**

Awesome job! You've now got control of your webpage, including the color, size and type of your font. To recap, we use the style attribute in the opening tag, like so:

a. font-size: 14px  
b. color: orange  
c. font-family: Bodoni

<p style = "font-size:14px; color: orange; font-family: Bodoni">

It's important to know that you can use the style attribute for paragraphs, headings, and even links!

**Background color**

The previous section covered a number of nice tricks to control how the text looks. Now we want to learn about how to change the color of the webpage's background.

We can use the style attribute again, and set it equal to "background-color: red" (or whatever color you want).

For example, here's how to change the background color of a <p> tag to red:

<p style="background-color: red;">Hello!</p>

**Aligning the text**

Often it is nice to be able to move the text around. To do so, we again use the **style** attribute. And then we use "text-align:left" (or right, or center) to determine the location of the text.

<h1 style="text-align:center">

**Strong words!**

We can change the appearance of words. What if we want to make them**bold**?

Surprise! We don't have to use the style attribute. Here are the steps:

1. Identify the word or words you want to **bold**.
2. Surround those words with opening tag <strong> and closing tag</strong>.
3. Celebrate how awesome you are at HTML!

**Emphasize words!**

Aside from bolding words, we often want to *italicize* words for **em**phasis. (Hint, hint.)

Like bolding, we do not need to use the style attribute. Instead:

1. Identify the word or words you want to italicize.
2. Surround the word or words with the opening tag <em> and closing tag</em>.
3. Be humble and grateful for your newfound powers.

## HTML: Tables, Divs and Spans

**What are tables?**

Tables are very useful. We use them to store tabular data so it is easy to read! When you want to present information neatly in a table with rows and columns—you guessed it—the<table> tag is what you need.

There are many tags associated with tables, but it all starts with the <table>tag, so let's add that first.

**Rows of information**

A table is just a bunch of information arranged in rows and columns.

We use the <tr> tag to create a **table row**. We'll learn how to create columns shortly, and everything will start to come together. (You don't really create columns in <table>s: instead, you tell each row how many cells to have, and that determines your number of columns).

**A single column**

Look at the HTML now. Can you tell that there are still three rows? We've added a little more whitespace to make it easier to deal with **table columns** and **table data**.

We've added a single <td> ("table data") cell to the first row, essentially creating a single column. If you view the Result tab now, you'll see that we've drawn a border around it. it's not that impressive, but don't worry: we're about to add more table data cells.

We're starting to add a lot of HTML elements now. Make sure to indent your tags as you nest them so you don't get confused!

**Adding a second column**

It may not have seemed like much, but you just created a single-column table in the last exercise. Nice work!

Now take a look at what we have on our page. Notice in the first table row we now have *two* table data cells.

Adding a second table data cell has the effect of adding a second table column, although if you go to the Result view, it may look funny because only the first row has two cells. Let's fix that!

**Head of the table**

Here's the table we made earlier. It's okay, but it just looks like we have a list of famous Hollywood people (monsters?) and their birth years. To make our table look a little more like a table, we'll use the <thead> and<tbody> tags. These go within the<table> tag and stand for **t**able head and **t**able body, respectively.

The <head> HTML tag contains information about a web page (*e.g.* its title) and the <body> tag contains the contents of the web page. In the same way, the <thead> tag can be thought of as containing information about a table and the <tbody> tag containing the actual tabular data.

**Table Heads**

We have just added a <thead> tag above the <tbody>.  
It will hold the heading for each column.

You add text to a <thead> similar to a<tbody>, like this:

<thead>

<tr>

<th>

Name

</th>

<th>

Favorite Color

</th>

</tr>

</thead>

1. First we have an opening <thead>tag for the table head.
2. Then we have an opening <tr> tag for the row. (to start the row)
3. After that, a <th></th> cell for the Name column heading. Notice that we use <th></th> for the table heading cells instead of<td></td>.
4. Then another <th></th> cell for the Favorite Color column heading.
5. Finally, we close the row element with a closing </tr> tag, and close out the table heading element with a closing </thead> tag.

**Naming your table**

Our table is missing a title. We want to add a title row that goes all the way across the top.

To do so, we need to use the **colspan** attribute for the <th> tag. By default, table cells take up 1 column. If we want a table cell to take up the space of 3 columns instead of 1, we can set the colspan attribute to 3.

It looks like this:

<th colspan="3">3 columns across!</th>

**Instruction**

**Style that head!**

Your table is starting to look great, but it's still a little bland. We've gone ahead and added some styling to the table to make it a bit easier to read. It's your job to add the finishing touches!

Feel free to play around with any of the style attributes we added; you'll learn much more about these things later during the CSS courses.

If you want to add more than one style, you just separate your styles with a semicolon, like so:

<th style="font-size:12px; color:red"

**'Div'ide and conquer**

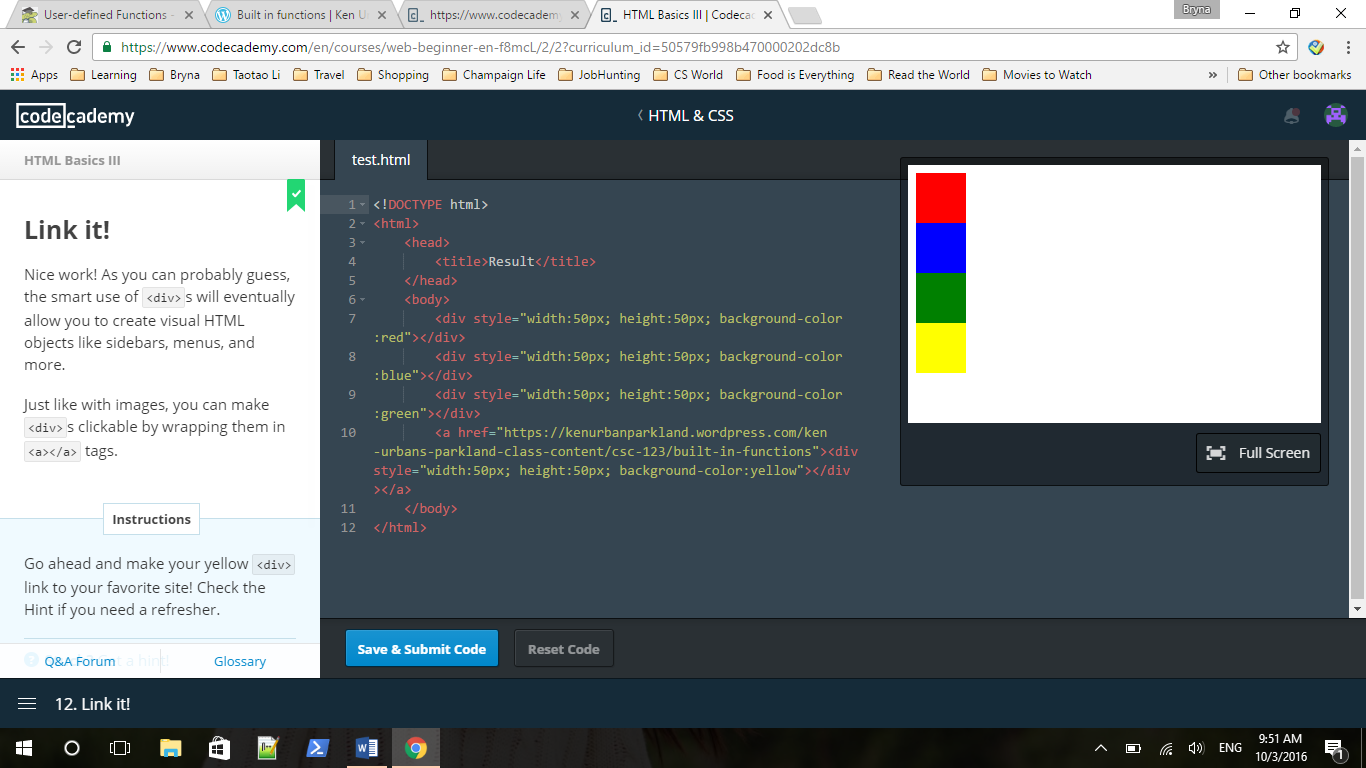
One of the most versatile structure tags available to you is the <div></div> tag. Short for "division," <div>allows you to divide your page into containers (that is, different pieces). This will come in handy when you begin learning CSS in the next unit: you'll be able to style different parts of your website individually!

Check out the Result tab. You should see three blocks: one red, one blue, and one green. Each one is its own<div> container.

**Link it!**

Nice work! As you can probably guess, the smart use of <div>s will eventually allow you to create visual HTML objects like sidebars, menus, and more.

Just like with images, you can make<div>s clickable by wrapping them in<a></a> tags.



**Spantastic**

While <div> allows you to divide your webpage up into pieces you can style individually, <span> allows you to control styling for smaller parts of your page, such as text. For example, if you always want the first word of your paragraphs to be red, you can wrap each first word in <span></span> tags and make them red using CSS!

 Wrap the word "red" in the editor in <span></span> tags and give the <span> tagstyle="color:red". Notice how only the word between the <span></span>tags changes color!

<!DOCTYPE html>

<html>

<head>

<title></title>

</head>

<body>

<p>This text is black, except for the word <span style="color: red">red!</span></p>

</body>

</html>

# CSS

## Intro to CSS

**What CSS is**

CSS (which stands for **C**ascading **S**tyle**S**heets) is a language used to describe the appearance and formatting of your HTML.

A **style sheet** is a file that describes how an HTML file should look. That's it!

We say these style sheets are **cascading** because the sheets can apply formatting when more than one style applies. For instance, if you say all paragraphs should have blue font, but you specifically single out one paragraph to have red font, CSS can do that! (We'll talk more about cascading in section four.)

**Why separate form from function?**

Great work! Look at you. You're already writing CSS.

There are two main reasons for separating your form/formatting (CSS) from your functional content/structure (HTML):

1. You can apply the same formatting to several HTML elements without rewriting code (*e.g.*style="color:red":) over and over
2. You can apply similar appearance and formatting to several HTML pages from a single CSS file

Look at the HTML in index.html. That's a lot of <span></span> tags! All those words are in regular font, but we want them to be super fancy.

**If it's in, it's out!**

We previously showed you how to do **inline styling** with HTML, like so:

<p style="color:red">Red font!</p>

This is a less awesome way to style your website for the reasons we just mentioned: you have to write the same code over and over, and if you want to make a big stylistic change to several elements, you have to change every single style tag. With a single CSS file, you only have to make the change in one place!

There are two ways to put CSS in one place. This first is to put your CSS between <style></style> tags, right in the same file as your HTML. These<style> tags go inside the <head></head> of your webpage; check out the example in the editor to the right.

**Link it up!**

But there's an even better way.

You know you should write your CSS in a totally separate file. But how do you make sure your HTML file can see that CSS information?

You do this by putting a <link> tag (as you saw in the first exercise of this course) between the <head>...</head>tags of your HTML page. Your <link>tag needs three attributes:

1. A type attribute that should always be equal to "text/css"
2. A rel attribute that should always be equal to "stylesheet"
3. A href attribute that should point to the web address of your CSS file

In the editor to the right, you'll see two files: index.html and stylesheet.css.

**PSA: Self-closing tags**

This brings us to a quick (but noteworthy!) concept in HTML: the **self-closing tag**.

Because nothing ever goes between<link></link> tags, it's okay to use a single set of <>s to be the opening *and* closing tags. You do that like so:

<link type="text/css" rel="stylesheet" href="CSS file address"/>

You may have noticed us do something similar with the <img> tag:

<img src="web address"/>

**Most tags are not self-closing**, but we'll point out the self-closing ones to help save you time and typing.

**Syntax for the wintax**

CSS syntax is different from the HTML you're used to, but don't worry: it's easy to pick up! The general format looks like this:

selector {

property: value;

}

A **selector** can be any HTML element, such as <p>, <img>, or <table>. You just take off the <>s! To make a paragraph's text red with CSS, you'd type:

p {

color: red;

}

A **property** is an aspect of a selector. For instance, you can change the font-family, color, and font-size of the text on your web pages (in addition to many more).

A **value** is a possible setting for a property. color can be red, blue, black, or almost any color; font-family can be a whole bunch of different fonts; and so on.

You need to end each property-value with a semi-colon (;). That's how CSS knows you're done with one pair and ready for the next.

**One selector, many properties**

Great work!

Another cool advantage of CSS is that you can set many properties for one selector. For instance, if you want to set a paragraph's font, font color, and font size, you can simply write:

p {

font-family: Arial;

color: blue;

font-size: 24px;

}

Remember: end each property-value pair with a semicolon!

**Please note:** If you have adjusted your browser's zoom, tests involving font-size and height will not work correctly. To remedy this, please type Command+0 or Ctrl+0 to reset your view.

**Color commentary**

Great! You're really getting the hang of this.

While it's important to get all your syntax down correctly, it's also a good idea to write **comments** as you go along. Good comments will help remind you why you did something a certain way (or will help someone else out if they're reading your code without you there to explain it).

As you've seen, HTML comments look like this:

*<!--I'm a comment!-->*

CSS comments, on the other hand, look like this:

*/\*I'm a comment!\*/*

Remember: the computer doesn't look at comments when figuring out what your HTML and CSS should do, but writing good comments is a good habit you want to pick up!

**Hexawhatnow?**

You've got the main ideas—now it's time to dive into the nitty-gritty.

You've noticed that when we've asked you to set color properties using CSS, we've been having you type things like color:red. You may have asked yourself: what if I want maroon? Or fire engine red? Or more of a red-orange? Does CSS know all those words?

The answer is no. It can, however, understand millions of colors in the form of **hexadecimal values**.

You're already extremely familiar with **decimal** values: it's everyday counting! You know when you see a number (*e.g.* 1,432) that each digit can only be the ten values 0 through 9. Because there are only ten possibilities, we say that regular counting is **base-10**.

Hexadecimal counting is **base-16**. Each digit can be the numbers 0 through 9 **or the letters a through f**! Crazy, right? Check it out:

**Roses are red...**

There are a lot of tools available on the Internet for looking up hexadecimal (or simply **hex**) color values.

Search for "hex color palette" or "hex color picker" with your favorite web browser to find a bunch of options!

Hex values always start with a pound sign (#), are up to six "digits" long, and are **case-insensitive**: that is, they don't care about capitalization.#FFC125 and #ffc125 are the same color.

**Pixels and ems**

Great work! We'll do more with colors as you learn more CSS.

When we've asked you to adjust font size, we've specified that the unit you should use is px (for "pixels"), like so:

p {

font-size: 10px;

}

A pixel is a dot on your computer screen. Specifying font sizes in pixels is great when you want the user to see exactly on their screen what you designed on yours, though it assumes your screens are of similar size.

What if the user is using a screen that's a very different size from yours, though (like a smartphone screen)? Enter **em**s. (Don't confuse these with the <em></em> tags we use for *emphasis*!)

The font-size unit **em** is a **relative** measure: one em is equal to the default font size on whatever screen the user is using. That makes it great for smartphone screens, since it doesn't try to tell the smartphone *exactly* how big to make a font: it just says, "Hey, 1em is the font size that you normally use, so 2em is twice as big and 0.5em is half that size!"

Check it out: we've set three different paragraphs to the font-sizes 1em, 0.5em, and 2em. For now, use whichever unit (px or em) you're more comfortable with—we just wanted to show you em now so you're not surprised when you see it later.

**A font of knowledge**

We've also asked you to change the font-family of certain elements using CSS. You've seen us use the fonts Verdana, Courier, and Garamond. But how many fonts does CSS know?

The answer is: it depends. Most computers will understand popular fonts like Verdana, Courier, and Garamond, but each individual computer has different fonts installed on it. The good news is that CSS has some built-in defaults meant to ensure your users see what you intend. They are:

**serif**: A font with little decorative bits on the ends of the strokes that make up letters. Do a search on "serif" to see what we mean.

**sans-serif**: A plain-looking font, like this one. It doesn't have the little doohickies on the ends of letters like a serif font does.

**cursive**: A scripty font! It looks like cursive writing.

We'll show you how to import your own fonts in a later course! This will help make sure the person viewing your page has all the fonts you want them to have.

**Backup values**

You don't have to jump straight to a default value like cursive or sans-serif: you can tell CSS to try several, going from one to the next if the one you want isn't available.

For example, if you write:

p {

font-family: Tahoma, Verdana, sans-serif;

}

CSS will first try to apply Tahoma to your paragraphs. If the user's computer doesn't have that font, it will try Verdana next, and if that doesn't work, it will show a default sans-serif font.

**Background color, height, and width**

Remember our friend <div>, and how we used it to make those multi-colored blocks? Time for you to build your own blocks! (Well, block. Let's not get ahead of ourselves.)

There are three properties you'll need to set values for:

1. background-color, which you set to a color or hex value
2. height, which you set to a value in pixels
3. width, which is also measured in pixels

These exercises will give you a brief overview of the different HTML elements you can select and what some of their property-value pairs are (like the new ones we mention above). We'll cover HTML element selection more in the next course!

**Bordering on insanity**

Many HTML elements support the border property. This can be especially useful with tables.

The border property in turn supports several values. For example, for a border 2 pixels thick, solid, and red, you'd type

selector {

border: 2px solid red;

}

Borders: pretty fancy.

**Links and text decoration**

Links have a lot of the same properties as regular text: you can change their font, color, size, and so on.

But links also have a property, text-decoration, that you can change to give your links a little more custom flair. You're probably used to seeing links that are blue and underlined, right? Well, that's not the way it has to be!

## CSS Classes and IDs

**Multiple Selectors**

As you've seen, it's possible to nest HTML elements inside one another, like so:

<div>

<div>

<p>I like tacos!</p>

So what if you want to grab <p>s that are inside two <div>s, and not *all* <p>s?

You select those in the CSS tab like this:

div div p {

/\*CSS stuff!\*/

}

**One selector to rule them all**

There's also a very special selector you can use to apply CSS styling to *every element* on the page: the \* selector. For example, if you type

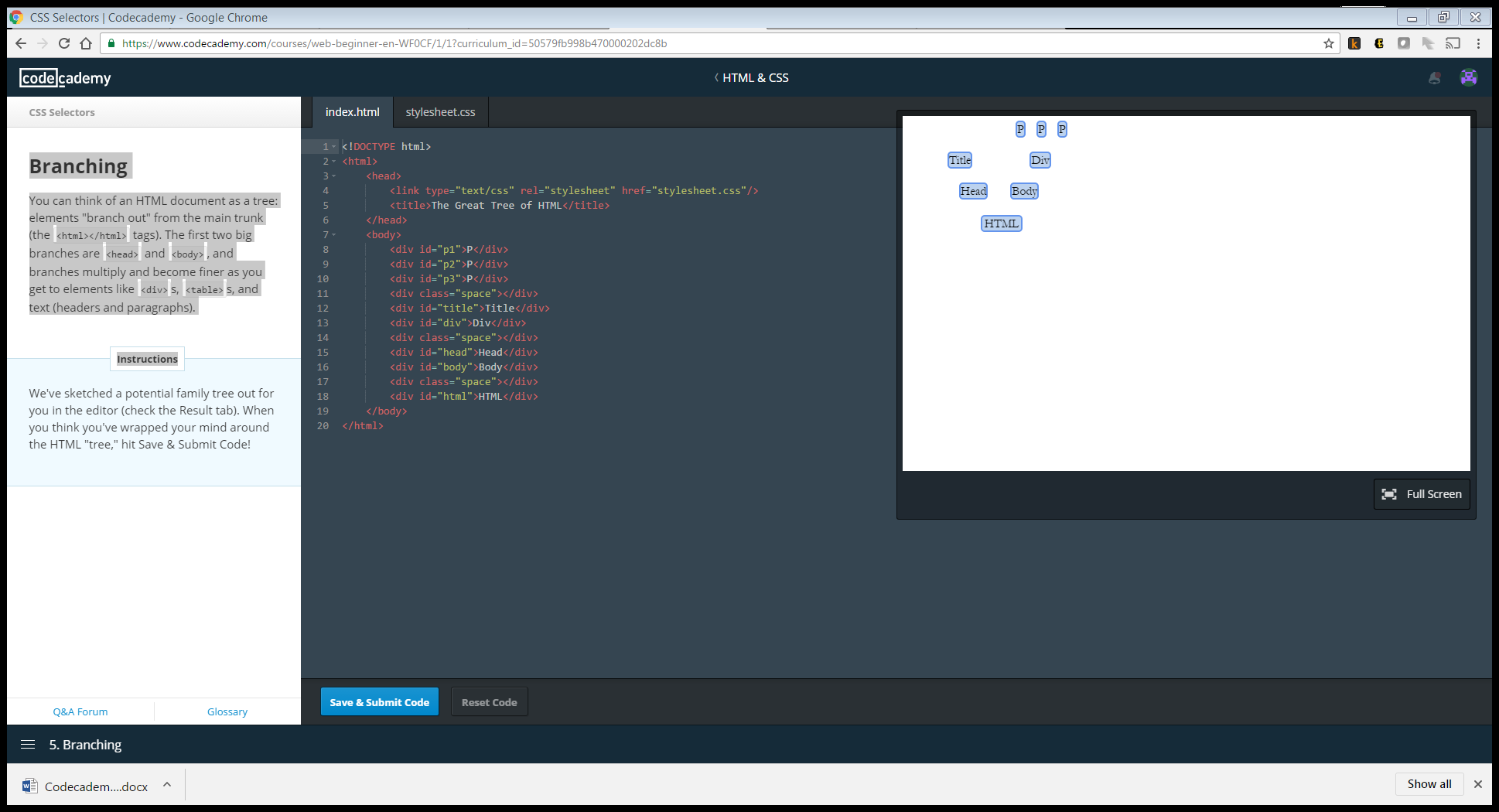
\* {

border: 2px solid black;

}

You'll create a two-pixel wide solid black border around *every* element on the HTML page.

**Branching**

You can think of an HTML document as a tree: elements "branch out" from the main trunk (the <html></html> tags). The first two big branches are <head> and <body>, and branches multiply and become finer as you get to elements like <div>s, <table>s, and text (headers and paragraphs). 

**Parents, children, and siblings**

If you think of the <html> tag as the trunk of the tree, you can think of its immediate branches—<head> and <body>—as its **children**. Both tags are children of <html>, and <html> is their **parent** element. Because they are both immediate children of <html>(that is, they are both only one element away), they are **siblings**.

Just like a real family, elements have children, grandchildren, great-grandchildren, and so on (though we don't make this distinction with HTML—a child of an element, and all that child's children, are children of the first parent).

**Can you swing it?**

Good work! Let's try something a little more involved.

Remember, you can reach an element that is a child of another element like this:

div div p { /\* Some CSS \*/ }

where in this case, we'd be grabbing any <p>that is nested *somewhere* inside a <div> that is nested *somewhere* inside another <div>. If you want to grab *direct children*—that is, an element that is *directly* nested inside another element, with no elements in between—you can use the > symbol, like so:

div > p { */\* Some CSS \*/* }

This only grabs <p>s that are nested *directly* inside of <div>s; it won't grab any paragraphs that are, say, nested inside lists that are in turn nested inside <div>s.

**See it to believe it**

Excellent! You've got the hang of this, and you're starting to learn more about cascading.

As we mentioned, certain selectors will "override" others if they have a greater **specificity value**. ul li p { is more specific CSS than just p {, so when CSS sees tags that are *both* <p> tags*and* happen to be inside unordered lists, it will apply the more specific styling (ul li p {) to the text inside the lists.

There are two selectors that are even more specific than nested selectors like the ones above: **classes** and **IDs**. Check them out in the editor to the right.

**Keeping it classy**

Classes are useful when you have a bunch of elements that should all receive the same styling. Rather than applying the same rules to several selectors, you can simply apply the same class to all those HTML elements, then define the styling for that class in the CSS tab.

Classes are assigned to HTML elements with the word class and an equals sign, like so:

<div class="square"></div>

<img class="square"/>

<td class="square"></td>

Classes are identified in CSS with a dot (.), like so:

.square {

height: 100px;

width: 100px;

}

This allows you to take elements of different types and give them the same styling.

**ID, please!**

IDs, on the other hand, are great for when you have exactly *one* element that should receive a certain kind of styling.

IDs are assigned to HTML elements with the word id and an equals sign:

<div id="first"></div>

<div id="second"></div>

<p id="intro"></p>

IDs are identified in CSS with a pound sign (#):

#first {

height: 50px;

}

#second {

height: 100px;

}

#intro {

color: #FF0000;

}

This allows you to apply style to a single instance of a selector, rather than *all* instances.

**Even finer control**

You've learned about class selectors. Now it's time to learn about **pseudo-class selectors**.

A **pseudo-class selector** is a way of accessing HTML items that aren't part of the document tree (remember the tree structure we talked about earlier?). For instance, it's very easy to see where a link is in the tree. But where would you find information about whether a link had been clicked on or not? It isn't there!

Pseudo-class selectors let us style these kinds of changes in our HTML document. For example, we saw we could change a link's text-decoration property to make it something other than blue and underlined. Using pseudo selectors, you can control the appearance of unvisited and visited links—even links the user is hovering over but hasn't clicked!

The CSS syntax for pseudo selectors is

selector:pseudo-class\_selector {

property: value;

}

It's just that little extra colon (:).

**Links**

There are a number of useful pseudo-class selectors for links, including:

a:link: An unvisited link.  
a:visited: A visited link.  
a:hover: A link you're hovering your mouse over.

Let's try a few!

**First child**

Another useful pseudo-class selector is first-child. It's used to apply styling to *only* the elements that are the first children of their parents. For instance:

p:first-child {

color: red;

}

Would make all paragraphs that are the first children of their parent elements red.

**Nth child**

Well done! You can actually select *any*child of an element after the first child with the pseudo-class selector nth-child; you just add the child's number in parentheses after the pseudo-class selector. For example,

p:nth-child(2) {

color: red;

}

Would turn every paragraph that is the*second* child of its parent element red.

The element that **is the child** goes before :nth-child; its parent element is the element that contains it.

**Multiple selectors**

Remember how to reach selectors nested inside others? If you have a paragraph inside a div that's inside *another* div, you can get to it like this:

div div p {

/\*Some CSS\*/

}

This will style all paragraphs nested inside two divs and will leave all paragraphs that don't meet these criteria alone.

**Please note:** If you have adjusted your browser's zoom, tests involving font-size and height will not work correctly. To remedy this, please type Command+0 or Ctrl+0 to reset your view.

**Class selectors**

You've also learned how to use class selectors to modify different elements (that is, you can give the same styling to an h3 header, a paragraph, a link, and a table).

**ID selectors**

You've also learned about ID selectors, and how they can be used to target a specific element.

*<!-- HTML -->*

<div id="menu"></div>

/\* CSS \*/

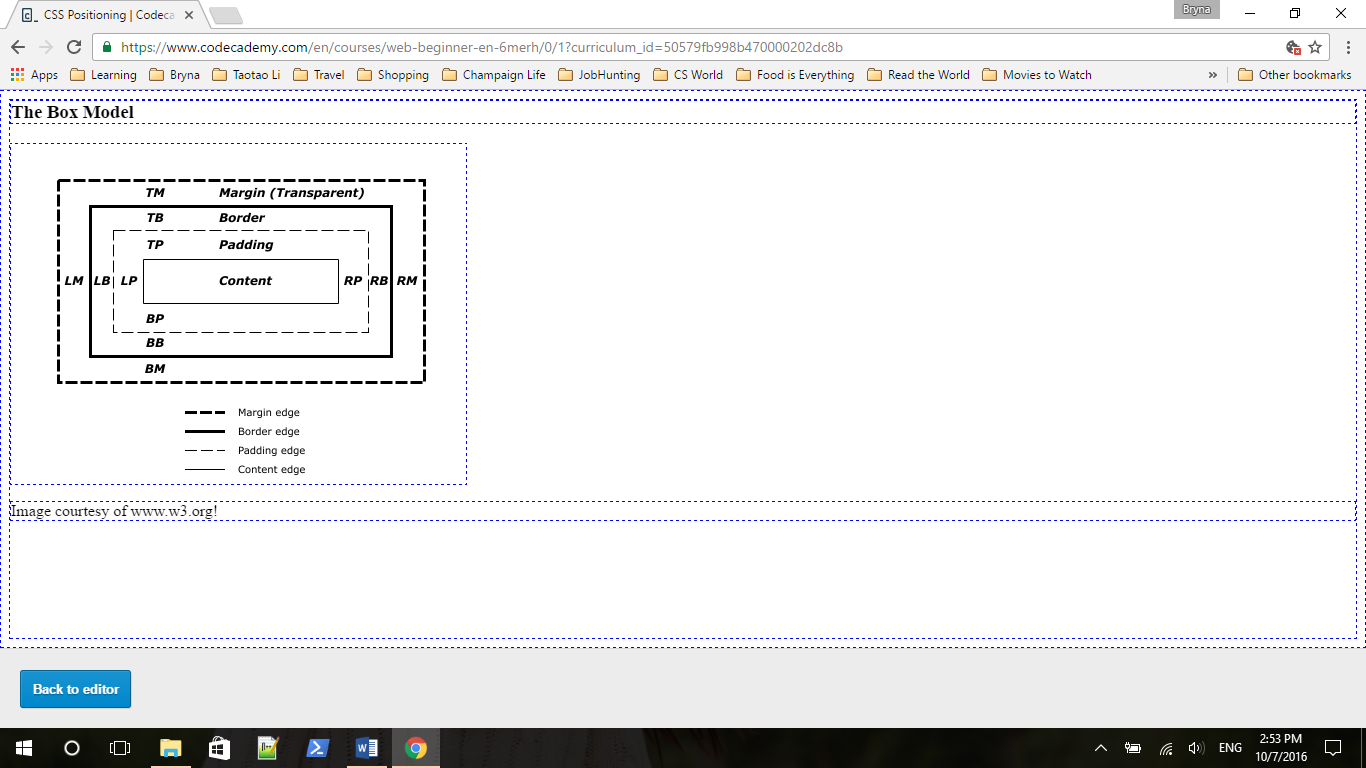
#menu {

color: #000000;

}

The example above is just a reminder.

## CSS Positioning



Controlling the position of HTML elements allows you incredibly fine control over how your pages look. No longer will your <div>s sit directly on top of one another! (Unless you want them to.)

As we mentioned, elements populate the page in what's known as the **CSS box model**. Each HTML element is like a tiny box or container that holds the pictures and text you specify.

Cool, right? Each HTML element gets its own box to live in.

As you saw, the outermost box of each element went all the way across the page. This is why until now, your HTML elements have been sitting on top of one another: by default, they take up the full width of the page.

We can change all this with the first positioning property we'll learn: the**display** property. We'll learn about four possible values.

**block**: This makes the element a block box. It won't let anything sit next to it on the page! It takes up the full width.

**inline-block**: This makes the element a block box, but will allow other elements to sit next to it on the same line.

**inline**: This makes the element sit on the same line as another element, but without formatting it like a block. It only takes up as much width as it needs (not the whole line).

**none**: This makes the element and its content disappear from the page entirely!

**Inline-block**

As mentioned, any element that comes in as a block (say, a paragraph) will automatically take up the full width of the page, no matter how much or how little content you put in.

If we specify a display of inline-block, however, our blocks are still blocks, but will be able to sit next to each other on the same line.

**Inline**

Did you see that? Your <div>s all moved onto the same line! You can already start to see how this type of positioning can be useful for navigation bars like the one at the top of the main Codecademy page (where you can click "Learn," "Teach," and so on).

The inline-block value allows you to put several block elements on the same line. The inline value places all your elements next to one another, but not as blocks: they don't keep their dimensions.

**None!**

The good news is, inline places all your elements on a single line. The bad news is that it doesn't maintain their "box"ness: as you saw, all your<div>s got squished to the smallest possible width!

The inline display value is better suited for HTML elements that are blocks by default, such as headers and paragraphs.

Finally, we'll try out the display value none. As you might expect, this prevents the page from displaying the selected element. As you might *not* expect, this removes the selected element from the page *entirely*, including any children and any content. Poof! Gone! (But not gone forever—changing the display value away from none will bring everything back.)

**Sketching it out**

Now that you understand more about the display property and the box model, we can delve into the details of how each individual box behaves.

Check out the diagram in the Result tab (it's the one from the first exercise in this lesson). As you can see, each box is made of layers. From the outermost to the innermost:

The **margin** is the space around the element. The larger the margin, the more space between our element and the elements around it. We can adjust the margin to move our HTML elements closer to or farther from each other.

The **border** is the edge of the element. It's what we've been making visible every time we set the border property.

The **padding** is the spacing between the content and the border. We can adjust this value with CSS to move the border closer to or farther from the content.

The **content** is the actual "stuff" in the box. If we're talking about a <p>element, the "stuff" is the text of the paragraph.

You'll see abbreviations like **TM**, **TB**, and **TP** in the diagram. These stand for "top margin," "top border," and "top padding." As we'll see, we can adjust the top, right, left, and bottom padding, border, and margin individually.

**Margin**

Let's start with our margins. Adjusting our margins not only moves our element relative to other elements on the page, but also relative to the "walls" of the HTML document.

For instance, if we take an HTML element with a specific width (such as our <div> in the editor) and set its margin to auto, this tells the document to auto matically put equal left and right margins on our element, centering it on the page.

**Margin top, right, bottom, left**

If you want to specify a particular margin, you can do it like this:

margin-top: */\*some value\*/*

margin-right: */\*some value\*/*

margin-bottom: */\*some value\*/*

margin-left: */\*some-value\*/*

You can also set an element's margins all at once: you just start from the top margin and go around clockwise (going from top to right to bottom to left). For instance,

margin: 1px 2px 3px 4px;

will set a top margin of 1 pixel, a right margin of 2, a bottom of 3, and a left of 4.

**Padding**

Good! Let's adjust the padding. Remember, the padding is the space between your border and your innermost layer: the actual content.

Padding can be set in two ways, just like your margins. You can either select them individually, like this:

padding-top: */\*some value\*/*

padding-right: */\*some value\*/*

padding-bottom: */\*some value\*/*

padding-left: */\*some-value\*/*

Or select them all in one declaration, like this:

padding: value value value value;

You should also know that if you want your padding to be the same for all four sides, you can declare that value only once. padding: 10px will give your HTML element 10 pixels of padding on all sides.

**Negative values**

Did you see that? Your <div> got huge! That's because the background color is the same for the content and the padding.

When you give CSS a positive padding or margin value, it puts that space between the element and its reference: for instance, if you have a<div> and you give it a margin-left of20px, it puts twenty pixels between the left margin of that <div> and the side of the screen. This effectively moves the <div> twenty pixels to the *right*.

If you want to move an element in the other direction, you can give CSS a *negative* value: margin-left: -20px will move the element twenty pixels to the *left*.

**To the right!**

Okay! So we know how our individual elements are constructed. But how do we determine where they go on the page?

One way is to use **floats**. When you float an element on the page, you're telling the webpage: "I'm about to tell you where to put this element, but you have to put it into the **flow** of other elements." This means that if you have several elements all floating, they all know the others are there and don't land on top of each other.

You can think of the HTML page as sort of like a sea, and floating elements as boats on it: all the boats have positions on the sea, and they all see and steer clear of each other.

(Some of the positioning methods we'll learn in upcoming sections *can* accidentally drop elements on top of each other.)

**Clearing elements**

Unfortunately, we sometimes mix large floating elements with non-floating ones, and elements *do* end up on top of each other.

See your footer (the blue bit between the two columns)? It's stuck back there because we haven't told it something very important: to clear the other elements on the page!

If you tell an element to clear: left, it will immediately move below any floating elements on the left side of the page; it can also clear elements on the right. If you tell it to clear: both, it will get out of the way of elements floating on the left *and* right!

The syntax is what you're used to:

element {

clear: */\*right, left, or both\*/*

}

**Static by default**

Great work so far! Now that you understand positioning elements with float, let's move on to slightly more complex positioning methods.

If you don't specify an element's positioning type, it defaults to static. This just means "where the element would normally go." If you don't tell an element how to position itself, it just plunks itself down in the document.

**Absolute positioning**

The first type of positioning is **absolute** positioning. When an element is set to position: absolute, it's then positioned in relation to the first parent element it has that *doesn't* have position: static. If there's no such element, the element with position: absolute gets positioned relative to <html>.

To show you how this works, we've set the #outer div to have absolute positioning. This means that when you position the #inner div, it will be relative to #outer. (If #outer had the default positioning of static, then#inner would get positioned relative to the entire HTML document.)