WEB Development Learning Path Notes

CSS

Table of Contents

[1.1 Using the <Style> tag 4](#_Toc32328674)

[1.2 Using the css file 4](#_Toc32328675)

[1.2.1 Dots in paths 5](#_Toc32328676)

[1.3 Tag Names 6](#_Toc32328677)

[1.4 Class Name 6](#_Toc32328678)

[1.5 ID Name 7](#_Toc32328679)

[1.6 Classes VS I’d’s 8](#_Toc32328680)

[1.7 Chaining Selectors 9](#_Toc32328681)

[1.8 Nested Elements 9](#_Toc32328682)

[1.9 Important 10](#_Toc32328683)

[1.10 Multiple Selectors 11](#_Toc32328684)

[2 Advanced Design With CSS 11](#_Toc32328685)

[2.1 The Box Model 11](#_Toc32328686)

[2.1.1 Height & Width 12](#_Toc32328687)

[2.1.2 Borders 13](#_Toc32328688)

[2.1.3 Padding 14](#_Toc32328689)

[2.1.4 Margins 15](#_Toc32328690)

[2.1.5 Minimum and Maximum Height and Width 18](#_Toc32328691)

[2.1.6 Resetting Defaults 20](#_Toc32328692)

[2.1.7 Visibility 20](#_Toc32328693)

[2.1.8 Changing the Box Model 21](#_Toc32328694)

[2.1.9 Border Box 22](#_Toc32328695)

[2.2 Flow of HTML 23](#_Toc32328696)

[2.2.1 Position 24](#_Toc32328697)

[2.2.2 Z Index 27](#_Toc32328698)

[2.2.3 Display 28](#_Toc32328699)

[2.2.4 Float 30](#_Toc32328700)

[2.2.5 Clear 31](#_Toc32328701)

[2.3 Color 32](#_Toc32328702)

[2.3.1 Hexadecimal colors 32](#_Toc32328703)

[2.3.2 RGB Colors 33](#_Toc32328704)

[2.3.3 Hue Saturation Lightness 33](#_Toc32328705)

[2.3.4 Opacity and Alpha 34](#_Toc32328706)

[2.4 Typography 35](#_Toc32328707)

[2.4.1 Font Family 35](#_Toc32328708)

[2.4.2 Font Weight 36](#_Toc32328709)

[2.4.3 Font Style 37](#_Toc32328710)

[2.4.4 Word Spacing 37](#_Toc32328711)

[2.4.5 Letter Spacing 38](#_Toc32328712)

[2.4.6 Text Transformation 38](#_Toc32328713)

[2.4.7 Text Alignment 38](#_Toc32328714)

[2.4.8 Line Height Anatomy 39](#_Toc32328715)

[2.4.9 Serif and Sans Serif 40](#_Toc32328716)

[2.4.10 Fallback Fonts 40](#_Toc32328717)

[2.4.11 Linking Fonts 41](#_Toc32328718)

[2.4.12 Font-Face 42](#_Toc32328719)

[3 Positioning crib sheet 43](#_Toc32328720)

[4 Sizing Elements 43](#_Toc32328721)

[4.1 EM 44](#_Toc32328722)

[4.2 REM 45](#_Toc32328723)

[4.3 Percenteges 45](#_Toc32328724)

[4.3.1 height & width 45](#_Toc32328725)

[4.3.2 Padding & Margins 46](#_Toc32328726)

[4.3.3 Width Min & Max 47](#_Toc32328727)

[4.3.4 min and max height 47](#_Toc32328728)

[4.3.5 Scaleing images and video 48](#_Toc32328729)

[4.3.6 Scaling Background images 49](#_Toc32328730)

[5 Responsive Web Design 50](#_Toc32328731)

[5.1 Media Queries 50](#_Toc32328732)

[5.2 Range 51](#_Toc32328733)

[5.3 Dots Per Inch (DPI) 52](#_Toc32328734)

[5.4 And Operator 52](#_Toc32328735)

[5.5 Comma Seperated List 53](#_Toc32328736)

[5.6 BreakPoints 53](#_Toc32328737)

[6 FlexBox 54](#_Toc32328738)

[6.1 Display:flex 55](#_Toc32328739)

[6.2 Display:inline-flex 55](#_Toc32328740)

[6.3 Justify-content 57](#_Toc32328741)

[6.4 Align-items 57](#_Toc32328742)

[6.5 Flex Grow 58](#_Toc32328743)

[6.6 Flex-shrink 59](#_Toc32328744)

[6.7 Flex-basis 60](#_Toc32328745)

[6.8 Flex 61](#_Toc32328746)

[6.9 Flex-wrap 62](#_Toc32328747)

## Using the <Style> tag

Inline styles are a fast way of styling HTML, but they also have limitations. If you wanted to style, for example, multiple <h1> elements, you would have to add inline styling to each element manually. In addition, you would also have to maintain the HTML code when additional <h1> elements are added.

Fortunately, HTML allows you to write CSS code in its own dedicated section with the <style> element. CSS can be written between opening and closing <style> tags. To use the <style> element, it must be placed inside of the <head> element.

<head>

<style>

</style>

</head>

After adding a <style> tag in the head section, you can begin writing CSS code.

<head>

<style>

p {

color: red;

font-size: 20px;

}

</style>

</head>

The CSS code in the example above changes the color of all paragraph text to red and also changes the size of the text to 20 pixels. Note how the syntax of the CSS code matches (for the most part) the syntax you used for inline styling. The main difference is that you can specify which elements to apply the styling to.

## Using the css file

Developers avoid mixing code by storing HTML and CSS code in separate files (HTML files contain only HTML code, and CSS files contain only CSS code).

You can create a CSS file by using the **.css** file name extension, like so: **style.css**

With a CSS file, you can write all the CSS code needed to style a page without sacrificing the readability and maintainability of your HTML file.

When HTML and CSS code are in separate files, the files must be linked. Otherwise, the HTML file won’t be able to locate the CSS code, and the styling will not be applied.

You can use the <link> element to link HTML and CSS files together. The <link> element must be placed within the head of the HTML file. It is a self-closing tag and requires the following three attributes:

1. href — like the anchor element, the value of this attribute must be the address, or path, to the CSS file.
2. type — this attribute describes the type of document that you are linking to (in this case, a CSS file). The value of this attribute should be set to text/css.
3. rel — this attribute describes the relationship between the HTML file and the CSS file. Because you are linking to a stylesheet, the value should be set to stylesheet.

When linking an HTML file and a CSS file together, the <link> element will look like the following:

<link href="https://www.codecademy.com/stylesheets/style.css" type="text/css" rel="stylesheet">

Note that in the example above the path to the stylesheet is a URL:

https://www.codecademy.com/stylesheets/style.css

Specifying the path to the stylesheet using a URL is one way of linking a stylesheet.

If the CSS file is stored in the same [directory](https://en.wikipedia.org/wiki/Directory_(computing)) as your HTML file, then you can specify a [relative path](https://en.wikipedia.org/wiki/Path_(computing)#Absolute_and_relative_paths) instead of a URL, like so:

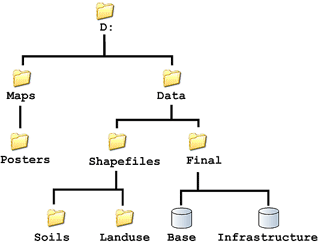
<link href="./style.css" type="text/css" rel="stylesheet">

Using a relative path is very common way of linking a stylesheet.

### Dots in paths

A relative path refers to a location that is relative to a current directory. Relative paths make use of two special symbols, a dot (.) and a double-dot (..), which translate into the current directory and the parent directory. Double-dots are used for moving up in the hierarchy. A single dot represents the current directory itself.

In the example directory structure below, assume you used Windows Explorer to navigate to D:\Data\Shapefiles\Soils. After navigating to this directory, a relative pathname will use D:\Data\Shapefiles\Soils as the current directory (until you navigate to a new directory, at which point the new directory becomes the current directory). The current directory is sometimes referred to as the root directory.



If you wanted to navigate to the Landuse directory from the current directory (Soils), you could type in the following in the Windows Explorer Address edit box:

..\Landuse

and Windows Explorer would navigate to D:\Data\Shapefiles\Landuse. A few more examples using D:\Data\Shapefiles\Landuse as the current directory are

.. *(D:\Data\Shapefiles)*

..\.. *(D:\Data)*

..\..\Final  *(D:\Data\Final)*

. *(D:\Data\Shapefiles\Landuse - the current directory)*

.\..\Soils *(D:\Data\Final\Soils)*

..\..\.\Final\..\Shapefiles\.\Landuse *(D:\Data\Shapefiles\Landuse)*

NOTE: You cannot type relative paths (using the dot and double-dot notation) in any ArcGIS application. Nor can you use relative paths in Python scripts.

NOTE: A relative path cannot span disk drives. For example, if your current directory is D:, you cannot use relative paths to navigate to any directory on E:

## Tag Names

CSS can select HTML elements by using an element’s tag name. A tag name is the word (or character) between HTML angle brackets.

For example, in HTML, the tag for a paragraph element is <p>. The CSS syntax for selecting <p> elements is:

p {

}

In the example above, all paragraph elements will be selected using a CSS selector. The selector in the example above is p. Note that the CSS selector matches the HTML tag for that element, but without the angle brackets.

In addition, two curly braces follow immediately after the selector (an opening and closing brace, respectively). Any CSS properties will go inside of the curly braces to style the selected elements.

## Class Name

CSS is not limited to selecting elements by tag name. HTML elements can have more than just a tag name; they can also have attributes. One common attribute is the class attribute. It’s also possible to select an element by its class attribute.

For example, consider the following HTML:

<p class="brand">Sole Shoe Company</p>

The paragraph element in the example above has a class attribute within the <p> tag. The class attribute is set to "brand". To select this element using CSS, we could use the following CSS selector:

.brand {

}

To select an HTML element by its class using CSS, a period (.) must be prepended to the class’s name. In the example above case, the class is brand, so the CSS selector for it is .brand.

We can use CSS to select an HTML element’s class attribute by name.

So far, we’ve selected elements using only one class name per element. If every HTML element had a single class, all the style information for each element would require a new class.

Luckily, it’s possible to add more than one class name to an HTML element’s class attribute.

For instance, perhaps there’s a heading element that needs to be green and bold. You could write two CSS rules like so:

.green {

color: green;

}

.bold {

font-weight: bold;

}

Then, you could include both of these classes on one HTML element like this:

<h1 class="green bold"> ... </h1>

We can add multiple classes to an HTML element’s class attribute by separating them with a space. This enables us to mix and match CSS classes to create many unique styles without writing a custom class for every style combination needed.

## ID Name

If an HTML element needs to be styled uniquely (no matter what classes are applied to the element), we can add an ID to the element. To add an ID to an element, the element needs an id attribute:

<h1 id="large-title"> ... </h1>

Then, CSS can select HTML elements by their id attribute. To select an id element, CSS prepends the id name with a hashtag (#). For instance, if we wanted to select the HTML element in the example above, it would look like this:

#large-title {

}

The id name is large-title, therefore the CSS selector for it is #large-title.

## Classes VS I’d’s

CSS can select HTML elements by their tag, class, and ID. CSS classes and IDs have different purposes, which can affect which one you use to style HTML elements.

CSS classes are meant to be reused over many elements. By writing CSS classes, you can style elements in a variety of ways by mixing classes on HTML elements.

For instance, imagine a page with two headlines. One headline needs to be bold and blue, and the other needs to be bold and green. Instead of writing separate CSS rules for each headline that repeat each other’s code, it’s better to write a .bold CSS rule, a .green CSS rule, and a .blue CSS rule. Then you can give one headline the bold green classes, and the other the bold blue classes.

While classes are meant to be used many times, an ID is meant to style only one element. As we’ll learn in the next exercise, IDs override the styles of tags and classes. Since IDs override class and tag styles, they should be used sparingly and only on elements that need to always appear the same.

Specificity is the order by which the browser decides which CSS styles will be displayed. A best practice in CSS is to style elements while using the lowest degree of specificity, so that if an element needs a new style, it is easy to override.

IDs are the most specific selector in CSS, followed by classes, and finally, tags. For example, consider the following HTML and CSS:

<h1 class="headline">Breaking News</h1>

h1 {

color: red;

}

.headline {

color: firebrick;

}

In the example code above, the color of the heading would be set to firebrick, as the class selector is more specific than the tag selector. If an ID attribute (and selector) were added to the code above, the styles within the ID selector’s body would override all other styles for the heading. The only way to override an ID is to add another ID with additional styling.

Over time, as files grow with code, many elements may have IDs, which can make CSS difficult to edit, since a new, more specific style must be created to change the style of an element.

To make styles easy to edit, it’s best to style with a tag selector, if possible. If not, add a class selector. If that is not specific enough, then consider using an ID selector.

ID 🡪Class 🡪Tag

ID will be applied with highest priority Tag with the lowest

## Chaining Selectors

When writing CSS rules, it’s possible to require an HTML element to have two or more CSS selectors at the same time.

This is done by combining multiple selectors, which we will refer to as chaining. For instance, if there was a .special class for h1 elements, the CSS would look like:

h1.special {

}

The code above would select only the h1 elements that have a class of special. If a p element also had a class of special, the rule in the example would not style the paragraph.

## Nested Elements

n addition to chaining selectors to select elements, CSS also supports selecting elements that are nested within other HTML elements. For instance, consider the following HTML:

<ul class='main-list'>

<li> ... </li>

<li> ... </li>

<li> ... </li>

</ul>

The nested <li> elements are selected with the following CSS:

.main-list li {

}

In the example above, .main-list selects the .main-list element (the unordered list element). The nested <li> are selected by adding li to the selector, separated by a space, resulting in .main-list li as the final selector (note the space in the selector).

Selecting elements in this way can make our selectors even more specific by making sure they appear in the context we expect.

In the last exercise, instead of selecting all h5 elements, you selected only the h5 elements nested inside the .description elements. This CSS selector was more specific than writing only h5. Adding more than one tag, class, or ID to a CSS selector increases the specificity of the CSS selector.

For instance, consider the following CSS:

p {

color: blue;

}

.main p {

color: red;

}

Both of these CSS rules define what a p element should look like. Since .main p has a class and a p tag as its selector, only the p elements inside the .main element will appear red. This occurs despite there being another more general rule that states p elements should be blue.

## Important

There is one thing that is even more specific than IDs: !important. !important can be applied to specific attributes instead of full rules. It will override any style no matter how specific it is. As a result, it should almost never be used. Once !important is used, it is very hard to override.

The syntax of !important in CSS looks like this:

p {

color: blue !important;

}

.main p {

color: red;

}

Since !important is used on the p selector’s color attribute, all p elements will appear blue, even though there is a more specific .main p selector that sets the color attribute to red.

The !important flag is only useful when an element appears the same way 100% of the time. Since it’s almost impossible to guarantee that this will be true throughout a project and over time, it’s best to avoid !important altogether. If you ever see !important used (or are ever tempted to use it yourself) we strongly recommend reorganizing your CSS. Making your CSS more flexible will typically fix the immediate problem and make your code more maintainable in the long run.

## Multiple Selectors

In order to make CSS more concise, it’s possible to add CSS styles to multiple CSS selectors all at once. This prevents writing repetitive code.

For instance, the following code has repetitive style attributes:

h1 {

font-family: Georgia;

}

.menu {

font-family: Georgia;

}

Instead of writing font-family: Georgia twice for two selectors, we can separate the selectors by a comma to apply the same style to both, like this:

h1,

.menu {

font-family: Georgia;

}

By separating the CSS selectors with a comma, both the h1 and the .menu elements will receive the font-family: Georgia styling.

# Advanced Design With CSS

## The Box Model

Browsers load HTML elements with default position values. This often leads to an unexpected and unwanted user experience, while limiting the views you can create. In this lesson you will learn about the *box model*, an important concept to understand how elements are positioned and displayed on a website.

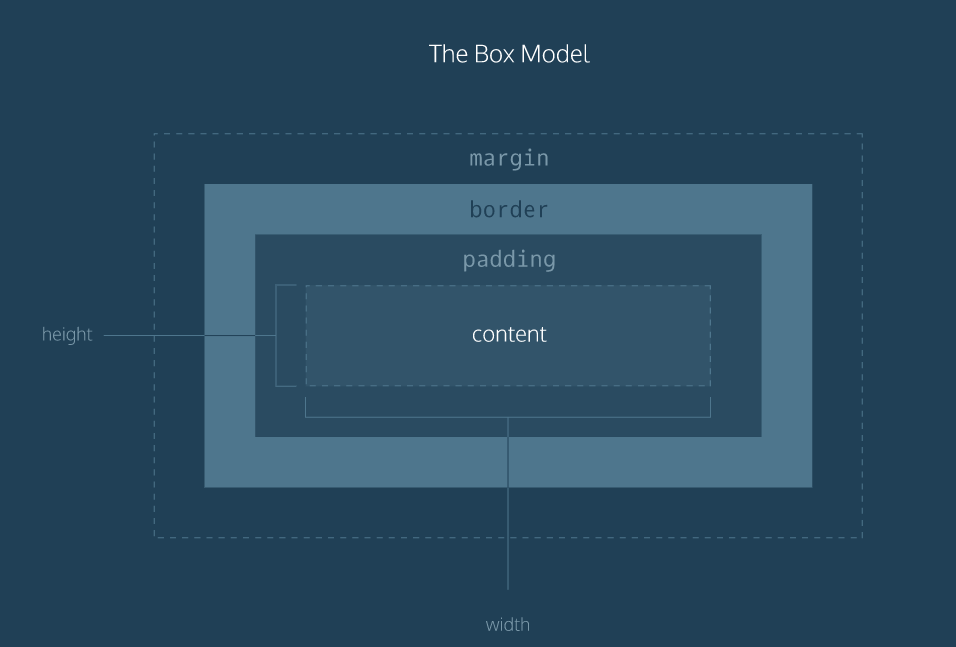
If you have used HTML and CSS, you have unknowingly seen aspects of the box model. For example, if you have set the background color of an element, you may have noticed that the color was applied not only to the area directly behind the element, but also to the area to the right of the element. Also, if you have aligned text, you know it is aligned relative to something. What is that something?

All elements on a web page are interpreted by the browser as “living” inside of a box. This is what is meant by the box model.

For example, when you change the background color of an element, you change the background color of its entire box.

In this lesson, you’ll learn about the following aspects of the box model:

1. The dimensions of an element’s box.
2. The borders of an element’s box.
3. The paddings of an element’s box.
4. The margins of an element’s box.



The box model comprises the set of properties which define parts of an element that take up space on a web page. The model includes the content area’s size (*width* and *height*) and the element’s *padding*, *border*, and *margin*. The properties include:

1. Width and height — specifies the width and height of the content area.
2. Padding — specifies the amount of space between the content area and the border.
3. Border — specifies the thickness and style of the border surrounding the content area and padding.
4. Margin — specifies the amount of space between the border and the outside edge of the element.

### Height & Width

An element’s content has two dimensions: a height and a width. By default, the dimensions of an HTML box are set to hold the raw contents of the box.

The CSS height and width properties can be used to modify these default dimensions.

p {

height: 80px;

width: 240px;

}

In this example, the height and width of paragraph elements are set to 80 pixels and 240 pixels, respectively — the px in the code above stands for *pixels*.

Pixels allow you to set the exact size of an element’s box (width and height). When the width and height of an element are set in pixels, it will be the same size on all devices — an element that fills a laptop screen will overflow a mobile screen.

### Borders

A border is a line that surrounds an element, like a frame around a painting. Borders can be set with a specific width, style, and color.

1. width — The thickness of the border. A border’s thickness can be set in pixels or with one of the following keywords: thin, medium, or thick.
2. style — The design of the border. Web browsers can render any of [10 different styles](https://developer.mozilla.org/en-US/docs/Web/CSS/border-style" \l "Values" \t "_blank). Some of these styles include: none, dotted, and solid.
3. color — The color of the border. Web browsers can render colors using a few different formats, including [140 built-in color keywords](https://developer.mozilla.org/en-US/docs/Web/CSS/color_value" \t "_blank).

p {

border: 3px solid coral;

}

In the example above, the border has a width of 3 pixels, a style of solid and a color of coral. All three properties are set in one line of code.

The default border is medium none color, where color is the current color of the element. If width, style, or color are not set in the CSS file, the web browser assigns the default value for that property.

p.content-header {

height: 80px;

width: 240px;

border: solid coral;

}

In this example, the border style is set to solid and the color is set to coral. The width is not set, so it defaults to medium.

#### Border Radius

Ever since we revealed the borders of boxes, you may have noticed that the borders highlight the true shape of an element’s box: square. Thanks to CSS, a border doesn’t have to be square.

You can modify the corners of an element’s border box with the border-radius property.

div.container {

border: 3px solid rgb(22, 77, 100);

border-radius: 5px;

}

The code in the example above will set *all four corners* of the border to a radius of 5 pixels (i.e. the same curvature that a circle with radius 5 pixels would have).

You can create a border that is a perfect circle by setting the radius equal to the height of the box, or to 100%.

div.container {

height: 60px;

width: 60px;

border: 3px solid rgb(22, 77, 100);

border-radius: 100%;

}

The code in the example above creates a <div> that is a perfect circle.

### Padding

The space between the contents of a box and the borders of a box is known as *padding*. Padding is like the space between a picture and the frame surrounding it. In CSS, you can modify this space with the padding property.

p.content-header {

border: 3px solid coral;

padding: 10px;

}

The code in this example puts 10 pixels of space between the content of the paragraph (the text) and the borders, on all four sides.

The padding property is often used to expand the background color and make content look less cramped.

If you want to be more specific about the amount of padding on each side of a box’s content, you can use the following properties:

1. padding-top
2. padding-right
3. padding-bottom
4. padding-left

Each property affects the padding on only one side of the box’s content, giving you more flexibility in customization.

p.content-header {

border: 3px solid fuschia;

padding-bottom: 10px;

}

In the example above, only the bottom side of the paragraph’s content will have a padding of 10 pixels.

Another implementation of the padding property lets you specify exactly how much padding there should be on each side of the content in a single declaration.

p.content-header {

border: 3px solid grey;

padding: 6px 11px 4px 9px;

}

In the example above, the four values 6px 11px 4px 9px correspond to the amount of padding in a clockwise rotation. In order, it specifies the amount of padding on the top (6 pixels), right (11 pixels), bottom (4 pixels), and left (9 pixels) sides of the content.

When using this implementation of the padding property, we must specify a padding value for all four sides of the element.

However, if the top and bottom values for padding will equal each other, and the left and right values for padding will also equal each other, you can use the following shortcut:

p.content-header {

padding: 5px 10px;

}

The first value, 5px, sets the padding value for the top and bottom sides of the content. The second value, 10px, sets the padding value for the left and right sides of the content.

### Margins

So far you’ve learned about the following components of the box model: content, borders, and padding. The fourth and final component of the box model is *margin*.

Margin refers to the space directly outside of the box. The margin property is used to specify the size of this space.

p {

border: 1px solid aquamarine;

margin: 20px;

}

The code in the example above will place 20 pixels of space on the outside of the paragraph’s box on all four sides. This means that other HTML elements on the page cannot come within 20 pixels of the paragraph’s border.

If you want to be even more specific about the amount of margin on each side of a box, you can use the following properties:

1. margin-top
2. margin-right
3. margin-bottom
4. margin-left

Each property affects the margin on only one side of the box, providing more flexibility in customization.

p {

border: 3px solid DarkSlateGrey;

margin-right: 15px;

}

In the example above, only the right side of the paragraph’s box will have a margin of 15 pixels. It’s common to see margin values used for a specific side of an element.

What if you don’t want equal margins on all four sides of the box?

A similar implementation of the margin property is used to specify exactly how much margin there should be on each side of the box in a single declaration.

p {

margin: 6px 10px 5px 12px;

}

In the example above, the four values 6px 10px 5px 12px refer to the amount of margin around the box in a clockwise rotation. In order, it specifies the amount of margin on the top (6 pixels), right (10 pixels), bottom (5 pixels), and left (12 pixels) sides of the box.

When using this implementation of the margin property, the margin value must be specified for all four sides of the box.

Just like the padding shortcut, when you’re certain that the top and bottom values for margin will equal each other, and that the left and right values for margin will also equal each other, you can use the following shortcut:

p {

margin: 6px 12px;

}

The first value, 6px, sets a margin value for the top and bottom of the box. The second value, 12px, sets a margin value for the left and right sides of the box.

#### Auto

The margin property also lets you center content. However, you must follow a few syntax requirements. Take a look at the following example:

div {

margin: 0 auto;

}

In the example above, margin: 0 auto; will center the divs in their containing elements. The 0 sets the top and bottom margins to 0 pixels. The auto value instructs the browser to adjust the left and right margins until the element is centered within its containing element.

The div elements in the example above should center within an element that fills the page, but this doesn’t occur. Why?

In order to center an element, a width must be set for that element. Otherwise, the width of the div will be automatically set to the full width of its containing element, like the <body>, for example. It’s not possible to center an element that takes up the full width of the page.

div.headline {

width: 400px;

margin: 0 auto;

}

In the example above, the width of the div is set to 400 pixels, which is less than the width of most screens. This will cause the div to center within a containing element that is greater than 400 pixels wide.

#### Margin Collapse

As you have seen, padding is space added inside an element’s border, while margin is space added outside an element’s border. One additional difference is that top and bottom margins, also called vertical margins, *collapse*, while top and bottom padding does not.

Horizontal margins (left and right), like padding, are always displayed and added together. For example, if two divs with ids #div-one and #div-two, are next to each other, they will be as far apart as the sum of their adjacent margins.

#img-one {

margin-right: 20px;

}

#img-two {

margin-left: 20px;

}

In this example, the space between the #img-one and #img-two borders is 40 pixels. The right margin of #img-one (20px) and the left margin of #img-two (20px) add to make a total margin of 40 pixels.

Unlike horizontal margins, vertical margins do not add. Instead, the larger of the two vertical margins sets the distance between adjacent elements.

#img-one {

margin-bottom: 30px;

}

#img-two {

margin-top: 20px;

}

In this example, the vertical margin between the #img-one and #img-two elements is 30 pixels. Although the sum of the margins is 50 pixels, the margin collapses so the spacing is only dependent on the #img-one bottom margin.

It may be helpful to think of collapsing vertical margins as a short person trying to push a taller person. The tall person has longer arms and can easily push the short person, while the person with short arms cannot reach the person with long arms.

### Minimum and Maximum Height and Width

Because a web page can be viewed through displays of differing screen size, the content on the web page can suffer from those changes in size. To avoid this problem, CSS offers two properties that can limit how narrow or how wide an element’s box can be sized to.

1. min-width — this property ensures a minimum width of an element’s box.
2. max-width — this property ensures a maximum width of an element’s box.

p {

min-width: 300px;

max-width: 600px;

}

In the example above, the width of all paragraphs will not shrink below 300 pixels, nor will the width exceed 600 pixels.

Content, like text, can become difficult to read when a browser window is narrowed or expanded. These two properties ensure that content is legible by limiting the minimum and maximum widths of an element.

You can also limit the minimum and maximum *height* of an element.

1. min-height — this property ensures a minimum height for an element’s box.
2. max-height — this property ensures a maximum height of an element’s box.

p {

min-height: 150px;

max-height: 300px;

}

In the example above, the height of all paragraphs will not shrink below 150 pixels and the height will not exceed 300 pixels.

What will happen to the contents of an element’s box if the max-height property is set too low? It’s possible for the content to spill outside of the box, resulting in content that is not legible. You’ll learn how to work around this issue in the next exercise.

#### Overflow

All of the components of the box model comprise an element’s size. For example, an image that has the following dimensions is 364 pixels wide and 244 pixels tall.

* 300 pixels wide
* 200 pixels tall
* 10 pixels padding on the left and right
* 10 pixels padding on the top and bottom
* 2 pixels border on the left and right
* 2 pixels border on the top and bottom
* 20 pixels margin on the left and right
* 10 pixels margin on the top and bottom

The total dimensions (364px by 244px) are calculated by adding all of the vertical dimensions together and all of the horizontal dimensions together. Sometimes, these components result in an element that is larger than the parent’s containing area.

How can we ensure that we can view all of an element that is larger than its parent’s containing area?

The overflow property controls what happens to content that spills, or overflows, outside its box. It can be set to one of the following values:

* hidden - when set to this value, any content that overflows will be hidden from view.
* scroll - when set to this value, a scrollbar will be added to the element’s box so that the rest of the content can be viewed by scrolling.
* visible - when set to this value, the overflow content will be displayed outside of the containing element. Note, this is the default value.

p {

overflow: scroll;

}

In the example above, if any of the paragraph content overflows (perhaps a user resizes their browser window), a scrollbar will appear so that users can view the rest of the content.

The overflow property is set on a parent element to instruct a web browser how to render child elements. For example, if a div’s overflow property is set to scroll, all children of this div will display overflowing content with a scroll bar.

### Resetting Defaults

All major web browsers have a default stylesheet they use in the absence of an external stylesheet. These default stylesheets are known as user agent stylesheets. In this case, the term “[user agent](https://en.wikipedia.org/wiki/User_agent)“ is a technical term for the browser.

User agent stylesheets often have default CSS rules that set default values for padding and margin. This affects how the browser displays HTML elements, which can make it difficult for a developer to design or style a web page.

Many developers choose to reset these default values so that they can truly work with a clean slate.

\* {

margin: 0;

padding: 0;

}

The code in the example above resets the default margin and padding values of all HTML elements. It is often the first CSS rule in an external stylesheet.

Note that both properties are both set to 0. When these properties are set to 0, they do not require a unit of measurement.

### Visibility

Elements can be hidden from view with the visibility property.

The visibility property can be set to one of the following values:

1. hidden — hides an element.
2. visible — displays an element.

<ul>

<li>Explore</li>

<li>Connect</li>

<li class="future">Donate</li>

<ul>

.future {

visibility: hidden;

}

In the example above, the list item with a class of future will be hidden from view in the browser.

Keep in mind, however, that users can still view the contents of the list item (e.g., Donate) by viewing the source code in their browser. Furthermore, the web page will *only* hide the contents of the element. It will still leave an empty space where the element is intended to display.

**Note:** What’s the difference between display: none and visibility: hidden? An element with display: none will be completely removed from the web page. An element with visibility: hidden, however, will not be visible on the web page, but the space reserved for it will.

### Changing the Box Model

The last lesson focused on the most important aspects of the box model: box dimensions, borders, padding, and margin.

The box model, however, has an awkward limitation regarding box dimensions. This limitation is best illustrated with an example.

<h1>Hello World</h1>

h1 {

border: 1px solid black;

height: 200px;

width: 300px;

padding: 10px;

}

In the example above, a heading element’s box has solid, black, 1 pixel thick borders. The height of the box is 200 pixels, while the width of the box is 300 pixels. A padding of 10 pixels has also been set on all four sides of the box’s content.

Unfortunately, under the current box model, the border thickness and the padding will affect the dimensions of the box.

The 10 pixels of padding increases the height of the box to 220 pixels and the width to 320 pixels. Next, the 1-pixel thick border increases the height to 222 pixels and the width to 322 pixels.

Under this box model, the border thickness and padding are added to the overall dimensions of the box. This makes it difficult to accurately size a box. Over time, this can also make all of a web page’s content difficult to position and manage.

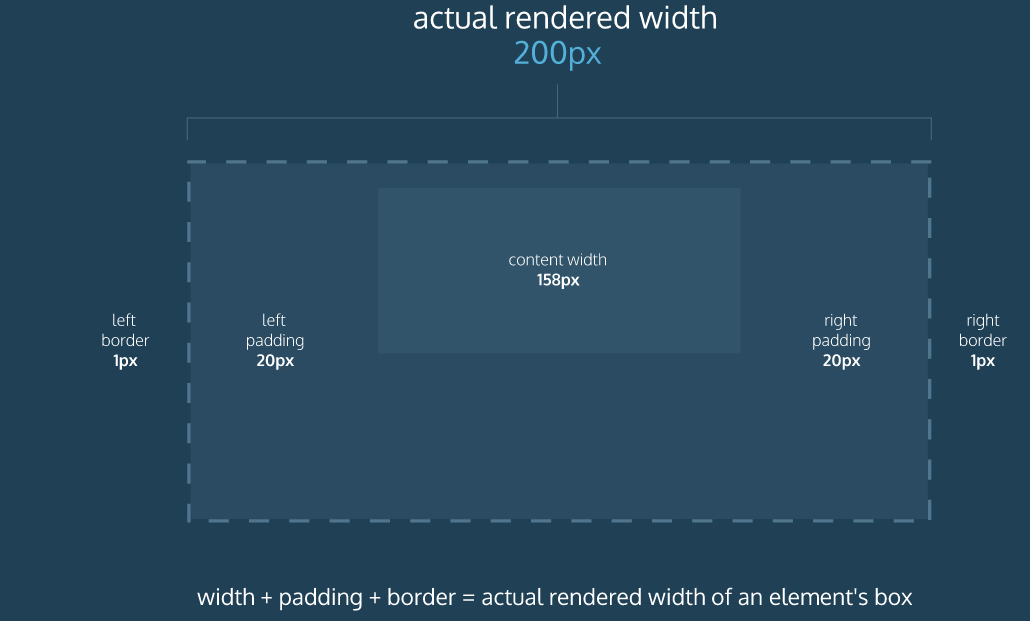
In this brief lesson, you’ll learn how to use a different technique that avoids this problem altogether.

Many properties in CSS have a default value and don’t have to be explicitly set in the stylesheet.

For example, the default font-weight of text is normal, but this property-value pair is not typically specified in a stylesheet.

The same can be said about the box model that browsers assume. In CSS, the box-sizing property controls the type of box model the browser should use when interpreting a web page.

The default value of this property is content-box. This is the same box model that is affected by border thickness and padding.



### Border Box

Fortunately, we can reset the entire box model and specify a new one: border-box.

\* {

box-sizing: border-box;

}

The code in the example above resets the box model to border-box for all HTML elements. This new box model avoids the dimensional issues that exist in the former box model you learned about.

In this box model, the height and width of the box will remain fixed. The border thickness and padding will be included inside of the box, which means the overall dimensions of the box do not change.

<h1>Hello World</h1>

\* {

box-sizing: border-box;

}

h1 {

border: 1px solid black;

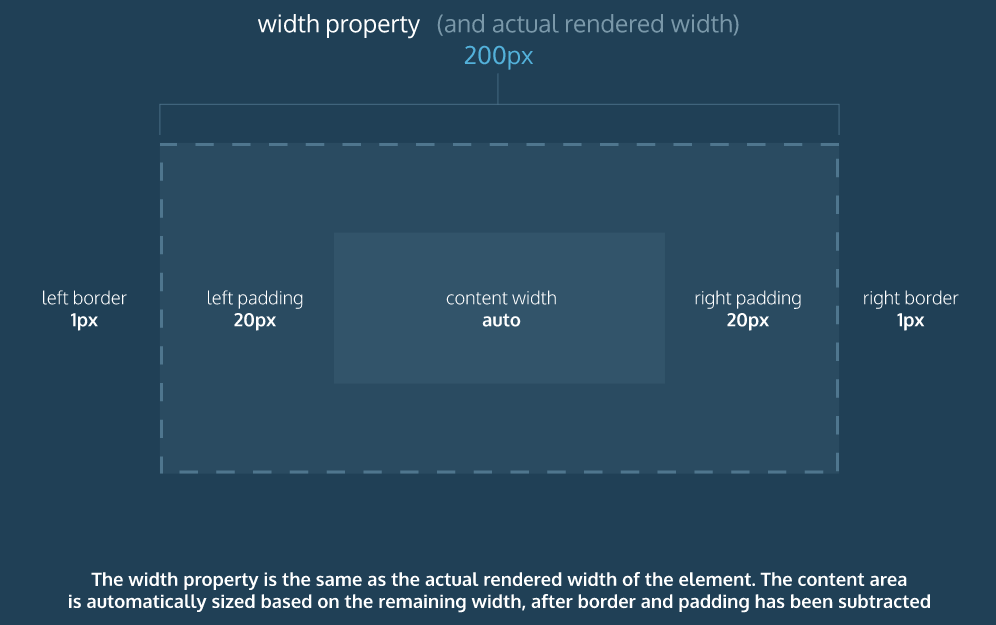
height: 200px;

width: 300px;

padding: 10px;

}

In the example above, the height of the box would remain at 200 pixels and the width would remain at 300 pixels. The border thickness and padding would remain entirely *inside* of the box.



Now that you know about the new box model, let’s actually implement it in the browser.

\* {

box-sizing: border-box;

}

It’s that simple! In the example above, the universal selector (\*) targets all elements on the web page and sets their box model to the border-box model.

## Flow of HTML

A browser will render the elements of an HTML document that has no CSS from left to right, top to bottom, in the same order as they exist in the document. This is called the *flow* of elements in HTML.

In addition to the properties that it provides to style HTML elements, CSS includes properties that change how a browser *positions* elements. These properties specify where an element is located on a page, if the element can share lines with other elements, and other related attributes.

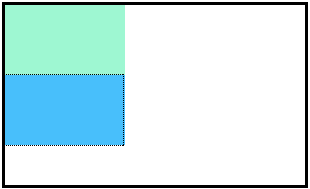
In this lesson, you will learn five properties for adjusting the position of HTML elements in the browser:

* position
* display
* z-index
* float
* clear

Each of these properties will allow us to position and view elements on a web page. They can be used in conjunction with any other styling properties you may know.

### Position

Take a look at the *block-level* elements in the image below:



Block-level elements like these boxes create a *block* the full width of their parent elements, and they prevent other elements from appearing in the same horizontal space. The boxes in the image above were created with the following CSS:

.boxes {

width: 120px;

height: 70px;

}

and the following HTML:

<div class="boxes"></div>

<div class="boxes"></div>

Notice the block-level elements in the image above take up their own line of space and therefore don’t overlap each other. In the browser to the right you can see block-level elements also consistently appear on the left side of the browser. This is the default *position* for block-level elements.

The default position of an element can be changed by setting its position property. The position property can take one of four values:

1. static - the default value (it does not need to be specified)
2. relative
3. absolute
4. fixed

In the next few exercises, you’ll learn about the values in items 2, 3, and 4 above. For now, it’s important to understand that if you favor the default position of an HTML element, you don’t need to set its position property.

#### Position Relative

One way to modify the default position of an element is by setting its position property to relative.

This value allows you to position an element *relative* to its default static position on the web page.

.box-bottom {

background-color: DeepSkyBlue;

position: relative;

}

Although the code in the example above instructs the browser to expect a relative positioning of the div, it does not specify where the div should be positioned on the page.

.box-bottom {

background-color: DeepSkyBlue;

position: relative;

top: 20px;

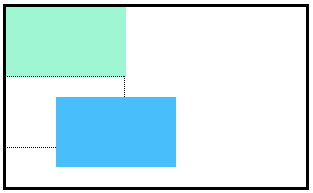
left: 50px;

}

In the example above, the <div> has been positioned using two of the four *offset properties*. The valid offset properties are:

1. top - moves the element down.
2. bottom - moves the element up.
3. left - moves the element right.
4. right - moves the element left.

In the example above, the <div> will be moved down 20 pixels and to the right 50 pixels from its default static position. The image below displays the new position of the box. The dotted line represents where the statically positioned (default) box was positioned.



Units for offset properties can be specified in pixels, ems, or percentages. Note that offset properties will not work if the value of the element’s position property is the default static.

#### Postion Absolute

Another way of modifying the position of an element is by setting its position to absolute.

When an element’s position is set to absolute all other elements on the page will *ignore* the element and act like it is not present on the page. The element will be positioned relative to its closest positioned parent element.

.box-bottom {

background-color: DeepSkyBlue;

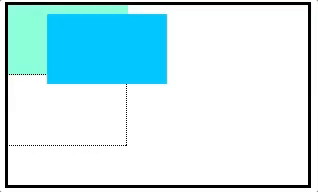
position: absolute;

top: 20px;

left: 50px;

}

In the example above, the .box-bottom <div> will be moved down and right from the top left corner of the view. If offset properties weren’t specified, the top box would be entirely covered by the bottom box. Take a look at the gif below:



The bottom box in this image (colored blue) is displaced from the top left corner of its container. It is 20 pixels lower and 50 pixels to the right of the top box.

In the next exercise, we will compare the scrolling of absolute elements with fixed elements.

#### Position Fixed

When an element’s position is set to absolute, as in the last exercise, the element will scroll with the rest of the document when a user scrolls.

We can *fix* an element to a specific position on the page (regardless of user scrolling) by setting its position to fixed.

.box-bottom {

background-color: DeepSkyBlue;

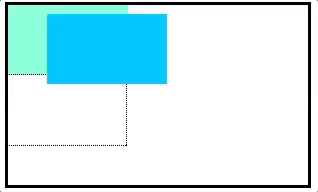
position: fixed;

top: 20px;

left: 50px;

}

In the example above, the .box-bottom <div> will remain fixed to its position no matter where the user scrolls on the page, like in the image below:



This technique is often used for navigation bars on a web page.

### Z Index

When boxes on a web page have a combination of different positions, the boxes (and therefore, their content) can overlap with each other, making the content difficult to read or consume.

.box-top {

background-color: Aquamarine;

}

.box-bottom {

background-color: DeepSkyBlue;

position: absolute;

top: 20px;

left: 50px;

}

In the example above, the .box-bottom <div> ignores the .box-top <div> and overlaps it as a user scrolls.

The z-index property controls how far “back” or how far “forward” an element should appear on the web page when elements overlap. This can be thought of the *depth* of elements, with deeper elements appearing behind shallower elements.

The z-index property accepts integer values. Depending on their values, the integers instruct the browser on the order in which elements should be displayed on the web page.

.box-top {

background-color: Aquamarine;

position: relative;

z-index: 2;

}

.box-bottom {

background-color: DeepSkyBlue;

position: absolute;

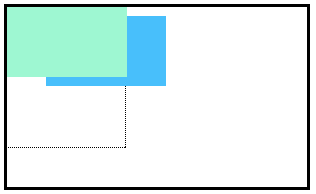
top: 20px;

left: 50px;

z-index: 1;

}

In the example above, we set the .box-top position to relative and the z-index to 2. We changed position to relative, because the z-index property does *not* work on static elements. The z-index of 2 moves the .box-top element forward, because it is greater than the .box-bottom z-index, 1. See the example image below:



In the image above, you can see the top box is moved in front of the bottom box.

### Display

Every HTML element has a default display value that dictates if it can share horizontal space with other elements. Some elements fill the entire browser from left to right regardless of the size of their content. Other elements only take up as much horizontal space as their content requires and can be directly next to other elements.

In this lesson, we’ll cover three values for the display property: inline, block, and inline-block.

#### Inline Display

The default display for some tags, such as <em>, <strong>, and <a>, is called inline. Inline elements have a box that wraps tightly around their content, only taking up the amount of space necessary to display their content and not requiring a new line after each element. The height and width of these elements cannot be specified in the CSS document. For example, the text of an anchor tag (<a>) will, by default, be displayed on the same line as the surrounding text, and it will only be as wide as necessary to contain its content. inline elements cannot be altered in size with the height or width CSS properties.

To learn more about <em>inline</em> elements, read <a href="#">MDN documentation</a>.

In the example above, the <em> element is inline, because it displays its content on the same line as the content surrounding it, including the anchor tag. This example will display:

To learn more about inline elements, read [MDN documentation](https://developer.mozilla.org/en-US/docs/Web/HTML/Inline_elements).

The CSS display property provides the ability to make any element an inline element. This includes elements that are not inline by default such as paragraphs, divs, and headings.

h1 {

display: inline;

}

The CSS in the example above will change the display of all <h1> elements to inline. The browser will render <h1> elements on the same line as other inline elements immediately before or after them (if there are any).

#### Block Display

Some elements are not displayed in the same line as the content around them. These are called block-level elements. These elements fill the entire width of the page by default, but their width property can also be set. Unless otherwise specified, they are the height necessary to accommodate their content.

Elements that are block-level by default include all levels of heading elements (<h1> through <h6>), <p>, <div> and <footer>. For a complete list of block level elements, visit [the MDN documentation](https://developer.mozilla.org/en-US/docs/Web/HTML/Block-level_elements).

strong {

display: block;

}

In the example above, all <strong> elements will be displayed on their own line, with no content directly on either side of them even though their contents may not fill the width of most computer screens.

#### Inline-Block

The third value for the display property is inline-block. Inline-block display combines features of both inline and block elements. Inline-block elements can appear next to each other and we can specify their dimensions using the width and height properties. Images are the best example of default inline-block elements.

For example, <div>s in the CSS below will be displayed on the same line and with the specified dimensions:

<div class="rectangle">

<p>I’m a rectangle!</p>

</div>

<div class="rectangle">

<p>So am I!</p>

</div>

<div class="rectangle">

<p>Me three!</p>

</div>

.rectangle {

display: inline-block;

width: 200px;

height: 300px;

}

In the example above, there are three rectangular divs that each contain a paragraph of text. The .rectangle <div>s will all appear inline (provided there is enough space from left to right) with a width of 200 pixels and height of 300 pixels, even though the text inside of them may not require 200 pixels by 300 pixels of space.

### Float

So far, you’ve learned how to specify the exact position of an element using offset properties. If you’re simply interested in moving an element as far left or as far right as possible on the page, you can use the float property.

The float property can be set to one of two values:

1. left - this value will move, or float, elements as far left as possible.
2. right - this value will move elements as far right as possible.

.boxes {

width: 120px;

height: 70px;

}

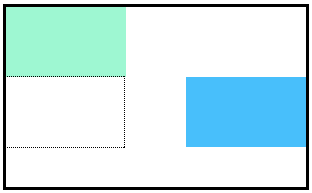
.box-bottom {

background-color: DeepSkyBlue;

float: right;

}

In the example above, we float the .box-bottom element to the right. This works for static and relative positioned elements. See the result of the code below:



Floated elements must have a width specified, as in the example above. Otherwise, the element will assume the full width of its containing element, and changing the float value will not yield any visible results.

### Clear

The float property can also be used to float multiple elements at once. However, when multiple floated elements have different heights, it can affect their layout on the page. Specifically, elements can “bump” into each other and not allow other elements to properly move to the left or right.

The clear property specifies how elements should behave when they bump into each other on the page. It can take on one of the following values:

1. left — the left side of the element will not touch any other element within the same containing element.
2. right — the right side of the element will not touch any other element within the same containing element.
3. both — neither side of the element will touch any other element within the same containing element.
4. none — the element can touch either side.

div {

width: 200px;

float: left;

}

div.special {

clear: left;

}

In the example above, all <div>s on the page are floated to the left side. The element with class special did not move all the way to the left because a taller <div> blocked its positioning. By setting its clear property to left, the special <div> will be moved all the way to the left side of the page.

## Color

CSS supports a wide variety of colors. These include *named colors*, like blue, black, and LimeGreen, along with colors described by a numeric value. Using a numeric system allows us to take advantage of the whole spectrum of colors that browsers support. In this lesson, we’re going to explore all the color options CSS offers.

Colors in CSS can be described in three different ways:

* *Named colors* — English words that describe colors, also called *keyword colors*
* *RGB* — numeric values that describe a mix of red, green, and blue
* *HSL* — numeric values that describe a mix of hue, saturation, and lightness

We’ll learn about and explore the benefits of each of these in depth. Using only named colors, you may feel like you’re picking labeled crayons out of a box. By the end of this lesson, you’ll feel like a painter mixing paints on a palette.

Before discussing the specifics of color, it’s important to make two distinctions about color. Color can affect the following design aspects:

1. The foreground color
2. The background color

Foreground color is the color that an element appears in. For example, when a heading is styled to appear green, the *foreground color* of the heading has been styled.

Conversely, when a heading is styled so that its background appears yellow, the *background color* of the heading has been styled

In CSS, these two design aspects can be styled with the following two properties:

1. color - this property styles an element’s foreground color.
2. background-color - this property styles an element’s background color.

h1 {

color: Red;

background-color: Blue;

}

In the example above, the text of the heading will appear in red, and the background of the heading will appear blue

### Hexadecimal colors

One syntax that we can use to specify colors is called hexadecimal. Colors specified using this system are called hex colors. A hex color begins with a hash character (#) which is followed by three or six characters. The characters represent values for red, blue and green.

DarkSeaGreen: #8FBC8F

Sienna: #A0522D

SaddleBrown: #8B4513

Brown: #A52A2A

Black: #000000 or #000

White: #FFFFFF or #FFF

Aqua: #00FFFF or #0FF

In the example above, you may notice that there are both letters and numbers in the values. This is because the hexadecimal number system has 16 digits (0-15) instead of 10 (0-9) like you are used to. To represent 10-15, we use A-F. [Here](https://developer.mozilla.org/en-US/docs/Web/CSS/color_value) is a list of many different colors and their hex values.

Notice that Black, White, and Aqua are all represented with both three characters and six characters. This can be done with hex colors whose number pairs are the same characters. In the example above, Aqua can be represented as #0FF because both of the first two characters are 0 and the second and third pairs of characters are both Fs. Keep in mind that all three character hex colors can be represented with six characters (by repeating each character twice) but the same is not true in reverse.

You can include hex colors just as you would include named colors: background-color: #9932cc;.

### RGB Colors

There is another syntax for representing RGB values that uses decimal numbers. It looks like this:

h1 {

color: rgb(23, 45, 23);

}

Here, each of the three values represents a color component, and each can have a decimal number value from 0 to 255. The first number represents the amount of red, the second is green, and the third is blue. These colors are exactly the same as hex, but with a different syntax and a different number system.

In general, hex and decimal color representations are equivalent. Which you choose is a matter of personal taste. That said, it’s good to choose one and be consistent throughout your CSS, because it’s easier to compare hex to hex and decimal to decimal.

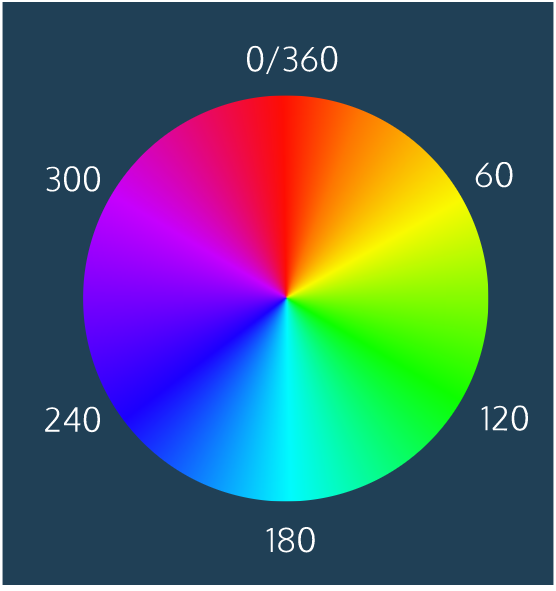
### Hue Saturation Lightness

The RGB color scheme is convenient because it’s very close to how computers represent colors internally. There’s another equally powerful system in CSS called the hue-saturation-lightness color scheme, abbreviated as *HSL*.

The syntax for HSL is similar to the decimal form of RGB, though it differs in important ways. The first number represents the degree of the hue, and can be between 0 and 360. The second and third numbers are percentages representing saturation and lightness respectively. Here is an example:

color: hsl(120, 60%, 70%);

*Hue* is the first number. It refers to an angle on a color wheel. Red is 0 degrees, Green is 120 degrees, Blue is 240 degrees, and then back to Red at 360. You can see an example of a color wheel below:



*Saturation* refers to the intensity or purity of the color. If you imagine a line segment drawn from the center of the color wheel to the perimeter, the saturation is a point on that line segment. If you spin that line segment to different angles, you’ll see how that saturation looks for different hues. The saturation increases towards 100% as the point gets closer to the edge (the color becomes more rich). The saturation decreases towards 0% as the point gets closer to the center (the color becomes more gray).

*Lightness* refers to how light or dark the color is. Halfway, or 50%, is normal lightness. Imagine a sliding dimmer on a light switch that starts halfway. Sliding the dimmer up towards 100% makes the color lighter, closer to white. Sliding the dimmer down towards 0% makes the color darker, closer to black.

HSL is convenient for adjusting colors. In RGB, making the color a little darker may affect all three color components. In HSL, that’s as easy as changing the lightness value. HSL is also useful for making a set of colors that work well together by selecting various colors that have the same lightness and saturation but different hues.

### Opacity and Alpha

All of the colors we’ve seen so far have been opaque, or non-transparent. When we overlap two opaque elements, nothing from the bottom element shows through the top element. In this exercise, we’ll change the *opacity*, or the amount of transparency, of some colors so that some or all of the bottom elements are visible through a covering element.

To use opacity in the HSL color scheme, use hsla instead of hsl, and four values instead of three. For example:

color: hsla(34, 100%, 50%, 0.1);

The first three values work the same as hsl. The fourth value (which we have not seen before) is the *alpha*. This last value is sometimes called the opacity.

Alpha is a decimal number from zero to one. If alpha is zero, the color will be completely transparent. If alpha is one, the color will be opaque. The value for half transparent would be 0.5.

You can think of the alpha value as, “the amount of the background to mix with the foreground”. When a color’s alpha is below one, any color behind it will be blended in. The blending happens for each pixel; no blurring occurs.

The RGB color scheme has a similar syntax for opacity, rgba. Again, the first three values work the same as rgb and the last value is the alpha. Here’s an example:

color: rgba(234, 45, 98, 0.33);

Alpha can only be used with HSL and RGB colors; we cannot add the alpha value to color: green color: #FFFFF.

There is, however, a named color keyword for zero opacity, transparent. It’s equivalent to rgba(0, 0, 0, 0). It’s used like any other color keyword:

color: transparent;

## Typography

### Font Family

If you’ve ever used a formatted word processor, chances are that you probably also used a feature that allowed you change the “type of font” you were typing in. The phrase “type of font” refers to the technical term [typeface](https://en.wikipedia.org/wiki/Typeface), or font family.

To change the typeface of text on your web page, you can use the font-family property.

h1 {

font-family: Garamond;

}

In the example above, the font family for all main heading elements has been set to Garamond.

When setting typefaces on a web page, keep the following points in mind:

1. The font specified in a stylesheet must be installed on a user’s computer in order for that font to display when a user visit the web page. We’ll learn how to work around this issue in a later exercise.
2. You’ve probably noticed that we haven’t been specifying a typeface in previous exercises of this course. How exactly does the browser know what typeface to use when displaying the web page? The default typeface for all most browsers is [Times New Roman](https://en.wikipedia.org/wiki/Times_New_Roman). You may be familiar with this typeface if you have ever used a formatted word processor.
3. It’s a good practice to limit the number of typefaces used on a web page to 2 or 3.
4. When the name of a typeface consists of more than one word, it must be enclosed in double quotes (otherwise it will not be recognized), like so:

h1 {

font-family: "Courier New";

}

### Font Weight

You’ve probably noticed **bold** text in websites you use, especially in news or text-heavy sites. It’s common to bold important headings or keywords. In CSS, we can style bold text with the font-weight property.

If we want to bold text in a web page, we can set the font-weight to bold.

p {

font-weight: bold;

}

If we want to ensure that text is not bold, we can set the font-weight to normal.

p {

font-weight: normal;

}

By default, the font-weight of most text elements is set to normal. Some elements, like headers, have built-in bold styling. A good approach is to check to see if the text element has any default styling, and use the font-weight property accordingly.

The font-weight property can also be assigned a number value to style text on a numeric scale ranging from 100 to 900. Valid values are multiples of 100 within this range such as 200 or 500.

When using numeric weights, there are a number of default font weights that we can use:

1. 400 is the default font-weight of most text.
2. 700 signifies a bold font-weight.
3. 300 signifies a light font-weight.

Let’s take a look at an example of how numeric fonts are used.

header {

font-weight: 800;

}

footer {

font-weight: 200;

}

Here, the header would appear as a deep bold, while the footer would appear rather light.

It’s important to note that not all fonts can be assigned a numeric font-weight. You can look up the font you are using to see which font-weight values are available.

### Font Style

You can also italicize text with the font-style property.

h3 {

font-style: italic;

}

The italic value causes text to appear in italics. The font-style property also has a normal value which is the default.

### Word Spacing

You can also increase the spacing between words in a body of text, technically known as word spacing.

To do so, you can use the word-spacing property:

h1 {

word-spacing: 0.3em;

}

The default amount of space between words is usually 0.25em. In the example above, the word spacing is set to 0.3em, which represents an increase of only .05em in word spacing.

It’s not common to increase the spacing between words, but it may help enhance the readability of bolded or enlarged text. Note, again, that the preferred unit is ems.

### Letter Spacing

You’ve learned how to increase the spacing between lines of text and words, but it’s possible to get even more detailed: increasing the spacing between individual letters.

The technical term for adjusting the spacing between letters is called tracking. Tracking can be adjusted with the letter-spacing property in CSS.

h1 {

letter-spacing: 0.3em;

}

Like word spacing, it’s not common to increase the tracking in text, but sometimes it enhances the readability of uppercase text.

### Text Transformation

Text can also be styled to appear in either all uppercase or lowercase with the text-transform property.

h1 {

text-transform: uppercase;

}

The code in the example above formats all <h1> elements to appear in uppercase, regardless of the case used for the heading within the HTML code. Alternatively, the lowercase value could be used to format text in all lowercase.

Since text can be directly typed in all uppercase or lowercase within an HTML file, what is the point of a CSS rule that allows you to format [letter case](https://en.wikipedia.org/wiki/Letter_case)?

Depending on the type of content a web page displays, it may make sense to always style a specific element in all uppercase or lowercase letters. For example, a website that reports breaking news may decide to format all <h1> heading elements such that they always appear in all uppercase, as in the example above. It would also avoid uppercase text in the HTML file, which could make code difficult to read.

### Text Alignment

No matter how much styling is applied to text (typeface, size, weight, etc.), text always appears on the left side of the browser.

To move, or align, text, we can use the text-align property.

h1 {

text-align: right;

}

The text-align property can be set to one of the following three values:

1. left - aligns text to the left hand side of the browser.
2. center - centers text.
3. right - aligns text to the right hand side of the browser.

Later in the course, you’ll learn exactly how the browser positions HTML elements by default, which will help you understand how the browser “aligns” text, since “align” is a relative term. For now, it’s enough to know that text can be moved to the left, center, or right side of the web page.

### Line Height Anatomy

Another property that we can set for text is line-height. This property modifies the leading of text.

The diagram to the right helps illustrate exactly what the terms “leading” and “line height” mean.



We often modify line-height to make text on a web page easier to read. When text is styled to appear larger, the vertical spacing between lines of text can decrease, creating text that is difficult to read, particularly in paragraphs.

We can use the line-height property to set how tall we want the line containing our text to be, regardless of the height of the text. Line heights can take one of several values:

1. A unitless number, such as 1.2. This number is an absolute value that will compute the line height as a ratio of the font size.
2. A number specified by unit, such as 12px. This number can be any valid CSS unit, such as pixels, percents, ems, or rems.

Generally, the unitless ratio value is the preferred method, since it is responsive and based exclusively on the current font size. In other words, if we change the font size, a unitless line-height would automatically readjust, whereas the pixel value would remain static.

p {

line-height: 1.4;

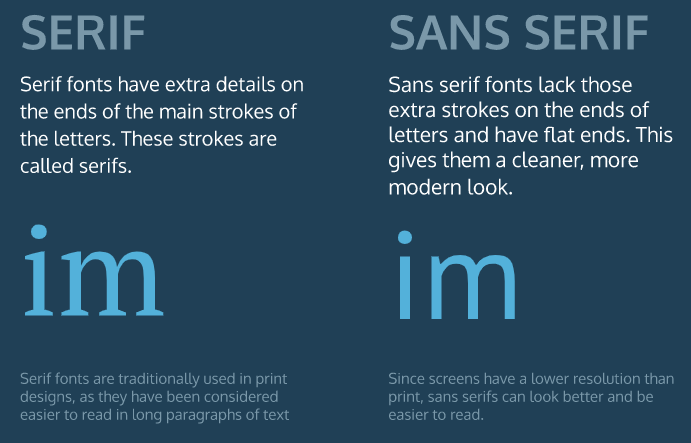
}

### Serif and Sans Serif

You’ve learned a lot of properties to modify text on a web page!

In the next exercise, you’ll set some text to be serif and some text to be sans-serif. What exactly do these words mean?

1. Serif — fonts that have extra details on the ends of each letter. Examples include fonts like Times New Roman or Georgia, among others.
2. Sans-Serif — fonts that do not have extra details on the ends of each letter. Instead, letters have straight, flat edges, like Arial or Helvetica.



### Fallback Fonts

What happens when a stylesheet requires a font that is not installed on a user’s computer? Most computers have a small set of typefaces pre-installed. This small set includes serif fonts like Times New Roman and sans-serif fonts like Arial.

These pre-installed fonts serve as *fallback fonts* if the stylesheet specifies a font which is not installed on a user’s computer.

To use fallback fonts, the following syntax is required:

h1 {

font-family: "Garamond", "Times", serif;

}

The CSS rule above says:

1. Use the Garamond font for all <h1> elements on the web page.
2. If Garamond is not available, use the Times font.
3. If Garamond and Times are not available, use any serif font pre-installed on the user’s computer.

The fonts specified after Garamond are the fallback fonts (Times, serif). Fallback fonts help ensure a consistent experience for the diverse audience of users that visit a site.

### Linking Fonts

With the number of fonts available with modern typography, it is unrealistic to expect users to have all fonts installed on their computers. New fonts are often centralized in directories made available for public use. We refer to these fonts as non-user fonts.

[Google Fonts](https://fonts.google.com/) is one such directory of thousands of open-source fonts, available for free use. Google Fonts gives us a way to retrieve the link for a single font, multiple fonts, or multiple fonts with the font-weight and font-style properties.

When we have the link to the font of our choice, we can add the font to the <head> section of the HTML document, using the <link> tag and the href.

Let’s take a look at a few examples:

1. A single linked font, using Droid Serif as an example:

<head>

<link href="https://fonts.googleapis.com/css?family=Droid+Serif" type="text/css" rel="stylesheet">

</head>

2. Multiple linked fonts, using the Droid Serif and Playfair Display fonts as an example:

<head>

<link href="https://fonts.googleapis.com/css?family=Droid+Serif|Playfair+Display" type="text/css" rel="stylesheet">

</head>

3. Multiple linked fonts, along with weights and styles. Here Droid Serif has font weights of 400, 700, and 700i, while Playfair Display has font weights of 400, 700, and 900i:

<head>

<link href="https://fonts.googleapis.com/css?family=Droid+Serif:400,700,700i|Playfair+Display:400,700,900i" rel="stylesheet">

</head>

Once a font is linked, we can create CSS selectors to target elements, just as we do with other fonts.

### Font-Face

There are other ways to link non-user fonts that don’t require the use of the <link> tag in the HTML document. CSS offers a way to import fonts directly into stylesheets with the @font-face property.

To load fonts with the @font-face property:

1. Instead of using the font’s link in the HTML document, enter the link into the URL bar in the browser.
2. The browser will load the CSS rules. You will need to focus on the rules that are directly labeled as /\* latin \*/. Some of the latin rules are on separate lines. You will need each of these.
3. Copy each of the CSS rules labeled latin, and paste the rules from the browser to the top of **style.css**.

It is important to stress the need to copy the @font-face rules to the top of the stylesheet for the font to load correctly in the project.

We can then use the fonts in the stylesheets as you would use any other font. Let’s practice loading an external font in our stylesheets using the @font-face property, and using the font to style our page.

While Google Fonts and other resources can broaden font selection, you may wish to use an entirely different font or abstain from using a font from an external service.

We can modify our @font-face rule to use local font files as well. We can supply the user with the desired font family and host it along with our site instead of depending on a different site.

@font-face {

font-family: "Roboto";

src: url(fonts/Roboto.woff2) format('woff2'),

url(fonts/Roboto.woff) format('woff'),

url(fonts/Roboto.tff) format('truetype');

}

Here, you’ll notice:

1. The main difference is the use of a relative filepath instead of a web URL.
2. We add a format for each file to specify which font to use. Different browsers support different font types, so providing multiple font file options will support more browsers.

As of now .woff2 appears to be the way of the future, due to greatly reduced file sizes and improved performance, but many browsers still don’t support it. There are lots of great sources to find fonts to use locally, such as [Font Squirrel](https://www.fontsquirrel.com/).

# Positioning crib sheet

Web developers should be familiar with:

position: static;

* default position
* positioned according to normal flow of HTML document
* offset properties (top, right, bottom, left) and z-index property have no effect

position: relative;

* positioned according to normal flow of the document (not taken out of the normal flow)
* will be affected by offset properties (top, right, bottom, left) and z-index
* if offset properties are used, the element will be offset in relation to its default position

position: absolute;

* element is taken out of the normal flow of the document
* positioned in relation to its nearest positioned parent element (grandparent, etc),
* if no positioned parent element exists, the absolutely positioned element will be positioned in relation to the <html> element
* will be affected by offset properties (top, right, bottom, left) and z-index

and

position: fixed;

* element is taken out of the normal flow of the document
* positioned in relation to the viewport of the browser
* will be affected by offset properties (top, right, bottom, left) and z-index

# Sizing Elements

Relative Measurements

Modern technology allows users to browse the Internet via multiple devices, such as desktop monitors, mobile phones, tablets, and more. Devices of different screen sizes, however, pose a problem for web developers: how can we ensure that a website is readable and visually appealing across all devices, regardless of screen size?

The answer: responsive design! Responsive design refers to the ability of a website to resize and reorganize its content based on:

1. The size of other content on the website.
2. The size of the screen the website is being viewed on.

In this lesson, we’ll size HTML content relative to other content on a website.

You’ve probably noticed the unit of [pixels](https://en.wikipedia.org/wiki/Pixel), or px, used in websites. Pixels are used to size content to exact dimensions. For example, if you want a div to be exactly 500 pixels wide and 100 pixels tall, then the unit of px can be used. Pixels, however, are fixed, [hard coded](https://www.google.com/search?q=hard+coding) values. When a screen size changes (like switching from landscape to portrait view on a phone), elements sized with pixels can appear too small, overflow the screen, or become completely illegible.

With CSS, you can avoid hard coded measurements and use relative measurements instead. Relative measurements offer an advantage over hard coded measurements, as they allow for the proportions of a website to remain intact regardless of screen size or layout.

## EM

Incorporating relative sizing starts by using units other than pixels. One unit of measurement you can use in CSS to create relatively-sized content is the *em*, written as em in CSS.

Historically, the em represented the width of a capital letter M in the typeface and size being used. That is no longer the case.

Today, the em represents the size of the base font being used. For example, if the base font of a browser is 16 pixels (which is normally the default size of text in a browser), then 1 em is equal to 16 pixels. 2 ems would equal 32 pixels, and so on.

Let’s take a look at two examples that show how em can be used in CSS.

.heading {

font-size: 2em;

}

In the example above, no base font has been specified, therefore the font size of the heading element will be set relative to the default font size of the browser. Assuming the default font size is 16 pixels, then the font size of the heading element will be 32 pixels.

.splash-section {

font-size: 18px;

}

.splash-section h1 {

font-size: 1.5em;

}

The example above shows how to use ems without relying on the default font size of the browser. Instead, a base font size (18px) is defined for all text within the splash-section element. The second CSS rule will set the font size of all h1 elements inside of splash-section relative to the base font of splash-section (18 pixels). The resulting font size of h1 elements will be 27 pixels.

## REM

The second relative unit of measurement in CSS is the *rem*, coded as rem.

Rem stands for *root em*. It acts similar to em, but instead of checking parent elements to size font, it checks the *root element*. The root element is the <html> tag.

Most browsers set the font size of <html> to 16 pixels, so by default rem measurements will be compared to that value. To set a different font size for the root element, you can add a CSS rule.

html {

font-size: 20px;

}

h1 {

font-size: 2rem;

}

In the example above, the font size of the root element, <html>, is set to 20 pixels. All subsequent rem measurements will now be compared to that value and the size of h1 elements in the example will be 40 pixels.

One advantage of using rems is that all elements are compared to the same font size value, making it easy to predict how large or small font will appear. If you are interested in sizing elements consistently across an entire website, the rem measurement is the best unit for the job. If you’re interested in sizing elements in comparison to other elements nearby, then the em unit would be better suited for the job.

## Percenteges

### height & width

To size non-text HTML elements relative to their parent elements on the page you can use *percentages*.

Percentages are often used to size box-model values, like width and height, padding, border, and margins. They can also be used to set positioning properties (top, bottom, left, right).

To start, let’s size the height and width of an element using percentages.

.main {

height: 300px;

width: 500px;

}

.main .subsection {

height: 50%;

width: 50%;

}

In the example above, .main and .subsection each represent divs. The .subsection div is nested within the .main div. Note that the dimensions of the parent div (.main) have been set to a height of 300 pixels and a width of 500 pixels.

When percentages are used, elements are sized relative to the dimensions of their parent element (also known as a container). Therefore, the dimensions of the .subsection div will be 150 pixels tall and 250 pixels wide. Be careful, a child element’s dimensions may be set erroneously if the dimensions of its parent element aren’t set first.

**Note:** Because the box model includes padding, borders, and margins, setting an element’s width to 100% may cause content to overflow its parent container. While tempting, 100% should only be used when content will not have padding, border, or margin.

### Padding & Margins

Percentages can also be used to set the padding and margin of elements.

When height and width are set using percentages, you learned that the dimensions of child elements are calculated based on the dimensions of the parent element.

When percentages are used to set padding and margin, however, they are calculated based only on the *width* of the parent element.

For example, when a property like margin-left is set using a percentage (say 50%), the element will be moved halfway to the right in the parent container (as opposed to the child element receiving a margin half of its parent’s margin).

Vertical padding and margin are also calculated based on the width of the parent. Why? Consider the following scenario:

1. A container div is defined, but its height is not set (meaning it’s flat).
2. The container then has a child element added within. The child element *does* have a set height. This causes the height of its parent container to stretch to that height.
3. The child element requires a change, and its height is modified. This causes the parent container’s height to also stretch to the new height. This cycle occurs endlessly whenever the child element’s height is changed!

In the scenario above, an unset height (the parent’s) results in a constantly changing height due to changes to the child element. This is why vertical padding and margin are based on the width of the parent, and not the height.

**Note:** When using relative sizing, ems and rems should be used to size text and dimensions on the page related to text size (i.e. padding around text). This creates a consistent layout based on text size. Otherwise, percentages should be used.

### Width Min & Max

Although relative measurements provide consistent layouts across devices of different screen sizes, elements on a website can lose their integrity when they become too small or large. You can limit how wide an element becomes with the following properties:

1. min-width — ensures a minimum width for an element.
2. max-width — ensures a maximum width for an element.

p {

min-width: 300px;

max-width: 600px;

}

In the example above, when the browser is resized, the width of paragraph elements will not fall below 300 pixels, nor will their width exceed 600 pixels.

When a browser window is narrowed or widened, text can become either very compressed or very spread out, making it difficult to read. These two properties ensure that content is legible by limiting the minimum and maximum widths.

**Note**: The unit of pixels is used to ensure hard limits on the dimensions of the element(s).

### min and max height

You can also limit the minimum and maximum *height* of an element.

1. min-height — ensures a minimum height for an element’s box.
2. max-height — ensures a maximum height for an element’s box.

p {

min-height: 150px;

max-height: 300px;

}

In the example above, the height of all paragraphs will not shrink below 150 pixels and the height will not exceed 300 pixels.

What will happen to the contents of an element if the max-height property is set too low for that element? It’s possible that content will overflow outside of the element, resulting in content that is not legible.

### Scaleing images and video

Many websites contain a variety of different media, like images and videos. When a website contains such media, it’s important to make sure that it is scaled proportionally so that users can correctly view it.

.container {

width: 50%;

height: 200px;

overflow: hidden;

}

.container img {

max-width: 100%;

height: auto;

display: block;

}

In the example above, .container represents a container div. It is set to a width of 50% (half of the browser’s width, in this example) and a height of 200 pixels. Setting overflow to hidden ensures that any content with dimensions larger than the container will be hidden from view.

The second CSS rule ensures that images scale with the width of the container. The height property is set to auto, meaning an image’s height will *automatically* scale proportionally with the width. Finally, the last line will display images as block level elements (rather than inline-block, their default state). This will prevent images from attempting to align with other content on the page (like text), which can add unintended margin to the images.

It’s worth memorizing the entire example above. It represents a *very common* design pattern used to scale images and videos proportionally.

**Note:** The example above scales the width of an image (or video) to the width of a container. If the image is larger than the container, the vertical portion of the image will overflow and will not display. To swap this behavior, you can set max-height to 100% and width to auto (essentially swapping the values). This will scale the *height* of the image with the height of the container instead. If the image is larger than the container, the horizontal portion of the image will overflow and not display.

Additional

Take a look at the images on the web page. Notice that they currently display incorrectly (too large). Let’s fix that.

First, in **style.css**, set the overflow property in .image-container to hidden. Run your code.

Take a look at the images once more. At this point, the images partially display. In reality, what we’ve done is constrain them to the dimensions of their container (.image-container). Any part of the image that overflows out of the container will be hidden from view. This will set us up to scale them proportionally.

Resize the width of the browser back and forth. Notice that the images expand and contract to show more (or less) of the image. Instead, let’s display the full image at all times.

In **style.css**, set the maximum width in .image-container img to 100%. This will ensure the full image is always displayed.

Great work! Take a look at the images on the web page again — they have been greatly improved!

Next, we’ll want to make sure the images automatically remain in proportion when the browser is resized.

In .image-container img, set the height to auto.

Finally, within the same CSS rule, set the display to block. This will instruct the images to behave as block-level elements and facilitate scaling (as opposed to their default inline behavior).

### Scaling Background images

Background images of HTML elements can also be scaled responsively using CSS properties.

body {

background-image: url('#');

background-repeat: no-repeat;

background-position: center;

background-size: cover;

}

In the example above, the first CSS declaration sets the background image (# is a placeholder for an image URL in this example). The second declaration instructs the CSS compiler to not repeat the image (by default, images will repeat). The third declaration centers the image within the element.

The final declaration, however, is the focus of the example above. It’s what scales the background image. The image will *cover* the entire background of the element, all while keeping the image in proportion. If the dimensions of the image exceed the dimensions of the container then only a portion of the image will display.

# Responsive Web Design

When someone visits a website, it’s possible they are viewing it on a phone, tablet, computer, or even a TV monitor. Because screen sizes can vary greatly across different devices, it’s important for websites to resize and reorganize their content to best fit screens of all sizes.

When a website doesn’t respond to different screen sizes, the website may look odd or become indecipherable on certain devices. This usually occurs on smaller screens, like phones. When a website responds to the size of the screen it’s viewed on, it’s called a *responsive* website.

Because websites can be displayed on thousands of different screen sizes, they must be able to respond to a change in screen size and adapt the content so that users can access it.

## Media Queries

CSS uses media queries to adapt a website’s content to different screen sizes. With media queries, CSS can detect the size of the current screen and apply different CSS styles depending on the width of the screen.

@media only screen and (max-width: 480px) {

body {

font-size: 12px;

}

}

The example above demonstrates how a media query is applied. The media query defines a rule for screens smaller than 480 pixels (approximately the width of many smartphones in [landscape](https://en.wikipedia.org/wiki/Page_orientation) orientation).

Let’s break this example down into its parts:

1. @media — This keyword begins a media query rule and instructs the CSS compiler on how to parse the rest of the rule.
2. only screen — Indicates what types of devices should use this rule. In early attempts to target different devices, CSS incorporated different media types (screen, print, handheld). The rationale was that by knowing the media type, the proper CSS rules could be applied. However, “handheld” and “screen” devices began to occupy a much wider range of sizes and having only one CSS rule per media device was not sufficient. screen is the media type always used for displaying content, no matter the type of device. The only keyword is added to indicate that this rule only applies to one media type (screen).
3. and (max-width : 480px) — This part of the rule is called a media feature, and instructs the CSS compiler to apply the CSS styles to devices with a width of 480 pixels or smaller. Media features are the conditions that must be met in order to render the CSS within a media query.
4. CSS rules are nested inside of the media query’s curly braces. The rules will be applied when the media query is met. In the example above, the text in the body element is set to a font-size of 12px when the user’s screen is less than 480px.

## Range

Specific screen sizes can be targeted by setting multiple width and height media features. min-width and min-height are used to set the minimum width and minimum height, respectively. Conversely, max-width and max-height set the maximum width and maximum height, respectively.

By using multiple widths and heights, a range can be set for a media query.

@media only screen and (min-width: 320px) and (max-width: 480px) {

/\* ruleset for 320px - 480px \*/

}

The example above would apply its CSS rules only when the screen size is between 320 pixels and 480 pixels. Notice the use of a second and keyword after the min-width media feature. This allows us to chain two requirements together.

The example above can be written using two separate rules as well:

@media only screen and (min-width: 320px) {

/\* ruleset for >= 320px \*/

}

@media only screen and (min-width: 480px) {

/\* ruleset for >= 480px \*/

}

The first media query in the example above will apply CSS rules when the size of the screen meets or exceeds 320 pixels. The second media query will then apply CSS rules when the size of the screen meets or exceeds 480 pixels, meaning that it can override CSS rules present in the first media query or apply additional CSS rules that are not already present in the first.

Both examples above are valid, and it is likely that you will see both patterns used when reading another developer’s code.

## Dots Per Inch (DPI)

Another media feature we can target is screen resolution. Many times we will want to supply higher quality media (images, video, etc.) only to users with screens that can support high resolution media. Targeting screen resolution also helps users avoid downloading high resolution (large file size) images that their screen may not be able to properly display.

To target by resolution, we can use the min-resolution and max-resolution media features. These media features accept a resolution value in either dots per inch (dpi) or dots per centimeter (dpc). Learn more about resolution measurements [here](https://en.wikipedia.org/wiki/Dots_per_inch).

@media only screen and (min-resolution: 300dpi) {

/\* CSS for high resolution screens \*/

}

The media query in the example above targets high resolution screens by making sure the screen resolution is at least 300 dots per inch. If the screen resolution query is met, then we can use CSS to display high resolution images and other media.

## And Operator

In previous exercises, we chained multiple media features of the same type in one media query by using the and operator. It allowed us to create a range by using min-width and max-width in the same media query.

The and operator can be used to require multiple media features. Therefore, we can use the and operator to require both a max-width of 480px *and* to have a min-resolution of 300dpi.

For example:

@media only screen and (max-width: 480px) and (min-resolution: 300dpi) {

/\* CSS ruleset \*/

}

By placing the and operator between the two media features, the browser will require both media features to be true before it renders the CSS within the media query. The and operator can be used to chain as many media features as necessary.

## Comma Seperated List

If only one of multiple media features in a media query must be met, media features can be separated in a comma separated list.

For example, if we needed to apply a style when only one of the below is true:

* The screen is more than 480 pixels wide
* The screen is in landscape mode

We could write:

@media only screen and (min-width: 480px), (orientation: landscape) {

/\* CSS ruleset \*/

}

In the example above, we used a comma (,) to separate multiple rules. The example above requires only one of the media features to be true for its CSS to apply.

Note that the second media feature is orientation. The orientation media feature detects if the page has more width than height. If a page is wider, it’s considered landscape, and if a page is taller, it’s considered portrait.

## BreakPoints

We know how to use media queries to apply CSS rules based on screen size and resolution, but how do we determine what queries to set?

The points at which media queries are set are called breakpoints. Breakpoints are the screen sizes at which your web page does not appear properly. For example, if we want to target tablets that are in landscape orientation, we can create the following breakpoint:

@media only screen and (min-width: 768px) and (max-width: 1024px) and (orientation: landscape) {

/\* CSS ruleset \*/

}

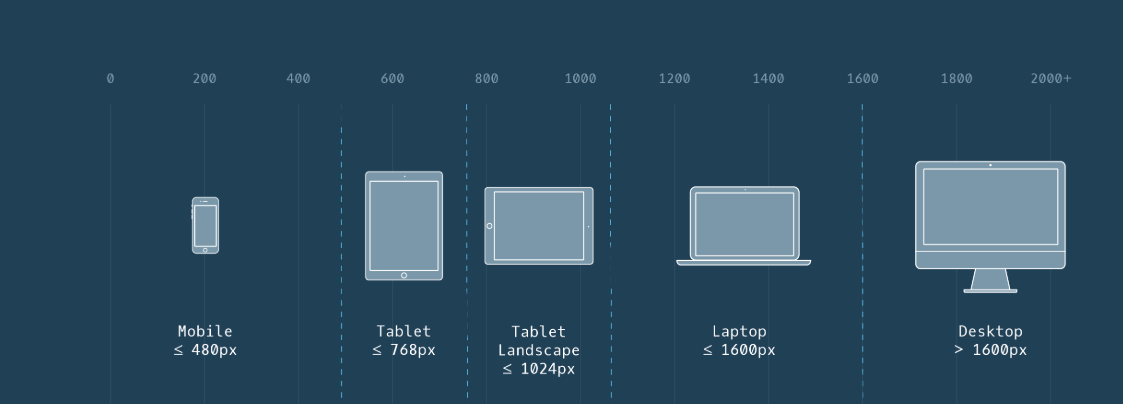
The example above creates a screen size range the size of a tablet in landscape mode and also identifies the orientation.

However, setting breakpoints for every device imaginable would be incredibly difficult because there are many devices of differing shapes and sizes. In addition, new devices are released with new screen sizes every year.

Rather than set breakpoints based on specific devices, the best practice is to resize your browser to view where the website naturally breaks based on its content. The dimensions at which the layout breaks or looks odd become your media query breakpoints. Within those breakpoints, we can adjust the CSS to make the page resize and reorganize.

By observing the dimensions at which a website naturally breaks, you can set media query breakpoints that create the best possible user experience on a project by project basis, rather than forcing every project to fit a certain screen size. Different projects have different needs, and creating a responsive design should be no different.

Check out [this](https://s3.amazonaws.com/codecademy-content/courses/freelance-1/unit-5/screen-sizes.png) list of breakpoints by device widths. Use it as a reference of screen widths to test your website to make certain it looks great across a variety of devices.



# FlexBox

CSS provides many tools and properties that you can use to position elements on a webpage. Codecademy’s lessons on the box model and CSS display introduce a couple of these techniques.

In this lesson, you will learn about *flexbox* or Flexible Box Layout, a new tool developed for CSS3 that greatly simplifies how to position elements. While flexbox is not meant to lay out entire pages, it is useful for positioning elements, whether individually or in groups.

There are two important components to a flexbox layout: *flex containers* and *flex items*. A flex container is an element on a page that contains flex items. All direct child elements of a flex container are flex items. This distinction is important because some of the properties you will learn in this lesson apply to flex containers while others apply to flex items.

To designate an element as a flex container, set the element’s display property to flex or inline-flex. Once an item is a flex container, there are several properties we can use to specify how its children behave. In this lesson we will cover these properties:

1. justify-content
2. align-items
3. flex-grow
4. flex-shrink
5. flex-basis
6. flex
7. flex-wrap
8. align-content
9. flex-direction
10. flex-flow

Flexbox is an elegant tool that makes it easy to address positioning issues that may have been difficult before. Let’s get started!

## Display:flex

Any element can be a flex container. Flex containers are helpful tools for creating websites that respond to changes in screen sizes. Child elements of flex containers will change size and location in response to the size and position of their parent container.

For an element to become a flex container, its display property must be set to flex.

div.container {

display: flex;

}

In the example above, all divs with the class container are flex containers. If they have children, the children are flex items. A div with the declaration display: flex; will remain block level — no other elements will appear on the same line as it.

However, it will change the behavior of its child elements. Child elements will not begin on new lines. In the exercises that follow, we will cover how the flex display property impacts the positioning of child elements.

## Display:inline-flex

In the previous exercise, you might have observed that when we gave a div — a block level element — the display value of flex that it remained a block level element. What if we want multiple flex containers to display inline with each other?

If we didn’t want div elements to be block-level elements, we would use display: inline. Flexbox, however, provides the inline-flex value for the display attribute, which allows us to create flex containers that are also inline elements.

<div class="container">

<p>I’m inside of a flex container!</p>

<p>A flex container’s children are flex items!</p>

</div>

<div class="container">

<p>I’m also a flex item!</p>

<p>Me too!</p>

</div>

.container {

width: 200px;

height: 200px;

display: inline-flex;

}

In the example above, there are two container divs. Without a width, each div would stretch the entire width of the page. The paragraphs within each div would also display on top of each other because paragraphs are block-level elements.

When we change the value of the display property to inline-flex, the divs will display inline with each other if the page is wide enough. As we progress through this lesson, we will cover in more detail how flex items are displayed.

Notice that in the example above, the size of the flex container is set. Currently, the size of the parent container will override the size of its child elements. If the parent element is too small, the flex items will shrink to accommodate the parent container’s size. We’ll explain why in a later exercise.

<div class="container">

<div class="child">

<h1>1</h1>

</div>

<div class="child">

<h1>2</h1>

</div>

</div>

.container {

width: 200px;

}

.child {

display: inline-flex;

width: 150px;

height: auto;

}

In the example above, the .child divs will take up more width (300 pixels) than the container div allows (200 pixels). The .child divs will shrink to accommodate the container’s size. In later exercises, we will explore several ways to handle this.

## Justify-content

In previous exercises, when we changed the display value of parent containers to flex or inline-flex, all of the child elements (flex items) moved toward the upper left corner of the parent container. This is the default behavior of flex containers and their children. We can specify how flex items spread out from left to right, along the *main axis*. We will learn more about axes in a later exercise.

To position the items from left to right, we use a property called justify-content.

.container {

display: flex;

justify-content: flex-end;

}

In the example above, we set the value of justify-content to flex-end. This will cause all of the flex items to shift to the right side of the flex container.

There are five values for the justify-content property:

1. flex-start — all items will be positioned in order starting, from the left of the parent container, with no extra space between or before them.
2. flex-end — all items will be positioned in order, with the last item starting on the right side of the parent container, with no extra space between or after them.
3. center — all items will be positioned in order, in the center of the parent container with no extra space before, between, or after them.
4. space-around — items will be positioned with equal space before and after each item, resulting in double the space between elements.
5. space-between — items will be positioned with equal space between them, but no extra space before the first or after the last elements.

In the definitions above, “no extra space” means that margins and borders will be respected, but no more space (than is specified in the style rule for the particular element) will be added between elements. The size of each individual flex item is not changed by this property.

## Align-items

In the previous exercise, you learned how to justify the content of a flex container from left to right across the page. It is also possible to align flex items vertically within the container. The align-items property makes it possible to space flex items vertically.

.container {

align-items: baseline;

}

In the example above, the align-items property is set to baseline. This means that the baseline of the content of each item will be aligned.

There are five values we can use for the align-items property:

1. flex-start — all elements will be positioned at the top of the parent container.
2. flex-end — all elements will be positioned at the bottom of the parent container.
3. center — the center of all elements will be positioned halfway between the top and bottom of the parent container.
4. baseline — the bottom of the content of all items will be aligned with each other.
5. stretch — if possible, the items will stretch from top to bottom of the container (this is the default value; elements with a specified height will not stretch; elements with a minimum height or no height specified will stretch).

These five values tell the elements how to behave along the *cross axis* of the parent container. In these examples, the cross axis stretches from top to bottom of the container. We’ll learn more about this in a future exercise.

You might be unfamiliar with the min-height and max-height properties, but you have used height and width before. min-height, max-height, min-width, and max-width are properties that ensure an element is at least a certain size or at most a certain size. You’ll see how these become useful as you move throughout this lesson.

## Flex Grow

In Exercise 3, we learned that all flex items shrink proportionally when the flex container is too small. However, if the parent container is larger than necessary then the flex items will not stretch by default. The flex-grow property allows us to specify if items should grow to fill a container and also which items should grow proportionally more or less than others.

<div class="container">

<div class="side">

<h1>I’m on the side of the flex container!</h1>

</div>

<div class="center">

<h1>I'm in the center of the flex container!</h1>

</div>

<div class=”side”>

<h1>I'm on the other side of the flex container!</h1>

</div>

</div>

.container {

display: flex;

}

.side {

width: 100px;

flex-grow: 1;

}

.center {

width: 100px;

flex-grow: 2;

}

In the example above, the .container div has a display value of flex, so its three child divs will be positioned next to each other. If there is additional space in the .container div (in this case, if it is wider than 300 pixels), the flex items will grow to fill it. The .center div will stretch twice as much as the .side divs. For example, if there were 60 additional pixels of space, the center div would absorb 30 pixels and the side divs would absorb 15 pixels each.

If a max-width is set for an element, it will not grow larger than that even if there is more space for it to absorb.

All of the previous properties we have learned are declared on flex containers, or the parent elements. This property — flex-grow — is the first we have learned that is declared on flex items.

## Flex-shrink

Just as the flex-grow property proportionally stretches flex items, the flex-shrink property can be used to specify which elements will shrink and in what proportions.

You may have noticed in earlier exercises that flex items shrank when the flex container was too small, even though we had not declared the property. This is because the default value of flex-shrink is 1. However, flex items do not grow unless the flex-grow property is declared because the default value of flex-grow is 0.

<div class="container">

<div class="side">

<h1>I'm on the side of the flex container!</h1>

</div>

<div class="center">

<h1>I'm in the center of the flex container!</h1>

</div>

<div class="side">

<h1>I'm on the other side of the flex container!</h1>

</div>

</div>

.container {

display: flex;

}

.side {

width: 100px;

flex-shrink: 1;

}

.center {

width: 100px;

flex-shrink: 2;

}

In the example above, the .center div will shrink twice as much as the .side divs if the .container div is too small to fit the elements within it. If the content is 60 pixels too large for the flex container that surrounds it, the .center div will shrink by 30 pixels and the outer divs will shrink by 15 pixels each. Margins are unaffected by flex-grow and flex-shrink.

Keep in mind, minimum and maximum widths will take precedence over flex-grow and flex-shrink. As with flex-grow, flex-shrink will only be employed if the parent container is too small or the browser is adjusted

## Flex-basis

In the previous two exercises, the dimensions of the divs were determined by heights and widths set with CSS. Another way of specifying the width of a flex item is with the flex-basis property. flex-basis allows us to specify the width of an item before it stretches or shrinks.

<div class="container">

<div class=”side”>

<h1>Left side!</h1>

</div>

<div class="center">

<h1>Center!</h1>

</div>

<div class="side">

<h1>Right side!</h1>

</div>

</div>

.container {

display: flex;

}

.side {

flex-grow: 1;

flex-basis: 100px;

}

.center {

flex-grow: 2;

flex-basis: 150px;

}

In the example above, the .side divs will be 100 pixels wide and the .center div will be 150 pixels wide if the .container div has just the right amount of space (350 pixels, plus a little extra for margins and borders). If the .container div is larger, the .center div will absorb twice as much space as the .side divs.

The same would hold true if we assigned flex-shrink values to the divs above as well.

## Flex

The flex property provides a convenient way for specifying how elements stretch and shrink, while simplifying the CSS required. The flex property allows you to declare flex-grow, flex-shrink, and flex-basis all in one line.

**Note:** The flex *property* is different from the flex *value* used for the display property.

.big {

flex-grow: 2;

flex-shrink: 1;

flex-basis: 150px;

}

.small {

flex-grow: 1;

flex-shrink: 2;

flex-basis: 100px;

}

In the example above, all elements with class big will grow twice as much as elements with class small. Keep in mind, this doesn’t mean big items will be twice as big as small items, they’ll just take up more of the extra space.

The CSS below declares these three properties in one line.

.big {

flex: 2 1 150px;

}

.small {

flex: 1 2 100px;

}

In the example above, we use the flex property to declare the values for flex-grow, flex-shrink, and flex-basis (in that order) all in one line.

.big {

flex: 2 1;

}

In the example above, we use the flex property to declare flex-grow and flex-shrink, but not flex-basis.

.small {

flex: 1 20px;

}

In the example above, we use the flex property to declare flex-grow and flex-basis. Note that there is no way to set only flex-shrink and flex-basis using 2 values.

The browser to the right has two flex containers, each with three flex items. In **style.css**, examine the values for each of these items. Notice that the flex-grow and flex-basis values are set for the blue divs.

Stretch the browser window to increase its width. Observe that once the top outer divs reach 100 pixels wide, they begin to grow faster than the top center div. Also notice that once the bottom center div reaches 100 pixels wide, it begins to grow faster than the outer divs.

Now, shrink the browser window and notice that once the top center div reaches 50 pixels wide it begins to shrink faster than the outer divs and when the bottom outer divs reach 75 pixels, they begin to shrink faster than the center div.

## Flex-wrap

Sometimes, we don’t want our content to shrink to fit its container. Instead, we might want flex items to move to the next line when necessary. This can be declared with the flex-wrap property. The flex-wrap property can accept three values:

1. wrap — child elements of a flex container that don’t fit into a row will move down to the next line
2. wrap-reverse — the same functionality as wrap, but the order of rows within a flex container is reversed (for example, in a 2-row flexbox, the first row from a wrap container will become the second in wrap-reverse and the second row from the wrap container will become the first in wrap-reverse)
3. nowrap — prevents items from wrapping; this is the default value and is only necessary to override a wrap value set by a different CSS rule.

<div class="container">

<div class="item">

<h1>We're going to wrap!</h1>

</div>

<div class="item">

<h1>We're going to wrap!</h1>

</div>

<div class="item">

<h1>We're going to wrap!</h1>

</div>

</div>

.container {

display: inline-flex;

flex-wrap: wrap;

width: 250px;

}

.item {

width: 100px;

height: 100px;

}

In the example above, three flex items are contained by a parent flex container. The flex container is only 250 pixels wide so the three 100 pixel wide flex items cannot fit inline. The flex-wrap: wrap; setting causes the third, overflowing item to appear on a new line, below the other two item.

**Note:** The flex-wrap property is declared on flex *containers*.

## Align-content

Now that elements can wrap to the next line, we might have multiple rows of flex items within the same container. In a previous exercise, we used the align-items property to space flex items from the top to the bottom of a flex container. align-items is for aligning elements within a single row. If a flex container has multiple rows of content, we can use align-content to space the rows from top to bottom.

align-content accepts six values:

1. flex-start — all rows of elements will be positioned at the top of the parent container with no extra space between.
2. flex-end — all rows of elements will be positioned at the bottom of the parent container with no extra space between.
3. center — all rows of elements will be positioned at the center of the parent element with no extra space between.
4. space-between — all rows of elements will be spaced evenly from the top to the bottom of the container with no space above the first or below the last.
5. space-around — all rows of elements will be spaced evenly from the top to the bottom of the container with the same amount of space at the top and bottom and between each element.
6. stretch — if a minimum height or no height is specified, the rows of elements will stretch to fill the parent container from top to bottom (default value).

<div class="container">

<div class=”child”>

<h1>1</h1>

</div>

<div class="child">

<h1>2</h1>

</div>

<div class="child">

<h1>3</h1>

</div>

<div class="child">

<h1>4</h1>

</div>

</div>

.container {

display: flex;

width: 400px;

height: 400px;

flex-wrap: wrap;

align-content: space-around;

}

.child {

width: 150px;

height: 150px;

}

In the example above, there are four flex items inside of a flex container. The flex items are set to be 150 pixels wide each, but the parent container is only 400 pixels wide. This means that no more than two elements can be displayed inline. The other two elements will wrap to the next line and there will be two rows of divs inside of the flex container. The align-content property is set to the value of space-around, which means the two rows of divs will be evenly spaced from top to bottom of the parent container with equal space before the first row and after the second, with double space between the rows.

Below, we will see each of the properties in action!

**Note:** The align-content property is declared on flex containers.

## Flex-direction

Up to this point, we’ve only covered flex items that stretch and shrink horizontally and wrap vertically. As previously stated, flex containers have two axes: a *major axis* and a *cross axis*. By default, the major axis is horizontal and the cross axis is vertical.

The major axis is used to position flex items with the following properties:

1. justify-content
2. flex-wrap
3. flex-grow
4. flex-shrink

The cross axis is used to position flex items with the following properties:

1. align-items
2. align-content

The major axis and cross axis are interchangeable. We can switch them using the flex-direction property. If we add the flex-direction property and give it a value of column, the flex items will be ordered vertically, not horizontally.

<div class="container">

<div class="item">

<h1>1</h1>

</div>

<div class="item">

<h1>2</h1>

</div>

<div class="item">

<h1>3</h1>

</div>

<div class="item">

<h1>4</h1>

</div>

<div class="item">

<h1>5</h1>

</div>

</div>

.container {

display: flex;

flex-direction: column;

width: 1000px;

}

.item {

height: 100px;

width: 100px;

}

In the example above, the five divs will be positioned in a vertical column. All of these divs could fit in one horizontal row. However, the column value tells the browser to stack the divs one on top of the other. As explained above, properties like justify-content will not behave the way they did in previous examples.

The flex-direction property can accept four values:

1. row — elements will be positioned from left to right across the parent element starting from the top left corner (default).
2. row-reverse — elements will be positioned from right to left across the parent element starting from the top right corner.
3. column — elements will be positioned from top to bottom of the parent element starting from the top left corner.
4. column-reverse — elements will be positioned from the bottom to the top of the parent element starting from the bottom left corner.

Below, we’ll investigate how these work.

**Note:** The flex-direction property is declared on flex containers.

## Flex-flow

Like the flex property, the flex-flow property is used to declare both the flex-wrap and flex-direction properties in one line.

.container {

display: flex;

flex-wrap: wrap;

flex-direction: column;

}

In the example above, we take two lines to accomplish what can be done with one.

.container {

display: flex;

flex-flow: column wrap;

}

In the example above, the first value in the flex-flow declaration is a flex-direction value and the second is a flex-wrap value. All values for flex-direction and flex-wrap are accepted.

**Note:** The flex-flow property is declared on flex containers.

## Nested Flexboxes

So far, we’ve had multiple flex containers on the same page to explore flex item positioning. It is also possible to position flex containers inside of one another.

<div class="container">

<div class="left">

<img class="small" src="#"/>

<img class="small" src="#"/>

<img class="small" src="#" />

</div>

<div class="right">

<img class="big" src="#" />

</div>

</div>

.container {

display: flex;

justify-content: center;

align-items: center;

}

.left {

display: inline-flex;

flex: 2 1 200px;

flex-direction: column;

}

.right {

display: inline-flex;

flex: 1 2 400px;

align-items: center;

}

.small {

height: 200px;

width: auto;

}

.large {

height: 600px;

width: auto;

}

In the example above, a div with three smaller images will display from top to bottom on the left of the page (.left). There is also a div with one large image that will display on the right side of the page (.right). The left div has a smaller flex-basis but stretches to fill more extra space; the right div has a larger flex-basis but stretches to fill less extra space. Both divs are flex items *and* flex containers. The items have properties that dictate how they will be positioned in the parent container and how their flex item children will be positioned in them.

We’ll use the same formatting above to layout the simple page to the right.

# Review Flex Boxes

You should be proud of yourself! You have learned the most important properties of flexbox. Flexbox is an art and a science; you can use it to make laying out multiple elements a piece of cake. You know everything necessary to begin using it in your own projects.

1. display: flex changes an element to a block-level container with flex items inside of it.
2. display: inline-flex allows multiple flex containers to appear inline with each other.
3. justify-content is used to space items along the major axis.
4. align-items is used to space items along the cross axis.
5. flex-grow is used to specify how much space (and in what proportions) flex items absorb along the major axis.
6. flex-shrink is used to specify how much flex items shrink and in what proportions along the major axis.
7. flex-basis is used to specify the initial size of an element styled with flex-grow and/or flex-shrink.
8. flex is used to specify flex-grow, flex-shrink, and flex-basis in one declaration.
9. flex-wrap specifies that elements should shift along the cross axis if the flex container is not large enough.
10. align-content is used to space rows along the cross axis.
11. flex-direction is used to specify the major and cross axes.
12. flex-flow is used to specify flex-wrap and flex-direction in one declaration.
13. Flex containers can be nested inside of each other by declaring display: flex or display: inline-flex for children of flex containers.