Johan Lindstrom

Life Time Fitness, Inc.  [Company address]

HTML & CSS Training

Headings are meant to emphasize or enlarge only a few words.

If you want to add blocks of text in HTML, you can use a *paragraph*, *div*, or *span*:

1. *Paragraphs* (<p>) simply contain a block of plain text.
2. <div>s can contain any text or other HTML elements. They are primarily used to *divide* HTML documents into sections.
3. <span>s contain short pieces of text or other HTML. They are primarily used to wrap small pieces of content that are on the same line as other content and do not break text into different sections.

Tags provided by HTML exist to organize and describe the content of web pages. Two of these HTML tags are <em>and <strong>. They are used to signal that the text within them should be "emphasized" or "strong."

Later, when you begin to style websites you will decide how you want browsers to display content within <em> and <strong> tags. However, browsers have built-in style sheets that will generally style these tags in this manner:

1. The <em> tag will generally render as *italic* emphasis.
2. The <strong> will generally render as **bold** emphasis.

The spacing between code in an HTML file doesn't affect the positioning of elements in the browser. If you *are* interested in modifying the spacing in the browser, you can use HTML's *line break* element: <br />.

The line break element is a self-closing tag. You can use it anywhere within your HTML code and a line break will be shown in the browser.

So far, all text has been in paragraph form. What if you want to display content in an easy-to-read list?

In HTML, you can use an *unordered list* tag (<ul>) to create a list of items in no particular order. An unordered list outlines individual *list items* with a bullet point.

The <ul> element cannot hold raw text and cannot automatically format raw text into an unordered list of items. Individual list items must be added to the unordered list using the <li> tag. The <li> or list item tag is used to describe an item in a list.

Great job! Some lists, however, will require a bit more structure. HTML provides the *ordered list* for when you need the extra ordering that unordered lists don't provide.

Ordered lists are like unordered lists, except that each list item is numbered. You can create the ordered list with the <ol> tag and then add individual list items to the list using <li> tags.

All of the elements you've learned about so far (headings, paragraphs, lists, and spans) share one thing in common: they're composed entirely of text! What if you want to add content to your web page that isn't composed of text, like images?

The <img> tag allows you to add an image to a web page. This is another example of a self-closing tag.

<img src="image-location.com" />

In addition to images, HTML also supports displaying videos. Like the <img> tag, the <video> tag requires a srcattribute with a link to the video source. Unlike the <img>tag however, the <video> element requires an opening and a closing tag.

<video src="myVi

<video src="myVideo.mp4" width="320" height="240" controls> Video not supported </video>

In this example, the video source (src=) is "myVideo.mp4." The source must link to a video file, not to a video on another site. After the src attribute, the width and heightattributes are used to set the size of the video displayed in the browser. The controls attribute instructs the browser to include basic video controls: pause, play and skip. The text, Video not supported, between the opening and closing video tags will only be displayed if the browser is unable to load the video.

You're off to a great start! So far you've learned how to add headings, paragraphs, lists, images, and videos to a web page. We wouldn't be taking advantage of the full power of HTML (and the Internet), however, if we didn't *link* to other web pages.

You can add links to a web page by adding an anchor element <a> and including the text of the link in between the opening and closing tags.

<a>This Is A Link To Wikipedia</a>

Wait a minute! Technically, the link in the example above is incomplete. How exactly is the link above supposed to work if there is no URL that will lead users to the actual Wikipedia page?

The anchor element in the example above is incomplete without the href *attribute*. This attribute stands for *hyperlink reference* and is used to link to a *file path*, or the address to where a file is located (whether it is on your computer or another location).

<a href="https://www.wikipedia.org/">This Is A Link To Wikipedia</a>

In the example above, the href attribute has been set to the value of the URL https://www.wikipedia.org/. The example now shows the correct use of an anchor element.

**Note:** When reading technical documentation, you may come across the term *hyperlink*. Not to worry, this is simply the technical term for link. These terms are often used interchangeably.

Have you ever clicked on a link and observed the resulting web page open in a new browser window? If so, you can thank the <a> element's target attribute.

The target attribute specifies that a link should open in a new window. Why is it beneficial to open links in a new window?

It's possible that one or more links on your web page link to an entirely different website. In that case, you may want users to read the linked website, but hope that they return to your web page. This is exactly when the target attribute is useful!

For a link to open in a new window, the target attribute requires a value of \_blank. The target attribute can be added directly to the opening tag of the anchor element, just like the href attribute.

<a href="https://en.wikipedia.org/wiki/Brown\_bear" target="\_blank">The Brown Bear</a>

In the example above, setting the target attribute to "\_blank" instructs the browser to open the relevant Wikipedia page in a new window.

**Note:** In this exercise, we've used the terminology "open in a new window." It's highly likely that you are using a modern browser that opens up websites in new *tabs*, rather than new windows. Before the advent of browsers with tabs, additional browser windows had to be opened to view more websites. The *target* attribute, when used in modern browsers, will open new websites in a new tab.

Thus far you have learned how to link to external web pages. Many sites also link to internal web pages like Home, About, and Contact.

Before we learn how to link between internal pages, let's establish where our files are stored. When making multi-page static websites, web developers often store HTML files in the *root directory*, or a main folder where all the files for the project are stored. As the size of the projects you create grows, you may use additional folders within the main project folder to organize your code.

about.html contact.html index.html

The example above shows three different files — **about.html**, **contact.html**, and **index.html** in one folder.

If the browser is currently displaying **index.html**, it knows that **about.html** and **contact.html** are in the same folder as **index.html**, also referred to as the *current* folder. Since the browser knows the current folder, other files in the folder can be linked using a *relative path*.

A relative path is a filename that shows the path to a *local file* (a file on the same website, such as ./index.html) versus an absolute path (a full url, like www.codecademy.com/learn/ruby which is stored in a different folder). The ./ in ./index.html tells the browser to look for the file in the current folder.

<a href="./contact.html">Contact</a>

In this example, the <a> tag is used with a relative path to link from the current HTML file to the contact.html file in the same folder. On the web page, Contact will appear as a link.

At this point, we have all the content we want on our page. Since we have so much content, it doesn't all fit on the screen. How do we make it easier for a user to jump to different portions of our page?

When users visit our site, we want them to be able to click a link and have the page automatically scroll to a specific section.

In order to link to a *target* on the same page, we must give the target an *id*, like this:

<p id="top"> This is the top of the page! </p> <h1 id="bottom">This is the bottom! </h1>

In this example, the <p> element contains id of top and the <h1> element contains id of bottom. An id can be added to most elements on a webpage.

An id should be descriptive to make it easier to remember the purpose of a link. The target link is a string containing the # character and the target element's id.

<ol> <li><a href="#top">Top</a></li> <li><a href="#bottom">Bottom</a></li> </ol>

In the example above, the links to <p id="top"> and <h1 id="bottom"> are embedded in an ordered list. These links appear in the browser as a numbered list of links. This is why we have been adding ids to our divs all along!

In the previous two exercises, you added numerous links to your page that allow a user to navigate content on the same page, to other pages on the same website, or to external websites.

Linking to elements on the same page or to other pages on the same site is called navigation. HTML has a special tag called <nav> that is used to wrap these links in order to organize the content on your web page.

Some of the tags we have used, such as <div>, are called *non-semantic* tags. This means that they do not describe the content that is inside of them. However, many tags are used to describe the content that they surround, which helps us modify and style our content later. These are called *semantic* tags and <nav> is one of them!

Setting up Building Environment

**3. ALL ABOUT URLS**

When you visit a website in your browser, you might navigate directly to the page you wish to see, such as <https://www.codecademy.com/learn/learn-html-css>. This is called a Uniform Resource Locator, or a URL. A URL tells a browser where to locate a resource (or file). The URL above tells the browser to request the learn-html-css resource from inside of the learn directory from codecademy.com.

A browser is simply a piece of software that can interpret and render HTML files (just like you might use a media player to listen to music or Microsoft Word to view a .docx file).

A URL is the same as a file path or the path to find a file on your computer. By default, most text editors display the file tree on the left side of the application. If your file tree is not visible in Atom, go to View > Toggle Tree View. Your file tree should look something like this:

At the top of the tree, there is a directory called projects and inside of projectsthere is helloHtml which contains **index.html**. If your “projects” folder is inside of the Documents folder on a Mac, your file path should look something like:

/Users/YourName/Documents/projects/helloHtml/index.html

This path guides your browser from your root directory, then to your YourName folder, then inside of Documents, then projects, then helloHtml, then index.html. The / symbol indicates different folders.

#### 4. LOCAL VS. REMOTE

Files that are saved on your computer are called local URLs:

/Users/YourName/Documents/projects/helloHtml/index.html

The file path refers to a file located on your local computer — this is considered a local file path.

There is a major difference between your local file path and https://www.codecademy.com/learn/learn-html-css, a remote path. Codecademy.com is not a directory on your computer - it is on Codecademy’s computer (or server). To view a file in Codecademy.com's directory, your computer makes a request to Codecademy. If Codecademy allows the request, then it sends a file, like **learn-html-css.html**, and your browser displays it.

#### 5. HYPERTEXT TRANSFER PROTOCOL

When you type the address of a website into your browser, the browser requests the website from its owner and renders it for you. The prefix http stands for **H**yper**T**ext **T**ransfer **P**rotocol, which refers to the protocol by which the HTML file from another server is transferred to your computer. In modern browsers, you don’t usually need to type http because the browser includes it for you.

#### 6. NAVIGATING TO A LOCAL URL

When you’re working on your website locally, it is a good practice to open it in your browser and see what it looks like as you make changes. There are several ways that you can do this.

* You can drag and drop your file from your file manager into Chrome.
* In the toolbar in Chrome you can click File > Open File and then navigate to index.html.
* You can type the path we found in section 2, beginning with file://. For example, you might type file:///Users/YourName/Documents/projects/hellohtml/index.html.
* In Atom, you can find your path by selecting the file in the file tree, right clicking, and selecting “Copy Full Path.” Paste it into the Chrome.

**Exercise IV: Previewing your HTML document in Chrome**

* Navigate to your index.html file and open it in your web browser. The browser tab should say “Hello, HTML!” and your header should be displayed in your browser window.

**Exercise V: Make a change**

* Currently, your website only has a single <h1> element. Add a paragraph of text to your HTML file in your text editor. Save your changes, then refresh the browser page in Chrome to see your changes.

Congratulations! You’ve set up the file tree for your first project, added the boilerplate code for the homepage of your website, and found and navigated to a local URL. Add some more content to your page using HTML tags and refresh the page to see your changes!

CSS Styling

Although CSS is a different language than HTML, it's possible to write CSS code directly within HTML code using *inline styles*.

To style an HTML element, you can add the style attribute directly to the opening tag. After you add the attribute, you can set it equal to the CSS style(s) you'd like applied to that element.

<p style="color: red;">I'm learning to code!</p>

The code in the example above demonstrates how to use inline styling. The paragraph element has a style attribute within its opening tag. Next, the style attribute is set equal to color: red;, which will set the color of the paragraph text to red within the browser.

You might be wondering about the syntax of the following snippet of code: color: red;. At the moment, the details of the syntax are not important; you'll learn more about CSS syntax in other exercises. For now, it's important to know that inline styles are a quick way of directly styling an HTML element.

If you'd like to add *more* than one style with inline styles, simply keep adding to the style attribute. Make sure to end the styles with a semicolon (;).

<p style="color: red; font-size: 20px;">I'm learning to code!</p>

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.

<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.

Again, the details of the CSS syntax in the example above aren't important at the moment. You will learn more about the details of CSS syntax in later lessons.

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.

Perfect! We successfully separated structure (HTML) from styling (CSS), but the web page still looks bland. Why?

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.

CSS is a styling language that can add color and structure to an HTML website. CSS works by selecting HTML elements, then applying styles to them. In this lesson, we’ll focus on how CSS selects HTML elements.

To style an HTML element, CSS first has to select it, using something called a CSS selector.

CSS always follows this two part process:

1. Select HTML elements.
2. Apply styles to the elements.

In this lesson, we’ll focus on how CSS selects HTML elements. There are lots of ways to select elements, and we'll show you a number of the most common.

Let's get started.

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.

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 classattribute. It's also possible to select an element by its classattribute.

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.

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.

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.

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.

In 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.

CSS Selectors Examples.

h1 {

color: maroon;

}

.title {

color: teal;

}

.uppercase {

text-transform: uppercase;

}

.publish-time{

color: gray;

}

.cursive{

font-family: cursive;

}

.capitalize {

text-transform: capitalize;

}

h2.destination{

font-family: cursive;

}

.description h5{

color: teal;

}

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

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.

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 h1and the .menu elements will receive the font-family: Georgia styling.

Throughout this lesson, you learned how to select HTML elements with CSS and apply styles to them. Let's review what you learned:

* CSS can change the look of HTML elements. In order to do this, CSS must select HTML elements, then apply styles to them.
* CSS can select HTML elements by tag, class, or ID.
* Multiple CSS classes can be applied to one HTML element.
* Classes can be reusable, while IDs can only be used once.
* IDs are more specific than classes, and classes are more specific than tags. That means IDs will override any styles from a class, and classes will override any styles from a tag selector.
* Multiple selectors can be chained together to select an element. This raises the specificity, but can be necessary.
* Nested elements can be selected by separating selectors with a space.
* The !important flag will override any style, however it should almost never be used, as it is extremely difficult to override.
* Multiple unrelated selectors can receive the same styles by separating the selector names with commas.

Great work this lesson. With this knowledge, you'll be able to use CSS to change the look and feel of websites to make them look great.

In CSS, the font-weight property controls how bold or thin text appears.

p { font-weight: bold; }

In the example above, all paragraphs on the web page would appear bolded.

The font-weight property has another value: normal. Why does it exist?

If we wanted *all* text on a web page to appear bolded, we could select all text elements and change their font weight to bold. If a certain section of text was required to appear normal, however, we could set the font weight of that particular element to normal, essentially shutting off bold for that element.

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

To align text we can use the text-align property. The text-align property will align text to the element that holds it, otherwise known as its *parent*.

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 its parent element, which in this case is the browser.
2. center — centers text inside of its parent element.
3. right — aligns text to the right hand side of its parent element.

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

* Foreground color
* 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:

* color: this property styles an element's foreground color
* 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.

Opacity is the measure of how transparent an element is. It's measured from 0 to 1, with 1 representing 100%, or fully visible and opaque, and 0 representing 0%, or fully invisible.

Opacity can be used to make elements fade into others for a nice overlay effect. To adjust the opacity of an element, the syntax looks like this:

.overlay { opacity: 0.5; }

In the example above, the .overlay element would be 50% visible, letting whatever is positioned behind it show through.

CSS has the ability to change the background of an element. One option is to make the background of an element an image. This is done through the CSS property background-image. Its syntax looks like this:

.main-banner { background-image: url("https://www.example.com/image.jpg"); }

1. The background-image property will set the element's background to display an image.
2. The value provided to background-image is a url. The url should be a url to an image. The url can be a file within your project, or it can be a link to an external site. To link to an image inside an existing project, you must provide a relative file path. If there was an image folder in the project, with an image named mountains.jpg, the relative file path would look like:

.main-banner { background-image: url("images/mountains.jpg"); }

Incredible work! You used CSS to alter text and images throughout a website. Throughout this lesson, you learned concepts including:

* CSS declarations are structured into property and value pairs.
* The font-family property defines the typeface of an element.
* font-size controls the size of text displayed.
* font-weight defines how thin or thick text is displayed.
* The align-text property places text in the left, right, or center of its parent container.
* Text can have two different color attributes: color and background-color. color defines the color of the text, while background-color defines the color behind the text.
* CSS can make an element transparent with the opacity property.
* CSS can also set the background of an element to an image with the background-image property.

**GitHub setup and commands**

Congratulations! You have now been introduced to the fundamental Git workflow. You learned a lot! Let's take a moment to generalize:

*Git* is the industry-standard version control system for web developers.

* Use Git commands to help keep track of changes made to a project:
  + git init creates a new Git repository.
  + git status inspects the contents of the working directory and staging area.
  + git add <filename> adds files from the working directory to the staging area.
  + git add . adds all files from the working directory to the staging area.
  + git commit -m <message> permanently stores file changes from the staging area in the repository.

*GitHub* is a service for hosting remote repositories on the web.

* git remote add origin <url> specifies the remote repository using Git
* git push -u origin master pushes the changes to the master branch on the remote repository, linking the local repository to the remote repository.
* git push origin master pushes the changes to the master branch on the remote repository, given that the local repository and the remote repository are already linked.

Positioning Elements in CSS



**Taking up space**

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

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

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

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

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

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

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

Example of boxes. HTML.

<!DOCTYPE html>

<html>

<head>

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

<title>Result</title>

</head>

<body>

<div id="one"></div>

<div id="two"></div>

<div id="three"></div>

<div id="four"></div>

</body>

</html>

**Example CSS boxes styles.**

**\* {**

**border: 1px dashed blue;**

**}**

**div {**

**height: 50px;**

**width: 100px;**

**border: 2px solid black;**

**border-radius: 5px;**

**display: block;**

**}**

**#one {**

**background-color: #FF0000;**

**}**

**#two {**

**background-color: #0000FF;**

**}**

**#three {**

**background-color: #FFD700;**

**}**

**#four {**

**background-color: #308014;**

**} The Box Model**



**Sketching it out**

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

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

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

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

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

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

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

**Margin**

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

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

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

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

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

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

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

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

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

margin: 1px 2px 3px 4px;

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

**Borders**

Well done! You can see how fine-tuning your margins will help you place elements where you'd like them to be on the page.

We've worked with borders before, but it never hurts to have extra practice.

**Negative values**

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

When you give CSS a positive padding or margin value, it puts that space between the element and its reference: for instance, if you have a <div> and you give it a margin-left of 20px, it puts twenty pixels between the left margin of that <div> and the side of the screen. This effectively moves the <div> twenty pixels to the *right*.

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

**To the right!**

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

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

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

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

**Clearing elements**

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

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

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

The syntax is what you're used to:

element {

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

}

**Instructions**

**Static by default**

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

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

**Display and Positioning**

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.

An element's content has two dimensions: a height and a width. By default, the dimension 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 is set in pixels it will be the same size on all devices — an element that fills a laptop screen will overflow a mobile screen

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#Values). 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](https://developer.mozilla.org/en-US/docs/Web/CSS/color_value) built-in color keywords.

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.

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 9pxcorrespond 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.

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.

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-items
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!

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 displayproperty 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.

In the previous exercise, you might have observed that when we gave a div — a block level element — the displayvalue 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.

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.

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-itemsproperty:

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 widthbefore. 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.

Now you’re going to see each of the five values above in action!

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 displayvalue of flex, so its three child divs will be positioned next to each other. If there is additional space in the .containerdiv (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.

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.

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.

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 smallitems, 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-basisvalues are set for the grey divs. Stretch the browser window to increase its width. Observe that once these divs reach 100 pixels wide, the center div begins to grow faster than the outer divs. Shrink the browser window and notice that something important happens: once the divs reach 100 pixels wide, they begin to shrink equally. This is because flex-shrink defaults to 1, whereas flex-grow defaults to 0.

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 will move down to the next line starting from the final item and working towards the first.
2. 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.
3. wrap-reverse — the wrapped element is displayed on top of the other elements in the flex container starting from the last and working toward the first (a mirror image of the wrap value).

<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.

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

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.

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-items 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.