**The Body**

Once the file has a body, many different types of content – including text, images, and buttons – can be added to the body.

<body>

<p>What's up, doc?</p>

</body>

**HTML Structure.** HTML is organized as a collection of family tree relationships. When an element is contained inside another element, it is considered the *child* of that element. The child element is said to be *nested* inside of the *parent* element.

<body>

<div>

<h1>Sibling to p, but also grandchild of body</h1>

<p>Sibling to h1, but also grandchild of body</p>

</div>

</body>

In this example, the <body> element is the parent of the <div> element. Both the <h1> and <p> elements are children of the <div> element. Because the <h1> and <p> elements are at the same level, they are considered siblings and are both grandchildren of the <body> element.

**Headings.** HTML follows a similar pattern. In HTML, there are six different *headings*, or *heading elements*. Headings can be used for a variety of purposes, like titling sections, articles, or other forms of content. <h1><h2><h3><h4><h5><h6>

<h1>BREAKING NEWS</h1>

**Divs**

One of the most popular elements in HTML is the <div> element. <div> is short for “division” or a container that divides the page into sections. These sections are very useful for grouping elements in your HTML together.

<div>s can contain any text or other HTML elements, such as links, images, or videos. Remember to always add two spaces of indentation when you nest elements inside of <div>s for better readability.

<body>

<div>

<h1>Why use divs?</h1>

<p>Great for grouping elements!</p>

</div>

</body>

**Attributes.** If we want to expand an element’s tag, we can do so using an *attribute*. Attributes are content added to the opening tag of an element and can be used in several different ways, from providing information to changing styling. Attributes are made up of the following two parts:

* The *name* of the attribute
* The *value* of the attribute

One commonly used attribute is the id. We can use the id attribute to specify different content (such as <div>s) and is really helpful when you use an element more than once. ids have several different purposes in HTML, but for now, we’ll focus on how they can help us identify content on our page. When we add an id to a <div>, we place it in the opening tag:

<div id="intro">

<h1>Introduction</h1>

</div>

**Displaying Text**. If you want to display text in HTML, you can use a *paragraph* or *span*:  *Paragraphs* (<p>) contain a block of plain text.

<span> contains short pieces of text or other HTML. They are used to separate small pieces of content that are on the same line as other content.

<div>

<h1>Technology</h1>

</div>

<div>

<p><span>Self-driving cars</span> are anticipated to replace up to 2 million jobs over the next two decades.</p>

</div>

In the example above, there are two different <div>. The second <div> contains a <p> with <span>Self-driving cars</span>. This <span> element separates “Self-driving cars” from the rest of the text in the paragraph.

It’s best to use a <span> element when you want to target a specific piece of content that is *inline*, or on the same line as other text. If you want to divide your content into *blocks*, it’s better to use a <div>.

**Styling Text**

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

<p><strong>The Nile River</strong> is the <em>longest</em> river in the world, measuring over 6,850 kilometers long (approximately 4,260 miles).</p>

In this example, the <strong> and <em> tags are used to emphasize the text to produce the following:

The Nile River is the *longest* river in the world, measuring over 6,850 kilometers long (approximately 4,260 miles).

As we can see, “The Nile River” is bolded and “longest” is in italics.

**Line Breaks.** 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 unique because it is only composed of a starting tag. You can use it anywhere within your HTML code and a line break will be shown in the browser.

<p>The Nile River is the longest river <br> in the world, measuring over 6,850 <br> kilometers long (approximately 4,260 <br> miles).</p>

The code in the example above will result in an output that looks like the following:

The Nile River is the longest river

in the world, measuring over 6,850

kilometers long (approximately 4,260

miles).

**Unordered Lists**

<ul>

<li>Limes</li>

<li>Tortillas</li>

<li>Chicken</li>

</ul>

In the example above, the list was created using the <ul> tag and all individual list items were added using <li> tags.

* Limes
* Tortillas
* Chicken

**Ordered Lists**

<ol>

<li>Preheat the oven to 350 degrees.</li>

<li>Mix whole wheat flour, baking soda, and salt.</li>

<li>Cream the butter, sugar in separate bowl.</li>

<li>Add eggs and vanilla extract to bowl.</li>

</ol>

**Images.** <img src="image-location.jpg" />

The <img> tag allows you to add an image to a web page. Most elements require both opening and closing tags, but the <img> tag is a *self-closing tag*. Note that the end of the <img> tag has a forward slash /. Self-closing tags may include or omit the final slash — both will render properly.

**Image Alts.** The alt attribute, which means alternative text, brings meaning to the images on our sites. The alt attribute can be added to the image tag just like the src attribute. The value of alt should be a description of the image.

* If an image fails to load on a web page, a user can mouse over the area originally intended for the image and read a brief description of the image. This is made possible by the description you provide in the alt attribute.
* Visually impaired users often browse the web with the aid of screen reading software. When you include the alt attribute, the screen reading software can read the image’s description out loud to the visually impaired user.
* The alt attribute also plays a role in Search Engine Optimization (SEO), because search engines cannot “see” the images on websites as they crawl the internet. Having descriptive alt attributes can improve the ranking of your site.

If the image on the web page is not one that conveys any meaningful information to a user (visually impaired or otherwise), the alt attribute should be left empty.

<img src="#" alt="A field of yellow sunflowers" />

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

<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 can be a video file that is hosted alongside your webpage, or a URL that points to a video file hosted on another webpage.

After the src attribute, the width and height attributes 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.

**Preparing for HTML.** This declaration is an instruction, and it must be the first line of code in your HTML document. It tells the browser what type of document to expect, along with what version of HTML is being used in the document. For now, the browser will correctly assume that the html in <!DOCTYPE html> is referring to HTML5, as it is the current standard.

**The <html> tag.** The <!DOCTYPE html> declaration provides the browser with two pieces of information (the type of document and the HTML version to expect), but it doesn’t actually add any HTML structure or content.

<!DOCTYPE html>

<html>

</html>

Anything between the opening <html> and closing </html> tags will be interpreted as HTML code. Without these tags, it’s possible that browsers could incorrectly interpret your HTML code.

**The Head.** The <head> element is part of this HTML metaphor. It goes above our <body> element.

The <head> element contains the *metadata* for a web page. Metadata is information about the page that isn’t displayed directly on the web page. Unlike the information inside of the <body> tag, the metadata in the head is information about the page itself. You’ll see an example of this in the next exercise.

The opening and closing head tags typically appear as the first item after your first HTML tag: <head></head>

**Page Titles.** What kind of metadata about the web page can the <head> element contain? If you navigate to the Codecademy [catalog](https://www.codecademy.com/learn/all) and look at the top of your browser, you’ll notice the words Full Catalog Courses & Tutorials | Codecademy, which is the *title* of the web page. A browser’s tab displays the title specified in the <title> tag. The <title> tag is always inside of the <head>.

<!DOCTYPE html>

<html>

<head>

<title>My Coding Journal</title>

</head>

</html>

**Where Does the Title Appear?** она будет написана в строке как Google вот так она будет отображаться.

**Linking to Other Web Pages.** This attribute stands for *hyperlink reference* and is used to link to a *path*, or the address to where a file is located (whether it is on your computer or another location). The paths provided to the href attribute are often URLs.

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

например написать: Learn more

**Opening Links in a New Window.** 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 how a link should open.

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!

**<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.

**Linking to Relative Page.** 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.

**project-folder/**

**|—— about.html**

**|—— contact.html**

**|—— index.html**

**<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.

**Linking At Will.** You’ve probably visited websites where not all links were made up of text. Maybe the links you clicked on were images or some other form of content. Thankfully, HTML allows you to turn nearly any element into a link by wrapping that element with an anchor element. With this technique, it’s possible to turn images into links by simply wrapping the <img> element with an <a> element.

**<a href="https://en.wikipedia.org/wiki/Opuntia" target="\_blank"><img src="https://www.Prickly\_Pear\_Closeup.jpg" alt="A red prickly pear fruit"/></a>**

In the example above, an image of a prickly pear has been turned into a link by wrapping the outside of the <img> element with an <a> element.

**Linking to Same Page.** 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 is assigned an id of “top” and the <h1> element is assigned “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. An id is especially helpful for organizing content belonging to a div!

**Whitespace.**

As the code in an HTML file grows, it becomes increasingly difficult to keep track of how elements are related. Programmers use two tools to visualize the relationship between elements: *whitespace* and *indentation*.

Both tools take advantage of the fact that the position of elements in a browser is independent of the amount of whitespace or indentation in the **index.html** file.

For example, if you wanted to increase the space between two paragraphs on your web page, you would *not* be able to accomplish this by adding space between the paragraph elements in the **index.html** file. The browser ignores *whitespace* in HTML files when it renders a web page, so it can be used as a tool to make code easier to read and follow.

<body><p>Paragraph 1</p><p>Paragraph 2</p></body>

This example is easier to read, because each element is on its own line. While the first example required you to read the entire line of code to identify the elements, this example makes it easy to identify the body tag and two paragraphs

<body>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</body>

**Indentation.** The second tool web developers use to make the structure of code easier to read is *indentation*. The spaces are inserted using the space and tab bars on your keyboard.

The [World Wide Web Consortium](https://www.w3.org/Consortium/), or W3C, is responsible for maintaining the style standards of HTML. At the time of writing, the W3C recommends 2 spaces of indentation when writing HTML code. Although your code will work without exactly two spaces, this standard is followed by the majority of professional web developers. Indentation is used to easily visualize which elements are nested within other elements.

<body>

<p>Paragraph 1</p>

<div>

<p>Paragraph 2</p>

</div>

</body>

In the example above, Paragraph 1 and the <div> tag are nested inside of the <body> tag, so they are indented two spaces. The Paragraph 2 element is nested inside of the <div> tag, so it is indented an additional two spaces.

**Comments.** HTML files also allow you to add comments to your code. Comments begin with <!-- and end with -->. Any characters in between will be ignored by your browser.

<!-- This is a comment that the browser will not display. -->

Including comments in your code is helpful for many reasons:

1. They help you (and others) understand your code if you decide to come back and review it at a much later date.
2. They allow you to experiment with new code, without having to delete old code.

<!-- Favorite Films Section -->

<p>The following is a list of my favorite films:</p>

In this example above, the comment is used to denote that the following text makes up a particular section of the page.

<!-- <p> Test Code </p> -->

In the example above, a valid HTML element (a paragraph element) has been “commented out.” This practice is useful when there is code you want to experiment with, or return to, in the future.

**Introduction to Tables.** There are many websites on the Internet that display information like stock prices, sports scores, invoice data, and more. This data is naturally tabular in nature, meaning that a table is often the best way of presenting the data.

In this part of the course, we’ll learn how to use the HTML <table> element to present information in a two-dimensional table to the users.

**Create a Table.** <table> </table>

**Table Rows.**

**<table>**

**<tr>**

**</tr>**

**<tr>**

**</tr>**

**</table>** In the example above, two rows have been added to the table.

**Table Data.**

**<table>**

**<tr>**

**<td>73</td>**

**<td>81</td>**

**</tr>**

**</table>**

In the example above, two data points (73 and 81) were entered in the one row that exists. By adding two data points, we created two cells of data. If the table were displayed in the browser, it would show a table with one row and two columns.

**Table Headings.** Table data doesn’t make much sense without titles to describe what the data represents.

To add titles to rows and columns, you can use the *table heading* element: <th>.

The table heading element is used just like a table data element, except with a relevant title. Just like table data, a table heading must be placed within a table row.

<table>

<tr>

<th></th>

<th scope="col">Saturday</th>

<th scope="col">Sunday</th>

</tr>

<tr>

<th scope="row">Temperature</th>

<td>73</td>

<td>81</td>

</tr>

</table>

What happened in the code above?

First, a new row was added to hold the three headings: a blank heading, a Saturday heading, and a Sunday heading. The blank heading creates the extra table cell necessary to align the table headings correctly over the data they correspond to.

In the second row, one table heading was added as a row title: Temperature.

Note, also, the use of the scope attribute, which can take one of two values:

1. row - this value makes it clear that the heading is for a row. **строка**
2. col - this value makes it clear that the heading is for a column. **столб**

**Table Borders.**

**<table border="1">**

**<tr>**

**<td>73</td>**

**<td>81</td>**

**</tr>**

**</table>**

The code in the example above is following is [deprecated](https://en.wikipedia.org/wiki/Deprecation), so please don’t use it. It’s meant to illustrate older conventions you may come across when reading other developers’ code. The browser will likely still interpret your code correctly if you use the border attribute, but that doesn’t mean the attribute should be used.

We use CSS to add style to HTML documents, because it helps us to separate the structure of a page from how it looks. You can learn more about CSS in our CSS courses. You can achieve the same table border effect using CSS.

table, td {

border: 1px solid black;

The code in the example above uses CSS instead of HTML to show table borders.

<tr>

<td>Davie's Burgers</td>

<td>2</td>

<td>Send Invoice</td>

</tr>

<tr>

<td>Baker's Bike Shop</td>

<td>3</td>

<td>Send Invoice</td>

</tr>

<tr>

<td>Miss Sally's Southern</td>

<td>4</td>

<td>Ship</td>

</tr>

<tr>

<td>Summit Resort Rentals</td>

<td>4</td>

<td>Ship</td>

</tr>

<tr>

<td>Strike Fitness</td>

<td>1</td>

<td>Enter Order</td>

</tr>

**Spanning Columns.** What if the table contains data that spans multiple columns? For example, a personal calendar could have events that span across multiple hours, or even multiple days.

Data can span columns using the colspan attribute. The attributes accepts an integer (greater than or equal to 1) to denote the number of columns it spans across.

<table>

<tr>

<th>Monday</th>

<th>Tuesday</th>

<th>Wednesday</th>

</tr>

<tr>

<td colspan="2">Out of Town</td>

<td>Back in Town</td>

</tr>

</table>

In the example above, the data Out of Town spans the Monday and Tuesday table headings using the value 2 (two columns). The data Back in Town appear only under the Wednesday heading.

**Spanning Rows.**

**<table>**

**<tr> <!-- Row 1 -->**

**<th></th>**

**<th>Saturday</th>**

**<th>Sunday</th>**

**</tr>**

**<tr> <!-- Row 2 -->**

**<th>Morning</th>**

**<td rowspan="2">Work</td>**

**<td rowspan="3">Relax</td>**

**</tr>**

**<tr> <!-- Row 3 -->**

**<th>Afternoon</th>**

**</tr>**

**<tr> <!-- Row 4 -->**

**<th>Evening</th>**

**<td>Dinner</td>**

**</tr>**

**</table>**

In the example above, there are four rows:

1. The first row contains an empty cell and the two column headings.
2. The second row contains the Morning row heading, along with Work, which spans two rows under the Saturday column. The “Relax” entry spans three rows under the Sunday column.
3. The third row only contains the Afternoon row heading.
4. The fourth row only contains the Dinner entry, since “Relax” spans into the cell next to it.

**Table Body.** Over time, a table can grow to contain a lot of data and become very long. When this happens, the table can be sectioned off so that it is easier to manage. Long tables can be sectioned off using the table body element: <tbody>.

<table>

  <tbody>

    <tr>

      <th></th>

      <th>Saturday</th>

      <th>Sunday</th>

    </tr>

    <tr>

      <th>Morning</th>

      <td rowspan="2">Work</td>

      <td rowspan="3">Relax</td>

    </tr>

    <tr>

     <th>Afternoon</th>

    </tr>

    <tr>

      <th>Evening</th>

      <td>Dinner</td>

    </tr>

  </tbody>

</table>

**Table** Head. the table’s headings were kept inside of the table’s body. When a table’s body is sectioned off, however, it also makes sense to section off the table’s column headings using the <thead> element.

<table>

  <thead>

    <tr>

      <th></th>

      <th scope="col">Saturday</th>

      <th scope="col">Sunday</th>

    </tr>

  </thead>

  <tbody>

    <tr>

      <th scope="row">Morning</th>

      <td rowspan="2">Work</td>

      <td rowspan="3">Relax</td>

    </tr>

    <tr>

     <th scope="row">Afternoon</th>

    </tr>

    <tr>

      <th scope="row">Evening</th>

      <td>Dinner</td>

    </tr>

  </tbody>

</table>

In the example above, the only new element is <thead>. The table headings are contained inside of this element. Note that the table’s head still requires a row in order to contain the table headings. Additionally, only the column headings go under the <thead> element. We can use the scope attribute on <th> elements to indicate whether a <th> element is being used as a "row" heading or a "col" heading.

**Table Footer.** The bottom part of a long table can also be sectioned off using the <tfoot> element.

<table>

  <thead>

    <tr>

      <th>Quarter</th>

      <th>Revenue</th>

      <th>Costs</th>

    </tr>

  </thead>

  <tbody>

    <tr>

      <th>Q1</th>

      <td>$10M</td>

      <td>$7.5M</td>

    </tr>

    <tr>

      <th>Q2</th>

      <td>$12M</td>

      <td>$5M</td>

    </tr>

  </tbody>

  <tfoot>

    <tr>

      <th>Total</th>

      <td>$22M</td>

      <td>$12.5M</td>

    </tr>

  </tfoot>

</table>

In the example above, the footer contains the totals of the data in the table. Footers are often used to contain sums, differences, and other data results.

**Styling with CSS.** Tables, by default, are very bland. They have no borders, the font color is black, and the typeface is the same type used for other HTML elements. CSS, or Cascading Style Sheets, is a language that web developers use to style the HTML content on a web page. You can use CSS to style tables. Specifically, you can style the various aspects mentioned above.

table, th, td {

  border: 1px solid black;

  font-family: Arial, sans-serif;

  text-align: center;

}

The code in the example above demonstrates just some of the various table aspects you can style using CSS properties.

**Introduction to HTML Forms.** Forms are a part of everyday life. When we use a physical form in real life, we write down information and give it to someone to process. Think of the times you’ve had to fill out information for various applications like a job, or a bank account, or dropped off a completed suggestion card — each instance is a form! Just like a physical form, an HTML <form> element is responsible for collecting information to send somewhere else. Every time we browse the internet we come into contact with many forms and we might not even realize it. There’s a good chance that if you’re typing into a text field or providing an input, the field that you’re typing into is part of a <form>! In this lesson, we’ll go over the structure and syntax of a <form> and the many elements that populate it.

**How a Form Works. (HTTP stands for Hypertext Transfer Protocol and is used to structure requests and responses over the internet. HTTP requires data to be transferred from one point to another over the network).** We can think of the internet as a network of computers which send and receive information. Computers need an HTTP request to know how to communicate. The HTTP request instructs the receiving computer how to handle the incoming information. The <form> element is a great tool for collecting information, but then we need to send that information somewhere else for processing. We need to supply the <form> element with both the location of where the <form>‘s information goes and what HTTP request to make. Take a look at the sample <form> below:

<form action="/example.html" method="POST">

</form>

In the above example, we’ve created the skeleton for a <form> that will send information to example.html as a POST request: The action attribute determines where the information is sent. The method attribute is assigned a HTTP verb that is included in the HTTP request. Note: HTTP verbs like POST do not need to be capitalized for the request to work, but it’s done so out of convention. In the example above we could have written method="post" and it would still work. The <form> element can also contain child elements. For instance, it would be helpful to provide a header so that users know what this <form> is about. We could also add a paragraph to provide even more detail. Let’s see an example of this in code:

<form action="/example.html" method="POST">

  <h1>Creating a form</h1>

  <p>Looks like you want to learn how to create an HTML form. Well, the best way to learn is to play around with it.</p>

</form>

The example above doesn’t collect any user input, but we’ll do that in the next exercise. For now, let’s practice making the foundation of an HTML <form>

**Text Input.**

If we want to create an input field in our <form>, we’ll need the help of the <input> element.

The <input> element has a type attribute which determines how it renders on the web page and what kind of data it can accept.

The first value for the type attribute we’re going to explore is "text". When we create an <input> element with type="text", it renders a text field that users can type into. It’s also important that we include a name attribute for the <input> — without the name attribute, information in the <input> won’t be sent when the <form> is submitted. We’ll explain more about submissions and the submit button in a later exercise. For now, let’s examine the following code that produces a text input field:

**<form action="/example.html" method="POST">**

**<input type="text" name="first-text-field">**

**</form>**

Here’s a screen shot of how the rendered form looks like on a web page for the Chrome browser (different browsers have different default rendering). When initially loaded, it will be an empty box:

rendered empty text field from input element type='text'

After users type into the <input> element, the value of the value attribute becomes what is typed into the text field. The value of the value attribute is paired with the value of the name attribute and sent as text when the form is submitted. For instance, if a user typed in “important details” in the text field created by our <input> element:

rendered filled text field which reads 'important details' 

When the form is submitted, the text: "first-text-field=important details" is sent to /example.html because the value of the name attribute is "first-text-field" and the value of value is "important details".

We could also assign a default value for the value attribute so that users have a pre-filled text field when they first see the rendered form like so:

<form action="/example.html" method="POST">

<input type="text" name="first-text-field" value="already pre-filled">

</form>

Which renders:

pre-filled text box due to assigned `value` attribute

**Adding a Label.**

In the previous exercise we created an <input> element but we didn’t include anything to explain what the <input> is used for. For a user to properly identify an <input> we use the appropriately named <label> element.

The <label> element has an opening and closing tag and displays text that is written between the opening and closing tags. To associate a <label> and an <input>, the <input> needs an id attribute. We then assign the for attribute of the <label> element with the value of the id attribute of <input>, like so:

<form action="/example.html" method="POST">

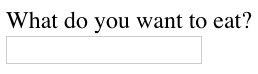
<label for="meal">What do you want to eat?</label>

<br>

<input type="text" name="food" id="meal">

</form>

The code above renders:



Look, now users know what the <input> element is for! Another benefit for using the <label> element is when this element is clicked, the corresponding <input> is highlighted/selected.

**Password Input.**

Think about all those times we have to put sensitive information, like a password or PIN, into a <form>. We wouldn’t want our information to be seen by anyone peeking over our shoulder! Luckily, we have the type="password" attribute for <input>!

An <input type ="password"> element will replace input text with another character like an asterisk (\*) or a dot (•). The code below provides an example of how to create a password field:

<form>

<label for="user-password">Password: </label>

<input type="password" id="user-password" name="user-password">

</form>

After a user types into the field, it would look like:

password field in a form with 6 dots showing text added to the field

Even though the password field obscures the text of the password, when the form is submitted, the value of the text is sent. In other words, if “hunter2” is typed into the password field, “user-password=hunter2” is sent along with the other information on the form.

**Number Input.** We’ve now gone over two type attributes for <input> related to text. But, we might want our users to type in a number — in which case we can set the type attribute to… (you guessed it)… "number"!

By setting type="number" for an <input> we can restrict what users type into the input field to just numbers (and a few special characters like -, +, and .). We can also provide a step attribute which creates arrows inside the input field to increase or decrease by the value of the step attribute. Below is the code needed to render an input field for numbers:

<form>

<label for="years"> Years of experience: </label>

<input id="years" name="years" type="number" step="1">

</form>

Which renders:

rendered number input field with arrows to the right hand side of the field

**Range Input.**

Using an <input type="number"> is great if we want to allow users to type in any number of their choosing. But, if we wanted to limit what numbers our users could type we might consider using a different type value. Another option we could use is setting type to "range" which creates a slider.

To set the minimum and maximum values of the slider we assign values to the min and max attribute of the <input>. We could also control how smooth and fluid the slider works by assigning the step attribute a value. Smaller step values will make the slider more fluidly, whereas larger step values will make the slider move more noticeably. Take a look at the code to create a slider:

<form>

<label for="volume"> Volume Control</label>

<input id="volume" name="volume" type="range" min="0" max="100" step="1">

</form>

The code above renders:rendered slider for volume control

In the example above, every time the slider moves by one, the value of the <input>‘s value attribute changes.

This practice has two new elements and an attribute that you may not be familiar with.

* <section> - An element used to represent a standalone section for which a more specific element can’t be found. This usually has a heading as a child element. A section should make sense in the outline of a document, whereas <div> is used for styling. This is a semantic element you’ll learn more about in a later lesson.
* class - A global attribute that has a list of classes pertaining to an element. You’ll see this used with <section> in the practice.
* <hr> - An element that is used to a break between paragraph-level elements. It is displayed as a horizontal line. This is also a semantic element that you’ll learn more about in a later lesson.

**Checkbox Input.** So far the types of inputs we’ve allowed were all single choices. But, what if we presented multiple options to users and allow them to select any number of options? Sounds like we could use checkboxes! In a <form> we would use the <input> element and set type="checkbox". Examine the code used to create multiple checkboxes:

<form>

<p>Choose your pizza toppings:</p>

<label for="cheese">Extra cheese</label>

<input id="cheese" name="topping" type="checkbox" value="cheese">

<br>

<label for="pepperoni">Pepperoni</label>

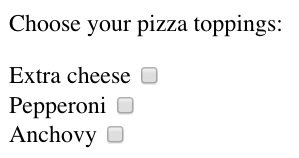
<input id="pepperoni" name="topping" type="checkbox" value="pepperoni">

<br>

<label for="anchovy">Anchovy</label>

<input id="anchovy" name="topping" type="checkbox" value="anchovy">

</form>

Which renders:

Notice in the example provided:

* there are assigned values to the value attribute of the checkboxes. These values are not visible on the form itself, that’s why it is important that we use an associated <label> to identify the checkbox.
* each <input> has the same value for the name attribute. Using the same name for each checkbox groups the <input>s together. However, each <input> has a unique id to pair with a <label>.

**Radio Button Input.**

Checkboxes work well if we want to present users with multiple options and let them choose one or more of the options. However, there are cases where we want to present multiple options and only allow for one selection — like asking users if they agree or disagree with the terms and conditions. Let’s look over the code used to create radio buttons:

<form>

<p>What is sum of 1 + 1?</p>

<input type="radio" id="two" name="answer" value="2">

<label for="two">2</label>

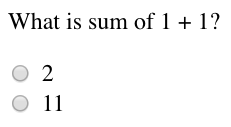
<br>

<input type="radio" id="eleven" name="answer" value="11">

<label for="eleven">11</label>

</form>

Which renders:



Notice from the code snippet, radio buttons (like checkboxes) do not display their value. We have an associated <label> to represent the value of the radio button. To group radio buttons together, we assign them the same name and only one radio button from that group can be selected.

**Dropdown list.**

Radio buttons are great if we want our users to pick one option out of a few visible options, but imagine if we have a whole list of options! This situation could quickly lead to a lot of radio buttons to keep track of. An alternative solution is to use a dropdown list to allow our users to choose one option from an organized list. Here’s the code to create a dropdown menu:

**<form>**

**<label for="lunch">What's for lunch?</label>**

**<select id="lunch" name="lunch">**

**<option value="pizza">Pizza</option>**

**<option value="curry">Curry</option>**

**<option value="salad">Salad</option>**

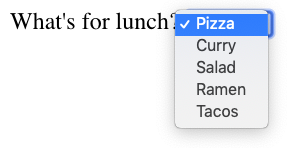
**<option value="ramen">Ramen</option>**

**<option value="tacos">Tacos</option>**

**</select>**

**</form>**

Which renders:rendered dropdown list with the first option showing

And if we click on the field containing the first option, the list is revealed:

Notice in the code that we’re using the element <select> to create the dropdown list. To populate the dropdown list, we add multiple <option> elements, each with a value attribute. By default, only one of these options can be selected.

The text rendered is the text included between the opening and closing <option> tags. However, it is the value of the value attribute that is used in <form> submission (notice the difference in the text and value capitalization). When the <form> is submitted, the information from this input field will be sent using the name of the <select> and the value of the chosen <option>. For instance, if a user selected Pizza from the dropdown list, the information would be sent as "lunch=pizza".

**Datalist Input.**

Even if we have an organized dropdown list, if the list has a lot of options, it could be tedious for users to scroll through the entire list to locate one option. That’s where using the <datalist> element comes in handy.

The <datalist> is used with an <input type="text"> element. The <input> creates a text field that users can type into and filter options from the <datalist>. Let’s go over a concrete example:

**<form>**

**<label for="city">Ideal city to visit?</label>**

**<input type="text" list="cities" id="city" name="city">**

**<datalist id="cities">**

**<option value="New York City"></option>**

**<option value="Tokyo"></option>**

**<option value="Barcelona"></option>**

**<option value="Mexico City"></option>**

**<option value="Melbourne"></option>**

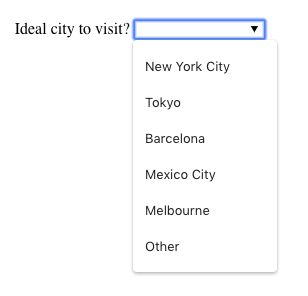
**<option value="Other"></option>**

**</datalist>**

**</form>**

Notice, in the code above, we have an <input> that has a list attribute. The <input> is associated to the <datalist> via the <input>‘s list attribute and the id of the <datalist>.

From the code provided, the following form is rendered:input field with a label 'Ideal city to visit?'

And when field is selected:

While <select> and <datalist> share some similarities, there are some major differences. In the associated <input> element, users can type in the input field to search for a particular option. If none of the <option>s match, the user can still use what they typed in. When the form is submitted, the value of the <input>‘s name and the value of the option selected, or what the user typed in, is sent as a pair.

**Textarea element.**

An <input> element with type="text" creates a single row input field for users to type in information. However, there are cases where users need to write in more information, like a blog post. In such cases, instead of using an <input>, we could use <textarea>.

The <textarea> element is used to create a bigger text field for users to write more text. We can add the attributes rows and cols to determine the amount of rows and columns for the <textarea>. Take a look:

<form>

<label for="blog">New Blog Post: </label>

<br>

<textarea id="blog" name="blog" rows="5" cols="30">

</textarea>

</form>

In the code above, an empty <textarea> that is 5 rows by 30 columns is rendered to the page like so:



If we wanted an even bigger text field, we could click and drag on the bottom right corner to expand it.

When we submit the form, the value of <textarea> is the text written inside the box. If we want to add a default value to text to <textarea> we would include it within the opening and closing tags like so:

<textarea>Adding default text</textarea. This code will render a <textarea> that contains pre-filled text: “Adding default text”

**Submit Form.**

Remember, the purpose of a form is to collect information that will be submitted. That’s the role of the submit button — users click on it when they are finished with filling out information in the <form> and they’re ready to send it off. Now that we’ve gone over how to create various input elements, let’s now go over how to create a submit button!

To make a submit button in a <form>, we’re going to use the reliable <input> element and set the type to "submit". For instance:

<form>

<input type="submit" value="Send">

</form>

Which renders:

rendered submit button

Notice in the code snippet that the value assigned to the <input> shows up as text on the submit button. If there isn’t a value attribute, the default text, Submit shows up on the button.

**FORM VALIDATION.**

Ever wonder how a login page actually works? Or why the combination of a username and password grants you access to a website? The answers lie in *validation*. Validation is the concept of checking user provided data against the required data.

There are different types of validation. One type is *server-side validation*, this happens when data is sent to another machine (typically a server) for validation. An example of this type of validation is the usage of a login page. The form on the login page accepts username and password input, then sends the data to a server that checks that the pair matches up correctly.

On the other hand, we use *client-side validation* if we want to check the data on the browser (the client). This validation occurs before data is sent to the server. Different browsers implement client-side validation differently, but it leads to the same outcome.

Shared among the different browsers are the benefits of using HTML5’s built-in client-side validation. It saves us time from having to send information to the server and wait for the server to send back confirmation or rejection of the data. This can also help us protect our server from malicious code or data from a malicious user. It also allows us to quickly give feedback to users for specific fields rather than having them fill in a form again if the data they input into the form was rejected.

In this lesson, we’ll learn how to add some validation checks to our <form>s!

**Requiring an Input.**

Sometimes we have fields in our <form>s which are not optional, i.e. there must be information provided before we can submit it. To enforce this rule, we can add the required attribute to an <input> element.

Take for example:

<form action="/example.html" method="POST">

<label for="allergies">Do you have any dietary restrictions?</label>

<br>

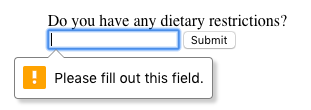
<input id="allergies" name="allergies" type="text" required>

<br>

<input type="submit" value="Submit">

</form>

This renders a text box, and if we try to submit the <form> without filling it out we get this message:



The styling of the message varies from browser to browser, the picture above depicts the message in a Chrome browser. We’ll also continue to show these messages as they appear in Chrome in later exercises.

Let’s see how it shows up in your browser!

**Set a Minimum and Maximum.**

Another built-in validation we can use is to assign a minimum or maximum value for a number field, e.g. <input type="number"> and <input type="range">. To set a minimum acceptable value, we use the min attribute and assign a value. On the flip side, to set a maximum acceptable value, we assign the max attribute a value. Let’s see this in code:

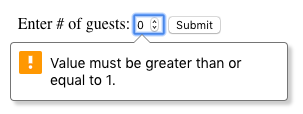
<form action="/example.html" method="POST">

<label for="guests">Enter # of guests:</label>

<input id="guests" name="guests" type="number" min="1" max="4">

<input type="submit" value="Submit">

</form>

If a user tries to submit an input that is less than 1 a warning will appear:

A similar message will appear if a user tries to input a number greater than 4. Let’s now see this action.

**Checking Text Length.**

In the previous exercise, we were able to use min and max to set acceptable minimum and maximum values in a number field. But what about text fields? There are certainly cases where we wouldn’t want our users typing more than a certain number of characters (think about the character cap for messages on Twitter). We might even want to set a minimum number of characters. Conveniently, there are built-in HTML5 validations for these situations.

To set a minimum number of characters for a text field, we add the minlength attribute and a value to set a minimum value. Similarly, to set the maximum number of characters for a text field, we use the maxlength attribute and set a maximum value. Let’s take a look at these attributes in code:

<form action="/example.html" method="POST">

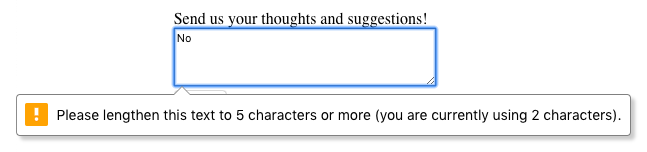
<label for="summary">Summarize your feelings in less than 250 characters</label>

<input id="summary" name="summary" type="text" minlength="5" maxlength="250" required>

<input type="submit" value="Submit">

</form>

If a user tries to submit the <form> with less than the set minimum, this message appears:



And if a user tries to type in more than the maximum allowed number of characters, they don’t get a warning message, but they can’t type it in!

**Matching a Pattern.**

In addition to checking the length of a text, we could also add a validation to check how the text was provided. For cases when we want user input to follow specific guidelines, we use the pattern attribute and assign it a *regular expression*, or regex. Regular expressions are a sequence of characters that make up a search pattern. If the input matches the regex, the form can be submitted.

Let’s say we wanted to check for a valid credit card number (a 14 to 16 digit number). We could use the regex: [0-9]{14,16} which checks that the user provided only numbers and that they entered at least 14 digits and at most 16 digits.

To add this to a form:

<form action="/example.html" method="POST">

<label for="payment">Credit Card Number (no spaces):</label>

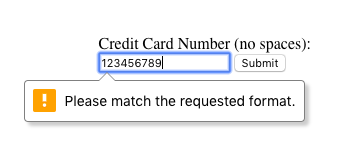
<br>

<input id="payment" name="payment" type="text" required pattern="[0-9]{14,16}">

<input type="submit" value="Submit">

</form>

With the pattern in place, users can’t submit the <form> with a number that doesn’t follow the regex. When they try, they’ll see a validation message like so:



If you want to find out more about Regex, read more at [MDN’s regex article](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions).

**SEMANTIC HTML.**

**Introduction to Semantic HTML.**

When building web pages, we use a combination of non-semantic HTML and *Semantic HTML*. The word semantic means “relating to meaning,” so semantic elements provide information about the content between the opening and closing tags.

By using Semantic HTML, we select HTML elements based on their meaning, not on how they are presented. Elements such as <div> and <span> are not semantic elements since they provide no context as to what is inside of those tags.

For example, instead of using a <div> element to contain our header information, we could use a <header> element, which is used as a heading section. By using a <header> tag instead of a <div>, we provide context as to what information is inside of the opening and closing tag.

**Why use Semantic HTML?**

* **Accessibility:** Semantic HTML makes webpages accessible for mobile devices and for people with disabilities as well. This is because screen readers and browsers are able to interpret the code better.
* **SEO:** It improves the website SEO, or *Search Engine Optimization*, which is the process of increasing the number of people that visit your webpage. With better SEO, search engines are better able to identify the content of your website and weight the most important content appropriately.
* **Easy to Understand:** Semantic HTML also makes the website’s source code easier to read for other web developers.

To better understand this, you can think of comparing non-semantic HTML to going into a store with no signs on the aisles. Since the aisles aren’t labeled, you don’t know what products are in those aisles. However, stores that do have signs for each aisle make it a lot easier to find the items you need, just like Semantic HTML.

**Header and Nav.**

Let’s take a look at some semantic elements that assist in the structure of a web page. A <header> is a container usually for either navigational links or introductory content containing <h1> to <h6> headings.

The example below shows <header> in action:

**<header>**

**<h1>**

**Everything you need to know about pizza!**

**</h1>**

**</header>**

This can be compared to the code below which uses a <div> tag instead of a <header> tag:

**<div id="header">**

**<h1>**

**Everything you need to know about pizza!**

**</h1>**

**</div>**

By using a <header> tag, our code becomes easier to read. It is much easier to identify what is inside of the <h1>‘s parent tags, as opposed to a <div> tag which would provide no details as to what was inside of the tag.

A <nav> is used to define a block of navigation links such as menus and tables of contents. It is important to note that <nav> can be used inside of the <header> element but can also be used on its own.

Let’s take a look at the example below:

<header>

<nav>

<ul>

<li><a href="#home">Home</a></li>

<li><a href="#about">About</a></li>

</ul>

</nav>

</header>

By using <nav> as a way to label our navigation links, it will be easier for not only us, but also for web browsers and screen readers to read the code.

**Main and Footer.**

Two more structural elements are <main> and <footer>. These elements along with <nav> and <header> help describe where an element is located based on conventional web development standards.

The element <main> is used to encapsulate the dominant content within a webpage. This tag is separate from the <footer> and the <nav> of a web page since these elements don’t contain the principal content. By using <main> as opposed to a <div> element, screen readers and web browsers are better able to identify that whatever is inside of the tag is the bulk of the content.

So how does <main> look when incorporated into our code? That’s a great question.

**<main>**

**<header>**

**<h1>Types of Sports</h1>**

**</header>**

**<article>**

**<h3>Baseball</h3>**

**<p>**

**The first game of baseball was played in Cooperstown, New York in the summer of 1839.**

**</p>**

**</article>**

**</main>**

As we see above, <main> contains an <article> and <header> tag with child elements that hold the most important information related to the page.

The content at the bottom of the subject information is known as the *footer*, indicated by the <footer> element. The footer contains information such as:

* Contact information
* Copyright information
* Terms of use
* Site Map
* Reference to top of page links

For example:

<footer>

<p>Email me at Codey@Codecademy.com</p>

</footer>

In the example above, the footer is used to contain contact information. The <footer> tag is separate from the <main> element and typically located at the bottom of the content.

**Article and Section.**

Now that we covered the body of Semantic HTML, let’s focus on what can go in the body. The two elements we’re going to focus on now are <section> and <article>.

<section> defines elements in a document, such as chapters, headings, or any other area of the document with the same theme. For example, content with the same theme such as articles about cricket can go under a single <section>. A website’s home page could be split into sections for the introduction, news items, and contact information.

Here is an example of how to use <section>:

<section>

<h2>Fun Facts About Cricket</h2>

</section>

In the code above we created a <section> element to encapsulate the code. In <section> we added a <h2> element as a heading.

The <article> element holds content that makes sense on its own. <article> can hold content such as articles, blogs, comments, magazines, etc. An <article> tag would help someone using a screen reader understand where the article content (that might contain a combination of text, images, audio, etc.) begins and ends.

Here is an example of how to use <article>:

**<section>**

**<h2>Fun Facts About Cricket</h2>**

**<article>**

**<p>A single match of cricket can last up to 5 days.</p>**

**</article>**

**</section>**

In the code above, the <article> element containing a fact about cricket was placed inside of the <section> element. It is important to note that a <section> element could also be placed in an <article> element depending on the context.

**The Aside Element.**

The <aside> element is used to mark additional information that can enhance another element but isn’t required in order to understand the main content. This element can be used alongside other elements such as <article> or <section>. Some common uses of the <aside> element are for:

* Bibliographies
* Endnotes
* Comments
* [Pull quotes](https://en.wikipedia.org/wiki/Pull_quote)
* Editorial sidebars
* Additional information

Here’s an example of <aside> being used alongside <article>:

<article>

<p>The first World Series was played between Pittsburgh and Boston in 1903 and was a nine-game series.</p>

</article>

<aside>

<p>

Babe Ruth once stated, “Heroes get remembered, but legends never die.”

</p>

</aside>

As shown above, the information within the <article> is the important content. Meanwhile the information within the <aside> enhances the information in <article> but is not required in order to understand it.

**Figure and Figcaption.**

With <aside>, we learned that we can put additional information next to a main piece of content, but what if we wanted to add an image or illustration? That is where <figure> and <figcaption> come in.

<figure> is an element used to encapsulate media such as an image, illustration, diagram, code snippet, etc, which is referenced in the main flow of the document.

<figure>

<img src="overwatch.jpg"/>

</figure>

In this code, we created a <figure> element so that we can encapsulate our <img> tag. In <figure> we used the <img> tag to insert an image onto the webpage. We used the src attribute within the <img> tag so that we can link the source of the image.

It’s possible to add a caption to the image by using <figcaption>.

<figcaption> is an element used to describe the media in the <figure> tag. Usually, <figcaption> will go inside <figure>. This is different than using a <p> element to describe the content; if we decide to change the location of <figure>, the paragraph tag may get displaced from the figure while a <figcaption> will move with the figure. This is useful for grouping an image with a caption.

<figure>

<img src="overwatch.jpg">

<figcaption>This picture shows characters from Overwatch.</figcaption>

</figure>

In the example above, we added a <figcaption> into the <figure> element to describe the image from the previous example. This helps group the <figure> content with the <figcaption> content.

While the content in <figure> is related to the main flow of the document, its position is independent. This means that you can remove it or move it somewhere else without affecting the flow of the document.

**Audio and Attributes.**

Now that we learned about text-based content, let us dig into <audio>! Surely everyone needs <audio>—how else would you listen to your Korean hip hop?

The <audio> element is used to embed audio content into a document. Like <video>, <audio> uses src to link the audio source.

<audio>

<source src="iAmAnAudioFile.mp3" type="audio/mp3">

</audio>

In this example, we created an <audio> element. Then we created a <source> element to encapsulate our audio link. In this case, iAmAnAudioFile.mp3 is our audio file. Then we specified the type by using type and named what kind of audio it is. Although not always necessary, it’s recommended that we state the type of audio as it helps the browser identify it more easily and determine if that type of audio file is supported by the browser.

We linked our audio file into the browser but now we need to give it controls. This is where *attributes* come in. Attributes provide additional information about an element.

Attributes allow us to do many different things to our audio file. There are many attributes for <audio> but today we’re going to be focusing on controls and src.

* controls: automatically displays the audio controls into the browser such as play and mute.
* src: specifies the URL of the audio file.

As you might have noticed, we already used the src attribute. Most attributes go in the opening tag of <audio>. For example, here’s how we could add both autoplay functionality and audio controls:

<audio autoplay controls>

**Video and Embed.**

As demonstrated in the previous exercise, media content can be a useful addition to a website. By using a <video> element, we can add videos to our website. The <video> element makes it clear that a developer is attempting to display a video to the user.

Some attributes that can alter a video playback include:

* controls: When added in, a play/pause button will be added onto the video along with volume control and a fullscreen option.
* autoplay: The attribute which results in a video automatically playing as soon as the page is loaded.
* loop: This attribute results in the video continuously playing on repeat.

Below is an example of <video> being used with the controls attribute:

<video src="coding.mp4" controls>Video not supported</video>

In the code above, a video file named coding.mp4 is being played. The “Video not supported” will only show up if the browser is unable to display the video.

Another tag that can be used to incorporate media content into a page is the <embed> tag, which can embed any media content including videos, audio files, and gifs from an external source. This means that websites that have an embed button have some form of media content that can be added to other websites. The <embed> tag is a self-closing tag, unlike the <video> element. Note that <embed> is a deprecated tag and other alternatives, such as <video>, <audio> and <img>, should be used in its place, but is being taught for legacy purposes.

Below we’ll take a look at <embed> being used in action.

<embed src="download.gif"/>

In the example above, <embed> is being used to add in a gif from a local file known as download.gif. Embed can be used to add local files as well as media content straight from some other websites.

**Learn CSS.**

**CSS SETUP AND SELECTORS.**

**Intro to CSS.**

The basic structure of every web page, HTML, is very plain on its own. The beautiful websites that you see across the internet are styled with a variety of tools, including CSS.

*CSS*, or Cascading Style Sheets, is a language that web developers use to *style* the HTML content on a web page. If you’re interested in modifying colors, font types, font sizes, shadows, images, element positioning, and more, CSS is the tool for the job!

In this lesson, you’ll learn how to select which HTML elements you wish to style and set up your CSS file structure.

**Inline Styles.**

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>**

**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. 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.

**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.

**Linking the CSS 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 a very common way of linking a stylesheet.

**Tag Name.**

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.

**Multiple Classes.**

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 and IDs.**

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.**

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.

**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.**

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.

**Chaining and Specificity.**

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.

**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.

**Review CSS Selectors.**

* 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.
* Multiple unrelated selectors can receive the same styles by separating the selector names with commas.

**Introduction To Visual Rules.**

In this lesson, you’ll learn the basic structure and syntax of CSS so that you can start styling web page elements.

**CSS Structure.**

To style an HTML element using CSS, you need to write a CSS declaration inside the body of a CSS selector.

h1 {

color: blue;

}

The example above selects the <h1> element. Inside of the selector’s body, we typed color: blue. This line is referred to as a CSS *declaration*. CSS declarations consist of a *property* and a *value*.

Property — the property you’d like to style of that element (i.e., size, color, etc.).

Value — the value of the property (i.e., 18px for size, blue for color, etc.).

In the example above, the property is color and the value is blue. The property and value are separated by a colon (:). A semicolon (;) should always be used at the end of a declaration.

Finally, the entire snippet of code in the example above is known as a CSS *rule* or *rule set*. A CSS rule consists of the selector (here, h1) and all declarations inside of the selector.

**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 font you were typing in. 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 visits the web page.
2. The default typeface for all HTML elements is Times New Roman. You may be familiar with this typeface if you have ever used a formatted word processor. If no font-family attribute is defined, the page will appear in Times New Roman.
3. It’s a good practice to limit the number of typefaces used on a web page to 2 or 3. This helps the page load faster in some cases and is usually a good design decision.
4. When the name of a typeface consists of more than one word, it’s a best practice to enclose the typeface’s name in quotes, like so:

h1 {

font-family: "Courier New";

}

**Font Size.**

Changing the typeface isn’t the only way to customize text. Often times, different sections of a web page are highlighted by modifying the *font size*.

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

**p {**

**font-size: 18px;**

**}**

In the example above, the font-size of all paragraphs was set to 18px. px means pixels and is a way to measure font size

**Font Weight.**

**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 a 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.**

**Text Align.**

**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.**

**Color.**

**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.**

**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.**

**Background Image.**

**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");**

**}**

**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.

One justification for using !important is when working with multiple stylesheets. For example, if we are using the [Material Design Lite](https://getmdl.io/) style library and want to override the styles for one specific HTML element, we can use the !important property.

**Review Visual Rules**

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 text-align 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.
* The !important flag will override any style, however it should almost never be used, as it is extremely difficult to override.

**Introduction to 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.**

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.

The image to the right is a visual representation of the box model.

Open [the box model image](https://content.codecademy.com/courses/freelance-1/unit-4/diagram-boxmodel.svg) in a new tab so you can reference the box model as you move through the lesson.

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

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 a radius of 5 pixels would have).

You can create a border that is a perfect circle by first creating an element with the same width and height and then setting the radius equal to the distance between the center of the circle and the width of the box, which is 50%.

div.container {

height: 60px;

width: 60px;

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

border-radius: 50%;

}

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

Padding I.

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.

Padding II.

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 I.**

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.

Margins II.

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.

Review

In this lesson, we covered the four properties of the box model: height and width, padding, borders, and margins. Understanding the box model is an important step towards learning more advanced HTML and CSS topics. Let’s take a minute to review what you learned.

1. The box model comprises a set of properties used to create space around and between HTML elements.
2. The height and width of a content area can be set in pixels or percentage.
3. Borders surround the content area and padding of an element. The color, style, and thickness of a border can be set with CSS properties.
4. Padding is the space between the content area and the border. It can be set in pixels or percent.
5. Margin is the amount of spacing outside of an element’s border.
6. Horizontal margins add, so the total space between the borders of adjacent elements is equal to the sum of the right margin of one element and the left margin of the adjacent element.
7. Vertical margins collapse, so the space between vertically adjacent elements is equal to the larger margin.
8. margin: 0 auto horizontally centers an element inside of its parent content area, if it has a width.
9. The overflow property can be set to display, hide, or scroll, and dictates how HTML will render content that overflows its parent’s content area.
10. The visibility property can hide or show elements.

Why Change 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.

Box Model: Content-Box.

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.

Box Model: 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.

**The New Box Model**

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.

**Review: Changing the Box Model**

In this lesson, you learned about an important limitation of the default box model: box dimensions are affected by border thickness and padding.

Let’s review what you learned:

1. In the default box model, box dimensions are affected by border thickness and padding.
2. The box-sizing property controls the box model used by the browser.
3. The default value of the box-sizing property is content-box.
4. The value for the new box model is border-box.
5. The border-box model is not affected by border thickness or padding.

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: 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;

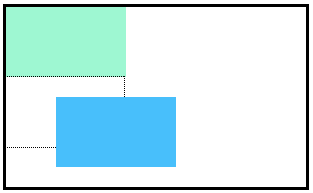
left: 50px;

}

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.

Position: 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;

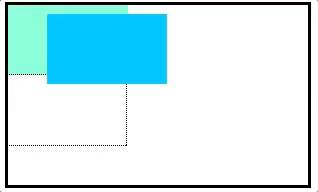
position: fixed;

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;

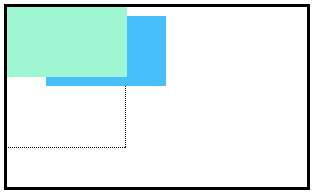
top: 20px;

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.

**Inline 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.

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 Display.**

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;

}

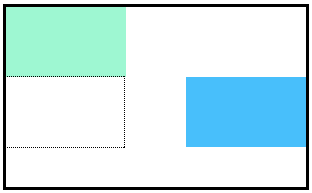
.box-bottom {

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.

**Review: Layout**

Great job! In this lesson, you learned how to control the positioning of elements on a web page.

Let’s review what you’ve learned so far:

1. The position property allows you to specify the position of an element in three different ways.
2. When set to relative, an element’s position is relative to its default position on the page.
3. When set to absolute, an element’s position is relative to its closest positioned parent element. It can be pinned to any part of the web page, but the element will still move with the rest of the document when the page is scrolled.
4. When set to fixed, an element’s position can be pinned to any part of the web page. The element will remain in view no matter what.
5. The z-index of an element specifies how far back or how far forward an element appears on the page when it overlaps other elements.
6. The display property allows you control how an element flows vertically and horizontally a document.
7. inline elements take up as little space as possible, and they cannot have manually-adjusted width or height.
8. block elements take up the width of their container and can have manually-adjusted heights.
9. inline-block elements can have set width and height, but they can also appear next to each other and do not take up their entire container width.
10. The float property can move elements as far left or as far right as possible on a web page.
11. You can clear an element’s left or right side (or both) using the clear property.

When combined with an understanding of the box model, positioning can create visually appealing web pages. So far, we’ve focused on adding content in the form of text to a web page. In the next unit, you’ll learn how to add and manipulate images to a web page.

Introduction to 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.

Foreground vs Background.

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.

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.

Hex and RGB

The hexadecimal and RGB color system can represent many more colors than the small set of CSS named colors. We can use this new set of colors to refine our web page’s style.

In both hex and decimal, we have three values, one for each color. Each can be one of 256 values. Specifically, 256 \* 256 \* 256 = 16,777,216. That is the amount of colors we can now represent. Compare that to the 147 named CSS colors!

Recall that we started with named colors, converted them to hex, and then converted some of the hex colors to decimal. Unless we made a mistake, all of the colors should still be the same, visually. Let’s use our broadened palette to make some more refined color choices.

Hue, Saturation, and 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:

color wheel

*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;

**Color Review.**

We’ve completed our extensive tour of the colors in CSS! Let’s review the key information we’ve learned.

There are four ways to represent color in CSS:

* Named colors — there are 147 named colors, which you can review [here](https://msdn.microsoft.com/en-us/library/aa358802(v=vs.85).aspx).
* Hexadecimal or hex colors
  + Hexadecimal is a number system with has sixteen digits, 0 to 9 followed by “A” to “F”.
  + Hex values always begin with # and specify values of red, blue and green using hexademical numbers such as #23F41A.
* RGB
  + RGB colors use the rgb() syntax with one value for red, one value for blue and one value for green.
  + RGB values range from 0 to 255 and look like this: rgb(7, 210, 50).
* HSL
  + HSL stands for hue (the color itself), saturation (the intensity of the color), and lightness (how light or dark a color is).
  + Hue ranges from 0 to 360 and saturation and lightness are both represented as percentages like this: hsl(200, 20%, 50%).
* You can add opacity to color in RGB and HSL by adding a fourth value, a, which is represented as a percentage.

Great job! Feel empowered to add a bit of color to each of your projects!

Typography.

In this lesson, we’ll focus on *typography*, the art of arranging text on a page. In particular, we’ll look at how to style fonts with CSS to make them legible and appealing and how to add external fonts to your web pages.

Some of the most important information a user will see on a web page will be textual. Styling text to make page content accessible and engaging can significantly improve user experience. Let’s begin!

**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 many 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.

**Font Weight II.**

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;

}

Note that it’s good to use em values in this case because em is dynamic — for word spacing, it sets the spacing based on the size of the font. In the example above, the word spacing is set to 0.3em. The default amount of space between words is usually 0.25em and can be set with the value normal. If you provide a value for word-spacing that’s not normal, then the value you provide is added to the default spacing. Therefore, since the word-spacing is set to 0.3em, your <h1> elements get a total of 0.55em word spacing when rendered.

It’s not common to increase the spacing between words, but it may help enhance the readability of bolded or enlarged text.

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.

Line Height.

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 I.**

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.

We’ll show you where to add this link in the next exercise.

**Linking Fonts II**

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

Let’s take a look at a few examples:

1. A single linked font, using Open Sans as an example:

<head>

<link href="https://fonts.googleapis.com/css2?family=Open+Sans" rel="stylesheet">

</head>

2. Multiple linked fonts, using the Open Sans and Playfair Display fonts as an example:

<head>

<link href="https://fonts.googleapis.com/css2?family=Open+Sans&family=Playfair+Display" rel="stylesheet">

</head>

3. Multiple linked fonts, along with weights and styles. Here Open Sans 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/css2?family=Open+Sans:ital,wght@0,400;0,700;1,700&family=Playfair+Display:ital,wght@0,400;0,700;1,900" 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 I.

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.

**Font-Face II.**

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.

Font-Face III.

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/).

Review

Great job! You learned how to style an important aspect of the user experience, typography.

Let’s review what you’ve learned so far:

* *Typography* is the art of arranging text on a page.
* Text can appear in any number of weights, with the font-weight property.
* Text can appear in italics with the font-style property.
* The vertical spacing between lines of text can be modified with the line-height property.
* *Serif* fonts have extra details on the ends of each letter. *Sans-Serif* fonts do not.
* *Fallback fonts* are used when a certain font is not installed on a user’s computer.
* Google Fonts provides free fonts that can be used in an HTML file with the <link> tag or the @font-face property.
* Local fonts can be added to a document with the @font-face property and the path to the font’s source.
* The word-spacing property changes how far apart individual words are.
* The letter-spacing property changes how far apart individual letters are.
* The text-align property changes the horizontal alignment of text.

What is 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.

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.

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

**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.

.container {

display: flex;

}

<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>

.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 *main axis* and a *cross axis*. By default, the main axis is horizontal and the cross axis is vertical.

The main 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 main 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="large" 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: Flexbox**

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 main 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 main axis.
6. flex-shrink is used to specify how much flex items shrink and in what proportions along the main 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 main 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.

Creating a Grid.

To set up a grid, you need to have both a *grid container* and *grid items*. The grid container will be a parent element that contains grid items as children and applies overarching styling and positioning to them.

To turn an HTML element into a grid container, you must set the element’s display property to grid (for a block-level grid) or inline-grid (for an inline grid). Then, you can assign other properties to lay out the grid.

Creating Columns.

By default, grids contain only one column. If you were to start adding items, each item would be put on a new row; that’s not much of a grid! To change this, we need to explicitly define the number of rows and columns in our grid.

We can define the columns of our grid by using the CSS property grid-template-columns. Below is an example of this property in action:

.grid {

display: grid;

width: 500px;

grid-template-columns: 100px 200px;

}

This property creates two changes. First, it defines the number of columns in the grid; in this case, there are two. Second, it sets the width of each column. The first column will be 100 pixels wide and the second column will be 200 pixels wide.

We can also define the size of our columns as a percentage of the entire grid’s width.

.grid {

display: grid;

width: 1000px;

grid-template-columns: 20% 50%;

}

In this example, the grid is 1000 pixels wide. Therefore, the first column will be 200 pixels wide because it is set to be 20% of the grid’s width. The second column will be 500 pixels wide.

We can also mix and match these two units. In the example below, there are three columns of width 20 pixels, 40 pixels, and 60 pixels:

.grid {

display: grid;

width: 100px;

grid-template-columns: 20px 40% 60px;

}

Notice that in this example, the total width of our columns (120 pixels) exceeds the width of the grid (100 pixels). This might make our grid cover other elements on the page! In a later exercise we will discuss how to avoid overflow.

Creating Rows.

We’ve learned how to define the number of columns in our grid explicitly. To specify the number and size of the rows, we are going to use the property grid-template-rows.

This property is almost identical to grid-template-columns. Take a look at the code below to see both properties in action.

.grid {

display: grid;

width: 1000px;

height: 500px;

grid-template-columns: 100px 200px;

grid-template-rows: 10% 20% 600px;

}

This grid has two columns and three rows. grid-template-rows defines the number of rows and sets each row’s height. In this example, the first row is 50 pixels tall (10% of 500), the second row is 100 pixels tall (20% of 500), and the third row is 600 pixels tall.

When using percentages in these two properties, remember that rows are defined as a percentage of the grid’s height, and columns are defined as a percentage of its width.

**Grid Template.**

The property grid-template can replace the previous two CSS properties. Both grid-template-rows and grid-template-columns are nowhere to be found in the following code!

.grid {

display: grid;

width: 1000px;

height: 500px;

grid-template: 200px 300px / 20% 10% 70%;

}

When using grid-template, the values before the slash will determine the size of each row. The values after the slash determine the size of each column. In this example, we’ve made two rows and three columns of varying sizes.

The same rules from before apply; when using percentages to set rows, each row will be a percentage of the grid’s total height. Columns are still a percentage of the total width.

Fraction.

You may already be familiar with several types of responsive units such as percentages (%), ems and rems. CSS Grid introduced a new relative sizing unit — fr, like fraction.

By using the fr unit, we can define the size of columns and rows as a fraction of the grid’s length and width. This unit was specifically created for use in CSS Grid. Using fr makes it easier to prevent grid items from overflowing the boundaries of the grid. Consider the code below:

.grid {

display: grid;

width: 1000px;

height: 400px;

grid-template: 2fr 1fr 1fr / 1fr 3fr 1fr;

}

In this example, the grid will have three rows and three columns. The rows are splitting up the available 400 pixels of height into four parts. The first row gets two of those parts, the second row gets one, and the third row gets one. Therefore the first row is 200 pixels tall, and the second and third rows are 100 pixels tall.

Each column’s width is a fraction of the available space. In this case, the available space is split into five parts. The first column gets one-fifth of the space, the second column gets three-fifths, and the last column gets one-fifth. Since the total width is 1000 pixels, this means that the columns will have widths of 200 pixels, 600 pixels, and 200 pixels respectively.

It is possible to use fr with other units as well. When this happens, each fr represents a fraction of the *available* space.

.grid {

display: grid;

width: 100px;

grid-template-columns: 1fr 60px 1fr;

}

In this example, 60 pixels are taken up by the second column. Therefore the first and third columns have 40 available to split between them. Since each gets one fraction of the total, they both end up being 20 pixels wide.

Repeat.

The properties that define the number of rows and columns in a grid can take a function as a value. repeat() is one of these functions. The repeat() function was created specifically for CSS Grid.

.grid {

display: grid;

width: 300px;

grid-template-columns: repeat(3, 100px);

}

The repeat function will duplicate the specifications for rows or columns a given number of times. In the example above, using the repeat function will make the grid have three columns that are each 100 pixels wide. It is the same as writing:

grid-template-columns: 100px 100px 100px;

Repeat is particularly useful with fr. For example, repeat(5, 1fr) would split your table into five equal rows or columns.

Finally, the second parameter of repeat() can have multiple values.

grid-template-columns: repeat(2, 20px 50px)

This code will create four columns where the first and third columns will be 20 pixels wide and the second and fourth will be 50 pixels wide.

**Minmax.**

So far, all of the grids that we have worked with have been a fixed size. The grid in our example has been 400 pixels wide and 500 pixels tall. But sometimes you might want a grid to resize based on the size of your web browser.

In these situations, you might want to prevent a row or column from getting too big or too small. For example, if you have a 100-pixel wide image in your grid, you probably don’t want its column to get thinner than 100 pixels! The minmax() function can help us solve this problem.

.grid {

display: grid;

grid-template-columns: 100px minmax(100px, 500px) 100px;

}

In this example, the first and third columns will always be 100 pixels wide, no matter the size of the grid. The second column, however, will vary in size as the overall grid resizes. The second column will always be between 100 and 500 pixels wide.

Grid Gap.

In all of our grids so far, there hasn’t been any space between the items in our grid. The CSS properties grid-row-gap and grid-column-gap will put blank space between every row and column in the grid.

.grid {

display: grid;

width: 320px;

grid-template-columns: repeat(3, 1fr);

grid-column-gap: 10px;

}

It is important to note that grid-gap does not add space at the beginning or end of the grid. In the example code, our grid will have three columns with two ten-pixel gaps between them.

Let’s quickly calculate how wide these columns are. Remember that using fr considers all of the *available* space. The grid is 320 pixels wide and 20 of those pixels are taken up by the two grid gaps. Therefore each column takes a piece of the 300 available pixels. Each column gets 1fr, so the columns are evenly divided into thirds (or 100 pixels each).

Finally, there is a CSS property grid-gap that can set the row and column gap at the same time. grid-gap: 20px 10px; will set the distance between rows to 20 pixels and the distance between columns to 10 pixels. Unlike other CSS grid properties, this shorthand does not take a / between values! If only one value is given, it will set the column gap and the row gap to that value.

**Grid Items.**

In this lesson, we have learned how to define a grid container. When explicitly defining a grid, you have to declare the quantity of rows and columns and their respective sizes.

In all of our examples, the items placed in the grid have always taken up exactly one square. This does not always need to be the case; we can drastically change the look of our grid by making grid items take up more than one row and one column. You can see this in the diagram to the right. Items A, B, C, and E span more than one row!

**Multiple Row Items.**

Using the CSS properties grid-row-start and grid-row-end, we can make single grid items take up multiple rows. Remember, we are no longer applying CSS to the outer grid container; we’re adding CSS to the elements sitting inside the grid!

.item {

grid-row-start: 1;

grid-row-end: 3;

}

In this example, the HTML element of class item will take up two rows in the grid, rows 1 and 2. The values that grid-row-start and grid-row-end accept are *grid lines*.

Row grid lines and column grid lines start at 1 and end at a value that is 1 greater than the number of rows or columns the grid has. For example, if a grid has 5 rows, the grid row lines range from 1 to 6. If a grid has 8 rows, the grid row lines range from 1 to 9.

The value for grid-row-start should be the row at which you want the grid item to begin. The value for grid-row-end should be one greater than the row at which you want the grid item to end. An element that covers rows 2, 3, and 4 should have these declarations: grid-row-start: 2 and grid-row-end: 5.

It is possible for the value of grid-row-start to be greater than that of grid-row-end. Both properties can also each have negative values. Consult the [documentation](https://developer.mozilla.org/en-US/docs/Web/CSS/grid-row-start) to learn more about how to use these features.

Grid Row.

We can use the property grid-row as shorthand for grid-row-start and grid-row-end. The following two code blocks will produce the same output:

.item {

grid-row-start: 4;

grid-row-end: 6;

}

.item {

grid-row: 4 / 6;

}

This code should look similar to the way grid-template is shorthand for grid-template-rowsand grid-template-columns. In this case, the starting row goes before the “/“ and the ending row goes after it. Again, the ending row is exclusive; this grid item will occupy rows four and five.

When an item spans multiple rows or columns using these properties, it will also include the grid-gap if any exists. For example, if an item spans two rows of height 100 pixels and there is a ten-pixel grid-gap, then the item will have a total height of 210 pixels.

Grid Column.

The previous three properties also exist for columns. grid-column-start, grid-column-end and grid-column work identically to the row properties. These properties allow a grid item to span multiple columns.

When using these properties, we can use the keyword span to start or end a column or row relative to its other end. Look at how span is used in the code below:

.item {

grid-column: 4 / span 2;

}

This is telling the item element to begin in column four and take up two columns of space. So item would occupy columns four and five. It produces the same result as the following code blocks:

.item {

grid-column: 4 / 6;

}

.item {

grid-column-start: 4;

grid-column-end: span 2;

}

.item {

grid-column-start: span 2;

grid-column-end: 6;

}

span is a useful keyword, because it avoids off-by-one errors (miscalculating the ending grid line) you might make when determining the ending grid line of an element. If you know where you want your grid item to start and how long it should be, use span!

**Grid Area**

We’ve already been able to use grid-row and grid-column as shorthand for properties like grid-row-start and grid-row-end. We can refactor even more using the property grid-area. This property will set the starting and ending positions for both the rows and columns of an item.

.item {

grid-area: 2 / 3 / 4 / span 5;

}

grid-area takes four values separated by slashes. The order is important! This is how grid-area will interpret those values.

1. grid-row-start
2. grid-column-start
3. grid-row-end
4. grid-column-end

In the above example, the item will occupy rows two and three and columns three through eight.

Using grid-area is an easy way to place items exactly where you want them in a grid.

**Review**

At this point, we’ve covered a great deal of different ways to manipulate the grid and the items inside it to create interesting layouts.

* grid-template-columns defines the number and sizes of the columns of the grid
* grid-template-rows defines the number and sizes of the rows of the grid
* grid-template is a shorthand for defining both grid-template-columns and grid-template-rows in one line
* grid-gap puts blank space between rows and/or columns of the grid
* grid-row-start and grid-row-end makes elements span certain rows of the grid
* grid-column-start and grid-column-end makes elements span certain columns of the grid
* grid-area is a shorthand for grid-row-start, grid-column-start, grid-row-end, and grid-column-end, all in one line

You have seen how to set up and fill in a grid and you now have one more CSS positioning technique to add to your toolkit! Let’s do some practice to solidify these skills.

**ADVANCED CSS GRID**

**Introduction.**

In the previous lesson, you learned all the foundational properties necessary to create a two-dimensional grid-based layout for your web pages! In this lesson, you’ll learn the following additional properties that you can use to harness the power of CSS Grid Layout:

* grid-template-areas
* justify-items
* justify-content
* justify-self
* align-items
* align-content
* align-self
* grid-auto-rows
* grid-auto-columns
* grid-auto-flow

You will also learn about the *explicit* and *implicit* grids and *grid axes*.

**Grid Template Areas**

The grid-template-areas property allows you to name sections of your web page to use as values in the grid-row-start, grid-row-end, grid-col-start,grid-col-end, and grid-area properties.

<div class="container">

<header>Welcome!</header>

<nav>Links!</nav>

<section class="info">Info!</section>

<section class="services">Services!</section>

<footer>Contact us!</footer>

</div>

.container {

display: grid;

max-width: 900px;

position: relative;

margin: auto;

grid-template-areas: "head head"

"nav nav"

"info services"

"footer footer";

grid-template-rows: 300px 120px 800px 120px;

grid-template-columns: 1fr 3fr;

}

header {

grid-area: head;

}

nav {

grid-area: nav;

}

.info {

grid-area: info;

}

.services {

grid-area: services;

}

footer {

grid-area: footer;

}

You may want to expand this section of the website to view the code above more clearly.

1. In the example above, the HTML creates a web page with five distinct parts.
2. The grid-template-areas declaration in the .container rule set creates a 2-column, 4-row layout.
3. The grid-template-rows declaration specifies the height of each of the four rows from top to bottom: 300 pixels, 120 pixels, 800 pixels, and 120 pixels.
4. The grid-template-columns declaration uses the fr value to cause the left column to use one fourth of the available space on the page and the right column to use three-fourths of the available space on the page.
5. In each rule set below .container, we use the grid-area property to tell that section to cover the portion of the page specified. The header element spans the first row and both columns. The nav element spans the second row and both columns. The element with class .info spans the third row and left column. The element with class .services spans the third row and right column. The footer element spans the bottom row and both columns.
6. That’s it! An entire page laid out in 40 lines of code.

This property is declared on grid containers.

**Overlapping Elements**

Another powerful feature of CSS Grid Layout is the ability to easily overlap elements.

When overlapping elements, it is generally easiest to use grid line names and the grid-area property.

<div class="container">

<div class="info">Info!</div>

<img src="#" />

<div class="services">Services!</div>

</div>

.container {

display: grid;

grid-template: repeat(8, 200px) / repeat(6, 100px);

}

.info {

grid-area: 1 / 1 / 9 / 4;

}

.services {

grid-area: 1 / 4 / 9 / 7;

}

img {

grid-area: 2 / 3 / 5 / 5;

z-index: 5;

}

In the example above, there is a grid container with eight rows and six columns. There are three grid items within the container — a <div> with the class info, a <div> with the class services, and an image.

The info section covers all eight rows and the first three columns. The services section covers all eight rows and the last three columns.

The image spans the 2nd, 3rd, and 4th rows and the 3rd and 4th columns.

The z-index property tells the browser to render the image element on top of the services and info sections so that it is visible.

**Justify Items**

We have referred to “two-dimensional grid-based layout” several times throughout this course.

There are two axes in a grid layout — the *column* (or block) axis and the *row* (or inline) axis.

The column axis stretches from top to bottom across the web page.

The row axis stretches from left to right across the web page.

In the following four exercises, we will learn and use properties that rely on an understanding of grid axes.

justify-items is a property that positions grid items along the inline, or row, axis. This means that it positions items from left to right across the web page.

justify-items accepts these values:

* start — aligns grid items to the left side of the grid area
* end — aligns grid items to the right side of the grid area
* center — aligns grid items to the center of the grid area
* stretch — stretches all items to fill the grid area

There are several other values that justify-items accepts, which you can read about on the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Box_Alignment_in_CSS_Grid_Layout#Justifying_Items_on_the_Inline_or_Row_Axis). The definitions for these values can also be found in the [documentation](https://developer.mozilla.org/en-US/docs/Web/CSS/justify-items#Values). It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

<main>

<div class="card">Card 1</div>

<div class="card">Card 2</div>

<div class="card">Card 3</div>

</main>

main {

display: grid;

grid-template-columns: repeat(3, 400px);

justify-items: center;

}

In the example above, we use justify-items to adjust the positioning of some elements on this web page.

1. There is a grid container with three columns that are each 400 pixels wide.
2. The container has three grid items that do not have a specified width.
3. Without setting the justify-items property, these elements will span the width of the column they are in (400 pixels).
4. By setting the justify-items property to center, the .card <div>s will be centered inside of their columns. They will only be as wide as necessary to contain their content (the words Card 1, etc).
5. If we specify a width for the .card elements, they will not stretch the width of their column.

This property is declared on grid containers.

Justify Content.

In the previous exercise, we learned how to position elements within their columns. In this exercise, we will learn how to position a grid within its parent element.

We can use justify-content to position the entire grid along the row axis.

It accepts these values:

* start — aligns the grid to the left side of the grid container
* end — aligns the grid to the right side of the grid container
* center — centers the grid horizontally in the grid container
* stretch — stretches the grid items to increase the size of the grid to expand horizontally across the container
* space-around — includes an equal amount of space on each side of a grid element, resulting in double the amount of space between elements as there is before the first and after the last element
* space-between — includes an equal amount of space between grid items and no space at either end
* space-evenly — places an even amount of space between grid items and at either end

There are several other values that justify-content accepts, which you can read about on the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Box_Alignment_in_CSS_Grid_Layout#Aligning_the_grid_tracks_on_the_block_or_column_axis). The definitions for these values can also be found in the [documentation](https://developer.mozilla.org/en-US/docs/Web/CSS/justify-content#Values). It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

<main>

<div class="left">Left</div>

<div class="right">Right</div>

</main>

main {

display: grid;

width: 1000px;

grid-template-columns: 300px 300px;

grid-template-areas: "left right";

justify-content: center;

}

1. In the example above, the grid container is 1000 pixels wide, but we only specified two columns that are 300 pixels each. This will leave 400 pixels of unused space in the grid container.
2. justify-content: center; positions the columns in the center of the grid, leaving 200 pixels on the right and 200 pixels on the left of the grid.

This property is declared on grid containers.

**Align Items.**

**In the previous two exercises, we learned how to position grid items and grid columns from left to right across the page. Below, we’ll learn how to position grid items from top to bottom!**

**align-items is a property that positions grid items along the block, or column axis. This means that it positions items from top to bottom.**

**align-items accepts these values:**

* **start — aligns grid items to the top side of the grid area**
* **end — aligns grid items to the bottom side of the grid area**
* **center — aligns grid items to the center of the grid area**
* **stretch — stretches all items to fill the grid area**

**There are several other values that align-items accepts, which you can read about on the** [**Mozilla Developer Network**](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Box_Alignment_in_CSS_Grid_Layout#Aligning_items_on_the_block_or_column_Axis)**. The definitions for these values can also be found in the** [**documentation**](https://developer.mozilla.org/en-US/docs/Web/CSS/justify-items#Values)**. It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.**

**<main>**

**<div class="card">Card 1</div>**

**<div class="card">Card 2</div>**

**<div class="card">Card 3</div>**

**</main>**

**main {**

**display: grid;**

**grid-template-rows: repeat(3, 400px);**

**align-items: center;**

**}**

**In the example above, we use align-items to adjust the positioning of some elements on this web page.**

1. **There is a grid container with three rows that are 400 pixels tall.**
2. **The container has three grid items that do not have a specified width.**
3. **Without setting the align-items property, these elements will span the height of the row they are in (400 pixels).**
4. **By setting the align-items property to center, the .card <div>s will be centered vertically inside of their rows. They will only be as tall as necessary to contain their content (the words Card 1, etc).**
5. **If we specify a height for the .card elements, they will not stretch the height of their row even if align-items: stretch; is set.**

**This property is declared on grid containers.**

**Align Content.**

In the previous exercise, we positioned grid items within their rows. align-content positions the rows along the column axis, or from top to bottom.

It accepts these positional values:

* start — aligns the grid to the top of the grid container
* end — aligns the grid to the bottom of the grid container
* center — centers the grid vertically in the grid container
* stretch — stretches the grid items to increase the size of the grid to expand vertically across the container
* space-around — includes an equal amount of space on each side of a grid element, resulting in double the amount of space between elements as there is before the first and after the last element
* space-between — includes an equal amount of space between grid items and no space at either end
* space-evenly — places an even amount of space between grid items and at either end

There are several other values that align-content accepts, which you can read about on the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Box_Alignment_in_CSS_Grid_Layout#Aligning_the_grid_tracks_on_the_block_or_column_axis). The definitions for these values can also be found in the [documentation](https://developer.mozilla.org/en-US/docs/Web/CSS/align-content#Values). It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

1. align-content: center; positions the rows in the center of the grid, leaving 100 pixels at the top a<main>

<div class="top">Top</div>

<div class="bottom">Bottom</div>

</main>

main {

display: grid;

height: 600px;

rows: 200px 200px;

grid-template-areas: "top"

"bottom";

align-content: center;

}

1. In the example above, the grid container is 600 pixels tall, but we only specified two rows that are 200 pixels each. This will leave 200 pixels of unused space in the grid container.
2. nd 100 pixels at the bottom of the grid.

This property is declared on grid containers.

Justify Self and Align Self.

The justify-items and align-items properties specify how all grid items contained within a single container will position themselves along the row and column axes, respectively.

justify-self specifies how an individual element should position itself with respect to the row axis. This property will override justify-items for any item on which it is declared.

align-self specifies how an individual element should position itself with respect to the column axis. This property will override align-items for any item on which it is declared.

They both accept these four properties:

* start — positions grid items on the left side/top of the grid area
* end — positions grid items on the right side/bottom of the grid area
* center — positions grid items on the center of the grid area
* stretch — positions grid items to fill the grid area (default)

align-self and justify-self accept the same values as align-items and justify-items. You can read about these values on the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Box_Alignment_in_CSS_Grid_Layout#Aligning_the_grid_tracks_on_the_block_or_column_axis). The definitions for these values can also be found in the [documentation](https://developer.mozilla.org/en-US/docs/Web/CSS/align-self#Values). It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

These properties are declared on grid items.

**Implicit vs. Explicit Grid.**

So far, we have been explicitly defining the dimensions and quantities of our grid elements using various properties. This works well in many cases, such as a landing page for a business that will display a specific amount of information at all times.

However, there are instances in which we don’t know how much information we’re going to display. For example, consider online shopping. Often, these web pages include the option at the bottom of the search results to display a certain quantity of results or to display ALL results on a single page. When displaying all results, the web developer can’t know in advance how many elements will be in the search results each time.

What happens if the developer has specified a 3-column, 5-row grid (for a total of 15 items), but the search results return 30?

Something called the *implicit* grid takes over. The implicit grid is an algorithm built into the specification for CSS Grid that determines default behavior for the placement of elements when there are more than fit into the grid specified by the CSS.

The default behavior of the implicit grid is as follows: items fill up rows first, adding new rows as necessary. New grid rows will only be tall enough to contain the content within them. In the next exercise, you’ll learn how to change this default behavior.

**Grid Auto Rows and Grid Auto Columns.**

CSS Grid provides two properties to specify the size of grid tracks added implicitly: grid-auto-rows and grid-auto-columns.

grid-auto-rows specifies the height of implicitly added grid rows. grid-auto-columns specifies the width of implicitly added grid columns.

grid-auto-rows and grid-auto-columns accept the same values as their explicit counterparts, grid-template-rows and grid-template-columns:

* pixels (px)
* percentages (%)
* fractions (fr)
* the repeat() function

In the e<body>

<div>Part 1</div>

<div>Part 2</div>

<div>Part 3</div>

<div>Part 4</div>

<div>Part 5</div>

</body>

body {

display: grid;

grid: repeat(2, 100px) / repeat(2, 150px);

grid-auto-rows: 50px;

}

xample above, there are 5 <div>s. However, in the section rule set, we only specify a 2-row, 2-column grid — four grid cells.

The fifth <div> will be added to an implicit row that will be 50 pixels tall.

If we did not specify grid-auto-rows, the rows would be auto-adjusted to the height of the content of the grid items.

These properties are declared on grid containers.

**Grid Auto Flow.**

In addition to setting the dimensions of implicitly-added rows and columns, we can specify the order in which they are rendered.

grid-auto-flow specifies whether new elements should be added to rows or columns.

grid-auto-flow accepts these values:

* row — specifies the new elements should fill rows from left to right and create new rows when there are too many elements (default)
* column — specifies the new elements should fill columns from top to bottom and create new columns when there are too many elements
* dense — this keyword invokes an algorithm that attempts to fill holes earlier in the grid layout if smaller elements are added

You can pair row and column with dense, like this: grid-auto-flow: row dense;.

This property is declared on grid containers.

**Review**

Great work! You have learned many new properties to use when creating a layout using CSS Grid! Let’s review:

* grid-template-areas specifies grid named grid areas
* grid layouts are two-dimensional: they have a row, or inline, axis and a column, or block, axis.
* justify-items specifies how individual elements should spread across the row axis
* justify-content specifies how groups of elements should spread across the row axis
* justify-self specifies how a single element should position itself with respect to the row axis
* align-items specifies how individual elements should spread across the column axis
* align-content specifies how groups of elements should spread across the column axis
* align-self specifies how a single element should position itself with respect to the column axis
* grid-auto-rows specifies the height of rows added implicitly to the grid
* grid-auto-columns specifies the width of columns added implicitly to the grid
* grid-auto-flow specifies in which direction implicit elements should be created.

**CSS Transitions.**

After a website is displayed, the visual appearances of various elements can change for many reasons. For example:

* Moving your mouse over a link may change the color or appearance of that link.
* Changing the size of the window may change the layout.
* Scrolling causes some elements to disappear and others to appear.

With *CSS transitions*, we can control how these changes happen.

These changes are a type of *state change*. CSS transitions allow us to control the timing of visual state changes. We can control the following four aspects of an element’s transition:

* Which CSS properties transition
* How long a transition lasts
* How much time there is before a transition begins
* How a transition accelerates

We’ll explore how different answers to each of these questions changes the animation. If any of these sound confusing, don’t worry! We will look at examples of each and learn how to use them together for a visually pleasing experience.

**Duration**

Duration

To create a simple transition in CSS, we must specify two of the four aspects:

1. The property that we want to transition.
2. The duration of the transition.

a {

transition-property: color;

transition-duration: 1s;

}

In the example above, transition-property declares which CSS property we will be animating, the text color. The second property, transition-duration, declares how long the transition will take — one second.

Many properties’ state changes can be transitioned. The type of transition depends on the property you choose. For a complete list of all animated properties, see [this resource](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_animated_properties).

Different properties transition in different ways, for example:

* Color values, like color and background-color, will blend to a new color.
* Length values like font-size, width, and height will grow or shrink.

Duration is specified in seconds or milliseconds, such as 3s, 0.75s, 500ms. The default value is 0s, or instantaneous, as if there is no transition.

When choosing a duration, think about how long actions take in real life. For example, a human eye blink takes around 400ms. People might expect the animation of clicking a button to be as sudden as a blink.

**Delay.**

Our next transition property is transition-delay. Much like duration, its value is an amount of time. Delay specifies the time to wait before starting the transition. As with the duration property, the default value for transition-delay is 0s, which means no delay.

transition-property: width;

transition-duration: 750ms;

transition-delay: 250ms;

In the example above, a change in the width of the element will start after a quarter of a second, and it will animate over three quarters of a second.

**Timing Function.**

The last transition property is transition-timing-function. The timing function describes the pace of the transition.

The default value is ease, which starts the transition slowly, speeds up in the middle, and slows down again at the end.

Other valid values include:

* ease-in — starts slow, accelerates quickly, stops abruptly
* ease-out — begins abruptly, slows down, and ends slowly
* ease-in-out — starts slow, gets fast in the middle, and ends slowly
* linear — constant speed throughout

transition-property: color;

transition-duration: 1s;

transition-timing-function: ease-out;

In the example above, the text color will be animated over one second. The timing function is ease-out which means it will begin abruptly and slow down as it ends.

If you’re interested in learning more about timing functions or seeing a full list of the possible values, we recommend [this resource](https://developer.mozilla.org/en-US/docs/Web/CSS/transition-timing-function) from the Mozilla Developer Network.

**Shorthand**

Now that we’ve explored each transition property, you may find yourself with many CSS rule sets that look like the code below.

transition-property: color;

transition-duration: 1.5s;

transition-timing-function: linear;

transition-delay: 0.5s;

Because these four properties are so frequently declared together, CSS provides a property that can be used to declare them all in one line: transition. This shorthand property describes each aspect of the transition puzzle in a single declaration. The properties must be specified in this order: transition-property, transition-duration, transition-timing-function, transition-delay.

transition: color 1.5s linear 0.5s;

In the example above, we have refactored the four lines of code in the previous example into one concise line. This example will cause any change in text color to transition at constant speed over 1.5 seconds, after a delay of 0.5 seconds.

Leaving out one of the properties causes the default value for that property to be applied. There is one exception: You must set duration if you want to define delay. Since both are time values, the browser will always interpret the first time value it sees as duration.

Combinations.

The shorthand transition rule has one advantage over the set of separate transition-<property> rules: you can describe unique transitions for multiple properties, and combine them.

To combine transitions, add a comma (,) before the semicolon (;) in your rule. After the comma, use the same shorthand syntax. For example:

transition: color 1s linear,

font-size 750ms ease-in 100ms;

The above code transitions two properties at once. The text color transitions over one second with linear timing and no delay. At the same time, the font size transitions over 750 milliseconds with an ease-in timing and a 100 millisecond delay. This “chaining” is a powerful tool for expressing complicated animations.

**All.**

Even with the shorthand, specifying transitions for many properties can be tedious. It is common to use the same duration, timing function, and delay for multiple properties. When this is the case you can set the transition-property value to all. This will apply the same values to all properties. To effect this, you can use all as a value for transition-property.

all means every value that changes will be transitioned in the same way. You can use all with the separate transition properties, or the shorthand syntax. This allows you to describe the transition of many properties with a single line:

transition: all 1.5s linear 0.5s;

In this example, any change will be animated over one and a half seconds after a half-second delay with linear timing.

**Review.**

*CSS Transitions* are a powerful tool for providing visual feedback to users. We’ve learned a lot about transitions, so let’s review:

CSS Transitions have 4 components:

* A *property* that will transition.
* The *duration* which describes how long the transition takes.
* The *delay* to pause before the transition will take place.
* The *timing function* that describes the transition’s acceleration.

A simple transition can be described with a property and a duration, which can be written like this:

transition-property: color;

transition-duration: 1s;

Many properties’ *state changes* can be transitioned, including color, background color, font size, width, and height. all is also a valid transition property that causes every changing property to transition.

The shorthand property transition can be used to describe all four components of a transition at once. By using the comma (,) operator, many transitions can be described in one CSS rule.

If you want to learn more about CSS Transitions, we recommend [this resource](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Transitions/Using_CSS_transitions).

Good work, you now have the tools to make your web pages come to life!