## **Programming Fundamentals**

* **The Web.** We'll talk about what the Web is and—at a basic level—how it works.
* **Tools for Editing.** You'll get practice editing web pages using a text editor and testing your work in a browser.
* **HTML.** You'll learn the fundamentals of HTML, the computer language we use to structure and write webpages.
* **URLs.** You'll also learn about URLs, the addresses that we use to refer to documents and sites on the web.

You'll put all of this together by taking notes on what you've learned and turning those notes into a web page, which you'll submit for your first lab.

* **Syntax.** Grammar rules of a computer language.
* **Formalism.** Computers take code literally, and will only do exactly what you tell it to do.
* **Nesting.** Bits of code going inside other bits of code in an orderly way.
* **There's a lot of help!** Make sure to use tools like documentation, or your favorite search engine if you need to.
* **Text files** contain:
  + Letters
  + Numbers
  + Symbols
  + Spaces
* **HTML** is written in text.
  + The layout of a **web page** seen by the user is put together by the web browser based on the code in the text file.

## **Installing a text editor (optional)**

If you don't have a text editor installed on your computer already, you may want to download and install one now. **This is entirely optional.** If you don't want to (or can't) install a text editor on your computer, don't worry—we will also provide a workspace right here in the Udacity classroom that has its own built-in text editor.

If you do wish to install a text editor, here are a couple we recommend:

* [**VS Code(opens in a new tab)**](https://code.visualstudio.com/) is a very popular editor created at Microsoft. It is available for Windows, Mac, and Linux.

Practice: The div Element

In the video on block and inline elements, you may (or may not) have noticed that one of the block-level elements Kelly mentioned was called div.

The div is short for **division**, because that's what this element is for—you can use it to *divide* up the page into different sections.

Like the paragraph p element, the division div element has an invisible box around it—and just like p, it can have a border, a margin, a width, a height, and so on.

But a p element is specifically meant to contain text. In contrast, the div element is a *generic* container for whatever other elements you might want to put inside. You can use the div element to organize the content and divide the page into sections.

That's a bit abstract. It might help to give it a try in the workspace below.

Block elements

Block elements are used for large sections of text, such as paragraphs, headlines, or lists; and also for some other features such as video players and tables.

A block element creates a (usually invisible) box in the browser display. By default, this box takes up the full width of the display. The beginning of a block always starts on a new line in the display.

Most block elements have a particular way they are displayed by default: paragraphs have margins around them; lists have bullet-points or numbered items; headlines are printed in large text. There is also a generic block element, div, which has no special defaults.

* p — Paragraph. Text in a paragraph is separated visually from other paragraphs by a small margin.
* ul and ol — Unordered and ordered lists. By default, ul lists are displayed with bullet points, and ol lists with numbered items.
* li — List items inside a ul or ol list. The li element has to be nested inside a ul or ol list; it can't occur on its own.
* Section headers, from h1 (largest) to h6 (smallest). Used for headlines, section titles, and the like.
* div — A logical *div*ision of a page or document. Other block elements such as paragraphs, lists, and headers can be nested inside a div.

You will see the div element much more in the next lesson. Because they don't have any default display settings, divs are heavily used with custom styling with CSS.

Inline elements

Inline elements do not create a full-width box on the display. They modify the display of text, or insert other things into the text — such as line breaks, images, or hyperlinks.

* em and strong — Emphasis. By default, text inside an em is displayed as *italic*, and text in strong is displayed as **boldface.**
* br — Line break. *(empty)* A line break does not create a new paragraph; it only marks the end of a line.
* sub and sup — Subscript and superscript. Useful for math and chemistry: I have x3+2x2 molecules of H2O.
* mark — Highlighting. Not very often used, but it's kind of cool.

Some of the inline elements you've seen require **attributes**, extra information besides the name of the element itself. Attributes are written inside the opening tag of the element.

* img — Images. Needs a src attribute with a URL, and an alt attribute with descriptive text.
* a — Hyperlinks. Needs an href attribute with a URL.

Images

The syntax for the img tag is like this:

<img src="Image URL here" alt="A description of the image">

The URL of an image may be an *absolute* URL, such as http://placebear.com/800/600, or it may be a *relative* URL such as images/wolves.jpg.

The alt text is used if the image can't be loaded, or if the user can't see images — such as if the user is using a [**screen reader(opens in a new tab)**](https://en.wikipedia.org/wiki/Screen_reader).

Links

Hyperlinks allow the user to navigate from one page to another. They are created using the a element. The destination of the link is written in an attribute called href; the link text appears as the contents of the a element. Here's an example:

<a href="https://en.wikipedia.org/wiki/Hypertext" target="\_blank">

Wikipedia's "Hypertext" article

</a>

**CSS**

CSS Rulesets

A CSS **ruleset** is made up of a **selector** and a **declaration block**.

Selectors

Selectors indicate which HTML elements the ruleset will apply to. For example, consider this ruleset:

li {

color: green;

}

In this example, li is the selector—meaning that whatever we put between the curly braces { } will be applied to all list item li elements on the page.

Declaration Blocks

A declaration block describes how the ruleset will modify the elements. In the above example, the declaration block is saying that all li elements will be modified by changing their font colors to green.

A declaration block can have multiple declarations inside of it. For example, consider this ruleset:

p {

background-color: blue;

font-size: 20px;

}

The declaration block is everything other than the selecter p:

{

background-color: blue;

font-size: 20px;

}

And this declaration block contains *two* individual declarations. One is:

background-color: blue;

And the other is

font-size: 20px;

Properties and Values

You can break the syntax of a declaration down even further. A declaration is made up of a **property** and a **value**. For example, in this declaration…

background-color: blue;

…the property would be background-color and the value would be blue.

CSS Selectors: Class and ID

The distinction between a class and an ID can be a bit confusing, so let's review them both.

Class

A **class** is a group of things with the same characteristics. To create a class, you need to use the attribute "class" in the HTML opening tag. Then in your CSS, you use the same name you gave the class attribute in your HTML, except you place a . in front of it.

Example:

//HTML

<div class="container">

<p> I'm inside a container! </p>

</div>

//CSS

.container {

border: 1px solid black;

}

In this example, the class name is container. Any element with the class attribute container will receive the styling of border: 1px solid black;. So for example, if you added class="container"to the paragraph element, that would also get this same styling.

ID

An **ID** is when you'd apply characteristics to *one* element. To create an ID, you need to use the attribute id in the HTML opening tag. Then in your CSS, you use the same name you gave the id attribute in your HTML, except you place a # in front of it.

Example:

//HTML

<h1 id="main-heading"> Welcome to My Page! </h1>

//CSS

#main-heading {

background-color: orange;

}

If you want to apply a style to more than one element, you should *always* use a class. You should *only* use an ID to style *one* element. IDs are unique.

Stylesheet *or* style element — not both!

Note that you can put your CSS code inside of a <style></style> element, like this ...

<style>

p{color:blue;}

</style>

... *or* inside of a linked *stylesheet*, like we just talked about.

But you would not want to do both at the same time! In other words, when you place your CSS in a stylesheet, you don't need to use a style element—you can just put the CSS directly into the stylesheet with nothing around it:

p{color:blue;}

This is what you should do for the exercise below—simply put the CSS directly into your style.css file (and *don't* use the style element).

Linking stylesheets

To link to a stylesheet in your HTML file, add a link element to the head of the HTML file. The syntax for the link element is just like this:

<link rel="stylesheet" href="style.css">

If you are linking to a stylesheet located on another web server, you will use a full URL in the href attribute. If you're linking to one that's in the same directory as your HTML file, you can just use the filename as a relative URL.

Creating a CSS file

You might think that creating a CSS file requires something special, but all you have to do is make a new text file (just plain old text!) and rename it with the extension .css. This is similar to how we create our HTML files—they are simply text files that have a .html file extension in the name.

For example, in the workspace below, you can hover over the plus button and select **New File**—and then simply name the file styles.css.

Font vs Typeface

If you have worked in a word processor (like Microsoft Word, for example), you may have seen things like *Times New Roman*, *Courier*, *Arial*, or *Helvetica*. For example:

This Text is in Times New Roman

In word processors (and everyday speech) we would call this the "Times New Roman font". But technically, "Times New Roman" is not a font—it is a **typeface**.

Wait, what?

Most typefaces have multiple versions that look a little different from one another. For example, we have:

*Times New Roman italic*

And also:

**Times New Roman bold**

These are all examples of the same **typeface**. but each one is a different version or **font**.

A **typeface** is a group of related fonts.  
A **font** is a specific version of a typeface.

Font Families

Another term we can use for a typeface (or group of related fonts) is a **font family**. That's the term that CSS uses.

In other words, to change the typeface of our text, we can use the font-family property. Like this:

font-family: Helvetica;

Or here's another example:

font-family: "Times New Roman";

**Note:** You might notice that one of these has quotes " " while the other does not. The quotes are recommended for font families that have spaces in the name. They help ensure that the name is read as one thing ("Times New Roman"), rather than potentially three separate things (Times, New, and Roman). The spaces don't usually cause a problem, so this is a recommended practice, not a requirement (you can leave them off and it will typically still work).

In the workspace below, you'll find all of the code that Kelly started with in the video. Go ahead and try applying the two flexbox properties that Kelly demonstrated (display and flex-wrap).

Note that these get added to the container element, not the inner boxes.

You should end up with a horizontal row of boxes that wraps when the browser window is resized.

.container{

width: 100%;

display: flex;

flex-wrap: wrap;

}

.box{

width: 100px;

height: 100px;

text-align: center;

font-size: 30px;

font-weight: bold;

font-family: sans-serif;

}

.red{

background-color: red;

}

.green{

background-color: green;

}

.yellow{

background-color: yellow;

}

Ruleset Syntax

The basic syntax of a CSS ruleset has two parts: a *selector*, and a group of *rules*, each of which consists of a *property* name and the *value* of that property.

selector {

property: value;

}

The selector is written first, and then the rules are written inside { curly brackets }. Each rule's property and value are separated by a : colon, and the rule always ends with a ; semicolon.

Selectors

The selector indicates which HTML elements the rule will apply to. You've seen a few different sorts of selector: the *element* selector, the *class* selector, the *id* selector, and the *descendant* selector.

A **type selector** applies to every HTML element of a particular type, such as p or em. This selector will apply to every p element:

p {

color: blue;

}

A **class selector** applies to all elements that share a class attribute. The class selector is written starting with a . (dot):

.narrow {

width: 20%;

}

In order for the class selector to apply, there have to be HTML elements on the page that use that class attribute:

<div class="narrow">

This will get the 20% width.

</div>

An **id selector** applies to an element with a particular id attribute. The id selector is written starting with a # sign:

#sidebar {

background-color: lightgray;

width: 20%;

float: left;

}

Within an HTML page, there should be only one element with that id attribute value.

<div id="sidebar">

This will get the background, width, and float values from the sidebar CSS rule.

</div>

A **descendant selector** is a compound of two simpler selectors. It applies only to an inner element that is a descendant (on the DOM tree) of a particular outer element.

li a {

color: pink;

}

The above selector will apply to a elements (hyperlinks), but only those inside an li element (list item):

<ul>

<li> <a href="https://www.udacity.com/"> Pink Udacity </a>

</ul>

<p> <a href="https://www.google.com/"> Non-pink Google </a>

Rules

A ruleset can be composed of several rules, each of which applies a particular *value* to a *property* of the selected elements. Properties are things such as the color, position, size, and shape of the element.

h1 { color: red; font-size: larger; }

This rule applies the value red to the property color, and the value larger to the property font-size.

Some properties allow values that are more than one word long, such as the font property:

body { font: 12pt bold Consolas, Monaco, monospace; }

Font stacks

The font-family and font properties allow you to specify a *font stack*, a list of font options separated by , commas. The browser will use the first font in the stack that is available on the user's system. Usually the last font in the stack should be a generic font name, such as serif, sans-serif, or monospace.

Colors

There are several ways to specify a color in CSS. Three common ones are hex codes, rgb triples, and color names.

.orange {

color: #ff9900;

}

.pink {

color: **rgb**(100%, 80%, 80%);

}

.chartreuse {

color: chartreuse;

}

Flexbox

To change the browser's layout from the default document-based layout to the flexible box layout, set display: flex on a container element (one that has other elements inside it).

.outer {

display: flex;

border: 2px dotted orange;

}

.inner {

width: 100px;

border: 1px solid black;

padding: 10px;

}

Flexbox can be heavily customized! The above will cause .inner HTML elements to be packed in a row within the .outer element:

JAVA SCRIP

const price = 15.00;

const money = 20.00;

if (money >= price) {

console.log("buy the hammer");

} else {

console.log("don't buy the hammer");

}

[MDN Web Docs - if else](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else)

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else

Logical operators

Logical operators can be used in conjunction with boolean values (true and false) to create complex logical expressions.

By combining two boolean values together with a logical operator, you create a *logical expression* that returns another boolean value. Here’s a table describing the different logical operators:

| Operator | Meaning | Example | How it works |
| --- | --- | --- | --- |
| && | Logical AND | value1 && value2 | Returns true if both value1 and value2 evaluate to true. |
| || | Logical OR | value1 || value2 | Returns true if either value1 or value2 (or even both!) evaluates to true. |
| ! | Logical NOT | !value1 | Returns the opposite of value1. If value1 is true, then !value1 is false. |

By using logical operators, you can create more complex conditionals like Julia’s weekend example.

**Logical AND and OR**

**Truth Tables**

&& (AND)

| **A** | **B** | **A && B** |
| --- | --- | --- |
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |
|  |  |  |

|| (OR)

| **A** | **B** | **A || B** |
| --- | --- | --- |
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

**Truth tables** are used to represent the result of all the possible combinations of inputs in a logical expression. A represents the boolean value on the left-side of the expression and B represents the boolean value on the right-side of the expression.

Truth tables can be helpful for visualizing the different outcomes from a logical expression. However, do you notice anything peculiar about the truth tables for logical AND and OR?

**Intro to Loops**

Loops Repeat Blocks of Code

Conditional statements are one way to control the flow of code -- if a certain condition is true, execute this block of code, otherwise, execute that other block of code.

Loops are another way to control the flow of code by allowing us to execute a block of code multiple times.

What We Will Cover in This Lesson

You will learn how to:

* Use while loops
* Use for loops
* Nest loops for more complex automation
* Use assignment operators to write more concise code

Along the way we'll give you a lot of practice writing loops.

Functions Are Awesome!

The ability to generalize code for a variety of possible inputs is a powerful tool when creating easy to understand, non-repetitive code.

**function** **reverseString**(reverseMe) {

**let** reversed = "";

**for** (**let** i = reverseMe.length - 1; i >= 0; i--) {

reversed += reverseMe[i];

}

**return** reversed;

}

**console**.**log**(**reverseString**("Julia"));

**Prints** "ailuJ"

* The function has one ***parameter*** -- a variable named reverseMe.
* reverseMewill store the ***argument*** -- the value of the string that we want the function to operate on.
* The variable reversed is intialized as an empty string. It will be used to store the reversed string as as it is being constructed.
* The function loops through each character the reverseMe string using string indexes, from the end to the beginning and adds each character to reversed.
* When the loop is complete, reversed is returned.

## **How to Declare a Function**

**Functions** allow you to package up lines of code that you can use (and often reuse) in your programs.

Sometimes they take **parameters** like the pizza button from the beginning of this lesson. reheatPizza() had one parameter: the number of slices.

**function** **reheatPizza**(numSlices) {

*// code that figures out reheat settings!*

}

The reverseString() function that you saw also had one parameter: the string to be reversed.

**function** **reverseString**(reverseMe) {

*// code to reverse a string!*

}

In both cases, the parameter is listed as a variable after the function name, inside the parentheses. And, if there were multiple parameters, you would just separate them with commas.

**function** **doubleGreeting**(name, otherName) {

*// code to greet two people!*

}

But, you can also have functions that don't have any parameters. Instead, they just package up some code and perform some task. In this case, you would just leave the parentheses empty. Take this one for example. Here's a simple function that just prints out "Hello!".

*// accepts no parameters! parentheses are empty*

**function** **sayHello**() {

**const** message = "Hello!"

**console**.**log**(message);

}

If you tried pasting any of the functions above into the JavaScript console, you probably didn't notice much happen. In fact, you probably saw undefined returned back to you. undefined is the default return value on the console when nothing is explicitly returned using the special return keyword.

## **How to Run a function**

Now, to get your function to do something, you have to **invoke** or **call** the function using the function name, followed by parentheses with any **arguments** that are passed into it. Functions are like machines. You can build the machine, but it won't do anything unless you also turn it on. Here's how you would call the sayHello() function from before, and then use the return value to print to the console:

*// declares the sayHello function*

**function** **sayHello**() {

**const** message = "Hello!"

**return** message; *// returns value instead of printing it*

}

*// function returns "Hello!" and console.log prints the return value*

**console**.**log**(**sayHello**());

**Prints:** "Hello!"

Parameters vs. Arguments

At first, it can be a bit tricky to know when something is either a parameter or an argument. The key difference is in where they show up in the code. A **parameter** is always going to be a *variable* name and appears in the function declaration. On the other hand, an **argument** is always going to be a *value* (i.e. any of the JavaScript data types - a number, a string, a boolean, etc.) and will always appear in the code when the function is called or invoked.

Try declaring and calling some functions on your own:

What You’ve Learned So Far:

* **Functions** package up code so you can easily use (and reuse) a block of code.
* **Parameters** are variables that are used to store the data that's passed into a function for the function to use.
* **Arguments** are the actual data that's passed into a function when it is invoked:

What You’ve Learned So Far:

* If an identifier is declared in **global scope**, it's available *everywhere*.
* If an identifier is declared in **function scope**, it's available *in the function* it was declared in (even in functions declared inside the function).
* If an identifier is declared in **block scope** with var, it is available *in the block and in the outer scope of the block* it was declared in.
* If an identifier is declared in **block scope** with let or const, it is only available *in the block* it was declared in.
* When trying to access an identifier, the JavaScript Engine will first look in the current function. If it doesn't find anything, it will continue to the next outer function to see if it can find the identifier there. It will keep doing this until it reaches the global scope.
* Global identifiers are a bad idea. They can lead to bad variable names, conflicting variable names, and messy code.

**Arrays**

Arrays Allow Us to Store Lists of Data

In this lesson we'll learn about:

* How to create arrays
* How to use arrays
* How arrays are structured

And we'll be talking a lot about donuts and how to keep track of all of the donuts in our shop!

**Why are we using const instead of let when we declare arrays?**

Another great question!

A simple way to think about the difference between let and const is that we use let when we anticipate that the value of a variable will change and const when we think it will be constant -- but that shorthand doesn't tell the whole story. The difference between let and const is not so much about ***change*** but about ***reassignment***

* let allows you to reassign the variable
* const doesn't allow you to reassign the variable

The decision about whether we need to be able to reassign the variable is based on what type of variable it is and how JavaScript stores those values.

**Primitive vs. Object Types in JavaScript**

***String***, ***Number***, ***Boolean***, ***undefined*** and ***null*** are considered [***Primitive Types*(opens in a new tab)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_values) in JavaScript. These relatively simple data types represent just one value which makes it easy for JavaScript to store that value. So when you assign a primitive value to a variable, JavaScript actually assigns that value.

***Arrays*** are more complicated because they consist of a list of values which makes storage much more complicated. Arrays are actually [***Object types*(opens in a new tab)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects) which means that instead of assigning all of the values of the list to the array, JavaScript simply assigns a *reference to where to find the values* in the array. Even if the values inside the object change, the reference address doesn't.

Here's an analogy that might help. Think of a JavaScript array as if it were a house. The house has a group of people who live inside it. If those people move out, and a new group of people moves in, the names of the people inside the house changes, but the house's postal address won't.

Arrays are Powerful Data Structures

Arrays have a number of properties and built-in methods.

Array properties includelengthwhich is similar to the length method for String types.

Common Array methods include:

* reverse: reverses the order of the elements in an array
* sort: sorts the elements in an array
* push: adds elements to an array
* pop: removes elements from an array

To see all of the properties and built-in methods for modifying arrays and accessing values in an array, check out the [**MDN Documentation(opens in a new tab)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

***TIP***: You can type []. into the JavaScript console for a list of all the available Array methods.

Array.length

You can find the **length** of an array by using its length property.

**const** donuts = ["glazed", "powdered", "sprinkled"];

**console**.**log**(donuts.length);

**Prints:** 3

To access the length property, type the name of the array, followed by a period . (you’ll also use the period to access other properties and methods), and the word length. The length property will then return the **number of elements** in the array.

**TIP:** Strings have a length property too! You can use it to get the length of any string. For example, "supercalifragilisticexpialidocious".length returns 34.

Pop

Alternatively, you can use the pop() method to remove elements from the *end of an array*.

**const** donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller", "cinnamon sugar", "sprinkled", "powdered"];

donuts.**pop**(); *// pops "powdered" off the end of the `donuts` array*

donuts.**pop**(); *// pops "sprinkled" off the end of the `donuts` array*

donuts.**pop**(); *// pops "cinnamon sugar" off the end of the `donuts` array*

**Returns:** "cinnamon sugar"  
**donuts array:** ["glazed", "chocolate frosted", "Boston creme", "glazed cruller"]

Splice

splice() is another handy method that allows you to add and remove elements from anywhere within an array.

While push() and pop() limit you to adding and removing elements from *the end of an array*, splice() lets you specify the index location to add new elements, as well as the number of elements you'd like to delete (if any).

**const** donuts = ["glazed", "chocolate frosted", "Boston creme", "glazed cruller"];

donuts.**splice**(1, 1, "chocolate cruller", "creme de leche"); *// removes "chocolate frosted" at index 1 and adds "chocolate cruller" and "creme de leche" starting at index 1*

**Returns**: ["chocolate frosted"]  
**donuts** array after calling the splice() method: ["glazed", "chocolate cruller", "creme de leche", "Boston creme", "glazed cruller"]

**Following is the syntax of** **splice()** **method**: arrayName.splice(arg1, arg2, item1, ....., itemX); where,

* arg1 = Mandatory argument. Specifies the starting index position to add/remove items. You can use a negative value to specify the position from the end of the array e.g., -1 specifies the last element.
* arg2 = Optional argument. Specifies the count of elements to be removed. If set to 0, no items will be removed.
* item1, ....., itemX are the items to be added at index position arg1

splice() method returns the item(s) that were removed.

Array Methods for Looping

Arrays have a set of special methods to help you iterate over and perform operations on collections of data. You can view the [**MDN Documentation list of Array methods(opens in a new tab)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), but a couple big ones to know are the forEach() and map() methods.

The forEach Method

The forEach() method gives you an alternative way to iterate over an array, and manipulate each element in the array with an inline function expression.

**const** donuts = ["jelly donut", "chocolate donut", "glazed donut"];

donuts.**forEach**(**function**(donut) {

donut += " hole";

donut = donut.**toUpperCase**();

**console**.**log**(donut);

});

**Prints:**

JELLY DONUT HOLE

CHOCOLATE DONUT HOLE

GLAZED DONUT HOLE

Notice that the forEach() method iterates over the array without the need of an explicitly defined index. In the example above, donut corresponds to the element in the array itself. This is different from a for or while loop where an index is used to access each element in the array:

**for** (**let** i = 0; i < donuts.length; i++) {

donuts[i] += " hole";

donuts[i] = donuts[i].**toUpperCase**();

**console**.**log**(donuts[i]);

}

forEach Syntax

The function that you pass to the forEach() method can take up to three parameters. We have named them element, index, and array, but you can name them whatever you like.

**const** myArray = ["A", "B", "C", "D"];

**function** **myAwesomeFunction**(element, index, array) {

**console**.**log**("Element: ", element);

**console**.**log**("Index: ", index);

**console**.**log**("Array: ", array);

}

myArray.**forEach**(myAwesomeFunction);

The forEach() method will call this function once *for each* element in the array (hence the name forEach.) Each time, it will call the function with different arguments. The element parameter will get the *value* of the array element. The index parameter will get the *index* of the element (starting with zero). The array parameter will get a reference to the whole array, which is handy if you want to modify the elements.

HHere's another example:

words = ["cat", "in", "hat"];

words.**forEach**(**function**(word, num, all) {

**console**.**log**("Word " + num + " in " + all.**toString**() + " is " + word);

});

**Prints:**  
Word 0 in cat,in,hat is cat\  
Word 1 in cat,in,hat is in\  
Word 2 in cat,in,hat is hat

***Did you know that...*** the toString() method converts and returns the value of an object or a variable into a string? For example, if you have a number or an array and you want to turn it into a readable string or manipulate it as text, you can use this method to achieve that.

On the next page, you'll do a quiz that uses the forEach() method to modify an array.

OPJECTS

Objects are Powerful!

Objects allow us to wrap up pieces of related data and functionality into one single container.

Arrays are a special type of object in JavaScript with methods including length() , .reverse() , sort() , push(), and pop() -- but objects are much more than arrays. JavaScript includes a lot of other built-in objects.

In this lesson, we'll learn all about how we can create our own objects that contain both data and functions that we can use to manipulate or operate on that data. By the end of this lesson you will be able to:

* Create a JavaScript object
* Use dot notation and bracket notation to access object elements
* Use typeof to access the type of a variable
* Add and access methods in JavaScript objects
* Use JavaScript naming conventions for property names

Let's get started!

Object-literal notation

**const** sister = {

name: "Sarah",

age: 23,

parents: [ "Alice", "Andy" ],

siblings: ["julia"],

favoriteColor: "purple",

pets: true

};

The syntax you see above is called **object-literal notation**. There are some important things you need to remember when you're structuring an object literal:

* The "key" (representing a **property** or **method** name) and its "value" are separated from each other by a **colon**
* The key: value *pairs* are separated from each other by **commas**
* The entire object is wrapped inside curly braces { }.

And, kind of like how you can look up a word in the dictionary to find its definition, the key in a key:value pair allows you to look up a piece of information about an object. Here's are a couple examples of how you can retrieve information about my sister's parents using the object you created.

*// two equivalent ways to use the key to return its value*

sister["parents"] *// returns ["Alice", "Andy" ]*

sister.parents *// also returns ["Alice", "Andy"]*

Using sister["parents"] is called **bracket notation** (because of the brackets!) and using sister.parents is called **dot notation** (because of the dot!).

**It's All About the DOM!**

In this lesson, we'll learn all about the Document Object Model a.k.a. the DOM.

We'll start with a discussion what the DOM is and how it gets created.

Next, we'll learn several different JavaScript methods for programmatically accessing DOM elements:

* .getElementById()
* .getElementsByClassName()
* .getElementsByTagName()
* .querySelector()
* .querySelectorAll()

We'll also learn about the web interfaces that facilitate this access, including:

* Element
* Node

We have a lot to cover in this lesson. Let's get started!

**Naming Best Practices**

* Don't use a number as the first character in a property name
* Don't use quotes around key names unless absolutely necessary
* Use camelCase when you need a multi-word variable name.

These naming conventions also apply to regular variable names too!

The DOM

In this section, we'll look at the **Document Object Model** - otherwise known as the DOM.

The words "the DOM" are used all over developer documentation sites and tutorials on writing interactive JavaScript code. But what is it? Perhaps you've even used the DOM and still aren't quite sure what it is. Is it the browser? Is it a special part of JavaScript? ¯\\_(ツ)\_/¯

Translating HTML to the DOM

When you request a website, no matter what backend language is powering that website, it will respond with HTML. This response is based on the HTML specification which contains a specific set of rules for how browsers should process the data they recieve.

The process works in this order:

1. HTML is received
2. HTML tags are converted to tokens
3. Tokens are converted to Nodes
4. Nodes are converted to the DOM

**The Details**

The browser receives a stream of HTML. The bytes are run through a complicated (but fully documented) parsing process that determines the different characters (e.g. the start tag character <, an attribute like href, a closing angle bracket like >). After parsing has occurred, a process called **tokenization** begins. Tokenization takes one character at a time and builds up **tokens**. The tokens are:

* DOCTYPE
* start tag
* end tag
* comment
* character
* end-of-file

Let's take a break for a second. At this state, the browser has received the bytes that've been sent by a server. The browser has converted the bytes to tags and has read through the tags to create a list of tokens.

This list of tokens then goes through the tree construction stage. The output of this stage is a tree-like structure - this is the DOM!

Two important quotes from Illya in the video:

a tree structure that captures the content and properties of the HTML and all the relationships between the nodes

the DOM is the full, parsed representation of the HTML

So the DOM is a *model* (representation) of the relationships and attributes of the HTML document that was received. Remember that DOM stands for "Document Object Model". Something that I've found to be true as I've been learning is that to break something down, just read the thing backwards:

Document Object Model

...would become…

Object Model of the Document!

Remember that a JavaScript object is a tree-like structure that has properties and values. So the DOM can be accessed using a special object provided by the browser: document

Exploring the DOM

Try this:

1. Open the console on this page
2. Type out the word document
   * careful not to declare it (const document)
   * careful not to wrap it in quotes ("document")
3. Press enter

The document object is provided by the browser and is a representation of the HTML document. This object is *not* provided by the JavaScript language. ECMAScript is the language specification that JavaScript is based on, and it only references the document object model in one place, in its "Global Object" section:

In addition to the properties defined in this specification the global object may have additional host defined properties. This may include a property whose value is the global object itself; for example, in the HTML document object model the window property of the global object is the global object itself. ([**source(opens in a new tab)**](https://www.ecma-international.org/ecma-262/#sec-global-object))

Basically, this says that the document object is not part of JavaScript, but is expected to *already exist* and be freely accessible to JavaScript code.

The DOM is standardized by the W3C. There are a number of specifications that make up the DOM, here are few:

* Core Specification
* Events Specification
* Style Specification
* Validation Specification
* Load and Save Specification

To see the full list of DOM specs, check out the standard at: [**https://www.w3.org/standards/techs/dom#w3c\_all**](https://www.w3.org/standards/techs/dom#w3c_all)

Selecting Multiple Elements At Once Recap

In this section, we learned two ways to select multiple DOM elements:

* .getElementsByClassName()
* .getElementsByTagName()

There are a few important things to keep in mind about these two methods:

* both methods use the document object
* both return multiple items
* the list that's returned is an HTMLCollection, not an array

*// select all elements that have the class "accent-color"*

document.**getElementsByClassName**('accent-color');

*// select all "span" elements*

document.**getElementsByTagName**('span');

Quick Recap and a Bit of History

We've been looking at these methods:

* .getElementById()
* .getElementsByClassName()
* .getElementsByTagName()

Now these DOM methods are standardized. However, not all browsers support every standard. They do *now*, for these three methods, but there are hundreds of other methods with varying levels of support.

That's why almost every method on MDN has a Browser compatibility table that lists when each browser started supporting that specific method.

You've Learned A Lot!!!

In this lesson, we learned all about the DOM (Document Object Model).

We started by learning what the DOM is and how it gets created.

Then we discovered several JavaScript methods for programmatically accessing DOM elements:

* .getElementById()
* .getElementsByClassName()
* .getElementsByTagName()
* .querySelector()
* .querySelectorAll()

Along the way we talked about the web interfaces that facilitate this access, including:

* Element
* Node

Up next, we'll learn new DOM methods that allow us to alter and control page content!

**Intro to Working with Browser Events**

Update Existing Content Recap

In this section, we looked at multiple ways to change page content:

* .innerHTML
* .textContent
* .innerText

We saw that to set HTML content for an element, out of the three properties list above, we can only use .innerHTML. Using .textContent will erroneously include the HTML characters as plain text inside the element.

We also looked at the difference between .textContent and .innerText. .textContent completely ignores any CSS styling and returns all of the element's HTML just as it's listed in the HTML. On the other hand, the .innerText property will take CSS styling into consideration and will return the text that is visibly rendered on the page.

Further Research

* [**innerHTML on MDN(opens in a new tab)**](https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML)
* [**textContent on MDN(opens in a new tab)**](https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent)
* [**innerText on MDN(opens in a new tab)**](https://developer.mozilla.org/en-US/docs/Web/API/Node/innerText)
* [**Article: The poor, misunderstood innerText(opens in a new tab)**](http://perfectionkills.com/the-poor-misunderstood-innerText/)
* [**Article: innertext vs. textcontent**](https://kellegous.com/j/2013/02/27/innertext-vs-textcontent/)