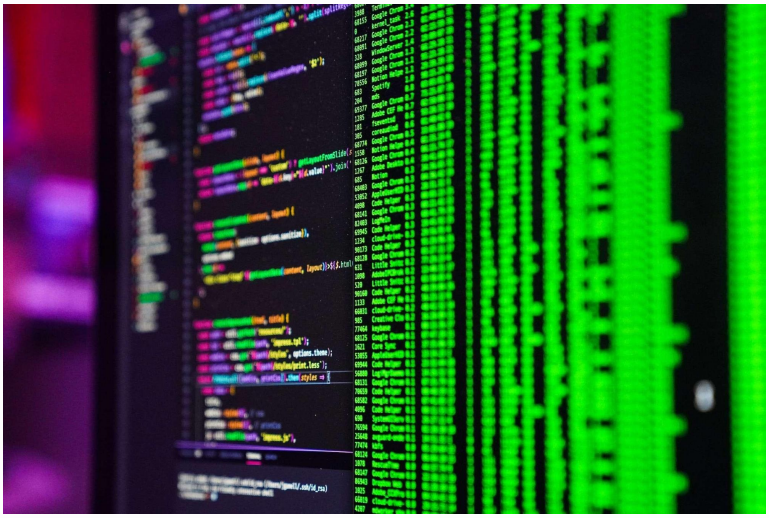
JavaScript Overview



cover image

Goals

By the end of this lesson, you should understand:
>
- What JavaScript is and where it came from
- Why it's beneficial to learn JavaScript
- How to create a basic JavaScript development setup
>
Introduction
After learning all the basics of HTML and CSS, in addition to the many other topics we have covered thus far in the course, it's now time to begin learning the crux of frontend web development – JavaScript! JavaScript is what creates all the sophisticated, elegant functionality and interactivity we have come to know, love, and indeed *expect*, whenever we visit a website. In this week's lessons, we'll start learning what it is and many of the different ways we can use it.

What is JavaScript?

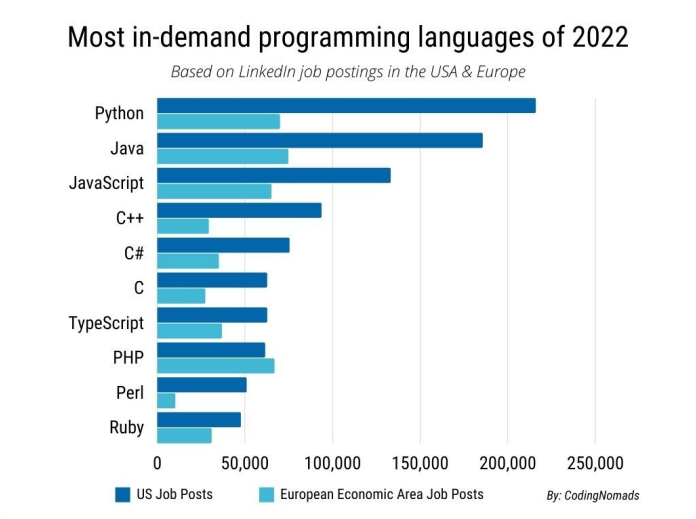
JavaScript is a high-level programming language that was designed for use in web browsers to make web pages more interactive and dynamic. It is technically a scripting language, meaning it is executed by web browsers rather than being compiled into machine code like many other programming languages.

The language was originally created by Brendan Eich in **just 10 days** in May 1995 while he was working at Netscape, whose web browser of the same name was the most popular at the time. JavaScript was originally called Mocha, but was soon renamed to LiveScript before finally being renamed yet again to JavaScript, a purely marketing-driven move in order to capitalize on the popularity of another programming language called Java. Initially designed to be a simple language for adding interactivity to web pages, JavaScript has since evolved to become a versatile language that can be used for building full-stack web applications, or even desktop and mobile applications utilizing certain frameworks!

Why learn JavaScript?

There are several reasons why learning JavaScript is worthwhile. For starters, it is the most widely used programming language for web development. But there are other reasons too – JavaScript today can be used both on the client-side *and* server-side (thanks in no small part to Node.js), and it has a large and very active developer community. This effectively means that there are a vast amount of resources available for continued learning, as well as solving problems which may arise during development of a given project.

Perhaps most importantly, there are a ton of available jobs for JavaScript developers! In fact, according to research conducted by CodingNomads, it is the third most in-demand programming language as of the end of 2022 – a trend which is continuing this year.



in-demand programming languages

Source: 10 Most In-Demand Programming Languages

JavaScript Versions

JavaScript versions are standardized by Ecma International in the document ECMA-262. For this reason JavaScript is sometimes referred to as ECMAScript; however, ECMAScript is the *standard* and JavaScript is the *language*.

Since the ECMA standard was created, there have been 13 ECMA editions. New JavaScript features are commonly referenced according to the ECMA edition that verifies them as standard; however, the compilers, polyfills, browsers, servers, and runtimes that use JavaScript add new features in a feature-by-feature manner and not by ECMA edition, so it is important as a JavaScript developer to know which features are widely supported enough to use in production.

There are many online resources available where you can check feature support, including ECMAScript Compatibility Table and Can I use?.

The most common, and most widely supported JavaScript version in use today by most web browsers is ES6 (which is also sometimes called EMCAScript 2015 or ES2015).

Setting up a Basic JavaScript Development Environment

> There are a number of ways to use JavaScript in a web development project, but the most basic is to write it directly in an HTML file.

In the code editor, you'll notice an HTML file with some basic boilerplate:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Basic JS Dev Setup</title>
</head>
<body>
  <h1>Basic JS Dev Setup</h1>

</body>
</html>
```

In order to add JavaScript to this file we will first create opening and closing script tags like so, just above the closing body tag:

```
<script>

</script>
```

Inside the script tags, write the following JavaScript code *(note that a semicolon ; is used to mark the end of a statement, and while semicolons are not strictly required many developers prefer to use them)*:

```
>
alert("Hello JavaScript!");
>
```

Now, refresh the Codio browser panel or click the button to open it in a new tab, and you should see an alert message popup while the page is loading, and whose content is similar the following:

Click the 'OK' button to close the alert. Congratulations! You just wrote your first line of JavaScript.

> `alert()` is a built-in function in JavaScript that displays a message in a dialog box.

Before wrapping up this section, it's important to note that the way we're utilizing JavaScript here (writing it directly in the HTML) is *not considered best practice.* The most common way to include JavaScript in a web page (and the way we'll be using, going forward) is to write the code in a separate file, then reference that file in the HTML.

Let's make one small change to the code we wrote above. Delete everything between the script tags, then add an **src** attribute to the opening script tag and give it a value of "**./script.js**":

```
>
<script src="./script.js"></script>
>
```

The **.js** is the file extension for JavaScript files. As is, this change won't do anything though. We still need to create the file this script tag is now referencing. Go ahead and create a new file called **script.js**, then open it and include our code from before:

```
>
alert("Hello JavaScript!");
>
```

Now refresh the browser preview again. If you see that same popup message we saw previously, then everything is working correctly!

Conclusion & Takeaway

> In this lesson, we started exploring what JavaScript is, where it came from, and why it is a good programming language to learn today. We covered its different versions, and then finished by setting up a basic development environment to work with JavaScript. We saw two of the different methods for including JavaScript in an HTML file, and wrote our first bit of JavaScript code. In the next lesson, we'll start diving much more into the mechanics of the JavaScript language.

Variables, Data Types and Commenting Code

Goals

By the end of this lesson, you should understand:
- What variables are and the different ways to create them
- The various data types available in JavaScript
- The syntax for commenting JavaScript code
>
Introduction
In the previous lesson, we saw a couple different ways to include JavaScript in an HTML file. In this lesson, we will truly start our JavaScript journey and begin learning programming fundamentals.
>
If this is your first experience with programming, these concepts may not seem intuitive at first. The best way to get comfortable is through practice! Ask lots of questions, and keep working on the exercises until they make sense. What you learn here will give you a foundation that can make learning other languages, such as Python and Java, much easier!
>
We will start by explaining what variables are and how they are used, the differences between variables and constants, and how to create each of them. We will then move on to discuss different data types in JavaScript (including primitive data types like **strings**, **numbers**, **booleans**, **null**, and **undefined**, as well as complex data types like **arrays** and **objects**). Finally, we will look at how to add comments to our code.

Intro to Variables

Intro to Variables

Variables are used in programming to store and manipulate data. In JavaScript, there are three different keywords used for declaring variables: **var**, **let**, and **const**. The main difference between these is something called “scope” which we will cover later.
>
1. **var**:
The **var** keyword is used to declare a variable globally or locally in a function. Variables declared with **var** are function-scoped, meaning they are only accessible within the function in which they are declared or in the global scope if they are declared outside a function
1. **let**:
The **let** keyword is used to declare a block-scoped variable. This means that variables declared with **let** are only accessible within the block they are declared in, including any nested blocks.
1. **const**:
The **const** keyword is used to declare a block-scoped variable that cannot be reassigned. This means that once a variable is declared with **const**, its value cannot be changed. However, note that the value of an object or an array declared with **const** can be modified, but the variable cannot be reassigned to a different value.

PRO TIP: The **var** keyword, though it is still valid, is an older way of declaring variables. Almost all modern JavaScript will use either **let** or **const**!
>When deciding between **let** and **const**, think about whether the value of the variable should be allowed to change or not. If it should not be changed, use **const**; otherwise, use **let**.
>
Next, let’s talk briefly about naming conventions. This applies to naming both variables and functions, when the name given is made up of several words. The most common convention in JavaScript is to use what is called **camel case**. This means that the first letter of the first word in the name is lowercase, while in each subsequent word in the name the first letter is uppercase. Let’s look at an example to illustrate this:

```
mySuperAwesomeVariableName
```
>
While this example may be a bit over the top, it illustrates the point. The reason why this is the most commonly used naming convention is because it helps greatly with code readability.

With that in mind, let's move on to another important point – namely, the difference between *declaring* a variable and *initializing* a variable. Declaring a variable means simply creating a variable (e.g. - **let myVariable**), while initializing a variable means giving it an initial value. With **let** you can declare a variable without initializing it; with **const**, however, you MUST also give it an initial value.

Let's look at some examples of declaring variables and constants in JavaScript:
>

```
let firstName; // declaring a variable named "firstName" without
initializing it
firstName = "John"; // initializing the "firstName" variable
with the value of "John"
let lastName = "Doe"; // declaring a variable named "lastName"
and initializing it with the value of "Doe"

let age = 25; // declaring a variable called "age" and
initializing it with the value of 25

const favoriteMovie = "Scott Pilgrim vs. The World"; //
declaring a constant called "favoriteMovie" and initializing it
with the value of "Scott Pilgrim vs. The World"
```

> Now let's get some practice with this! Note that the preview window has been configured to show the console output. This way we can see whatever we log to the console in our JavaScript code. For now, there is no output because we haven't written anything in **script.js** yet; but we're going to change that by declaring some variables:
>
> 1. In **script.js**, underneath the comment that says "Write your code below", use `let` to declare and initialize the variables `firstName` and `lastName` with your first and last name.
> 2. Next, let's add a `console.log()` statement and inside the parentheses write `firstName + " " + lastName`.
> 3. Refresh the preview, and you should now see your first and last name separated by a space!
>
> Strings are a data type used to represent text and must be enclosed in quotation marks. Variable names do not require quotation marks because they are identifiers used to represent values in the code.
>
> When using the **let** keyword, it's also possible to declare multiple variables at once. So, we could have even declared the first and last name variables above as follows:

```
let firstName = "John", lastName = "Doe";
```

Primitive Data Types

> In JavaScript, there are several primitive data types: **strings**, **numbers**, **booleans**, **null** and **undefined**. These data types are used to represent basic values. Let's take a closer look at each of them.

1. **Strings**: A string is a sequence of characters enclosed in single or double quotes. Strings are used to represent text in JavaScript. Here are some examples of strings:

```
let message = "Hello, world!"; // a string variable
let name = 'Bob'; // another string variable
```

We can concatenate strings using the + operator (we'll learn more about operators in a later lesson):
>

```
let greeting = "Hello, " + name + "!"; // concatenating the
"Hello, " and "!" strings with the value of the name variable
```

> Let's log the **greeting** variable to the console:

```
console.log(greeting) // Output: Hello, Bob!
```

> 2. **Numbers**: Numbers are used to represent numerical values in JavaScript. They can be integers or decimals. Here are some examples of numbers:

```
let count = 10; // an integer variable
let price = 9.99; // a decimal variable
```

We can perform arithmetic operations on numbers, such as addition, subtraction, multiplication, and division:

```
let total = count * price; // multiplying the "count" and
"price" variables
```

> Think of how a shopping website takes your order and calculates the total; it's not much different from this!

Now, let's display the total by using `console.log` to output the price.

3. **Booleans**: A boolean represents a logical value, and can be either **true** or **false**. Here are some examples of booleans:

```
let isStudent = true; // a boolean variable
let isEmployed = false; // another boolean variable
```

Booleans are most useful in conditional statements, such as *if-else* statements (which we will cover in the next lesson):
>

```
if (isStudent) {
  console.log("You are a student.");
} else {
  console.log("You are not a student.");
}
```

4. **Null and Undefined**: **null** and **undefined** are special values in JavaScript that represent the absence of a value. **null** is explicitly set by the developer, while **undefined** is the default value for variables that have not been initialized. Here are some examples:

```
let value1 = null; // explicitly setting the value of a variable
to null
let value2; // declaring a variable without initializing it
(defaults to undefined)
console.log(value1); // output: null
console.log(value2); // output: undefined
```

Complex Data Types

> In addition to primitive data types, JavaScript also has some complex data types. For now, we're going to focus on two of them specifically: **arrays** and **objects**.
>
> 1. **Arrays**: an array is an ordered collection of values. Every value in an array has a sort of defined "address" (called an index) which is denoted by a number, beginning with 0. Here's an example of an array:

```
let numbers = [1, 2, 3, 4, 5];
```

You can then access any of the values in an array using it's index:

```
console.log(numbers[0]); // output: 1
console.log(numbers[2]); // output: 3
```

You can also modify the values in an array in a similar manner:

```
numbers[0] = 10; // changing the value at index 0 to 10
console.log(numbers); // output: [10, 2, 3, 4, 5]
```

> 2. **Objects**: An object is an unordered collection of key-value pairs, and can store values of any data type. Here's an example of an object:

```
let person = {
  name: "John",
  age: 30,
  isStudent: true
};
```

You can access the values in an object by their keys, using dot notation:

```
console.log(person.name); // output: "John"
console.log(person.age); // output: 30
```

You can also add new key-value pairs to an object:

```
person.email = "john@example.com"; // add an email key-value
pair to the object
console.log(person); // output: {name: "John", age: 30,
isStudent: true, email: "john@example.com"}
```

Commenting in JavaScript

> Commenting code in JavaScript, as with HTML and CSS, allows us to add notes to the source code that helps other developers (or anyone looking at our code) to understand what the code does, why it does it, and how it works. Comments are ignored when the code is executed, and they are indicated using special characters that are not part of the code itself.
>
> In JavaScript, there are two types of comments: **single-line comments** and **multi-line comments**. **Single-line comments** begin with two forward slashes (//) and continue until the end of the line. **Multi-line comments**, on the other hand, begin with /* and end with */. Multi-line comments can span multiple lines, making them useful for longer explanations.

Here are some examples:

```
// This is a single-line comment
  let num = 42; // Single-line comments can also be added at the
end of a line of code

  /*
    This is an example of a
    multi-line comment
  */
```

Comments can be used to explain the purpose of functions, describe complex code segments, and provide context for variable names or values. They can also be used to temporarily disable a piece of code, as shown below:

```
/*
  let anotherNum = 57
*/
```

By surrounding the above code with a multi-line comment, it will not be executed when the code is run. This can be useful in many debugging or testing scenarios.
>
In general, it is considered good practice to comment code thoroughly to make it easier for other developers to understand and modify the code in the future. However, it is also important to avoid over-commenting, as this

can end up having the opposite effect. As with many things, it's all about finding the right balance.
>

Conclusion & Takeaway

> In this lesson, we covered the basics of variables, constants and data types in JavaScript. We explored what variables are and how they are used, as well as the difference between *declaring* and *initializing* them. In addition, we went over primitive data types (strings, numbers, booleans, null, and undefined), and complex data types (arrays and objects). Finally, we learned the syntax for adding comments to JavaScript code. In the next lesson, we will continue building on this foundation by examining the key concepts of type coercion, “truthy” and “falsey”, and have a look at a wide variety of operators!

Basic Operators: Introduction

> **Goals**
>
> By the end of this lesson, you should understand:
> - Basic operators in JavaScript, and how to use them
> - Type coercion, and how it can affect code output
> - The concepts of “truthy” and “falsy”, and why they’re important

Introduction

In this lesson, we will cover the basic operators available in JavaScript. Operators are symbols used to perform mathematical or logical operations on variables or values. In addition, we will briefly touch on the topic of **type coercion** in JavaScript, as well as the essential concepts of “truthy” and “falsy” - which are especially important when using logical operators or conditional statements (which we’ll learn about later).
>

Arithmetic Operators

> Arithmetic operators are used to perform mathematical calculations on numbers. There are several arithmetic operators available in JavaScript.
>
> For each of the following examples, add the code to **script.js**!
>
> 1. **Addition** (+): Adds two or more numbers together.

```
let num1 = 10;
let num2 = 5;

let additionExample = num1 + num2;

console.log(additionExample); // Output: 15
```

> 2. **Subtraction** (-): Subtracts one number from another.

```
let subtractionExample = num1 - num2;

console.log(subtractionExample); // Output: 5
```

3. **Multiplication** (__*__): Multiplies two or more numbers together.

```
let multiplicationExample = num1 * num2;

console.log(multiplicationExample); // Output: 50
```

4. **Division** (/): Divides one number by another.

```
let divisionExample = num1 / num2;

console.log(divisionExample); // Output: 2
```

5. **Modulus** (%): Returns the remainder of a division operation.

```
let modulusExample = num1 % num2;

console.log(modulusExample); // Output: 0
```

6. **Increment** (++): Adds one to a number.

> **NOTE:** With this operator (as well as the decrement operator, which we'll look at next) the returned value will be different based on where the operator is placed: if we place it **after** the variable we're trying to increment then assign the result to a new variable and console log it, we will get the original value! However, if we place it **before** the variable we'll get the value we expect in this context. Be aware, though, that this operator will actually update the value of the original variable as well!

```
let incrementExample = ++num1;

console.log(incrementExample); // Output: 11
```

7. **Decrement** (– –): Subtracts one from a number. In the following example, since the value of **num1** has been incremented to 11, we should expect to get 10 again after we decrement the value:

```
let decrementExample = --num1;

console.log(decrementExample); // Output: 10
```

8. **Compound Assignment** (+= and -=): Adds or subtracts a given value and assigns the result to a variable. In JavaScript, the += and -= operators are known as compound assignment operators. They allow you to perform an arithmetic operation and assign the result to a variable in a single step. The += operator performs addition and assignment, while the -= operator performs subtraction and assignment.

Consider the following code snippet:

```
let x = 10;
x += 5; // equivalent to x = x + 5;

console.log(x); // output: 15


let y = 20;
y -= 10; // equivalent to y = y - 10;

console.log(y); // output: 10
```

In the first example, the value of **x** is incremented by 5 and then assigned back to **x**. The output of **console.log(x)** is 15.
>
In the second example, the value of **y** is decremented by 10 and then assigned back to **y**. The output of **console.log(y)** is 10.
>
In Summary: Assignment is the act of storing a value in a variable. Compound assignment is a shorthand way of combining an assignment with an arithmetic operation. Instead of writing an assignment and an operation separately, you can use a compound assignment operator to do both in one step. Using compound assignment operators can make your code more concise and easier to read.

Order of Operations

> Arithmetic operators, as you might expect, follow the mathematical order of operations - meaning that multiplication and division are performed before addition and subtraction. However, you can use parentheses to change the order of operations, just like in math. For example:

```
let orderOfOperationsExample = (10 + 5) * 2;

console.log(orderOfOperationsExample); // Output: 30
```

In this example, the addition is performed first inside the parentheses, resulting in 15. Then, the multiplication is performed, resulting in the final output of 30.
>

Type Coercion

> It's important to note that arithmetic operators only work with numbers. If you try to use an arithmetic operator on a non-number value, it can yield unpredictable outputs in different scenarios.

Let's say you have two variables; one is a number and the other is a string. If you try to add them together, you might get a surprising result:

```
let val1 = 10;
let val2 = "5";


let typeCoercionExample = val1 + val2;


console.log(typeCoercionExample); // Output: "105"
```

Wait... "105"?? Yep! That's because JavaScript does what is called **type coercion** when it performs an operation like this on two values that are not of the same type.

In this case, instead of converting **val2** to a number and performing the addition operation, JavaScript's type coercion effectively converted **val1** to a string and performed a different operation called **string concatenation**, which uses the same operator as addition. We'll cover strings more in depth later in the course.
>

Comparison Operators

> ### Comparison Operators
>
> Comparison operators are used in JavaScript to compare values and return a Boolean value (**true** or **false**). There are several comparison operators available in JavaScript:
> 1. **Equal to** (==): Returns **true** if two values are equal, regardless of their data types.

```
let num1 = 10;
let num2 = "10";

console.log(num1 == num2); // Output: true
```

In this example, the == operator returns **true** even though **num1** is a number and **num2** is a string. This is because JavaScript automatically converts the string “10” to the number 10 before comparing them.
2. **Not equal to** (**!**=): Returns **true** if two values are not equal.

```
let num1 = 10;
let num2 = 5;


console.log(num1 != num2); // Output: true
```

In this example, the **!**= operator returns **true** because **num1** and **num2** are not equal.
3. **Strict equal to** (===): Returns **true** if two values are equal and have the same data type.

```
let num1 = 10;
let num2 = "10";

console.log(num1 === num2); // Output: false
```

In this example, the === operator returns **false** because **num1** is a number and **num2** is a string, even though their values are the same.
4. **Strict not equal to** (**!**==): Returns **true** if two values are not equal or do not have the same data type.

```
let num1 = 10;
let num2 = "10";

console.log(num1 !== num2); // Output: true
```

In this example, the **!==** operator returns **true** because **num1** is a number and **num2** is a string.
5. **Greater than** (>): Returns **true** if one value is greater than another.

```
let num1 = 10;
let num2 = 5;

console.log(num1 > num2); // Output: true
```

In this example, the > operator returns **true** because **num1** is greater than **num2**.
6. **Less than** (<): Returns **true** if one value is less than another.

```
let num1 = 10;
let num2 = 5;

console.log(num1 < num2); // Output: false
```

In this example, the < operator returns **false** because **num1** is not less than **num2**.
7. **Greater than or equal to** (>=): Returns **true** if one value is greater than or equal to another.

```
let num1 = 10;
let num2 = 5;

console.log(num1 >= num2); // Output: true
```

In this example, the >= operator returns **true** because **num1** is greater than **num2**.
8. **Less than or equal to** (<=): Returns **true** if one value is less than or equal to another.

```
let num1 = 10;
let num2 = 5;

console.log(num1 <= num2); // Output: false
```

In this example, the <= operator returns **false** because **num1** is not less than or equal to **num2**.

Comparison operators can be used with variables or with literal values, and are often used in conditional statements and loops. For example, here's how you might use comparison operators in an **if-else** statement:

```
let age = 25;

if (age >= 18) {
  console.log("You are an adult");
} else {
  console.log("You are not an adult");
}
```

In this example, the >= operator is used to check if the **age** variable is greater than or equal to 18. If it is, the program outputs "You are an adult". Otherwise, it outputs "You are not an adult".

> A quick note about checking equality in JavaScript: because of type coercion it's generally best to use the strict equality operator === whenever possible, unless you have a good reason not to!

In summary, comparison operators are used to compare values and they return a Boolean value (**true** or **false**). They are often used in conditional statements and loops to control the flow of code execution. In addition, it's important to be aware of the data types you're working with and how JavaScript will automatically convert them with type coercion if needed.
>

Logical Operators

> Logical operators are used to combine or modify Boolean values. There are three logical operators in JavaScript: **&&** (AND), **||** (OR), and **!** (NOT).

The **&&** operator returns **true** if both operands are **true**, otherwise it returns **false**. Here's an example:

```
let x = 5;
let y = 10;

if (x < 10 && y > 5) {
  console.log("Both conditions are true");
} else {
  console.log("At least one condition is false");
}
```

In this example, the **&&** operator is used to check both if the **x** variable is less than 10 and if the **y** variable is greater than 5. Since both conditions are true, our code should output: "Both conditions are true".

The **||** operator, on the other hand, returns **true** as long as *one* operand is **true**, otherwise it returns **false**. Here's an example:

```
let a = 2;
let b = 3;

if (a > 3 || b < 5) {
  console.log("At least one condition is true");
} else {
  console.log("Both conditions are false");
}
```

In this example, the **||** operator is used to check if either the **a** variable is greater than 3 or the **b** variable is less than 5. Since the second condition is true, our code should output: "At least one condition is true".

The **!** operator is used to invert a Boolean value. If the operand is **true**, **!** returns **false**, and if the operand is **false**, **!** returns **true**. Here's an example:

```
let value = true;

if (!value) {
  console.log("The value is false");
} else {
  console.log("The value is true");
}
```

In this example, the **!** operator is used to check if the **value** variable is **false**. Since the **value** variable is actually **true**, our code should output: “The value is true”.
>
Logical operators are often used in combination with comparison operators in conditional statements and loops. For example:

```
let x = 5;
let y = 10;

if (x < 10 && y > 5) {
  console.log("Both conditions are true");
} else if (x >= 10 || y <= 5) {
  console.log("At least one condition is true");
} else {
  console.log("Both conditions are false");
}
```

In this example, the **&&** operator is used to check if both the **x** variable is less than 10 and the **y** variable is greater than 5. If this condition is not met, the next condition is examines, which uses the || operator to check if either the **x** variable is greater than or equal to 10 or the **y** variable is less than or equal to 5. If this condition is also not met, our code would output: “Both conditions are false”.

“Truthy” or “Falsy”

> In JavaScript, another important concept is the idea of “truthy” or “falsy”: a value is considered “truthy” if it evaluates to **true** when coerced to a boolean, and “falsy” if it evaluates to **false**.

Here are the values that are considered “falsy” in JavaScript:
- **false**
- **null**
- **undefined**
- **0**
- **-0**
- **NaN**
- **""** (empty string)
>
All other values are considered “truthy”.

This concept of “truthy” or “falsy” applies not only to literal values, but also variables as well. Even functions (and their return values), arrays, objects, and just about anything, really, will evaluate to either “truthy” or “falsy”!

Here are some examples to demonstrate this:

```
if (false) {
  console.log("This will not be printed");
}


if (null) {
  console.log("This will not be printed");
}


if (undefined) {
  console.log("This will not be printed");
}


if (0) {
  console.log("This will not be printed");
}


if (-0) {
  console.log("This will not be printed");
```

```
}


if (NaN) {
  console.log("This will not be printed");
}


if ("") {
  console.log("This will not be printed");
}


if ("Hello, world!") {
  console.log("This will be printed");
}


if (42) {
  console.log("This will be printed");
}


if ([]) {
  console.log("This will be printed");
}


if ({}) {
  console.log("This will be printed");
}
```

Here we are using an **if** statement to test the "truthiness" of different values. We will learn more about **if** statements later.

If you run the code above you will see that the first seven conditions will not execute the code block inside the **if** statement because they are "falsy", whereas the last four conditions will because they are "truthy".

> An empty array `[]` and an empty object `{}` are considered truthy in JavaScript because they are both objects, and in JavaScript, all objects are considered truthy values.

Understanding "truthy" and "falsy" values is very important, especially when working with conditional logic!
>

Conclusion & Takeaway

> In this lesson, we covered the different types of operators in JavaScript, as well as the concepts of “truthy” and “falsy” and type coercion. We learned how perators are used to perform mathematical or logical operations on variables or values, and also how pretty much everything in JavaScript can evaluate to either “truthy” or “falsy”, which can be handy in a number of different scenarios. All of these concepts are important to master as you continue progressing as a developer.

Control Structures Part 1: Introduction

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The syntax and usage of *if/else* statements and *switch* statements in JavaScript
> - What ternary expressions are and why they are helpful in certain scenarios

> ### Introduction
>
> In JavaScript, as in other programming languages, there are what are known as *control structures* which, as the name suggests, control the flow of code execution. In this lesson, we will begin exploring one such control structure: **conditional statements**. Understanding the different conditional statements available in JavaScript and how to use them will enable us to write more dynamic logic in our frontend code.

Conditional Statements

> Conditional statements, simply put, are used to execute code based on certain conditions. There are three types of conditional statements in JavaScript: *if/else* statements, *ternary expressions* and *switch* statements.

1. **If/Else Statements**
   If/else statements are a fundamental control structure that allow you to execute different code blocks based on a condition. The syntax for an *if/else* statement, as we've already seen a couple of times, is as follows:

```
if (condition) {
  // code block to execute if condition is true
} else {
  // code block to execute if condition is false
}
```

Here's how it works:
1. The condition is evaluated to a Boolean value (true or false).
2. If the condition is true, the code block inside the **if** statement is executed.
3. If the condition is false, the code block inside the **else** statement is executed (if included).

Here's an example of how to use an *if/else* statement to check if a number is positive or negative:

```
let num = 5;

if (num >= 0) {
  console.log("The number is positive");
} else {
  console.log("The number is negative");
}
```

In this example, we use an *if/else* statement to check if the **num** variable is greater than or equal to 0. If it is, the message "The number is positive" is printed to the console. If it's not, the message "The number is negative" is printed to the console.

You can also use *if/else* statements with more complex conditions using comparison operators such as ==, **!**=,>, <, >=, and <=. For example:

```
let age = 18;

if (age >= 18) {
  console.log("You are an adult");
} else if (age >= 13) {
  console.log("You are a teenager");
} else {
  console.log("You are a child");
}
```

In this example, we use an *if/else if/else* statement to check the value of the **age** variable. If it's greater than or equal to 18, the message "You are an adult" is printed to the console. If it's between 13 and 18 (inclusive), the message "You are a teenager" is printed to the console. If it's less than 13, the message "You are a child" is printed to the console.

If/else statements can also be nested to handle more complex conditions:

```
let num = 5;

if (num >= 0) {
  if (num == 0) {
    console.log("The number is zero");
  } else {
    console.log("The number is positive");
  }
} else {
  console.log("The number is negative");
}
```

In this example, we first check if the **num** variable is greater than or equal to 0. If it is, we then check if it's *equal* to 0. If it is, the message "The number is zero" is printed to the console. If it's greater than 0, the message "The number is positive" is printed to the console. If it's less than 0, the message "The number is negative" is printed to the console.
>

Ternary Expressions

> Ternary expressions, also known as ternary operators, are a shorthand way of writing *if/else* statements. The syntax for a ternary expression is as follows:

```
condition ? expressionIfTrue : expressionIfFalse
```

Here's how it works:
1. The condition is evaluated to a Boolean value (true or false).
2. If the condition is true, the **expressionIfTrue** segment is returned.
3. If the condition is false, the **expressionIffFalse** segment is returned.

Here's an example of how to use a ternary expression to check if a number is positive or negative:

```
let num = 5;

let message = num >= 0 ? "The number is positive" : "The number
is negative";

console.log(message);
```

In this example, we use a ternary expression to check if the **num** variable is greater than or equal to 0. If it is, the message "The number is positive" is stored in the **message** variable. If it's not, the message "The number is negative" is stored in the **message** variable. The **console.log** statement then prints the **message** variable to the console.

You can also use ternary expressions with more complex conditions using comparison operators such as ==, **!**=, >, <, >=, and <=. For example:

```
let age = 18;

let message = age >= 18 ?
   "You are an adult"
  : age >= 13 ?
   "You are a teenager"
  : "You are a child";

console.log(message);
```

In this example, we use nested ternary expressions to check the value of the **age** variable. If it's greater than or equal to 18, the message "You are an adult" is stored in the **message** variable. If it's between 13 and 18 (inclusive), the message "You are a teenager" is stored in the **message** variable. If it's less than 13, the message "You are a child" is stored in the **message** variable. The **console.log** statement then prints the **message** variable to the console.

Ternary expressions are similar to *if/else* statements in that they both allow you to execute different code based on a condition. However, ternary expressions are more concise and can be easier to read in some cases. Additionally, ternary expressions can be used as part of an expression or assignment statement, while *if/else* statements can only be used as a standalone control structure.

That being said, *if/else* statements are more flexible than ternary expressions, and can be nested more easily, making them better suited for handling more complex conditions and logic.

In summary, *if/else* statements are better suited for handling complex logic, while ternary expressions are better suited for simple conditions and concise code.

Statement	Syntax	Advantages	Uses	Best Practices
Ternary Operator	`(condition) ? true : false;`	Concise and easy to read in simple cases	Can be used as part of an expression or assignment statement	Use for simple conditions and concise code; avoid nesting or complex expressions
If/Else	`if (condition) { code block }`	More flexible and can handle complex logic	Used as a standalone block of code that executes when the condition is true or false	Use for complex conditions and logic; structure code for readability and ease of maintenance

Switch Statements

> *Switch* statements are a type of control structure in JavaScript that allow you to compare a single value against multiple cases and execute different code blocks depending on the matched case.

The syntax for a *switch* statement is as follows:

```
switch(expression) {
  case value1:
    // code block
    break;
  case value2:
    // code block
    break;
  default:
    // code block
}
```

Here's how it works:
1. The *switch* expression is evaluated once and its value is compared to each case value.
2. If there is a match between the *switch* expression and a case value, the code block for that case is executed.
3. The **break** keyword is used to indicate the end of each case block. If a **break** statement is not included, the code execution will "fall through" to the next case block.
4. If there is no match between the *switch* expression and any of the case values, the code block in the default section is executed (if included).
>
Here's an example of how to use a *switch* statement to output a message based on the day of the week:

```
let day = new Date().getDay();
let message;

switch (day) {
  case 0:
    message = "Today is Sunday";
    break;
  case 1:
    message = "Today is Monday";
    break;
  case 2:
    message = "Today is Tuesday";
    break;
  case 3:
    message = "Today is Wednesday";
    break;
  case 4:
    message = "Today is Thursday";
    break;
  case 5:
    message = "Today is Friday";
    break;
  case 6:
    message = "Today is Saturday";
    break;
  default:
    message = "Invalid day";
}

console.log(message);
```

In this example, we use the **getDay()** method to get the current day of the week (as a number between 0 and 6), and then use a **switch** statement to set the value of the **message** variable based on the day of the week. If the day value is not between 0 and 6, the **default** case is executed, and the message “Invalid day” is assigned to the **message** variable.

> **Date()** and **getDay()** are part of the built-in JavaScript Date object.

Switch statements are useful when you have multiple possible conditions to check against a single variable or expression. However, they should be used judiciously to avoid overly complex or hard-to-read code.
>

Conclusion & Takeaway

> Conditional statements are fundamental in programming because they enable us to make decisions and execute code based on specific conditions. By mastering the different conditionals we covered in this lesson - *if-else* statements, ternary expressions, and *switch* statements - we can write efficient and structured JavaScript code that responds dynamically to various scenarios.

Control Structures Part 2: Introduction

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The syntax and functionality of *for* loops
> - How and when to use *while* or *do/while* loops
>
> ### Introduction
>
> Loops are another important control structure to master - they allow us to execute code repeatedly until specific conditions are met. There are primarily three types of loops in JavaScript: *for* loops, *while* loops, and *do/while* loops. In this lesson, we will explore each of these three types of loops, their syntax, and which scenario each is best suited for.

For Loops

> **For Loops**
> *for* loops are used to repeat a block of code a certain number of times, based on a set of conditions. Here's the basic syntax of a *for* loop:

```
for (initialization; condition; iteration) {
  // code to be executed
}
```

The **initialization** statement is executed before the loop starts. This is where you can declare and initialize any variables that will be used in the loop.

The **condition** statement is evaluated at the beginning of each iteration. If it evaluates to *true*, the loop continues. If not, the loop ends.

The **iteration** statement is executed at the end of each iteration. This is where you can update any variables that are used in the loop.

Here's an example of a *for* loop that counts from 0 to 9 and outputs each number to the console:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

In this example, **let i = 0** initializes the **i** variable to 0, **i < 10** is the condition that checks if **i** is less than 10, and **i++** increments **i** by 1 at the end of each iteration.

You can also use a *for* loop to iterate over an array or object. For example, here's a *for* loop that iterates over an array of numbers and outputs the sum of all the numbers:

```
let numbers = [1, 2, 3, 4, 5];
let sum = 0;

for (let i = 0; i < numbers.length; i++) {
  sum += numbers[i];
}

console.log(sum);
```

In this example, **numbers.length** is used as the condition to ensure that the loop runs for as many times as there are elements in the **numbers** array. The **sum** variable is initialized to 0 before the loop starts, and inside the loop, **sum** += **numbers[i]** adds each number in the array to the **sum** variable.

> **.length** is a property in JavaScript that is used to get the number of characters in a string or the number of elements in an array.

You can also use **break** and **continue** statements inside a *for* loop to control the flow of the loop. The **break** statement is used to exit the loop completely, while the **continue** statement is used to skip to the next iteration. Here's an example that uses both **break** and **continue**:

```
for (let i = 0; i < 10; i++) {
  if (i === 3) {
    continue; // skip iteration if i equals 3
  }

  if (i === 7) {
    break; // exit loop if i equals 7
  } console.log(i);
}
```

In this example, the **continue** statement is used to skip the iteration where **i** equals 3, meaning the number 3 will not show up in the output. The **break** statement is used to exit the loop completely when **i** equals 7.
>

While Loops

> **While Loops**
> *while* loops are used to repeatedly execute a block of code while a specified condition is true. The basic syntax of a *while* loop is as follows:

```
while (condition) {
  // code to be executed
}
```

The **condition** is evaluated before each iteration of the loop. If it's true, the code inside the loop is executed. If it's false, the loop is exited.

Using the same example from our *for* loop, which printed the numbers from 0 to 9 to the console, we can acheive basically the same output - this time using a *while* loop:

```
let i = 0;

while (i < 10) {
  console.log(i);
  i++;
}
```

In this example, **let i** = **0** initializes the i variable to 0, and **i** < **10** is the condition that checks if i is less than 10. Inside the loop, **console.log(i)** outputs the value of i to the console, and **i**++ increments i by 1.

As with *for* loops, you can also use a *while* loop to iterate over an array or object. For example:

```
let numbers = [1, 2, 3, 4, 5];
let sum = 0;
let i = 0;

while (i < numbers.length) {
  sum += numbers[i];
  i++;
}

console.log(sum);
```

You can use **break** and **continue** statements inside a *while* loop as well:

```
let i = 0;

while (i < 10) {
  if (i === 3) {
    i++; // increment i to skip iteration if i equals 3
    continue;
  }

  if (i === 7) {
    break; // exit loop if i equals 7
  }

  console.log(i); i++;
}
```

Note that in this example **i**++ is used twice inside the loop to increment **i** in both the **continue** and the normal case.
>

Do/While Loops

> **Do/While Loops**
> *do/while* loops are similar to *while* loops, but with one important difference: a *do/while* loop always executes its code block at least once, even if the condition is false. The basic syntax of a *do/while* loop is as follows:

```
do {
  // code to be executed
} while (condition);
```

The code block is executed first, and then the condition is checked. If the condition is true, the loop executes again. If the condition is false, the loop exits.

Here's an example of a *do/while* loop that prompts a user to enter a number and keeps prompting until a valid number is entered:

```
let number;

do {
  number = prompt("Enter a number:");
} while (isNaN(number));

console.log("You entered the number: " + number);
```

In this example, the code block inside the **do** statement prompts the user to enter a number. The condition, **isNaN(number)**, checks if the entered value is a valid number or not. If the condition is true, the loop executes again and prompts the user to enter a number again. If the condition is false, the loop exits and the program outputs the number entered by the user.

> **isNaN()** is a built-in function in JavaScript that is used to determine if a value is NaN (Not a Number).

Let's build slightly on the previous example:

```
let input;
let isValid = false;

do {
  input = prompt("Please enter a positive integer:");
  // Validate the input
  if (!isNaN(input) && parseInt(input) > 0) {
    isValid = true;
  } else {
    alert("Invalid input. Please try again.");
  }
} while (!isValid);

alert("You entered: " + input);
```

In this example, the *do/while* loop will continue to prompt the user for **input** until they enter a valid positive integer. We first initialize the input variable to **undefined** and the **isValid** variable to **false**. Then, we use a *do/while* loop to repeatedly prompt the user for input until their input is valid.

Inside the loop, we use the **prompt** function to get the user's input. We then validate the input by ensuring that it is a number (using **parseInt()**) and checking if it is greater than 0. If the input is valid, we set the **isValid** variable to **true**, which will cause the loop to exit. Otherwise, we display an error message and the loop will continue.

> **parseInt()** is a built-in function in JavaScript that parses a value as a string and returns a number.

After the loop exits, we display a message to the user that shows the valid input they entered.

All of this helps ensure that the user provides valid input before moving on to the next step. Without this validation, our program may encounter errors or produce incorrect results.

As with both other types of loops we've looked at, we can use a *do/while* loop to iterate over an array or object:

```
let numbers = [1, 2, 3, 4, 5];
let sum = 0;
let i = 0;

do {
  sum += numbers[i];
  i++;
} while (i < numbers.length);

console.log("The sum of the numbers is: " + sum);
```

In this example, the code block inside the **do** statement adds each number in the **numbers** array to the **sum** variable, and the i variable is incremented by 1. The condition, **i < numbers.length**, ensures that the loop executes for as many times as there are elements in the **numbers** array.

You can use **break** and **continue** statements in this type of loop as well:

```
let i = 0;

do {
  if (i === 3) {
    i++; // increment i to skip iteration if i equals 3
    continue;
  }

  if (i === 7) {
    break; // exit loop if i equals 7
  }

  console.log(i);
  i++;

} while (i < 10);
```

Conclusion & Takeaway

> In this lesson, we went over different looping mechanisms in JavaScript. Loops come in different flavors to suit various scenarios. *For* loops are ideal when the number of iterations is predetermined, and they allow for easy iteration over arrays or objects. *While* loops allow us to execute code as long as a given condition remains true, enabling dynamic repetition. The unique feature of *do/while* loops, on the other hand, lies in their guarantee of executing the code block at least once. In the next and final lesson for today, we will dive into functions - which, some might argue, are the bread and butter of JavaScript programming. So grab a coffee and stay tuned!

Functions: Introduction

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - What functions are and how to create them
> - Different kinds of functions and their syntax
> - Parameters, arguments, and the *return* keyword
> - The concept of “scope”
>
> ### Introduction
>
> Functions are one of the most fundamental building blocks of programming; they are used to group together a set of statements to perform a specific task. They can greatly improve the readability and maintainability of our code, as well as help avoid code repetition. In this lesson, we will cover the basics of functions in JavaScript, including how to create them, how to invoke (execute) them, the concepts of arguments and parameters, default parameters, the *return* keyword, scope and a few different function syntaxes.

Creating Functions

> There are actually several ways of creating functions in JavaScript: function declarations, function expressions, arrow functions, and what are called “immediately invoked function expressions” (IIFE). In this section we will take a look at the different syntaxes used to create each of them, along with some examples.
>
> ### Function Declaration
>
> A function declaration is defined using the **function** keyword, followed by the name of the function, a comma-separated list of parameters enclosed in parentheses, and the function body enclosed in curly braces.
>
> Here is an example of a simple function with no parameters:

```
function greet() {
  console.log("Hello world!");
}
```

Here we defined a function named **greet** using the function declaration syntax. The function body consists of a single statement that simply logs a greeting to the console.
>
However, as is this code won’t do anything! That is because we defined the function, but we still have to *invoke* it.
>
Let’s try adding the line `greet();` below the function definition, so that our code now looks like this:

```
function greet() {
  console.log("Hello world!");
}

greet();
```

Refresh the preview window and we should now see our “Hello world!” message.

> ### Function Expression
>
> A function expression is, basically, a function that is assigned to a variable. The syntax for a function expression isn't too different from a function declaration; the main difference is that the **function** keyword is not followed by a name in this case, since the name is given by the variable itself. Here is an example of a function expression:

```
const greet2 = function() {
  console.log("Hello world!");
};
```

In this example, we have created our same **greet** function, but this time using the function expression syntax. The function body is the same as our traditional function example.

Arrow Functions (ES6)

Arrow functions were introduced in ES6 and provide a more concise syntax for creating functions. Arrow function syntax is similar to function expression syntax, except they are created using an arrow (=>) symbol and do not include the **function** keyword. Here is an example of an arrow function:

```
const greet3 = () => {
  console.log("Hello world!");
};
```

In this example, we created the same basic **greet** function as in the previous two examples, now using arrow function syntax.

Arrow functions have a few key differences from the other types of functions we have seen:
- Arrow functions do not have their own **this** context. They use the **this** context of the enclosing lexical scope, meaning the environment in which they are defined (more on the concept of scope shortly).
- Arrow functions cannot be used as constructors. This means that you cannot use the **new** keyword with an arrow function.
- Arrow functions do not have an **arguments** object. Instead, you can use rest parameters (**...args**) to get an array of all the arguments passed to the function.
>
Here is an example of using an arrow function with rest parameters:

```
const sum = (...args) => {
  let total = 0;

  for (let i = 0; i < args.length; i++) {
    total += args[i];
  }

  console.log(total);
};

sum(1, 2, 3, 4, 5); // logs 15 to the console
```

In this example, we have created an arrow function named **sum** that uses rest parameters to get all the arguments passed to the function. The function body then loops over the arguments and adds them together, returning the total.
>

Immediately Invoked Function Expressions (IIFE)

An Immediately Invoked Function Expression (IIFE) is a function that is immediately invoked after it is defined. This can be useful for creating a private scope for your code and avoiding naming collisions with other code. Here is an example:

```
(function() {
  const privateVariable = "I am private!";
  console.log(privateVariable); // logs "I am private!" to the
console
})();

console.log(privateVariable); // ReferenceError: privateVariable
is not defined
```

You'll notice that the syntax for this entails wrapping a normal function declaration (without a name) inside parentheses () then including another set of parentheses right after which effectively calls or invokes the function.
>
Here we have defined an IIFE that in turn defines a private variable named **privateVariable**. Inside the IIFE, we can access the private variable. Outside the IIFE, we get a **ReferenceError** if we try to access it.
>

Hoisting

> Another important thing to note when it comes to the different types of functions and function invocation: that is the concept of **hoisting**. **Hoisting** refers to the ability of certain functions to be invoked before they are defined. Let's see an example of this:

```
function greet4 () {
  const message = defineGreeting();

  console.log(message);
}


greet4(); // logs "Hello!" to the console


function defineGreeting() {
  return "Hello!"
}
```

Notice how we're invoking the **greet4** function, which in turn calls the **defineGreeting** function as part of the code block it executes, before the **defineGreeting** function is defined. This works because **defineGreeting** uses the function declaration syntax. If we were to rewrite **defineGreeting** as an arrow function (or a function expression), our code would break:
>

```
function greet4 () {
  const message = defineGreeting();

  console.log(message);
}


greet4(); // Uncaught ReferenceError: defineGreeting is not
defined


const defineGreeting = () => {
  return "Hello!"
}
```

```
Only function declarations can be hoisted; function expressions and arrow
functions need to be defined first!
>
```

Scope

> Scope refers to the visibility of variables in different parts of the code. In JavaScript, variables defined inside a function are not accessible outside the function. This is called **local scope**. This means that these variables are only visible and can be used within the function in which they are defined.

Variables defined outside a function, on the other hand, are accessible both inside and outside the function. This is called **global scope**. This means that these variables can be used anywhere in the code, including inside functions. Here is an example:

```
const globalVariable = "I am global!"; // defined in global
scope

function myFunction() {
  const localVariable = "I am local!"; // defined in myFunction
scope
  console.log(globalVariable); // logs "I am global!" to the
console
  console.log(localVariable); // logs "I am local!" to the
console
}

myFunction();


console.log(globalVariable); // logs "I am global!" to the
console
console.log(localVariable); // ReferenceError: localVariable is
not defined
```

In this example, `globalVariable` is defined in the global scope, so it can be accessed both inside and outside of our function.

`localVariable`, on the other hand, is defined within the scope of the function, so it can only be accessed inside that function.

When we try to access `localVariable` outside of the function, we get a `ReferenceError`, since `localVariable` is not defined in the global scope.

Let's also talk briefly about the term **lexical scope**, which we mentioned when going over arrow functions.
>

> In programming, the term “lexical” refers to the way that the code is written and structured in the source code file.
>
The concept of **lexical scope** in JavaScript basically means that a given variable’s scope is derived from the place in which it was defined. In the code above, the **lexical scope** of `globalVariable` would be the **global scope**, since it was defined outside of any function. The **lexical scope** of `localVariable`, on the other hand, would be the **local scope** of `myFunction()`, since it was defined inside the function.
>

Parameters and Arguments

> Functions can take one or more parameters, which are placeholders for the values that we pass to the function when we invoke it. Parameters are specified in the function definition. Here is an example of a function that takes two parameters:

```
function add(a, b) {
  console.log(a + b);
}

add(2, 3); // logs 5 to the console
```

In this example, we have defined a function named **add** that takes two parameters, **a** and **b**. The function body consists of a single statement that returns the sum of **a** and **b**. We then invoke the **add** function with the arguments **2** and **3**, which returns **5**, and we log the result to the console.
>
Default Parameters
We can also set default values for parameters in case no argument is passed to the function for that parameter. We can do this by assigning a default value to the parameter in the function definition. Here is an example using the **greet** function we were working with previously:

```
function greet5(name = "world") {
  console.log("Hello, " + name + "!");
}

greet5(); // logs "Hello, world!" to the console

greet5("Bob"); // logs "Hello, Bob!" to the console
```

When we invoke the function without any arguments, the default value is used for the **name** parameter. When we invoke the function with the argument "**Bob**", the argument value is used instead of the default value.
>
Arguments vs. Parameters
Parameters and arguments are two important concepts to understand when working with functions.
>
Sometimes you may hear these two terms used interchangeably, but in fact a **parameter** refers to the placeholder variable used in the function

definition, whereas an **argument** refers to the actual value that is passed to the function when it is called. Consider the `add()` function we defined previously:

```
function add(a, b) {
  console.log(a + b);
}

add(2, 3);
```

In this case, **a** and **b** are parameters of the `add()` function, and the values **2** and **3** (which are passed to the function when we invoke it) are the arguments.
>

The return Keyword

Functions can also return values using the **return** keyword. The **return** keyword stops the execution of the function and returns a value. To see this in action, let's modify our `add()` function as follows:

```
function add(a, b) {
  return a + b;
}

const result = add(2, 3);
console.log(result); // logs 5 to the console
```

In this example, using our `add()` function from before, we now return the value from the function instead of directly logging it to the console from inside the function. Then, we create a variable to store the return value of calling the `add()` function with the arguments 2 and 3. Finally, to see the output, we then log the variable to the console.
>

Conclusion & Takeaway

> Functions are a fundamental concept in programming and are used to organize and modularize code. They allow us to reuse pieces of logic and help to make our code more readable and maintainable. In this lesson, we covered various different function syntaxes, function invocation, parameters and arguments, default parameters, the *return* keyword, scope, and IIFEs (Immediately Invoked Function Expressions). In the next few lessons we will add yet more to our JavaScript toolbox as we begin delving into arrays and objects!

Arrays: Introduction

> ### Goals
>
> By the end of this lesson, you should understand:
> - The different ways to create arrays in JavaScript
> - Various built-in array properties and methods
> - Where you can find resources for additional learning and practice
>
> ### Introduction
>
> In this lesson, we are going to talk more in depth about one of the complex data types we introduced earlier this week: **arrays**. As previously mentioned, arrays are ordered collections of values; thus, they are extremely useful when we need to store multiple related data in a single variable, where the order of that data matters. We will explore the different ways of creating arrays in JavaScript, and the various properties and methods we can use to work with and manipulate them.

Creating Arrays

> There are several ways we can go about creating an array in JavaScript. The most common way of creating an array is by declaring a variable using either **let** or **const**, and setting it equal to square brackets **[ ]** containing a comma-separated list of values:
>
> `let arrayName = [value1, value2, value3];`
>
> For example, the following code declares an array called **myArray** and assigns three values to it:

```
let myArray = [1, 2, 3];

console.log('myArray: ' + myArray);
```

We can also create an empty array and then add values to it later by assigning them to a particular index:
>

```
const myArray2 = [];
myArray2[0] = "a";
myArray2[1] = "b";
myArray2[2] = "c";

console.log('myArray2: ' + myArray2)
```

A third way to create an array is with the Array constructor prefixed by the **new** keyword. Here's what that syntax looks like:
>

```
const arrayName = new Array();
```

>
When using this syntax you can not only create an array but also specify its size. This is done by passing only a single integer value as a parameter to the Array constructor:
>

```
let myArray3 = new Array(3);
```

> Here we have created an array with three undefined elements. We can then assign values to these elements using their index:

```
myArray3[0] = 1;
myArray3[1] = "ring to rule them all";
myArray3[2] = true;

console.log('myArray3: ' + myArray3)
```

Properties and Methods

> ### Properties
>
> Arrays have a few different properties that we can access and work with in our code:
> 1. **length**: Returns the number of elements in an array.

```
let myArray4 = [1, 2, 3];

console.log('Length of myArray4: ' + myArray4.length); //
Output: 3
```

2. **constructor**: Returns the constructor function that was used to create the array. You probably won't use this property very often, if at all. *(It can be useful in situations where you need to identify the type of an object)*

```
const myArray5 = [1, 2, 3];

console.log('myArray5 constructor: ' + myArray5.constructor); //
Output: Array()
```

> The constructor property is used for different purposes in JavaScript, and its usage can vary depending on the type of object.

3. **prototype**: Allows us to add custom properties and methods to the Array prototype. As with the constructor property, you probably won't need to use this in most instances, but it's good to know in case the need arises *(an example would be creating custom array methods)*.

Here's an example of how we can add a custom method to the Array prototype, and then invoke that method on `myArray5` which we created a moment ago:

```
Array.prototype.multiplyByTwo = function() {
  const newArray = [];

  for (let i = 0; i < this.length; i++) {
    newArray[i] = this[i] * 2;
  }

  return newArray;
}

console.log('multiplyByTwo result: ' +
myArray5.multiplyByTwo()); // Expected result: [2, 4, 6]
```

In this example, we added a custom method called `multiplyByTwo()` to the `Array.prototype` object. This method takes an array and returns a new array where each element is multiplied by two. We then called the `multiplyByTwo()` method on `myArray5`, which returns a new array with the all the elements of the original array multiplied by two.

> warning
>
> The `prototype` property of an array should be used with care. Adding custom methods and properties to the `Array.prototype` object can affect the behavior of all arrays in your codebase!

Built-in Methods

Before we get into the different built-in methods (which are essentially functions) for working with arrays, we need to talk about an important point. As mentioned earlier in this lesson, arrays can be stored in variables using either **let** or **const**. You may be wondering as you go through some of the following methods why we can still use **const** when we're seemingly changing the array, since variables declared with the **const** keyword can't be changed. Right?
>
Well, here is a key distinction: yes, we are "changing" the array with some of these methods, but we are not changing the *value of the variable*; it is still equal to an array. The only thing we're doing is modifying what's inside the array. This same distinction also applies to **objects**, which we will get to in the next lesson.

> With the `const` keyword, you cannot reassign the variable itself to a different value, but you can still modify the contents of the object or array that the variable refers to.

Let's now begin to look at the various built-in methods of arrays that we can make use of:
>
1. **push()**: Adds one or more elements to the end of the array and returns the new length of the array.

```
myArray5.push(4);
```

You can also push multiple elements at once:

```
myArray5.push(5, 6, 7);

console.log('myArray5 after push(): ' + myArray5); // Output:
[1, 2, 3, 4, 5, 6, 7]
```

> 2. **pop()**: Removes the last element from the array and returns it:

```
const removedElement1 = myArray5.pop();

console.log('myArray5 after pop(): ' + myArray5); // Output: [1,
2, 3, 4, 5, 6]
console.log('Removed element: ' + removedElement1); // Output: 7
```

> 3. **shift()**: Removes the first element from the array and returns it:

```
const removedElement2 = myArray5.shift();

console.log('myArray5 after shift(): ' + myArray5); // Output:
[2, 3, 4, 5, 6]
console.log('Removed element: ' + removedElement2); // Output: 1
```

> 4. **unshift()**: Adds one or more elements to the beginning of the array and returns the new length of the array:

```
myArray5.unshift(0);
```

You can also unshift multiple elements at once:

```
myArray5.unshift(-3, -2, -1);

console.log('myArray5 after unshift(): ' + myArray5); // Output:
[-3, -2, -1, 0, 2, 3, 4, 5, 6]
```

Properties and Methods (cont'd)

5. **concat()**: Joins two or more arrays and returns a new array:

```
const myArray6 = [1, 2, 3];
const myArray7 = [4, 5, 6];
const myArray8 = [7, 8, 9];
const combinedArray = myArray6.concat(myArray7, myArray8);

console.log('Combined array: ' + combinedArray); // Output: [1,
2, 3, 4, 5, 6, 7, 8, 9]
```

> 6. **slice()**: Returns a new array containing a portion of the original array:

```
const myArray9 = [1, 2, 3, 4, 5];
const slicedArray = myArray9.slice(1, 4);

console.log('myArray9 after slice()' + slicedArray); // Output:
[2, 3, 4]
```

The **slice** method takes two arguments: the starting index and the ending index (exclusive). If the ending index is not specified, the method will return all elements from the starting index to the end of the array.
>

7. **splice()**: Adds or removes elements from the array at a specific index:

```
const myArray10 = [1, 2, 3, 4, 5];
myArray10.splice(2, 1);

console.log('myArray10 after splice()' + myArray10); // Output:
[1, 2, 4, 5]
```

The **splice** method takes (usually) two arguments: the starting index and the number of elements to remove. You can also add elements to the array at the specified starting index by passing additional arguments after the second argument:

```
myArray10.splice(2, 0, "a", "b", "c");

console.log('myArray10 after adding elements with splice()' +
myArray10); // Output: [1, 2, 4, "a", "b", "c", 5]
```

> 8. **indexOf()**: Returns the first index at which a specified element can be found in the array, or -1 if it is not present:

```
let myArray11 = [1, 2, 3, 4, 5];
let index1 = myArray11.indexOf(3);

console.log('Index of the number 3 in myArray11: ' + index1); //
Output: 2
```

> 9. **lastIndexOf()**: Returns the last index at which a specified element can be found in the array, or -1 if it is not present:

```
let myArray12 = [1, 2, 3, 4, 5, 3];
let index2 = myArray12.lastIndexOf(3);

console.log('Last index of the number 3 in myArray12: ' +
index2); // Output: 5
```

10. **forEach()**: Calls a function for each element in the array. For example:

```
let myArray13 = [1, 2, 3, 4, 5];

myArray13.forEach(function(element) {
  console.log(element);
});

// Output:
/*
  1
  2
  3
  4
  5
*/
```

> 11. **map()**: Calls a function for each element in the array and returns a new array with the results:

```
const newArray1 = myArray13.map(function(element) {
  return element * 2;
});

console.log('newArray1: ' + newArray1); // Output: [2, 4, 6, 8,
10]
```

12. **filter()**: Calls a function for each element in the array and returns a new array with the elements that pass the test. For example:

```
const newArray2 = myArray13.filter(function(element) {
  return element > 2;
});

console.log('newArray2' + newArray2); // Output: [3, 4, 5]
```

13. **reduce()`**: Calls a function for each element in the array and returns a single value that is the result of the function. For example:

```
const sum = myArray13.reduce(function(total, element) {
  return total + element;
}, 0);

console.log('sum: ' + sum); // Output: 15
```

The **reduce** method takes two arguments: a function that adds the current element to the running total, and an initial value of 0 for the running total.
>

Additional Resources

Additional Resources for Learning JavaScript Arrays

This page provides a list of external resources that you can use to enhance your understanding of JavaScript arrays.

If you're having trouble with arrays, focus on learning the basics of creating and manipulating arrays before moving on to more advanced topics like array methods.

Room to Grow

Use these links to help with the basics.

Arrays Tutorial

A comprehensive tutorial from W3Schools to reinforce what you have learned today and continue practicing with additional examples.

Array Methods and Properties

A listing of methods and properties available to use on arrays.

Lift Off!

Once you are ready for the next step, use these links to continue practicing and deepen your knowledge.

Array Const

Further reading and practice of the `const` keyword.

Array Constructor

A brief definition and usage of the `constructor` property.

Array Methods Practice

Examples and practice for several array methods.

Conclusion and Takeaway

> Arrays are a fundamental data structure in JavaScript and you will find yourself using them quite extensively in your code. They are extremely useful for storing several related values together, which we can then modify or manipulate in a variety of ways. In this lesson, we covered the various syntaxes which can be used to create arrays, how to access elements in arrays using the given element's index, as well as the different built-in properties and methods of arrays. Next up, we will do a similar deep dive into objects!

Objects: Introduction

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The different ways to create object literals in JavaScript
> - The basics of the **this** keyword and how it applies to object literals
> - Various built-in properties and methods of object literals
>
> ### Introduction
>
> An object literal is a complex data type which provides a way to store and manipulate related data using what are called **key/value pairs**. Object literals are a core concept in JavaScript and, like arrays, you will find yourself using them a lot. In this lesson, we will explore the syntax of object literals and many of their built-in properties and methods, as well as the **this** keyword.

> **NOTE**: 'Object literal' is an important name distinction because, technically, in JavaScript almost *everything* is an object! There are a few exceptions, of course, such as **null**, **undefined**, **strings**, **numbers**, **booleans**, and **symbols**. That being said, for simplicity's sake, going forward in the lesson we will refer to 'object literals' as 'objects'.

Creating Objects

> In JavaScript, objects are created using a set of curly braces `{}`. Within the curly braces, you can define a list of properties and their values, separated by a comma. The syntax for creating an object using the object literal syntax looks like this:

```
let person1 = {
  name: 'John',
  age: 30,
  city: 'New York'
};

console.log('person1: ', person1);
```

In this example, we define an object called **person** with three properties: **name**, **age**, and **city**. The values for these properties are '**John**', **30**, and '**New York**', respectively.

Another way to create an object is to use the **Object()** constructor:

```
let person2 = new Object();

person2.name = 'Alice';
person2.age = 28;
person2.city = 'Wonderland';

console.log('person2: ', person2);
```

In this example, we create a new object called **person2** using the **Object()** constructor. We then added three properties to the object using dot notation.

In addition to these two ways of creating objects, we can also create an object using what is called a *constructor function*. Here what that looks like:

```
function Person(name, age, city) {
  this.name = name;
  this.age = age;
  this.city = city;
}


let person3 = new Person('Bruce Wayne', 45, 'Gotham City');
let person4 = new Person('Clark Kent', 32, 'Metropolis');


console.log(person3); // Output: { name: 'Bruce Wayne', age: 45,
city: 'Gotham City' }
console.log(person4); // Output: { name: 'Clark Kent', age: 32,
city: 'Metropolis' }
```

In this example, we created a constructor function called **Person** that takes three parameters: **name**, **age**, and **city**. Inside the constructor function, we use the **this** keyword to define properties with the same name as the parameters. We then created two new objects using the **new** keyword and the **Person** constructor function.
>

Quick Primer on the ‘this’ Keyword

> You’ve seen the **this** keyword used a couple of times already, and perhaps you wondered what the heck **this** refers to. In JavaScript, the **this** keyword refers to the object that is currently executing a given piece of code. In general, it is a reference to the object that “owns” the code being executed.

When used inside an object method, **this** refers to the object that the method is called on. Let’s look at **this** by creating another person object and adding a method that console logs the person’s name:

```
let person5 = {
  name: 'Frodo Baggins',
  age: 50,
  home: 'The Shire',
  printName: function() {
    console.log(this.name)
  }
}
```

Here, **this** refers to our person object, because it “owns” the **printName()** method.
>
When used inside a function that is not a method of an object or a constructor function as we saw a few moments ago, **this** refers to the global object (which, in the case of the web browser, is the **window** object – more on that in a later lesson).
>

Accessing Properties and Methods

> ### Accessing Properties and Methods
>
> In JavaScript, properties can be accessed using dot notation or bracket notation. Let's take a look at this using the Frodo object we created a moment ago:

```
console.log('Name: ' + person5.name); // Output: 'Frodo Baggins'
console.log('Age' + person5['age']); // Output: 50
```

Here we are accessing the **name** property using *dot notation* and the **age** property using *bracket notation*. Note that with bracket notation, we need to surround the property name with quotes.

> Both dot notation and bracket notation are valid ways to access object properties, and they can be used interchangeably. One example of when to use bracket notation is when the property name contains special characters or spaces, as these cannot be used with dot notation.

> To access methods, which are simply functions that are stored inside a particular object (or class, which we will learn about in a later module), we would typically use dot notation. Let's look at another example:

```
let wizard = {
  name: 'Harry Potter',
  age: 13,
  defendAgainstDementors: function() {
    console.log('Expecto patronum!');
  }
};

wizard.defendAgainstDementors(); // Output: 'Expecto patronum!'
```

In the example above, **defendAgainstDementors** is a method defined on our **wizard** object, in exactly the same way as properties are defined, except we use a function for the value.
>

Built-in Object Methods

> JavaScript has several built-in object methods that can be used to manipulate objects in various ways. In this lesson, we will explore some of the most common built-in methods.

Object.assign()

The **Object.assign()** method is used to copy the values of all enumerable properties from one or more source objects to a target object:

> Enumerable properties are the properties of an object that can be accessed using a loop.

```
const target = {
  a: 1,
  b: 2
};


const source = {
  b: 4,
  c: 5
};


Object.assign(target, source);


console.log(target); // { a: 1, b: 4, c: 5 }
```

In this example, we created a **target** object with two properties, **a** and **b**. We also created a **source** object with properties **b** and **c**. We then use **Object.assign()** to copy the properties from **source** to **target**. The value of **b** in **target** is overwritten with the value from **source**, while **a** and **c** are added as new properties.
>

Object.hasOwnProperty()

The **hasOwnProperty** method is used to determine whether an object has a property with a specified name. This method returns a boolean value indicating whether the object has the specified property as a direct property of that object.

```
const obj = {
  name: "John",
  age: 30,
  country: "USA"
};

console.log(obj.hasOwnProperty("name")); // Output: true
console.log(obj.hasOwnProperty("gender")); // Output: false
```

> ### Object.keys()
>
> The **Object.keys()** method returns an array of all the enumerable property names of an object.

```
const objectKeys = Object.keys(obj);

console.log(objectKeys); // Output: ["name", "age", "country"]
```

Object.values()

The **Object.values()** method returns an array of all the enumerable property values of an object.

```
const objectValues = Object.values(obj);

console.log(objectValues); // Output: ["John", 30, "USA"]
```

Object.entries()

The **Object.entries()** method returns an array of all the enumerable properties of an object in the form of an array, where each sub-array contains two elements: the first is the property key, and the second is the property value.

```
const objectEntries = Object.entries(obj);

console.log(objectEntries); // Output: [["name", "John"],
["age", 30],
["country", "USA"]]
```

Additional Resources

Additional Resources for Learning JavaScript Objects

This page provides a list of external resources that you can use to enhance your understanding of JavaScript objects.

Make sure you have a solid understanding of what an object is, how to create one, and how to access its properties and methods. Review the syntax and examples until you feel comfortable with these concepts.

Room to Grow

Use these links to help with the basics.

JavaScript Objects

A tutorial from W3Schools to reinforce what you have learned today and continue practicing with additional examples.

Lift Off!

Once you are ready for the next step, use these links to continue practicing and deepen your knowledge.

The JavaScript this Keyword

Definition and usage of the `this` keyword.

Object Methods Practice

Examples and practice for several object methods.

Conclusion & Takeaway

> In this lesson, we covered the basics of objects in JavaScript, including the different ways we can create objects, how to define and access properties and methods in objects, as well as some of the built-in methods JavaScript gives us for working with objects in a variety of ways. Like arrays, objects are an important data structure and are used extensively in web development. Having now covered many of the fundamentals of the JavaScript language, in the next lesson we will begin looking at how we can use JavaScript to interact with and manipulate our HTML and CSS!

Web APIs (BOM and DOM)

Goals

By the end of this lesson, you should understand:

- What the Browser Object Model (BOM) and Document Object Model (DOM) are
- Various BOM and DOM methods
- Different ways to select DOM elements in JavaScript
- The basics of DOM manipulation and event handling

Introduction

Web APIs (Application Programming Interfaces) allow web developers to interact with various parts of a web browser or a web page. There are many Web APIs available in JavaScript, but the two we will focus on for now are the Browser Object Model (BOM) and the Document Object Model (DOM). In this lesson, we will explore these two APIs and their importance in web development.

Browser Object Model (BOM)

> The Browser Object Model (BOM) is a JavaScript API that provides access to the browser window and other browser-related objects. The BOM is not standardized, which means that the objects and methods available can vary between different browsers.

Here, we will discuss three common objects in the BOM: the window object, the location object, and the history object.
>
Window Object:
The window object represents the browser window that contains the current web page. Some of the most commonly used properties of the window object are:
- **window.innerWidth** - Returns the width of the browser window, excluding the toolbars and scrollbar.
- **window.innerHeight** - Returns the height of the browser window, excluding the toolbars and scrollbar.
- **window.document** - Returns a reference to the Document object for the current web page.
- **window.localStorage** - allows you to access the local storage object in the user's browser (more on this in a later module!)
>
Location Object:
The location object represents the URL of the current web page and provides methods for working with it. Some of the most commonly used methods of the location object are:
- **location.reload()** - Reloads the current web page.
- **location.assign(**url**)** - Loads a new web page at the specified URL.
- **location.replace(**url**)** - Replaces the current web page with a new web page at the specified URL. Note that when using the **replace()** method you cannot hit the back button to go back to the original page!
>
History Object:
The history object provides access to the browser's session history, allowing you to navigate forward and backward through the user's browsing history. Some of the most commonly used methods of the history object are:
>
- **history.back()** - Moves the user back one page in their browsing history.
- **history.forward()** - Moves the user forward one page in their browsing history.
- **history.go(**number**)** - Moves the user to a specific page in their browsing history, where number is the number of pages to move forward or backward. For example, **history.go(-2)** would move the user back two pages.

Document Object Model (DOM)

> What is the Document Object Model? The Document Object Model (DOM) is a programming interface for HTML documents. It represents the web page as a hierarchical tree-like structure, where each element in the tree corresponds to a part of the web page.
>
> Why is the DOM important? Well, it provides a way to dynamically update the content and style of a web page in response to user actions or other events. Without the DOM, it would be impossible to create the dynamic and interactive web applications that we see today. If functions are the bread and butter of JavaScript, then the DOM (or, more specifically, *manipulating* the DOM), is the porterhouse steak (sorry, vegetarians!)
>
> ### Node Types
>
> In the DOM, each item in the HTML document is represented by a node, and there are several different types of nodes that can be present in the document tree. The most common node types are:

1. Element nodes - These represent HTML elements, such as `<div>`, `<p>`, and `<img>`.
2. Attribute nodes - These represent the attributes of an HTML element, such as **src**, **href**, and **class**.
3. Text nodes - These represent the text content of an HTML element, such as the text within a `<p>` element.
4. Comment nodes - These represent comments within the HTML document, which are not displayed on the web page.
   >

Elements vs. Nodes

It's important to note that elements are a type of node, but not all nodes are elements. Elements are nodes that represent HTML elements, such as `<div>`, `<p>`, and `<img>`. Non-element nodes, such as text nodes and comment nodes, do not represent HTML elements.
>

Selecting Elements

> Before we can manipulate an HTML element with JavaScript, we need to select it from the DOM. There are several ways to do this, including:
> - **getElementById()**: This method selects an element with a specific ID attribute.
> - **getElementsByClassName()**: This method selects all elements with a specific class name.
> - **getElementsByTagName()**: This method selects all elements with a specific tag name.
> - **querySelector()**: This method selects the first element that matches a specific CSS selector.
> - **querySelectorAll()**: This method selects all elements that match a specific CSS selector.
>
> Let's get some practice with these by selecting elements from the HTML file we have open. In **script.js** add the following:

```
// Select an element by ID
const header = document.getElementById('header');


// Select all elements with a class name
const paragraphs = document.getElementsByClassName('paragraph');


// Select all elements with a tag name
const asideElements = document.getElementsByTagName('aside');


// Select the first element that matches a CSS selector
const firstMatchingElement = document.querySelector('#footer');


// Select all elements that match a CSS selector
const allMatchingElements = document.querySelectorAll('.p2');
```

In general, which of these methods we use to select elements depends to some degree on preference and more so on the situation; however, the most common ones to use (in *most* scenarios) are: `getElementById()`, `querySelector()`, and `querySelectorAll()`. The point here is just to get some practice with all of them.

Next, we'll have a look at handling user events for each of these selected elements!
>

Intro to Event Handling

> Events are actions or occurrences that happen in the browser, such as clicking a button or submitting a form. JavaScript enables us to handle these events and perform actions in response to them.

We can add event listeners to elements using the **addEventListener()** method. This method takes two arguments: the name of the event, and the function to be executed when the event occurs.

Here is the syntax to create an event listener for the 'click' event (there are many other kinds of events, which we'll cover in more depth in a later module):

```
// Add a click event listener to an element
myElement.addEventListener('click', function() {
  // some code to run when the element is clicked
});


// You can also create event listeners using an arrow function
myOtherElement.addEventListener('click', () => {
  // some code to run when the element is clicked
});
```

Now, let's try adding 'click' event listeners to the elements we selected previously. For now, we'll just create the event listeners themselves; we won't add any code inside them just yet.

> **NOTE**: Because some of our element selection methods select multiple elements at once, we'll have to use a **for** loop to iterate over each one and add an event listener!

```
// getElementById event listener
header.addEventListener('click', () => {
   // event listener code
});


// getElementsByClassName event listener
for (i = 0; i < paragraphs.length; i++) {
  paragraphs[i].addEventListener('click', (e) => {
    // event listener code
  });
}


// getElementsByTagName event listener
for (i = 0; i < asideElements.length; i++) {
  asideElements[i].addEventListener('click', (e) => {
    // event listener code
  });
}


// querySelector event listener
firstMatchingElement.addEventListener('click', () => {
  // event listener code
});


// querySelectorAll event listener
for (i = 0; i < allMatchingElements.length; i++) {
  allMatchingElements[i].addEventListener('click', (e) => {
   // event listener code
  });
}
```

Notice that for some of these, inside the parentheses of the `addEventListener` callback function, we added an “e”. This is a parameter representing the event which triggers the function. We called it “e” (for “event”) but you could give it whatever name you want. Using “e” is just a common convention.

The reason we added this parameter variable is because for those event listeners, in order to make them work correctly, we will have to perform our modification of the corresponding DOM element using the event *target* – or `e.target`. This will make more sense shortly!

DOM Manipulation: First Steps

Manipulating Styles Directly

JavaScript allows us to manipulate CSS styles for HTML elements. We can do this using the style property of an element, which contains an object representing the element's styles.

We can set individual style properties using the syntax **element.style.property.** For example:

```
// Set the background color of an element
myElement.style.backgroundColor = 'red';

// Set the font size of an element
myElement.style.fontSize = '1.25rem';
```

We can also use the **getComputedStyle()** method to get the computed styles of an element. This method returns an object representing the styles applied to the element, including styles inherited from parent elements.

Here is an example:

```
// Get the computed styles of an element
  const myStyles = getComputedStyle(myElement);

  // Get the background color of an element
  const myBackgroundColor = myStyles.backgroundColor;
```

Manipulating CSS Classes

As you know, CSS classes are great for applying styles to groups of elements. JavaScript provides a few methods for working with CSS classes, instead of setting styles directly:
>
- **classList.add()**: This method adds a class to an element's class list.
- **classList.remove()**: This method removes a class from an element's class list.
- **classList.toggle()**: This method toggles a class in an element's class list, removing or adding it depending on whether it is present or not when the method is invoked
>
Here are some examples of these methods:

```
 // Add a class to an element
  myElement.classList.add('my-class');

  // Remove a class from an element
  myElement.classList.remove('my-class');

  // Toggle a class in an element
  myElement.classList.toggle('my-class');
```

Add to our code

Now, let's apply some of the above concepts and add some code inside our event listeners:

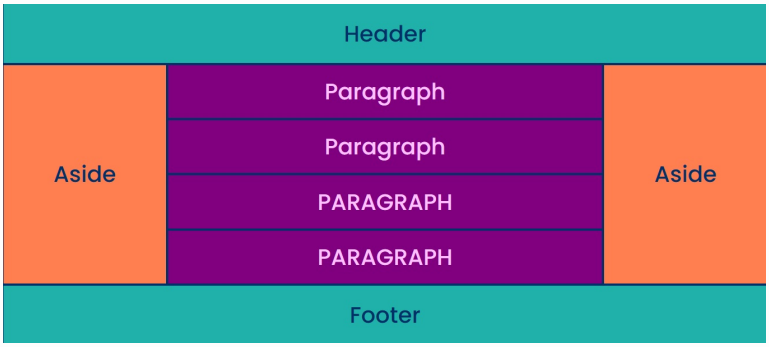
```
// getElementById event listener
header.addEventListener('click', () => {

   header.style.backgroundColor = "lightseagreen";

});



// getElementsByClassName event listener
for (i = 0; i < paragraphs.length; i++) {
  paragraphs[i].addEventListener('click', (e) => {

    e.target.classList.toggle('purple');

  });
}



// getElementsByTagName event listener
for (i = 0; i < asideElements.length; i++) {
  asideElements[i].addEventListener('click', (e) => {

    e.target.classList.add('coral')

  });
}



// querySelector event listener
```

```
firstMatchingElement.addEventListener('click', () => {

  firstMatchingElement.style.backgroundColor = "lightseagreen";

});



// querySelectorAll event listener
for (i = 0; i < allMatchingElements.length; i++) {
  allMatchingElements[i].addEventListener('click', (e) => {

   e.target.style.textTransform = 'uppercase';

  });
}
```

You may want to click the button to open the preview in a new browser tab, so things are a little easier to see. After doing that, go ahead and start clicking on elements to see what our code does!

DOM Manipulation: Wrap-up

If we did everything correctly, after clicking once on every element, this is what it should look like:



DOM manipulation example

Well, what we have created is... truly hideous 😱😆

But, for the time being, that doesn't matter! The point of this exercise was just to get a taste of manipulating the DOM. There is *much* more we can do with regards to DOM manipulation, and we will delve more and more into it as we continue learning JavaScript!

Conclusion & Takeaway

> In this lesson, we covered the Browser Object Model and some of its objects: the window object, the location object, and the history object, as well as the most commonly used properties and methods of these objects. We then went over the Document Object Model (DOM), which provides a way for JavaScript to interact with the contents of a web page. In addition, saw how we can select elements and attach event listeners to them, then finished up by working through a basic example of DOM manipulation. In the next lesson, we will get more practice working with the DOM by going through some code-along exercises!

Mini-Project: Dark/Light Mode Toggle

> ### Goals
>
> In this exercise, you will:
>
> - Get some practice implementing several of the concepts from this module
> - Build a dark/light mode toggle
>
> ### Introduction
>
> Here is where the proverbial rubber begins to meet the road! In this lesson, we will build the first of two small projects to practice what we've learned. The first project will be simple dark/light mode toggle! By building out this very common UI component, we will have a chance to implement a lot of the topics covered throughout this module, particularly the basics of DOM manipulation.

Set up the HTML

> Let's get started by writing the HTML for our toggle switch. Inside **index.html**, underneath the `<h1>` element in the `<body>`, add the following:

```
<div class="container">
  <h2 class="mode-status">Currently in Light Mode</h2>
  <label class="toggle-switch">
    <input type="checkbox" id="mode-toggle">
    <span class="slider"></span>
  </label>
</div>
```

It's a very basic setup; not a lot going on in terms of elements, but that's fine for this project. We have a `<div>` with the class of **container**, which houses an `<h2>` that indicates the current theme (dark or light), and finally a toggle switch composed of a `<label>`, a `<span>`, and an checkbox `<input>` element.

Add the CSS

Next, let's switch to our **style.css** file. Some default styles are already defined for the body, headings, and container.

We'll begin by adding a class called **.dark-mode** which we will utilize in our JavaScript to toggle the dark/light theme, and we will also create styles for the different elements that make up the toggle switch:

```
.dark-mode {
  background-color: #333;
  color: #f0f0f0;
  transition: all 0.4s ease-in;
}

.toggle-switch {
  position: relative;
  display: inline-block;
  width: 3.75rem;
  height: 2.13rem;
  margin-bottom: 0.63rem;
  margin-inline: auto;
}

.toggle-switch input {
  opacity: 0;
  width: 0;
  height: 0;
}

.slider {
  position: absolute;
  cursor: pointer;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background-color: #ccc;
  transition: all 0.4s ease-in;
  border-radius: 2.13rem;
}

.slider:before {
  position: absolute;
```

```
  content: "";
  height: 1.63rem;
  width: 1.63rem;
  left: 0.25rem;
  bottom: 0.25rem;
  background-color: white;
  transition: 0.4s;
  border-radius: 50%;
}

input:checked + .slider {
  background-color: #2196f3;
}

input:focus + .slider {
  box-shadow: 0 0 2px #2196f3;
}

input:checked + .slider:before {
  transform: translateX(1.63rem);
}
```

If you take a look at the 'dark-mode' class we defined, you'll notice that different colors will get applied to the background and the text whenever that class is present, and we're using a transition of 0.4 seconds with the 'ease-in' timing function so that the change between dark mode and light mode will appear smooth.

The **.toggle-switch** class positions the toggle switch container. It uses `position: relative`; so that the inner elements of the switch can be positioned absolutely within it. We also target the input inside the element with the **.toggle-switch** in order to hide it and use our own custom styles for the switch.

The **.slider** class actually represents the background of our custom toggle switch. We set `cursor: pointer` to indicate it's clickable. The 'background-color' sets its initial color, and the transition, again, ensures a smooth color change. We also apply some styles to the **:before** pseudo-element, which represents the actual switch that moves left and right, setting its position to 'absolute', relative to its parent – the **.slider** element. It will have a white background initially and transition smoothly to the dark background.

Then with the **input:checked** + **.slider** rule, we add styles so that when the checkbox is checked (i.e., when the user activates the switch), the background color of the **.slider** element changes.

With the **input:focus** + **.slider** rule, we're saying that when the checkbox (which is hidden but still focusable) gains focus a slight blue shadow should be applied around the slider, providing a visual cue for accessibility.

And finally, the **input:checked** + **.slider:before** rule moves the pseudo-element (the switch part that actually moves) 1.63rem to the right when the checkbox is checked, thereby indicating that the toggle switch is activated.
>

Add the JavaScript

> Now, let's hop into our **script.js** file. The first thing we need to do is select some elements from the DOM:

```
const body = document.querySelector('body');
const modeToggle = document.getElementById('mode-toggle');
const modeStatus = document.querySelector('.mode-status');
```

> Here we are accessing both the **body** element and an **h2** element using the **querySelector** method, and also the **input** element with an **id** of mode-toggle using **getElementById**.

Next, let's add a JavaScript function called **toggleMode** with the logic to switch between the dark and light mode based on the checkbox's state (checked or unchecked):

```
function toggleMode() {
  body.classList.toggle('dark-mode');

  const modeMessage = body.classList.contains('dark-mode') ?
    'Dark Mode'
    : "Light Mode"

  modeStatus.innerText = "Currently in " + modeMessage;
}
```

Great! Now the last thing we need to do is attach an click listener to our switch that invokes our `toggleMode` function as a callback:

```
modeToggle.addEventListener('click', toggleMode);
```

To recap, here's a breakdown of what our JavaScript code does:

- Gets references to the `<body>`, the toggle `<input>`, and the `<h2>` where we'll display the current theme
- Defines a function called `toggleMode` that toggles the '.dark-mode' class on the `<body>` when the switch is clicked
- Uses a ternary expression to update the `<h2>` text to indicate the current theme, based on whether the body element's classList contains 'dark-mode' or not. If it does, the text is set to "Currently in Dark Mode"; otherwise, the text is set to "Currently in Light Mode".
- Attaches an event listener to the switch, which calls the `toggleMode`

function when clicked
>
Now, refresh the browser preview in Codio and try clicking the switch! Then, copy and paste your HTML, CSS and JavaScript code into the respective boxes below and submit.
>

Conclusion and Takeaway

> In this code-along lesson, we covered how to create a Dark/Light Mode toggle feature, starting basically from scratch and adding all of the HTML, CSS, and JavaScript to accomplish this functionality. In the next lesson, we'll build another small feature commonly found in many websites: an image slider!

Mini-Project: Image Slider

> ### Goals
>
> In this exercise, you will:
>
> - Get more practice with the concepts from this module
> - Build a simple image slider
>
> ### Introduction
>
> For this mini-project, we will build an image slider –- which you've also likely seen on many websites and will likely have occasion to include in many future projects. We will add our own flair to this component by having the images fade in and out, instead of *sliding* in and out. Not that there's anything wrong with the traditional sliding animation – it just never hurts to add a personal spin on common UI components! This will also give us the chance to get more practice with DOM manipulation.

Set up the HTML

> As with the dark/light mode toggle mini-project, let's start by writing the markup in **index.html**. Just below the opening `<body>` tag, add the following:

```
  <div class="container">
    <div class="project-heading">
      <h1 class="title">Image Slider</h1>
      <div class="subtitle">Click the buttons to change the
image!</div>
    </div>

    <div id="slider">
      <img src="https://picsum.photos/id/1005/500/400"
alt="Image 1">
      <img src="https://picsum.photos/id/1012/500/400"
alt="Image 2">
      <img src="https://picsum.photos/id/1015/500/400"
alt="Image 3">
      <img src="https://picsum.photos/id/1024/500/400"
alt="Image 4">
      <button id="prev">&lt;</button>
      <button id="next">&gt;</button>
    </div>
  </div>
```

> If you would like, feel free to upload four of your own images into Codio and then update the **src** attributes on the `<img>` elements accordingly!
>
> Other than that, our markup here is pretty straightforward – we have an outer container `div` with two child elements: one with class of **project-heading** that holds a title and subtitle element, and the other with an id of **slider** which has all our `img` elements and the buttons to change the image being displayed. Next, we'll add some styling to this to get it looking right!

Add the CSS

> The first thing we need to do is add the basic styling for **#slider** element:

```
#slider {
  position: relative;
  width: 500px;
  height: 400px;
}
```

> This sets the slider to have a relative position, which is crucial for child elements that will have an absolute position. The dimensions are set to 500 pixels by 400 pixels, providing a canvas where our images and buttons will reside.
>
> Now,
> let's move on to styling the images *inside* the slider:

```
#slider img {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  opacity: 0;
  transition: all 1s ease-in-out;
}
```

> Here, each image is set to have an absolute position so that they stack on top of each other within the **#slider** element. The opacity is set to zero, hiding all images initially, and any change in any property will transition smoothly over one second, thanks to the transition property.
>
> Now, let's also define a ruleset for the "active" image (meaning the image currently displayed):

```
#slider img.active {
  opacity: 1;
}
```

> The **.active** class will make the associated image fully visible by setting its opacity to 1.
>
> Next up, let's style the slider buttons, beginning with basic styles which should apply to *both* buttons:

```
button {
  position: absolute;
  top: 50%;
  transform: translateY(-50%);
  border: 3px solid #fff;
  border-radius: 50%;
  width: 70px;
  height: 70px;
  background-color: #717171;
  color: #f1f1f1;
  font-size: 32px;
  font-stretch: condensed;
  cursor: pointer;
  transition: all 200ms ease-in-out;
}
```

> This will position our buttons absolutely within the slider, as well as vertically centering them. We're also saying here that the buttons should be round (`border-radius: 50%`), with a diameter of 70 pixels (set via the width and height properties), and should have a solid white border with a dark grey background. We also added a transition so any changes (i.e. - hover) will be smooth instead of choppy, and added a font-stretch of **condensed** to make the arrows look a little better.
>
> Speaking of 'hover', let's add a ruleset for that:

```
button:hover {
  background-color: steelblue;
}
```

> Now, when we hover over the button with our mouse the background changes to the color 'steelblue', giving a visual indication that the button is interactive.
>
> Finally, we tweak the individual "Previous" and "Next" buttons:

```
button#prev {
  left: -35px;
  padding: 7px 17px 7px 15px;
}

button#next {
  right: -35px;
  padding: 7px 15px 7px 17px;
}
```

> These specific rules ensure that the “Previous” and “Next” buttons are slightly outside the main slider box, making them easier to interact with and providing a unique look. Additionally, The padding here helps to center the arrows inside the buttons.
>
> On to JavaScript!

Add the JavaScript

> First, we need to select all the image elements within our slider, as well as the “Previous” and “Next buttons”:

```
const images = document.querySelectorAll('#slider img');
const previousImage = document.getElementById("prev");
const nextImage = document.getElementById("next");
```

> Next, we’ll need to create a variable named **currentIndex** which we will use to keep track of which image is currently visible, and initialize it to **0**. We’ll use **let** for this, since the value will change:

```
let currentIndex = 0;
```

> Now let’s create a reset function which will be responsible for removing the **active** class from all images. We’ll use a *for* loop to iterate over the image elements:

```
function reset() {
  for (let i = 0; i < images.length; i++) {
    images[i].classList.remove('active');
  }
}
```

> We also need to create a function to set the initial state of our slider when the page loads. Let’s call it **initializeSlider**, and inside we will first call the **reset** function to begin with a clean slate, then add the **active** class to the first image using the **currentIndex** variable:

```
function initializeSlider() {
  reset();
  images[currentIndex].classList.add('active');
}
```

> Now we need to define two functions to handle navigating through the images – one called **slideLeft** and one called **slideRight**:

```
function slideLeft() {
  reset();
  currentIndex--;
  if (currentIndex < 0) {
    currentIndex = images.length - 1;
  }
  images[currentIndex].classList.add('active');
}

function slideRight() {
  reset();
  currentIndex++;
  if (currentIndex >= images.length) {
    currentIndex = 0;
  }
  images[currentIndex].classList.add('active');
}
```

> The **slideLeft** function moves to the previous image by decrementing the **currentIndex** variable and adding the **active** class to the image at that index. It also checks if the current index is less than 0 and, if so, it will set **currentIndex** to the last image using **images.lenth - 1**.
>
> Conversely, the **slideRight** moves to the next image by *incrementing* **currentIndex**, adds the **active** class, but if **currentIndex** is greater than **images.length** it will reset the variable's value to 0.
>
> Lastly, we need to call our **initializeSlider** function and add event listeners to the "Previous" and "Next" buttons so they actually *do* something when clicked:

```
initializeSlider();

previousImage.addEventListener('click', function() {
  slideLeft();
});

nextImage.addEventListener('click', function() {
  slideRight();
});
```

> And there you have it – our image slider is done! Let's refresh the browser preview and give it a try!
>
> Go ahead and submit your HTML, CSS, and JavaScript below:

Conclusion & Takeaway

> In this lesson we built a simple image slider component utilizing basic DOM manipulation concepts learned in this module. As with the first mini-project, we got practice with selecting DOM elements, defining functions, and adding event listeners to make the selected elements respond to user interaction. Though these were “simple” projects, we incorporated a lot of what we learned.
>
> None of this is considered easy! It’s okay to struggle and ask lots of questions. It is important to be sure you understand what is happening in the code, and how the pieces work together.

Attribution

- David Marby (https://dmarby.se) & Nijiko Yonskai (https://github.com/nijikokun). (n.d.). Lorem Picsum. Lorem Picsum. https://picsum.photos/

Project: Add JavaScript to your personal website

Instructions

For project 9 there are two steps outlined below, the first of which is required. Step Two, while not strictly *required*, is highly **suggested**:

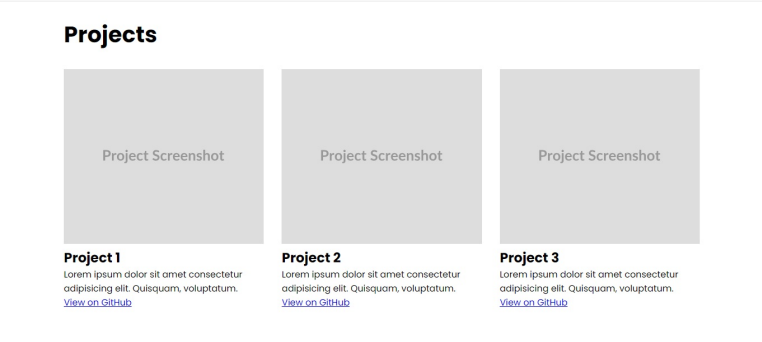
Step One

Implement some of what you learned in this module in your personal website!
>
Not sure where to start? One suggestion would be to add either an **image slider** or a **dark/light mode toggle** (or both), using the mini-projects you just completed as a guide. You can style them differently if you want, but the functionality should be same!
>
Alternatively, you can do some independent research and come up with an entirely different piece of functionality you'd like to add to your website. For example, you could research how to create a **modal** for your contact form, instead of having it on the page directly.
>
You can decide which of these suggestions you want to incorporate, but **pick at least one**.

> **Note**: if you decide to implement the dark/light mode toggle, be aware of contrast and accessibility! Be sure that the contrast between the darker background and the light text is enough so that things are readable, but choose colors for both the background and the text that don't create a strain on the eyes.

Step Two (Optional, but highly suggested!)

Build out a projects section for your website (or a projects *page*, if your website has multiple pages). It doesn't have to be too complicated, but choose 2 to 3 things you've coded (either from this course or from outside this course), push the code to GitHub, and then for each include the following elements on your website: an **image** (screenshot of the UI component or full page you built), a **title**, a **brief description**, and a **link to the GitHub repository**. The finished result should look something like this:



Projects section example

> **HINT**: You can achieve this kind of layout using either Grid or Flexbox.

The idea is to keep updating this section (or page) over time as you build more and more complex projects.

When you're finished with both steps, submit your HTML, CSS, and JavaScript below.

Happy coding!
>

> It always seems impossible until it is done.
> –Nelson Mandela