

# Comments & Shortcuts

---

All programming languages have commenting functionality. Comments are used for documenting your code and explaining things in a human readable way. Often times, multiple developers work on the same code base and comments are used to explain what a certain piece of code does.

Comments are also used to disable code. This is useful when you want to test something out without deleting the code.

You may also use comments as a todo list. This is useful when you want to keep track of what you need to do next.

## Single Line Comments

In JavaScript, single line comments are created by using two forward slashes `//`. Anything after the two forward slashes will be ignored by the JavaScript interpreter. This is useful when you want to add a comment to a single line of code.

```
// This is a single line comment

console.log('Hello World'); // This is a single line comment
```

## Multi Line Comments

In JavaScript, multi line comments are created by using `/*` and `*/`. Anything between the two symbols will be ignored by the JavaScript interpreter. This is useful when you want to add a comment to multiple lines of code.

```
/*
This is a multi line comment
*/
```

## Useful Shortcuts

I just want to go over some helpful keyboard shortcuts. These work for VS Code, but many of them are universal and will work in most text editors.

- `shift + up/down` - Highlight lines of code up and down
- `shift + right/left` - Highlight code right and left
- `cmd + right/left arrow` - move cursor to beginning/end of line
- `cmd + up/down arrow` - move cursor to beginning/end of file
- `option + up/down arrow` - move line up/down
- `shift + option + up/down arrow` - copy line up/down
- `cmd + /` - Comment out a line of code

- `cmd + shift + /` - Comment out multiple lines of code
- `cmd + d` - Select the next instance of the selected word
- `cmd + shift + l` - Select all instances of the selected word
- `cmd + shift + f` - Search for a string in the project
- `cmd + shift + o` - Search for a file in the project

You aren't going to remember all of these right now. Try and use a few of them while you're coding and it will become second nature. I use these shortcuts all the time and they make my life a lot easier.

# Variables

---

Variables are containers for pieces of data. That data can be one of many different data types. It's important to know and understand those data types and we will go over them in the next lesson, but right now, we are just going to look at the syntax for creating and re-assigning variables as well as the differences between how we declare them.

## Declaring a Variable

In JavaScript, we need to first declare a variable with one of three keywords

- var
- let
- const

In modern JavaScript, you probably won't see `var` very much. `var` was the original declaration, but in [ES2015](#) also known as [ES6](#), which was a huge update to the language, they introduced `let` and `const`. One of the reasons for this is because of [block scoping](#). Now, I'm not going to talk about things like scope right now, because we haven't gotten to functions or anything, so if I talk about scope it'll go right over a lot of people's heads. I just want to focus on how we declare and assign variables and also work with constants. Just know that on the [global scope](#), meaning not inside of a function or any kind of control structure, `var` and `let` work in a very similar same way. `const` is a bit different. I'll be using `let` and `const` throughout the course, unless there is a specific reason for me to use `var`.

So let's say that we want a variable called `firstName` and `lastName`. Remember that [string](#) data types need to be wrapped in either single or double quotes. You can also use backticks. Backticks (`) have a special use which I'll talk about later.

```
let firstName = 'John';
let lastName = 'Doe';

// We can do a console.log to show the value
console.log(firstName, lastName);
```

We can store other data types such as numbers:

```
let age = 30;

console.log(age);
```

## Variable Naming Conventions

So different languages have different rules and conventions when it comes to naming things. There are some rules that you have to follow when it comes to the formatting of your variable names.

- Only letters, numbers, underscores and dollar signs
- Can not start with a number

I wouldn't suggest starting your variable names with a dollar sign or an underscore either.

## Multi-Word Variables

When it comes to variables as well as functions and classes with multiple words, it's really up to you on how you format the case. What you'll typically see in JavaScript and what I usually do is **camelCase**. This is where we start with a lowercase letter but every word after that starts with an uppercase letter.

```
let firstName = 'John';
```

You may also see underscores like this

```
let first_name = 'Sara';
```

There's also pascal case, where the first word is also capitalized. You typically see this for class names in object oriented programming.

```
let FirstName = 'Tom';
```

You might also see all lowercase, which I wouldn't recommend

```
let firtstname = 'Bob';
```

## Reassigning Values

Alright, so if we want to reassign a value we can do that here since we're using **var**. When it comes to directly reassigning a primitive type like a number, you can't use **const**. Const stands for **constant**. So you have to use **var** or **let** if you want to re-assign a variable value. Again, you're not going to see **var** very much, so I will use **let** for this.

```
let x = 100;
```

Let's reassign x to 200

```
x = 200;
```

Now, in some cases, you may want to simply declare a variable and not assign a value to it.

```
let score;
```

In this case, score would be equal to **undefined**, which is actually one of the seven primitive data types that we're gonna talk about in the next video.

If I want to assign a value to it at any time, I can.

```
score = 1
```

One reason you might do this is because you have a conditional that says if score equals one thing, do this, if another, then do something else. Here is an example

```
let score;

if(userScores) {
  score += 1;
} else {
  score -= 0;
}
```

## Constants

Alright, let's look at **const**, which works a bit differently than **let** or **var**. So if I declare a name like this...

```
const x = 100
```

and then I try and re-assign that value

```
x = 200; // Results in error
```

I get an error. You can't directly re-assign a value to a constant. You also can't initialize a constant as **undefined**.

```
const score1; // Results in error
```

That will also throw an error. It has to be declared with a value.

Now where this can be a little confusing is when we use `const` with values that are not primitive like objects or reference types such as `arrays` and `object literals`. In that case we can't directly re-assign, but we can change them.

So let's say that we have an array

```
const arr = [1,2,3,4]
```

What I can't do is re-assign

```
arr = [1,2,3,4,5] // Results in error
```

But I can for instance, add to that array with the `push` method. I can even take a specific index of the array and change the value that way.

```
arr.push(5); // [1, 2, 3, 4, 5]
arr[0] = 200; // [200, 2, 3, 4, 5]
```

We can do the same type of thing with object literals. Now I know some of you have no clue what an object literal is, we'll get to that, but I just want to show you that the object itself can be manipulated, even with `const`.

```
const person = {
  name: 'Brad',
};

person.name = 'John';
person.email = 'john@gmail.com';

console.log(person);

/*
Results in...
{
  name: 'John',
  email: 'john@gmail.com'
}
*/
```

## Declaring multiple values at once

We don't have to declare variables line by line, we are able to declare multiple values at once. With `let`, we can initialize, with `const`, we have to assign a value.

```
let a, b, c;  
const d = 10, e = 20, f = 30;
```

## Let or Const - Which to Use?

So how do we figure out which to use when it comes to let and const or even var? Well it comes down to preference. What I like to do is always use `const` unless it's a primitive value that I think I may need to re-assign at some point. The score example above is a good example. That score number will be re-assigned throughout the game. So I would use let. You'll find in most cases that you don't need to explicitly re-assign values. We're usually dealing objects where we manipulate them but don't re-assign them.

Some people do the opposite and always use `let` no matter what. Which is fine, but I think using `const` is a bit more robust because you know your values can't be re-assigned by mistake. So, I would suggest `const` unless you know you need to either initialize as undefined or re-assign. It's all preference though.

# Data Types

---

We're going to spend a little time on data types. When we create a variable, the data that we define has a type. There are **primitive types** and **reference types**. The big difference under the hood is the way that they're stored and accessed from memory.

1. **Primitive Types** - Stored directly in the location that the variables is accessed
2. **Reference Types (Objects)** - Accessed by reference

## Primitive Data Types

That may not make much sense right now, but in the next video I'm going to go more in depth on how this works under the hood. So first well talk about the 7 primitive types of JavaScript.

The 7 primitive data types are the following

- **String** - a sequence of characters. In JavaScript, strings can be enclosed within the single or double quotes
- **Number** - Represent both integer and floating-point numbers
- **Boolean** - Represent a logical entity and can have two values: true or false
- **Null** - Represents the intentional absence of any object value
- **Undefined** - A variable that has not been assigned a value is undefined.
- **Symbol** - It is a built-in object whose constructor returns a symbol-that is unique
- **BigInt** - New data type used for numbers that are greater than the Number type can handle.

## Dynamic vs Static Types

JavaScript is a "dynamically-typed" language. This means that when we create a variable or a function, we don't explicitly assign the type. JavaScript assigns the type of that value dynamically. In other words, the type is associated with the **value**, not the **variable**. So you can have a variable value be a string and then later in the script, change it to a number. You probably won't do that very often, but you can.

There are other languages that are "statically-typed". This is where you would explicitly define the type of data. Java is an example of a statically-typed language. There's also a language called TypeScript, which is essentially JavaScript with some extra features, including types. So in TypeScript, you could do this.

```
const y:number = 100
```

You can see we defined it as a **number**. So now that **y** variable's value HAS to always be a number.

This is not something we can do with vanilla JavaScript. The advantage of static types is that it makes your code more robust and less prone to errors. The downside is you need to write a bit more code.

## Assigning Variables

String

A string is a "string" of characters wrapped in either single or double quotes. Strings can include any number, letter or symbol

```
const firstName = 'Sara';
```

## Number

Any number in JavaScript is the **Number** type, including floats and decimals. Some languages have separate types for floats and integers. JavaScript does not. Numbers are not wrapped in quotes.

```
const age = 30;
const temp = 98.9
```

## Boolean

A boolean is a true or false value

```
const hasKids = true;
```

## Null

Intentional absence

```
const aptNumber = null;
```

## Undefined

Undefined represents a variable that has not been defined a value.

```
let score;
const score = undefined;
```

## Symbol

A "symbol" represents a unique identifier. We will talk more about what they are used for later, but this is how we can create one

```
const id = Symbol('id');
```

## BigInt

BigInt is a new primitive type and represents integers that are out of the range of the Number type

```
const n = 9007199254740991n;
```

## typeof Operator

To check the type of a variable in JavaScript, you can use the typeof operator

```
console.log(typeof name)
```

**Tip:** If you run typeof on a variable that holds `null`, you will not get null, you will get `object`. This is generally regarded as a mistake. More info [here](#)

```
console.log(typeof aptNumber)
```

# Reference Data Types (Objects)

---

We talked about the primitive data types that are stored directly in memory where they are accessed. Reference types work a little differently under the hood. There are basically 3 types that are passed by reference

- `Arrays`
- `Functions`
- `Objects`

Technically, these are all "objects", meaning that they have properties and methods.

Reference types or "objects" are a non-primitive value and when assigned to a variable, the variable is given a `reference` to that value. The reference points to the object's location in memory. Unlike primitives, where the variable contains the actual value. I'll talk more about this in the next video.

## Arrays

I know we haven't gone over arrays yet so don't worry if you've never worked with them. They're essentially a data structure that can hold multiple values.

So if we write

```
const numbers = [1,2,3,4]
```

We have created an array in memory and a variable that points to the address or location of that array.

## Object Literals

Objects are comma separated lists of **name-value pairs**. We'll get into them later, but just to give you an example here

```
{  
  name: 'John',  
  age: 30  
}
```

## Functions

Functions are also objects in JavaScript. They can have **properties** and **methods**. What distinguishes them from other objects is that they can be called.

```
const sayHello = function() {  
  return 'Hello';  
}  
  
sayHello()
```

If we check the type with `typeof`, we'll get "object" for arrays and object literals, but we actually get "function" for a function. This is still an object, it's just identified as a function object, which can be called as you can see [here](#)

# Stack vs Heap Memory Allocation

---

Now we're going to look at how **primitive** and **reference** type data is stored in memory. One thing I really want to mention is that if you're a beginner or even intermediate, this may seem a little confusing but that's absolutely fine. In all honestly, I have met senior developers that don't know some of this stuff, so don't let this overwhelm you.

What's really important right now is that you just know the types and know how to create variables. I was iffy on even talking about some of this stuff at this point, but I want to this to be a very thorough course. You can always just take in the basics and come back to these videos later. And I'll do my best to let you know what is really important for the day to day and what is more behind the scenes knowledge.

## Garbage Collection

Before we look at how data is stored in memory, I just want to mention that JavaScript uses something called **garbage collection**. With some low-level languages such as **C** and **C++**, you actually have to manage your own memory. When you create variables, you have to allocate the memory yourself and when you are done, you need to free up that space. It makes programming much more difficult. More modern interpreted languages like **JavaScript** and **Python**, do not have you do this. It is automated with something called garbage collection. That's why you don't technically need to know how this stuff works to start writing JavaScript.

## Memory Allocation

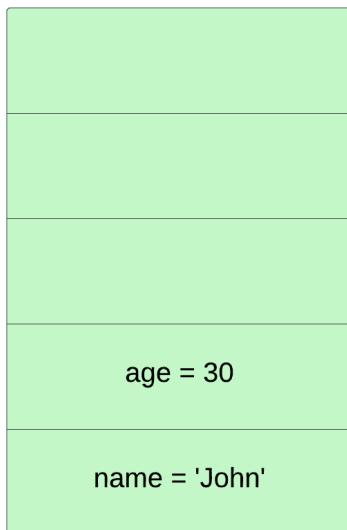
When we create a variable, the JavaScript engine allocates them into two different locations called the **stack** and the **heap**.

Primitive values like strings, numbers, etc are static and immutable data that are fixed. Because of this, the size of the data doesn't change at all, so the space is allocated as a fixed amount and is stored on the stack.

So let's take the following example

```
const name = 'John';
const age = 30;
```

To help visualize this, I created a simple image of what the memory stack would look like

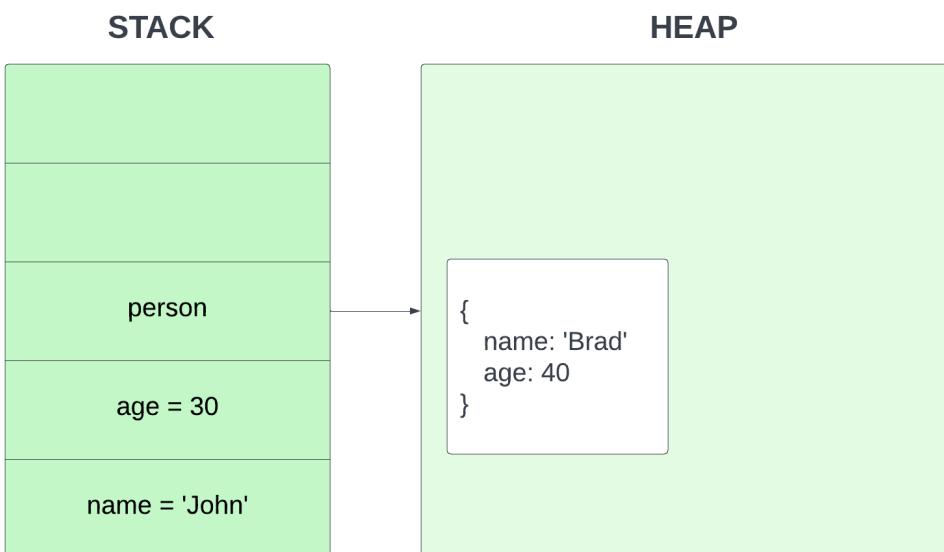


Notice that the memory was allocated on the stack for the variable and the value for name and age. This is because they are static primitive values.

Now let's create a **person** object

```
const person = {  
    name: 'Brad',  
    age: 40  
}
```

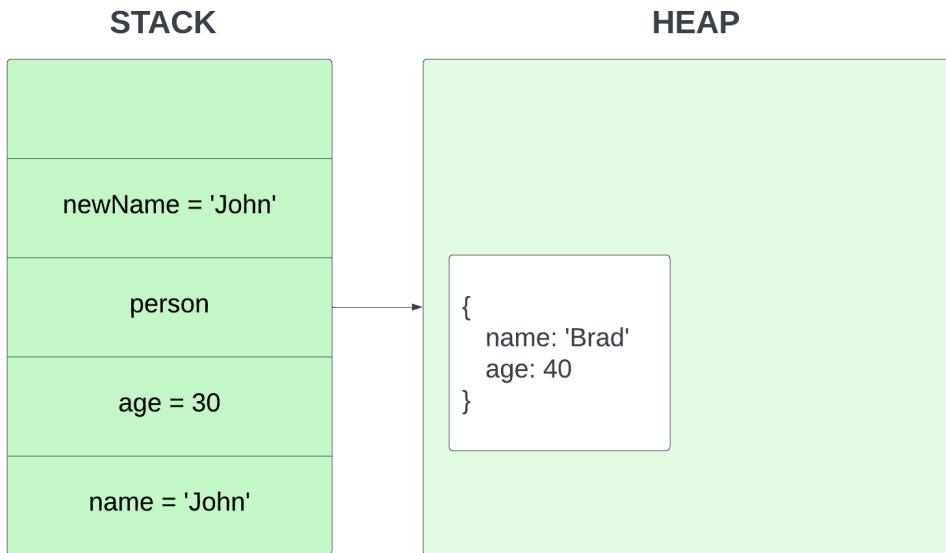
Since this is a reference type that is not static and we can add and remove values from it, the object gets stored in the heap



Lets set a new variable called **newName** to reference the primitive **name** variable

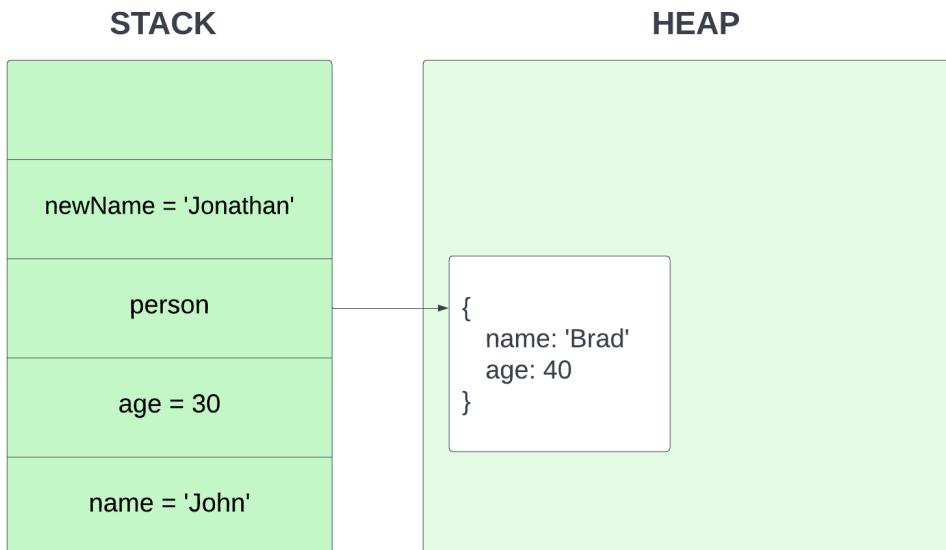
```
const newName = name;
```

Under the hood, JavaScript copies the primitive value of '**John**' and assigns it to **newName**



Now, let's change the value of the **newName** to **Jonathan**

```
newName = 'Jonathan'
```

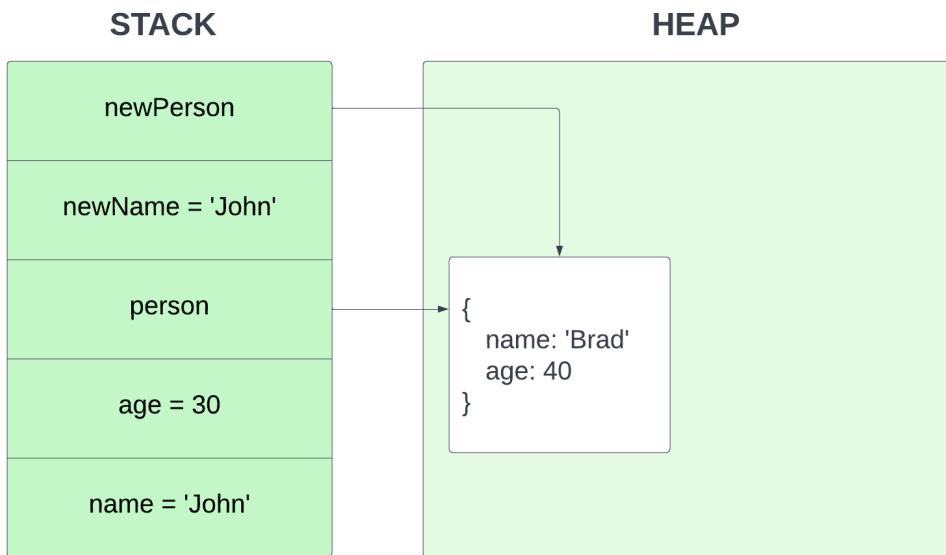


The **name** value stays the same and only the **newName** is changed. That's because it was passed by the value.

Now lets create a new variable called **newPerson** and set it to **person**

```
const newPerson = person;
```

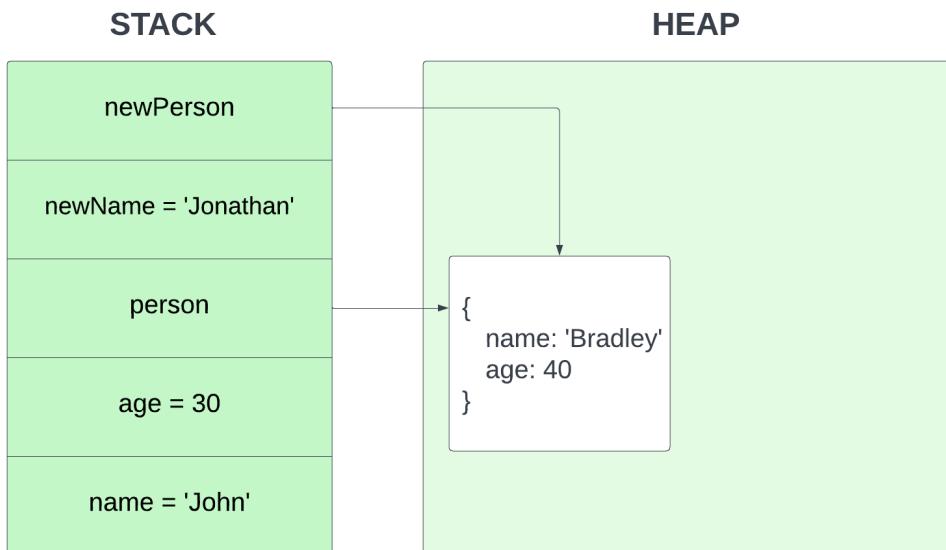
Now the **newPerson** variable references the same value on the heap.



If I were to update **newPerson** object's name value

```
newPerson.name = 'Bradley'
```

It would change the reference in the heap as well and now both **person** and **newPerson** will have the name Bradley. This is because it is passed by reference.



So at some point, this knowledge will come in handy when you get into more advanced programming. It's not really beginner stuff, but I do think learning it now gives you a leg up.

# Type Conversion

---

In programming, **type conversion** or **type casting** is the process of explicitly converting a value from one type to another. For instance, you may have a string value of '**5**' and you want to convert it to a number value of **5** so that you can do some arithmetic operations on it.

**Type Coercion**, can also be explicit, but also refers to the automatic or implicit conversion of values from one data type to another. We will talk about this in another video.

## Converting Types

There are a few different ways that we can convert types.

We have a variable with the string of '100'

```
let amount = '100';
```

### **parseInt()**

We can convert it to a number type with the **parseInt()** function

```
amount = parseInt(amount);
```

### **toString()**

If we want to convert it back to a string, we can use the **toString()** method

```
amount = amount.toString();
```

A **method** is just a function that belongs to an object.

This brings up a question though. The **amount** variable is a primitive number. How come it has a method of **toString()**?

The answer is that JavaScript will actually create a temporary wrapper object to run **toString()** on

### **Number()**

Another way to convert a string to a number is by wrapping it in the **Number()** method

```
amount = Number(amount);
```

## String()

We can convert something to a string using the `String()` method

```
amount = String(amount);
```

## Decimals

If you want to convert a string to a decimal, you don't want to use `parseInt()` because an "integer" is either a negative or positive whole number.

Use `parseFloat()` to convert to a decimal

```
let amount2 = 5.5;
amount2 = parseFloat(amount2);
```

## Boolean Values

Strings and numbers are not all that we can convert. We can use `Number()` with booleans

The following would give us a number of 1. If false, it would give us 0.

```
let x = true;
x = Number(x);
```

We also have a `Boolean()` method to turn a number into a boolean. Later I will talk about "truthy and falsey" values.

```
x = Boolean(x);
```

The following will give us the string `true`

```
x = String(x);
```

## NaN

You saw above that we could run the string of '5' through `Number()` and get the number 5. Well what happens if the string is something like 'hello'?

```
let str = 'Hello';
let num = Number(str)
```

NaN	script.js:30
number	script.js:31
>	

We actually get a special "number" called **NaN**, which stands for **Not a Number**... Yeah, that's JavaScript for you. A number that is "not a number". There are some really strange and quirky aspects of JavaScript. The good news is that most of it doesn't get in your way in practical development. It's just when we start to dig down deeper.

**NaN** is a property of the global object and is a non-writable property. There are 5 different types of operations that return NaN:

- Number cannot be parsed (e.g. parseInt("blabla") or Number(undefined))
- Math operation where the result is not a real number (e.g. Math.sqrt(-1))
- Operand of an argument is NaN (e.g. 7 \*\* NaN)
- Indeterminate form (e.g. 0 \* Infinity, or undefined + undefined)
- Any operation that involves a string and is not an addition operation (e.g. "foo" / 3)

# Operators & Arithmetic

---

In order to look at implicit type coercion in the next video, we'll be using some operators, so I thought that this would be a good time to go over JavaScript operators. Most of these are in just about every language. This is really easy stuff, but I want to make sure we cover everything.

We have a few different types of operators

## Arithmetic Operators

Addition, subtraction, multiplication and division

```
5 + 5; // 10
```

```
10 - 5; // 5
```

```
5 * 5; // 25
```

```
10 / 2; // 5
```

Concatenation: The `+` operator can also be used to put 2 or more strings together. The use of `+` depends on the data type

```
'Hello, ' + 'World!'; // Hello World!
```

Modulus is used to get a division remainder

```
10 % 3; // 1
```

Exponentiation returns the result of the first operand to the power of the second.

```
2**3; // 8
```

Increment is used to **increment** or add 1 to a value

---

```
x = 10;  
x++; // 11
```

Decrement is used to **decrement** or remove 1 from a value

```
x = 10;  
x--; // 9
```

## Assignment Operators

Assignment operators do something with the value on the right side to set the variable on the left side.

Value assignment

```
x = 10;
```

Addition assignment operator adds the amount on the right side

```
x = 10;  
x += 10; // 20  
// Same as x = x + 10
```

Subtraction assignment operator subtracts the amount on the right side

```
x = 10;  
x -= 10; // 0  
// Same as x = x - 10
```

Multiplication assignment operator multiplies the amount on the right side

```
x = 10;  
x *= 10; // 100  
// Same as x = x \* 10
```

Division assignment operator divides the amount on the right side

```
x = 10;  
x /= 10; // 1
```

```
// Same as x = x / 10
```

Modulo assignment operator gets division remainder of the amount on the right side

```
x = 10;  
x %= 10; // 0  
// Same as x = x % 10
```

Exponentiation assignment operator gets exponent of the amount on the right side

```
x = 10;  
x **= 10; // 100  
// Same as x = x \*\* 10
```

## Comparison Operators

Comparison operators are used to compare values

Equal to operator - The following will return true even if the types do not match

```
2 == 2 // true  
2 == '2' // true
```

Equal value & equal type - The types have to match or it will return false

```
2 === 2 // true  
2 === '2' // false
```

Not equal to operator - returns true if not equal

```
2 != 2 // false  
2 != '2' // false
```

Not equal value & equal type

```
2 !== 2 // false  
2 !== '2' // true
```

Greater than

```
10 > 5 // true
```

Less than

```
10 < 5 // false
```

Greater than or equal

```
10 >= 5 // true
```

Less than or equal

```
10 <= 5 // false
```

## `==` VS `===`

As I said above, the `==` operator is used to compare values. The `===` operator is used to compare values and types.

The double equals (`==`) performs type coercion, which means it makes sure that the values are the same type before comparing them.

Which one you use is up to you, but I personally use the triple equals (`===`) because it is more explicit and your code will be less prone to errors. In many situations, it doesn't matter, but I prefer to use it unless there is a specific reason not to.

Later on we will talk about **truthy** and **falsy** values and I will show you some situations where the double equals can cause issues.

# Type Coercion

---

Type Coercion is when data types are converted implicitly by JavaScript. Although the conversion that we did in the previous video can also be called explicit coercion. There's so many different words for the same thing in programming.

Anyway, there are a few situations where implicit coercion can occur. For the most part, you have 3 types of coercion

- to string
- to number
- to boolean

Typically coercion happens when you apply operators to values of different types.

Let's take a look at some examples. I will warn you, some of this will get weird. But again, this stuff usually isn't a big issue in everyday development.

Let's look at our first example

```
5 + '5'; // 55 (string)
```

So as you can see, if we use the `+` operator on the **number** 5 and a **string** with the character '5', we get a **string** of 55. Some of you may have expected 10, which is the answer to  $5 + 5$ .

The reason that this happened is because the **number** 5 is being implicitly coerced into a **string** due to the `+` operator also being used for concatenation, as I talked about in the last video.

This is an example of automatic or implicit coercion because we applied an operator to values of different types.

If I wanted to add these together, I would first convert the string '55' to a number like this

```
5 + Number('5'); // 10
```

Although the string would most likely be a variable in this case.

Now, let's take the **number** 5 and multiply by the **string** 5.

```
5 * '5'; // 25 (number)
```

For this expression, we do get the result of  $5 * 5$ , which is a **number** of 25. So in this case, JavaScript looked at the expression and coerced the **string** of '5' into the **number** 5.

This happened because it makes the most sense. The \* can not do anything else but multiply in this expression.

Let's try some more weird stuff and see the results

```
5 + null; // 5
```

So in this case, we get 5. Reason being that null is coerced to a number of 0. We can see what a value would be as a number by doing the following

```
Number(null); // 0
```

Let's see what the boolean values of true and false would be as a number

```
Number(true); // 1  
Number(false); // 0
```

So with that what do you think the answer would be if we added 5 + true and 5 + false?

```
5 + true; // 51  
5 + false; // 50
```

The null and false being 0, brings us to something called "falsey values". We'll get more into that when we talk about conditionals though.

Now let's look at the following expression

```
5 + undefined; // NaN
```

So the result is NaN or **not a number**. We talked a little about this in the last video. The reason is because NaN is the result of a failed number operation. If we run undefined through the Number() method, we also get NaN.

If we try and add NaN like this, we also get NaN.

```
5 + NaN; // NaN
```

Want to see something really strange?

```
NaN == NaN // false
```

You read that right. NaN is NOT equal to NaN. Kind of mind blowing. This is because not all NaN numbers are created equal. You can read more about it [here](#), but just know that this will always equate to false.

Now there may be times when you need to check for a NaN value. Again, I know we haven't got into functions or conditionals yet, but we do have a function called **isNaN** that we can use like this.

```
isNaN(NaN); // true
```

However, this will return true for ANYTHING that is "not a number", such as

```
isNaN('Hello') // true
```

If you really want to check for the specific value of NaN, you can use the isNaN method on the Number object like this

```
Number.isNaN(NaN) // true  
Number.isNaN('Hello') //false
```

# Working with Strings

---

Alright, so now we're going to work with strings. I'm going to show you how to concatenate strings, use something called **template literals** as well as look at some methods and properties that we can use on string objects.

## Concatenation

So we briefly looked at concatenating strings, but let's take a deeper look.

The `+` operator is used to concatenate. Let's take two strings and put them together.

```
'Hello' + 'World';
```

Pretty simple right? We usually use concatenation to work variables into a string like so

```
const name = 'John';
const age = 30;

'Hello, my name is ' + name + ' and I am ' + age + ' years old';
```

I learned PHP before JavaScript and in PHP, we use the period(.) character in the same way, so I picked up on this pretty quickly.

## Template Literals

So in 2015, JavaScript had a major update called [ES2015](#), also known as [ES6](#). In that update, template literals or "template strings" were introduced. These make it much easier and cleaner to include variables in a string.

```
const name = 'John';
const age = 30;

`Hello, my name is ${name} and I am ${age} years old`;
```

So as you can see, instead of using single or double quotes around the string, we use backticks(`).

Then, if we want to use a variable or an expression, we simply surround it in curly braces and prefix it with a dollar sign like this...  `${myVariable}`. This makes it much easier because we can just put the variables right into the string and not worry about ending and beginning it again with a `+` sign. You can also go on to multiple lines. You can not do that with regular concatenation.

## String Properties & Methods

Strings are **primitive** in JavaScript. We have already established that. Primitives do not have properties (attributes) and methods (functions), however if we try and use a method like `.toUpperCase()` on a string, it will work. This is because the JavaScript engine will actually promote them to full blown **string objects**. Same goes for numbers, or at least variables that store numbers.

We can actually create a string object ourselves by using the `new` keyword and then the **String constructor**

```
const s = new String('Hello World');
```

This is what happens behind the scenes when we use a property or method on a string. If you check the `typeof s`, you will see **object**.

```
console.log(s);
```

```
▼ String {'Hello World'} ⓘ script.js:70
  0: "H"
  1: "e"
  2: "l"
  3: "l"
  4: "o"
  5: " "
  6: "W"
  7: "o"
  8: "r"
  9: "l"
  10: "d"
  length: 11
▶ [[Prototype]]: string
[[PrimitiveValue]]: "Hello World"
```

If you log the actual value of `s`, you will see an object and each character with a key/value pair. The first key or "index" is always **0**. We can actually get a specific character by its index.

```
console.log(s[3]); // prints "l"
```

We get the letter "l" because it is the letter at index 3.

Along with the key/value pairs, we get the primitive type and you'll also see something called a **prototype**.

If you un-collapse this, you will find all of the properties and methods available on the string object. We will get much more into prototypes later when we talk about object oriented programming, etc, but I wanted to show you that this is available in case you want to experiment with it.

You can also access the prototype directly with

```
console.log(s.__proto__);
```

Alright, so now that we know how primitive values seem to have properties and methods, let's look at some of them.

There is really only one property that we're going to look at and that is **length**. The rest are methods. Methods are functions, so they always end with parentheses ()

## length

Length is pretty common for multiple data types. On an array, it will tell us the number of elements in it. If we use it on a number or string object, it will give us the number of characters

```
const greeting = 'Hello World';
greeting.length; // 11
```

## Changing case

In some cases, you may want to change your case to upper or lower case. There are methods we can use for that

```
greeting.toUpperCase();
greeting.toLowerCase();
```

## charAt()

Returns the character at the specified index. All characters in a string have an index that starts from 0. In fact, when we logged the "string object", we saw the key/value pairs. This method will tell you the character or value at the specified index

```
greeting.charAt(0); // H
```

## indexOf()

indexOf() will do the reverse of charAt(). Instead of passing the index, we pass the value and it will give us the index of the first occurrence of the character(s)

```
greeting.indexOf('o'); // 4
```

## **substring()**

Search a string for a specified value. Pass in the start index and the length. Let's say we want to pull out a substring of 'Hell' from 'Hello World', we would start at character 0 and go 4 in length

```
greeting.substring(0, 4); // Hell
```

## **slice()**

Extracts a part of a string and returns a new string. It is very similar to substring, but there are some differences, for instance, you can use negative numbers with slice.

```
greeting.slice(0, 4); // Hell  
// We can also use negative numbers  
greeting.slice(-11, -7) // Hell
```

## **trim()**

Trim the whitespace of a string

```
x = '      Hello World!      ';  
x = x.trim(); // Hello World!
```

## **replace()**

Replace all instances of a string

```
const url = 'https://traversymedia.com';  
url.replace('traversymedia', 'google'); // https://google.com
```

## **includes()**

Returns true if a searched string is found

```
url.includes('media'); // true
```

## **valueOf()**

Returns the primitive value of a variable

```
url.valueOf(); // https://traversymedia.com
```

## split()

Splits a string into an array and takes in a separator parameter. If we add a space, it will put every word in its own array item.

```
greeting.split(' '); // ['Hello', 'World']
```

# Capitalize Challenge

---

We have arrived at our first challenge. Every once in a while I'm going to ask you to figure out a problem or do something using what we have already learned. I will not ask you to do a challenge that involves doing something that we have not yet gone over.

If you feel stuck and can not figure it out on your own, that is absolutely fine. I will walk you through the solution(s), so that you can understand how to solve the problem.

## Instructions:

Take the variable `myString` and capitalize the first letter of the word using some of the methods that we talked about in the last video. Put the result in a variable called `myNewString`.

Create multiple solutions if you would like.

## Expected Result:

```
const myString = 'developer';

console.log(myNewString); // 'Developer'
```

## Hints:

1. You can use the `charAt()` method as well as `string[index]` to get the character at a specific index.
2. The `.toUpperCase()` method will make the entire string uppercase
3. `substring()` or `slice()` will return a specific portion of a string

## ► Click For Solution

There are many ways to do this. Let's take a look at a few

```
// Solution 1
const myNewString = myString.charAt(0).toUpperCase() + myString.substring(1);

// Solution 2 (Uses string[0] instead of string.charAt(0))
const myNewString = myString[0].toUpperCase() + myString.substring(1);

// Solution 3 (Uses template literal and slice())
const myNewString = `${myString[0].toUpperCase()}${myString.slice(1)}`;
```

In all of these, we get the first character of the string, then we use the `substring()` or `slice()` method to get the rest of the string. We then use the `toUpperCase()` method to capitalize the first character and then we concatenate the result with the rest of the string.

# Working with Numbers

---

So we looked at strings and the properties and methods that are available to us. Now let's look at numbers. We'll also look at the `Math` object.

Like with strings, when we add a method to a number, a new number object is created and we can use that object to call the method. Let's create the object ourselves to see the available methods in the prototype

```
const num = new Number(5);
console.log(num)
```

```
▼ Number {5} ⓘ
  ▼ [[Prototype]]: Number
    ► constructor: f Number()
    ► toExponential: f toExponential()
    ► toFixed: f toFixed()
    ► toLocaleString: f toLocaleString()
    ► toPrecision: f toPrecision()
    ► toString: f toString()
    ► valueOf: f valueOf()
    ► [[Prototype]]: Object
      [[PrimitiveValue]]: 0
      [[PrimitiveValue]]: 5
```

[script.js:3](#)

## **toString()**

Returns a string representation of the number

```
num.toString(); // "5"
```

`Number` types and objects do not have a `length` property. If you want to find the length, one thing that you could do is convert it to a string and then use `length`

```
num.toString().length; // 1
```

## **toFixed()**

Returns a string representation of the number with a specified number of decimals

Let's assume `num` is equal to 5 right now

---

```
num.toFixed(2); // 5.00
num.toFixed(1); // 5.0
```

### **toPrecision()**

returns a number with the specified length

```
const num2 = 94.4058;
num.toPrecision(3); // 94.4
```

### **toExponential()**

Returns a string representation of the number in exponential notation

```
num.toExponential(2); // "5.00e+0"
```

### **toLocaleString()**

Returns a string representation of the number in the current locale

```
let num = 5000000;
num.toLocaleString(); // "5,000,000"
```

It defaults to the browser's locale, which for me is "en-US", but let's say we want to use India's locale

```
num.toLocaleString('en-IN'); // "50,00,000"
```

## Number Object Properties & Values

The Number object has a few properties and methods that are available.

### **Max Value**

Largest possible value of a number

```
Number.MAX_VALUE; // 1.7976931348623157e+308
```

### **Min Value**

Smallest possible value of a number

```
Number.MIN_VALUE; // 5e-324
```

There are methods as well. For instance, we already looked at `isNaN()`, which will tell us if the value is actually NaN

```
Number.isNaN(NaN); // true
```

# Math Object

---

JavaScript has a built-in object called **Math**. This object has a bunch of methods that have to do with mathematical operations. It comes in handy when you need to do things like find the square root or absolute value of a number, when you need to round up or down or when you need to generate a random number. Let's see some examples.

You can see what is available with:

```
console.log(Math);
```

## **Math.abs()**

Returns the absolute value of a number

```
Math.abs(-5); // 5
```

## **Math.round()**

Returns the rounded value of a number

```
Math.round(5.5); // 6
```

## **Math.ceil()**

Returns the smallest integer greater than or equal to a number (rounds up)

```
Math.ceil(5.5); // 6
```

## **Math.floor()**

Returns the largest integer less than or equal to a number (rounds down)

```
Math.floor(5.5); // 5
```

## **Math.sqrt()**

Returns the square root of a number

```
Math.sqrt(25); // 5
```

### **Math.pow()**

Returns the value of a number raised to the specified power

```
Math.pow(5, 2); // 25
```

### **Math.min()**

Returns the smallest of the two numbers

```
Math.min(5, 10); // 5
```

### **Math.max()**

Returns the largest of the two numbers

```
Math.max(5, 10); // 10
```

### **Math.random()**

Returns a random number between 0 and 1

```
Math.random(); // 0.83929
```

This returns a decimal. If you want let's say a random integer between 1 and 10, you can multiply the result of Math.random() by 10 and round it down.

```
Math.floor(Math.random() * 10) + 1; // 5
```

We had to round down first with `Math.floor()` or we would get a decimal. We also added 1 otherwise we would get a number between 0 and 9. This is important to understand for the next challenge.

To get a random integer between two numbers, you can do

```
const min = 10;
const max = 20;
x = Math.floor(Math.random() * (max - min + 1)) + min; // 17
```

# Number Challenge

---

## Instructions:

Create a variable called `x` that is a random number between 1 and 100 along with a variable called `y` that is a random number between 1 and 50.

Create a variable for the sum, difference, product, quotient and remainder of `x` and `y`. Log the output in a string that shows the two numbers of `x` and `y` along with the operator and result.

- You can log the output string directly or put them in separate variables and log them like below.
- You can use string concatenation or template literals for the output.

## Expected Result:

```
console.log(sumOutput); // 31 + 15 = 46
console.log(differenceOutput); // 31 - 15 = 16
console.log(productOutput); // 31 * 15 = 465
console.log(quotientOutput); // 31 / 15 = 2.0666666666666667
console.log(rmOutput); // 31 % 15 = 1
```

## Hints:

1. The `Math.random()` function returns a floating-point, pseudo-random number in the range 0 to less than 1
2. The `Math.floor()` function will round a number down to the nearest integer

## ► Click For Solution

```
x = Math.floor(Math.random() * 100) + 1;
y = Math.floor(Math.random() * 50) + 1;

// Get the sum
const sum = x + y;
const sumOutput = `${x} + ${y} = ${sum}`;
console.log(sumOutput);

// Get the difference
const difference = x - y;
const differenceOutput = `${x} - ${y} = ${difference}`;
console.log(differenceOutput);

// Get the product
const product = x * y;
const productOutput = `${x} * ${y} = ${product}`;
console.log(productOutput);
```

```
// Get the quotient
const quotient = x / y;
const quotientOutput = `${x} / ${y} = ${quotient}`;
console.log(quotientOutput);

// Get the remainder
const rm = x % y;
const rmOutput = `${x} % ${y} = ${rm}`;
console.log(rmOutput);
```

# Working With Dates

---

Dates and times are very important in any programming language. JavaScript has a **Date** object that represents a point in time and let's you do some basic operations on it.

We can instantiate a **Date** object with the **new** keyword.

```
let d;  
d = new Date(); // Fri Jul 22 2022 08:35:10 GMT-0400
```

If do not include any arguments, the Date object will be set to the current date and time and you will also get the timezone information. Sometimes you will need to specify the timezone because it can cause some weird issues.

## Converting to a string

If we look at the type of the variable, it will show **object**. You can always convert it to a string using the **toString()** method.

```
console.log(typeof d); // object  
  
d.toString(); // "Fri Jul 22 2022 08:35:10 GMT-0400"  
  
console.log(typeof d); // string
```

## Specific dates

If you want a specific date and time, you can pass in the year, month, day, hour, minute, second, and millisecond.

One caveat is that the month is 0-indexed, so January is 0 and December is 11.

```
d = new Date(2022, 6, 10); // Fri Jul 10 2022 00:00:00 GMT-0400
```

If you want to add a time, you can. Let's make it 12:30:00.

```
d = new Date(2022, 6, 10, 12, 30, 0); // Fri Jul 10 2022 12:30:00 GMT-0400
```

You can also pass in different date/time strings. You would use the actual month number here.

```
d = new Date('2022-07-10T12:30:00'); // July 10, 2022 12:30:00
d = new Date('07/10/2022 12:30:00'); // July 10, 2022 12:30:00
d = new Date('2022-07-10 12:30:00'); // July 10, 2022 12:30:00
```

You can pass a date without time as well.

Note that if you use the **YYYY-MM-DD** format with hyphens, it may be one day off due to timezones.

```
d = new Date('2022-07-10'); // May be July 09, 2022
```

If you use the **MM-DD-YYYY** format, it should not have this issue.

```
d = new Date('07-10-2022'); // July 10, 2022

// Backslashes will also work
d = new Date('07/10/2022'); // July 10, 2022
```

## Timestamps

The **UNIX timestamp** is a system for describing a point in time. It is an integer that represents the number of seconds elapsed since January 1 1970 (An arbitrary date).

The timestamp in JavaScript is expressed in milliseconds.

You can get the current timestamp using:

```
Date.now(); // 1658497991311
```

To get the timestamp of a specific date, you can use the `getTime()` or the `valueOf()` method.

```
d = new Date('07/22/2022'); // July 10, 2022
d.getTime(); // 1658497991311
d.valueOf(); // 1658497991311
```

You can also create date objects from a timestamp.

```
d = new Date(1658497991311); // Fri Jul 22 2022 08:35:10 GMT-0400
```

The JavaScript timestamp is expressed in milliseconds. To convert it to seconds:

```
Math.floor(Date.now() / 1000); // 1658498058
```

# Date Methods

---

There are a lot of useful methods that we can use on the Date object for getting information about the date. Let's create a new date object to work with. I will just use today's date, which is July 22, 2022.

```
let d = new Date();
```

We already looked at a few methods such as `toString()` and `getTime()`.

```
// Convert the date to a string  
d.toString();  
  
// Get the time in milliseconds  
d.getTime();  
d.valueOf();
```

Let's look at some methods for getting specific parts of the date.

```
// Get the year  
d.getFullYear();  
  
// Get the month number (0-11)  
d.getMonth();  
d.getMonth() + 1; // Actual month number  
  
// Get the day of the month (1-31)  
d.getDate();  
  
// Get the day of the week (0-6)  
d.getDay();
```

We also have methods for getting time values.

```
// Get the hour (0-23)  
d.getHours();  
// Get the minute (0-59)  
d.getMinutes();  
// Get the second (0-59)  
d.getSeconds();  
// Get the millisecond (0-999)  
d.getMilliseconds();
```

We can create custom date using methods

```
`${d.getFullYear()}-${d.getMonth() + 1}-${d.getDate()}`; // 2022-7-22
```

## Intl.DateTimeFormat API

The Intl.DateTimeFormat API is a more modern and powerful way to format dates and times in a locale-sensitive way. It fixes some of the locale issues we can run into.

We can pass in a locale argument to the constructor to get a date formatted in a specific locale.

```
new Intl.DateTimeFormat('en-US').format(d); // July 22, 2022
new Intl.DateTimeFormat('en-GB').format(d); // 22 July, 2022

// You can pass in 'default' to get your default locale
new Intl.DateTimeFormat('default').format(d); // July 22, 2022
```

By default, it will return a string in the above format, but we can specify what we want in the second argument. Let's say I just want the month name

```
new Intl.DateTimeFormat('en-GB', { month: 'long' }).format(d); // July
```

## toLocaleString()

A shorter way to use this API is to use the `toLocaleString()` method. This is what I would usually recommend.

```
d.toLocaleString('default', { month: 'long' }); // July
```

if we want more information:

```
d.toLocaleString('default', {
  weekday: 'long',
  year: 'numeric',
  month: 'long',
  day: 'numeric',
  hour: 'numeric',
  minute: 'numeric',
  second: 'numeric',
  timeZone: 'America/Los_Angeles',
}); // Friday, July 22, 2022 at 7:30:57 AM
```



# The JavaScript Console

---

When we create front-end websites and applications, we work in the browser environment. We can use JavaScript to manipulate the DOM (Document Object Model) and display different elements and values, but many times, we just need a quick way to show what a value is. We also need a place to see any errors or warnings in our script or any script that runs. This is where the JavaScript console comes in.

The console is part of the **Developer Tools** in the browser. In most browsers, you can open the dev tools with **F12** on a Windows PC and **CMD+OPT+I** on a Mac. From there you have a bunch of tabs and one (usually the second) is the console. You can also use **CMD+OPT+J** or **CTRL+ALT+J** to go right to the console.

From here you can actually run JavaScript. Try typing in the following directly into the browser console and hit **enter**.

```
alert('Hello from the console')
```

It should show a browser alert with the text.

We can write just about any single line JavaScript expression directly in the console, however, you probably won't do this very much. We usually use the console to output information and values from our script/code.

## Console methods

The **global object** in the browser environment gives us access to a **console** object that has a bunch of useful methods that we can use from our JavaScript file to interact with the JavaScript console. When I say a method, I just mean a function that is attached to an object. In this case, the "console" object.

### **console.log()**

This is the most common console method. We pass whatever we want to log to the console as an argument to the **log()** method. This can be a string, a number, a boolean, an object, an array, or even a function. It will output the value to the console.

#### **Log a number**

Different data types will have different colors in the console

```
console.log(123);
```

#### **Log a string**

```
console.log('Hello World');
```

## Log multiple values

You can separate values with a comma

```
console.log(123, 'Hello', true);
```

## Log a variable

For the most part, we use the console to debug and log out variables or the result of a function or network request.

```
x = 100;  
console.log(x);
```

## console.error()

using console.error() will make the text red

```
console.error('This is an error');
```

## console.warn()

Using console.warn() will make the text yellow

```
console.warn('This is a warning');
```

## console.clear()

```
console.clear();
```

## console.table()

If you want to log an object, you can format it as a table

```
console.table({ name: 'Brad', city: 'Boston' });
```

## console.group()

```
console.group('simple');
console.warn('warning!');
console.error('error here');
console.log('Hello World');
console.groupEnd('simple');
console.log('new section');
```

## Log with style

```
const styles = 'padding: 10px; background-color: white; color: green;';
console.log('%cHello', styles);
```

There are even more methods. See all of them [here](#)

# Array Methods

---

As we talked about earlier, arrays are a special kind of **object**. Objects have **properties** and **methods**. In JavaScript, those methods are stored in the **prototype chain**. We'll talk much more about prototypes later on, but we already saw that both **string objects** and **number objects** have properties and methods in their prototypes.

## Array methods

### **push()**

Push adds an element to the end of the array.

```
arr.push(6); // [1, 2, 3, 4, 5, 6]
```

### **pop()**

Pop removes the last element of the array.

```
arr.pop(); // [1, 2, 3, 4, 5]
```

### **unshift()**

Adds an element to the beginning of the array.

```
arr.unshift(0); // [0, 1, 2, 3, 4, 5]
```

### **shift()**

Takes an element off of the start of the array.

```
arr.shift(); // [1, 2, 3, 4, 5]
```

### **reverse()**

I'll let you figure out what this does.

```
arr.reverse(); // [5, 4, 3, 2, 1, 0]
```

## Returning values from an array

The methods above are all used to manipulate a current array. These methods are typically used to return some kind of value from an array.

### **includes()**

Returns `true` if the array contains the value you pass in.

```
arr.includes(5); // true
```

### **indexOf()**

Returns the index of the value you pass in.

```
arr.indexOf(5); // 4
```

### **Return string from array**

```
arr.toString(); // "1,2,3,4,5"  
arr.join(); // "1,2,3,4,5"
```

## **slice() and splice()**

These are similar in that they both are used to return a new array using a subset of the current array.

### **slice()**

`slice()` takes in two parameters. One being the starting index and one being the ending index.

```
const newArr = arr.slice(1, 3); // [2, 3]
```

### **splice()**

`splice()` takes in the starting index and the number of elements from the starting index. If you leave off the number of elements, it will take from the starting index and on.

```
const newArr = arr.splice(1, 3); // [2, 3, 4]
```

Another difference is that slice() does not modify the original array and splice() does. If you log the original array, you will see those elements are gone.

To pluck out a single value, use splice and pass the start index and then 1 as the second arg because you only want to remove that element.

```
const newArr = arr.splice(2, 1); // [3] the original array would be [1, 2, 4, 5]
```

You can also start from the end with a negative value

```
const newArr = arr.slice(-2); // [4, 5]
```

## Chaining methods

You can also chain these methods together. Just be aware of what is being returned from the previous method.

```
x = arr.concat(arr2).reverse();
x = arr3.slice(1, 3).toString();
```

# Nesting, concat, spread & Array Object Methods

---

To get you more familiar with arrays, we are going to look at nesting, concatenation and the spread operator

Let's create our fruit array again

```
const fruits = ['apple', 'pear', 'orange'];
```

and let's create a "berries" array

```
const berries = ['strawberry', 'blueberry', 'rasberry'];
```

Now let's nest the berries array inside the fruit array

```
fruits.push(berries); // ['apples', 'oranges', 'pears', ['strawberries', 'blueberries', 'raspberries']]
```

Now if we want to access blueberries for example, we can do so

```
fruits[3][1]; // blueberries
```

We could also create a new variable and nest both arrays:

```
const allFruits = [fruits, berries];
```

```
allFruits[1][1]; // blueberries
```

## concat()

We may not want to nest arrays, but we can concatenate them together into the same array using **concat()**.

```
const newArr = fruits.concat(berries); // ['apples', 'oranges', 'pears', 'strawberries', 'blueberries', 'raspberries']
```

---

So now we have a single array with all of the values of both arrays. There is no nesting going on.

### **Spread operator (...)**

The spread operator is a very useful operator that allows us to spread out an array into another array. We can use it to achieve the same type of stuff that concat() does. The spread operator also works with object literals, which we will talk about very soon.

```
const newArr2 = [...fruits, ...berries]; // ['apples', 'oranges', 'pears',  
'strawberries', 'blueberries', 'raspberries']
```

### Flatten an array

You can use the `flat()` method to flatten an array. It takes in a parameter for the depth. If you don't want a limit to the depth, you could use `Infinity`.

```
const arr = [1, 2, [3, 4, 5], 6, [7, 8, [9, 10, [11, 12]]]];  
x = arr.flat(Infinity);
```

### Static properties on the Array object

The Array object in JavaScript actually has some helpful methods directly associated with it that we can use.

#### **Array.from()**

Check if something is an array.

```
Array.isArray(fruits); // true  
Array.isArray('Hello'); // false
```

#### **Array.from**

Convert an array-like object into an array.

```
Array.from('12345'); // ['1', '2', '3', '4', '5']
```

#### **Array.of()**

Create an array from a variable number of arguments.

```
const a = 1;
const b = 2;
const c = 3;
y = Array.of(a, b, c); // [1, 2, 3]
```

# Array Challenges

---

Now that we have gone over some basic array stuff, let's try some simple challenges. I would suggest trying these using just what you remember from the past sections, but if you need to look back, that's fine as well.

**Note:** When I assign a challenge, I will never ask you to do something that we have not went over yet in the course. For example, I would not ask you to create a function because we haven't talked about functions yet. Because of this, some of the earlier challenges may seem a bit easy to some of you that already have experience with JavaScript.

Challenge 1:

**Instructions:**

Use some of the array methods that we looked at to mutate the following array to = the expected result below:

```
const arr = [1, 2, 3, 4, 5];
```

**Expected Result:**

```
console.log(arr);
// [6, 5, 4, 3, 2, 1, 0];
```

**Hint:** No hints. This one is pretty easy 😊

► [Click For Solution](#)

```
const arr = [1, 2, 3, 4, 5];

arr.unshift(0);
arr.push(6);
arr.reverse();

console.log(arr); // [6, 5, 4, 3, 2, 1, 0];
```

Challenge 2:

**Instructions:**

Combine `arr1` and `arr2` into a new array called `arr3` with the following elements:

```
const arr1 = [1, 2, 3, 4, 5];
const arr2 = [5, 6, 7, 8, 9, 10];
```

Notice that both `arr1` and `arr2` include the number 5. You will have to find a way to get rid of the extra 5.

**Expected Result:**

```
console.log(arr3);
// [1,2,3,4,5,6,7,8,9,10]
```

**Hint:** There are many ways to do this, but think of the `concat()` method or the spread operator as well as the `slice()` or `splice()` methods

► Click For Solution

```
const arr1 = [1, 2, 3, 4, 5];
const arr2 = [5, 6, 7, 8, 9, 10];

// Solution 1
const arr3 = arr1.slice(0, 4).concat(arr2);

console.log(arr3); // [1,2,3,4,5,6,7,8,9,10]

// Solution 2
const arr4 = [...arr1, ...arr2];
arr4.splice(4, 1);

console.log(arr4); // [1,2,3,4,5,6,7,8,9,10]
```

# Object Literals

I want to talk about a very common JavaScript data structure that holds key/value pairs called an **object literal**. We can also create objects using a **constructor**, but I will get into that later on. The main difference between using a constructor and an object literal is that the object literal is a **singleton**, meaning that it is a **single instance** of the object. With a constructor, we can create multiple instances of the object. Most of the time, we will use the object literal syntax.

```
const person = {  
    name: 'John Doe',  
    age: 30,  
    location: 'Boston'  
}
```

KEYS      {    } VALUES

The image above shows us the syntax to create a new object. Essentially, all an object is, is a set of **properties** with **key/value** pairs. The **key** is often called the **property name**. The values go by the same types and principles as anything you would set in a regular variable (string, number, boolean, etc)

```
const person = {  
    name: 'John',  
    age: 30,  
    location: 'Boston',  
    hobbies: ['movies', 'music'],  
    isAdmin: false,  
};
```

We can also nest objects, but I'm going to go over that in next video.

## Accessing property values

To access a value, we can use both dot notation and bracket notation.

```
// Dot notation  
person.name; // John  
  
// Bracket notation  
person['name']; // John
```

We will typically use dot notation and use brackets with arrays. Although, technically arrays are just objects with a numeric index.

To access the first hobby in the array:

```
person.hobbies[0]; // movies
```

## Adding & updating properties

I just manually added and changed some of the properties directly in the object above, but to programmatically add or change properties, we could do this:

```
person.email = 'john@gmail.com'; // Add a new property
person.isAdmin = true; // Update an existing property
```

## Removing properties

To remove a property, we can use the `delete` keyword.

```
delete person.isAdmin;
```

## Adding functions to objects

Not only can we store primitives, arrays and other objects, we can also store functions as an object's property.

```
person.greet = function () {
  console.log('Hello, my name is ' + this.name);
};
```

Notice I used the keyword `this` to access the `name` property. This is because `this` refers to the object that the function is being called on. This is where `scope` comes into play. If we are in the `global scope`, then the `this` keyword refers to the `global object` or the `window` in the browser. We'll get more into scope in the next section.

We can call the `greet` function like this:

```
person.greet(); // Hello, my name is John
```

## Using multiple words in property names

If you want to use multiple words as a property name or key, you can use quotes or backticks to wrap the property name.

```
const person2 = {  
  'first name': 'Brad',  
  'last name': 'Traversy',  
};
```

We have to use bracket notation when using property names with multiple words.

```
person2['first name']; // Brad
```

# Object Nesting, spreading, methods and more

---

So we saw how to create object literals in JavaScript. I want to go a bit deeper and look at nesting, the spread operator as well as some static methods on the Object constructor.

First, I do want to show you another way of creating objects that you may run into and that is by using the Object constructor.

```
const todo = new Object();
// Same as
const todo = {};
```

We can add properties to the object using dot notation.

```
todo.id = 1;
todo.title = 'Buy milk';
todo.completed = false;
```

Later on when we get more into OOP, I'll show you how we can create our own object constructors that we can use to create multiple instances of the same object. So that we can do something like this:

```
const todo1 = new Todo();
```

For now, we'll stick with static object literals.

## Nesting objects

We can put objects inside other objects.

```
const person = {
  address: {
    coords: {
      lat: 42.3601,
      lng: -71.0589,
    },
  },
};
```

If we want to access nested objects, we simply use dot notation

```
person.address.coords.lat;
```

## Spread operator

The spread operator is used to **spread** across object properties into a new object. Let's first look at another example of nesting.

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 1, d: 4 };

const obj3 = { obj1, obj2 }; // { { a: 1, b: 2 }, { c: 1, d: 4 } }
```

## Object.assign()

We can do the same thing we did with the spread operator with the Object.assign() method.

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 1, d: 4 };

const obj3 = Object.assign({}, obj1, obj2); // { a: 1, b: 2, c: 1, d: 4 }
```

The first param we pass in is an empty object, then each one after is an object we want to spread into the empty object.

In the code above, we are simply nesting. We can use the spread operator to spread the properties instead of just nesting the entire object.

```
const obj3 = { ...obj1, ...obj2 }; // { a: 1, b: 2, c: 1, d: 4 }
```

## Arrays of objects

In many cases, you will be dealing with arrays of objects.

```
const people = [
  { id: 1, name: 'John' },
  { id: 2, name: 'Sara' },
  { id: 3, name: 'Mike' },
];
```

We can access the name **Sara** like this:

```
people[1].name; // Sara
```

## Object methods

The `Object()` constructor has some helpful methods that we can use, just like the `Number()` and `String()` constructors. We already looked at `Object.assign()`, let's look at some others.

### `keys()`

Get an array of all the keys in an object.

```
Object.keys(person); // [ 'name', 'age', 'address' ]
```

### Getting the length of an object

We can't use the `length` property directly on an object. If I do the following, it will look for a 'length' property.

```
person.length; // undefined
```

However, we can first get the keys of the object and then use the `length` property on the keys array.

```
Object.keys(person).length; // 3
```

### `values()`

We can get an array of all the values in an object.

```
Object.values(person); // [ 'John', 30, { address } ]
```

### `entries()`

Get an array of key/value pairs

```
Object.entries(person); // [ [ 'name', 'John' ], [ 'age', 30 ], [ 'address', { address } ] ]
```

### `hasOwnProperty()`

Get a boolean indicating if the object has the property

```
person.hasOwnProperty('name'); // true
```

# Destructuring & Naming

---

## Variables with the same name

In some cases, we may have a variable that we want to add as an object property value. If a property name (key) is the same as the variable name for the value, we can withdraw the variable name of the value. For example:

```
const firstName = 'John';
const lastName = 'Doe';
const age = 30;
```

Here I will put the variables above as the values for each property of the object. I will use the same variable names as the property names.

```
const person = {
  firstName: firstName
  lastName: lastName
  age: age
};
```

We can shorten this code by removing the variable names from the property values. The variable names are the same as the property names, so we can remove them.

```
const person = {
  firstName
  lastName
  age
};
```

## Destructuring an object

Destructuring allows us to pull values/variables out of an object

```
const todo = {
  id: 1,
  title: 'Take out trash',
};

const { id, title } = todo;

console.log(id, title); // 1, 'take out the trash'
```

We can also go multiple levels deep:

```
const todo = {  
  id: 1,  
  title: 'Take out trash',  
  user: {  
    name: 'John',  
  },  
};  
  
const {  
  user: { name },  
} = todo;  
  
console.log(name); // John
```

If we want to rename the properties, we can do that too:

```
const { id: todoId } = todo;  
  
console.log(todoId); // 1
```

## Destructuring an array

We can also destructure arrays. I personally don't destructure arrays as much as I do objects

```
const numbers = [10, 20, 30, 40];  
  
const [firstNumber, secondNumber] = numbers;  
  
console.log(firstNumber, secondNumber) // 10, 20
```

## The rest operator

The spread operator (...) can be used here, but in this situation, we call it the **rest operator**. It puts the "rest" of the values in an array

```
const [firstNumber, secondNumber, ...rest] = numbers;  
  
console.log(rest) // [30, 40]
```

# An Intro to JSON

---

**JSON** is something that you will be working with a lot as a JavaScript developer. **JSON** stands for **JavaScript Object Notation** and is a lightweight data-interchange format. It is essentially a way of storing data in a simple, human-readable format.

In web development, especially in JavaScript, we work with APIs that send and receive data to and from a server. Several years ago, XML (Extensible Markup Language) was the standard for sending data, but in more recent years, JSON has become the standard.

An example of an API that you can make a request to right now and see the JSON response would be the Github API. In your browser, go to the following URL:

<https://api.github.com/users/bradtraversy>

You will see the result is in JSON format.

The reason I want to talk about JSON right now is because we are talking about JavaScript object literals and the syntax that JSON uses is extremely similar. JSON uses curly braces of key/value pairs. Let's look at an example of a JSON object:

```
{  
  "name": "John",  
  "age": 30,  
  "city": "New York"  
}
```

The main difference here is that the keys have double quotes around them. These are required and they must be double quotes. As far as the values, strings must be wrapped in double quotes. Numbers and booleans do not.

We have methods available to turn JavaScript objects into JSON strings and vice versa.

```
const obj = {  
  name: 'John',  
  age: 30,  
  city: 'New York',  
};  
  
// Turn object into JSON string  
const str = JSON.stringify(obj);  
console.log(str); // {"name":"John", "age":30, "city":"New York"}  
  
// Turn JSON string into object  
const obj2 = JSON.parse(str);  
console.log(obj2); // {name: "John", age: 30, city: "New York"}
```

Aside from `stringify()` and `parse()`, there isn't really anything else to learn about the JSON object. You will use these two methods a lot though, whether you are on the front-end or back-end.

We can not access properties from a JSON string.

```
console.log(str.name); // undefined
```

If we try and get the index of a JSON string, we will get the character, because it is treated as a string.

```
console.log(str[0]); // {"
```

## JSON Arrays

We have JSON objects like the one above, but JSON can also have arrays. We can also set arrays as a property value.

```
[  
  {  
    "name": "John",  
    "age": 30,  
    "city": "New York",  
    "hobbies": ["basketball", "cooking"]  
  },  
  {  
    "name": "Jane",  
    "age": 20,  
    "city": "LA",  
    "hobbies": ["movies", "sports"]  
  },  
  {  
    "name": "Jack",  
    "age": 25,  
    "city": "Paris",  
    "hobbies": ["music", "painting"]  
  }  
]
```

We can parse it into a JavaScript object with `JSON.parse()`. Then we can access the properties.

```
const arr = JSON.parse(str);  
console.log(arr[0].name); // John
```

# Object Challenge

---

## Step 1

Create an array of objects called `library`. Add 3 objects with a property of `title`, `author`, `status`. Title and author should be strings (whatever value you want) and status should be another object with the properties of `own`, `reading` and `read`. Which should all be boolean values. For all status, set `own` to `true` and `reading` and `read` to false.

## Step 2

You finished reading all of the books. Set the `read` value for all of them to `true`. Do not edit the initial object. Set the values using dot notation.

## Step 3

Destructure the title from the first book and rename the variable to `firstBook`

## Step 4

Turn the library object into a JSON string. There is a specific function that we looked at in the last section that we can use to do this.

► [Click For Solution](#)

### Step 1 Solution

```
const library = [
  {
    title: 'The Road Ahead',
    author: 'Bill Gates',
    status: {
      own: true,
      reading: true,
      read: false,
    },
  },
  {
    title: 'Steve Jobs',
    author: 'Walter Isaacson',
    status: {
      own: true,
      reading: false,
      read: false,
    },
  },
  {
    title: 'Mockingjay: The Final Book of The Hunger Games',
```

```
author: 'Suzanne Collins',
status: {
  own: true,
  reading: false,
  read: true,
},
};

];
```

## Step 2 Solution

```
library[0].status.read = true;
library[1].status.read = true;
library[3].status.read = true;
```

## Step 4 Solution

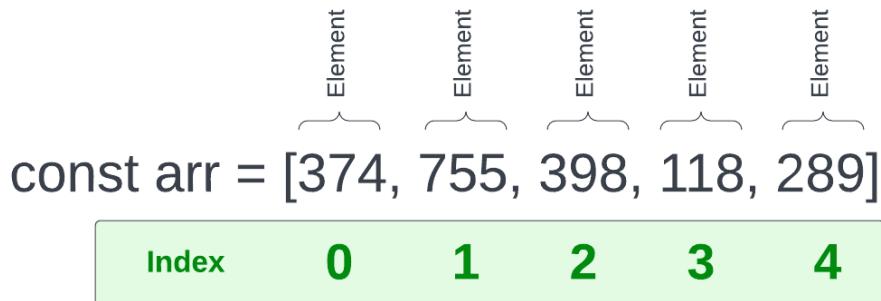
```
const { title: firstBook } = library[0];
console.log(firstBook);
```

## Step 4 Solution

```
const libraryJSON = JSON.stringify(library);
console.log(libraryJSON);
```

# Array Basics

Arrays are a special type of object in JavaScript. Arrays are also what we call a data structure, which is just that, a way of structuring data. Lets look at a basic Array



We create a variable and set it to some brackets with some values inside. The values could be of any type. In this case, we have an array of numbers.

The values in an array are called elements. Each element has an index. The index is the position of the element in the array. The first element in an array has an index of 0. The second element has an index of 1, and so on. In just about every language, arrays are zero-based.

Let's create this same array in our code

```
const arr = [374, 755, 398, 118, 289];
```

If we log the array in the console, we'll see that like other types of objects, it has a prototype property. These are all kinds of properties and methods that can be used with arrays. We'll get more into those in a little bit.

```
console.log(arr);
```

## Array Constructor

There is a second way to create an array that is less common. The one that we have already used is called an array literal. The second is by using the **Array** constructor with the **new** keyword

```
const arr2 = new Array(1, 2, 3, 4, 5);
```

You can use either one to create an array. They both create an array object.

## Getting values by index

So to get one of these values, we can use the index of the element we want to get. To get the first one

```
arr[0];
```

If we wanted to use an array value in an expression, we could

```
arr[0] + arr[3]; //492
```

Arrays can have any type of data within it. In this case, we have an array of numbers, but we could create an array of strings

```
const fruit = ['apples', 'oranges', 'pears'];
```

We can even mix types of data within an array and even have an array within an array

```
const mixed = [1, 'string', true, null, undefined, [1, 2, 3]];
```

## Array length

We can get the length of an array using the `.length` property, which is very useful

```
console.log(arr.length); // 5
```

## Mutating arrays

Arrays are mutable, meaning we can add, change and remove elements. There are many methods that we can use and I'll get into those in a little bit, but lets say we want to add a new element to the end without using any kind of method.

We know that the `length` property gives us the total count of elements and since the array is 0-based, we can simply add the length as the index of the new element

```
fruit[fruit.length] = 'Peaches'; // ['apples', 'oranges', 'pears', 'peaches']
```

In the code above, `fruit.length` was `3` so we set the value of `fruit[3]`, which was the next index.

If we want to target a specific index and change the value, we can do that easily

```
```javascript
fruit[1] = 'mangos';
```

We can also set the array length to a new value, which will shorten the array

```
fruit.length = 2;
```

Now the array would look like this

```
[ 'apples', 'oranges'];
```

# Function Basics

---

Functions are a way to group code together. They make code more readable and easier to understand. Let's look at some of the basics.

## Creating a function

Creating or defining a function is simple. We use the `function` keyword, followed by the name of the function, followed by parentheses. Then we create the function body wrapped in curly braces.

```
function sayHello() {  
  console.log('Hello World');  
}
```

When we create a function, we don't add a semi-colon at the end of the curly braces.

Just creating the function does not execute any code. We need to "call" or "invoke" the function to execute the code within it.

```
sayHello(); // Hello World
```

## Function Parameters

You can define parameters for functions to allow the user to input specific data when calling the function.

```
function add(x, y) {  
  console.log(x + y);  
}  
  
add(5, 5); // 10
```

## Parameters vs. Arguments

- Parameters are the names of the variables that are used to pass data into a function.
- Arguments are the values that are passed into the function

## Returning a value

The function that we created just `console.log`s a message. In the real world, you'll usually want to return a value from a function.

Anything that you put after the return will not get executed because you are essentially ending the function and returning to the parent scope.

```
function subtract(x, y) {
    return x - y;

    // This code will not execute
    console.log('After the return');
}

// This will not output anything
subtract(10, 5);

// You could log the value like This
console.log(subtract(10, 5)); // 5

// Usually we do something with the return value, like store it in a variable
const result = subtract(10, 5);
```

You can also return without a value. This is useful for functions that don't return anything. You may have a function to update or delete some data on a server, but don't have a need to return anything.

# More on Params & Arguments

---

When we create a function, we can allow data to be passed in by creating **parameters** or "params". The data that we pass in when we invoke a function is called an **argument**. I want to look a little more into them.

```
function registerUser(user) {  
  return user + ' is registered';  
}  
  
console.log(registerUser('John')); // John is registered
```

First off, if it isn't obvious, the variable of `user` is only available inside the function. The area within the function is called the function's **scope**.

If I try to log the name variable outside of the function, I will get an error.

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

We will talk more about scope soon.

Let's see what happens if we run the `registerUser()` function, but without passing in a parameter.

```
console.log(registerUser()); // Undefined is registered
```

## Default Parameters

Sometimes, we may want to make parameters optional and have a default value for them if not passed in. In the ES2015 update, they added a new and easy syntax for this. Before that update, we would do something like this.

```
function registerUser(user) {  
  if (!user) {  
    user = 'Bot';  
  }  
  return user + ' is registered';  
}
```

But now, we can simply add the default value with the "=" sign

```
function registerUser(user = 'Bot') {
  return user + ' is registered';
}

console.log(registerUser()); // Bot is registered
```

## Rest Parameters

Usually, you will know exactly which parameters will be passed to your functions. In the case of a function that takes in multiple arguments that we won't know beforehand, we can use the "rest" operator to collect all of the arguments into an array.

```
function sum(...numbers) {
  return;
}

sum(1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]
```

As you can see, all the params we pass in are put into an array.

We have not learned about loops yet, but let's say that we want to add all of the numbers passed in. We could do the following.

```
function sum(...numbers) {
  let total = 0;

  for (const num of numbers) {
    total += num;
  }

  return total;
}

console.log(sum(1, 2, 3, 4)); // 10
```

## Passing objects as arguments

We don't always pass in a primitive value as an argument. Sometimes we want to pass in an object.

```
function registerUser(user) {
  console.log(`User ${user.id}(${user.name}) has been registered`);
}

const user = {
  id: 1,
```

```
    name: 'John',  
};  
  
registerUser(user); // User 1(John) has been registered  
  
// or we can pass the object directly:  
  
registerUser({  
  id: 1,  
  name: 'John',  
});
```

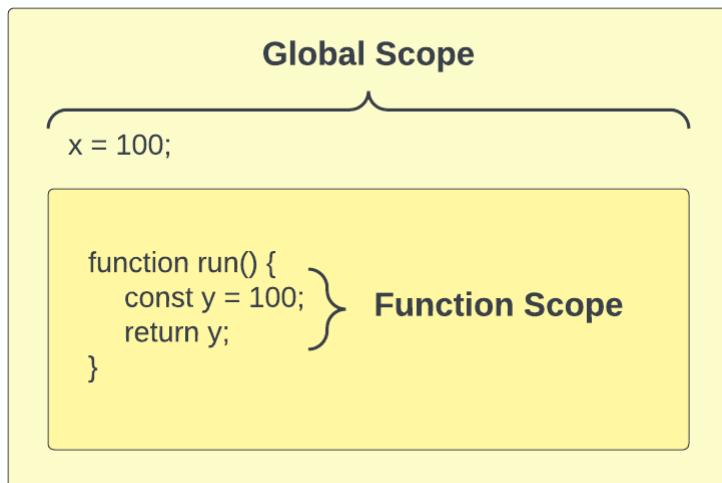
## Passing arrays as arguments

We may want to pass in an array. Let's create a function that takes in an array and returns a random number.

```
function getRandom(arr) {  
  // get random index value  
  const randomIndex = Math.floor(Math.random() * arr.length);  
  
  // get random item  
  const item = arr[randomIndex];  
  
  return item;  
}  
  
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
  
console.log(getRandom(numbers)); // 4
```

# Global & Function Scope

**Scope** is an important concept, not only in JavaScript, but in many other programming languages. Scope is the current area or context of a specific piece of code. There are certain rules for what is accessible in specific scopes.



## Global Scope

In JavaScript, when we write code in the **global scope** it is available everywhere, including functions. If we are NOT inside of a function or any kind of block, such as an if statement or a loop, then we are in the **global scope**.

### The **window** object

The browser creates a global object called **window**. This object has a ton of methods and properties on it that are available to us that we'll be looking at later on in the course.

The **alert** method belongs to the window object.

```
window.alert('Hello World');
```

Since the **window** object is the most top-level object in the browser environment, we don't need to use **window**

```
alert('Hello World');
```

There is an **innerWidth** property on the **window** object. We could use that anywhere as well

```
console.log(window.innerWidth);
```

We can use it in a function as well because it is global

```
function run() {
  console.log(window.innerWidth);
}
```

### Creating globally scoped variables

If I simply create a variable at the top of a JavaScript file, this is a global variable in the global scope and I can access it anywhere.

```
const x = 100;
console.log(x); // 100
```

If we try to access x in the `run()` function, we can because it's global.

```
function run() {
  console.log(x); // 100
}
```

### Function Scope

Function scope is the scope that is available to all code inside of a function. Any variables we define here will be available only inside of the function.

```
function add() {
  const y = 50;
}

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

Since `x` is global, I could use that in the `add()` function

```
function add() {
  const y = 50;
  console.log(x + y); // 150
}
```

If I create a variable called `x` in the function, it will overwrite the global variable and I can no longer access it. This is called **variable shadowing**.

```
function add() {  
  const x = 1;  
  const y = 50;  
  console.log(x + y); // 51  
}
```

## Local Scope

Whatever scope we are currently working in or talking about is called the `local scope`.

# Block Scope

---

**Block scope** is the scope that is available to all code inside of a **block**. A block is something like an if statement or any kind of loop. I know we haven't talked about these yet, but we'll get to them soon. Just know that they have their own scope.

```
const x = 100;

if (true) {
  console.log(x); // 100
  const y = 200;
  console.log(x + y); // 300
}

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

As you can see, we can not access **y** in the global scope because it belongs to the if statement block.

## Loop Example

I know we have not gone over loops yet, but I want to show you that they do have their own block scope.

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

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

As you can see, **i** is only available inside of the loop.

## let & const vs var

For all of these examples, I used **const** to define my variables. Both **const** and **let** are block scoped. **var** on the other hand is NOT. Let's try an example with var.

```
if (true) {
  const a = 500;
  let b = 600;
  var c = 700;
}

console.log(a); // ReferenceError: a is not defined
console.log(b); // ReferenceError: b is not defined
console.log(c); // 500
```

If you use `var` with a for loop, it will also be accessible outside of the loop, which you do not want.

```
for (var i = 0; i <= 10; i++) {  
    console.log(i);  
}  
  
console.log(i); // 11
```

One thing I do want to mention is that `var` is at least **function-scoped**. So if you create a variable in a function with `var`, it will not be accessible outside of the function.

```
function run() {  
    var d = 100;  
    console.log(d);  
}  
  
run();  
  
console.log(d); // ReferenceError: d is not defined
```

When it comes to which of the 3 to use, I would suggest using `let` or `const`. Unless you have a good reason to use `var`. Having variables that are not accessible outside of their scope is usually a good thing.

Another difference with `var` is that when you create a global variable using it, that variable is put on the `window` object.

```
const foo = 1;  
var bar = 2;
```

You can check by typing `window` in your console and you will see `bar` will be there

# Nested Scope

---

## Nested Functions

In JavaScript, we can also define a function within a function. This is called a **nested function**. This relates to something called **closures**, which we will talk about later. When it comes to scope, any variable defined in the parent function is available to the nested/child function, but not the other way around.

```
function first() {  
    const x = 500;  
  
    function second() {  
        const y = 600;  
        console.log(x); // 500  
    }  
  
    console.log(y); // ReferenceError: y is not defined  
  
    second();  
}  
  
first();
```

As you can see, the **x** variable is defined in **first()**, so we have access to it in the child function, **second()**. However, the **y** variable is defined in **second()**, so we can't access it in the parent function, **first()**. We can access variables from parents but can not access variables from child functions.

## Nested If

Just like with functions, we can access the parent block scope, but not the nested/child scope

```
if (true) {  
    const x = 100;  
    // Nested if  
    if (x === 100) {  
        const y = 200;  
        console.log(x + y); // 300  
    }  
  
    // console.log(y); // ReferenceError: y is not defined  
}  
  
// console.log(y); // ReferenceError: y is not defined
```

# Function Declaration vs Function Expression

---

Up to this point, we know how to declare a function and call or invoke it. There is another way that we can create functions and that is with a function expression. Let's first declare a function just like we've been doing.

```
function addDollarSign(value) {
  return '$' + value;
}

addDollarSign('10'); // '$10'
```

A function expression is when we assign a function to a variable.

```
const addPlusSign = function (value) {
  return '+' + value;
};

addPlusSign('10'); // '$10'
```

Notice that we invoke them the same exact way, but we define them differently. The expression is just a variable assigned to a function. Semi-colons are optional in JS, for the most part, but if you are using them, you would put one at the end of the function expression, because again, we are assigning a variable.

## Hoisting

I am not going to explain the ins and outs of hoisting here, I will do that soon when we get into something called **execution context**, but in very simple terms, hoisting is the process of moving all of the function and variable **declarations** to the top of the current scope. before the code is run.

I do want to mention that both functions and variables/function expressions are hoisted, but only function declarations are available before the code is run.

Let's try and access the `addDollarSign()` function **before** it is defined.

```
const money = addDollarSign('10'); // '$10'

function addDollarSign(value) {
  return '$' + value;
}
```

This still works, however, if we try this with a function expression, it will not work

```
addPlusSign('10'); // Cannot access 'addPlusSign' before initialization

const addPlusSign = function (value) {
  return '+' + value;
};
```

I will get more into hoisting later, but for now, just know that you can define a function declaration and a function expression. Which one to use really comes down to preference. It may seem that declarations are better because of the way they're hoisted, but in reality, it's not good practice to use functions before they are defined. Using expressions actually makes your code less prone to errors in my own opinion.

# Arrow Functions

Arrow functions were introduced in **ES6** or **ES2015**. They are a shorter syntax for writing functions with some advantages such as being more compact, implicit returns and lexical scope. Let's look at the syntax.

First off, let's look at a simple traditional function:

```
function add(a, b) {  
    return a + b;  
}  
  
add(1, 2); // 3
```

Now let's look at the same thing written as an arrow function:

```
const add = (a, b) => {  
    return a + b;  
};  
  
add(1, 2); // 3
```

There are a few things to note here. First, we took away the `function` keyword and instead added a variable declaration. This is because arrow functions are always anonymous functions so we need to assign them to a variable in this case.

Secondly, we added what we call a **fat arrow** to the function, which is an equal sign and a greater-than sign. Sometimes, arrow functions are referred to as **fat-arrow functions**.

## Implicit Return

We can actually shorten this particular function a bit more, because it only has one line of code that is a single expression. In this case, we can remove the curly braces and the return statement. If there were multiple lines of code (statements), we would need to add the curly braces and the return statement.

```
const add = (a, b) => a + b;  
  
add(1, 2); // 3
```

Instead of explicitly returning the result, `a + b` will be returned implicitly. When we don't use the `return` keyword, it is called an **implicit return**. If there were more than one line of code in the function, we would have to use the curly braces and the `return` keyword.

## Returning an object literal

Where you can run into an issue with implicit returns is when you want to return an object literal.

Be sure that you use parentheses around the curly braces if you want to return an object literal implicitly.

```
const createObj = () => ({ name: 'John' });
```

## Single argument shorthand

Another cool feature of arrow functions is that we can remove the parentheses around the argument if it is a single argument.

If you are using VS Code with prettier as I am, it may add the parentheses automatically.

```
const add10 = (a) => a + 10;  
  
add10(1); // 11
```

## Anonymous Functions

Many times we don't want to use a named function, but instead want to use an anonymous function. This is useful when we want to use a function as a callback. We can use an arrow function in this case as well.

I know we have not gone over high order array methods like `forEach()` yet (we will very soon), but just to give you an idea of how this works. Let's create an array with some numbers.

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

The `forEach()` method can be used to iterate over an array. It takes a callback function as an argument. The callback function will be called for each element in the array.

```
numbers.forEach(function (number) {  
    console.log(number);  
});
```

To shorten this to one line, we can use an arrow function:

```
numbers.forEach((number) => console.log(number));
```

# Immediately Invoked Function Expressions (IIFE)

---

Usually when we deal with functions, we define them and then when we want to run them, we call/invoke them. There may be times where you want to define a function and run it at the same time. In JavaScript, we can do this with an **immediately invoked function expression**, also called an IIFE (pronounced "Iffy").

## Why Use an IIFE?

There are a few reasons why you may want to use an IIFE. One of the biggest reasons is to avoid **global scope pollution**.

If I have a second JavaScript file loading, whether I created it or it is some 3rd party library, etc and in that file is something like this.

```
const user = 'Brad';
```

and then in my script I don't know about that global variable and I create a variable with the same name, it will break my script. Because we are defining two variables in the same scope.

```
const user = 'John'; // Results in error
```

To fix this, I could create an IIFE

## IIFE Syntax

The syntax for an IIFE is a little strange. You have to wrap the function keyword and the function body in parentheses. You then have another set of parentheses to indicate that it is an invocation, just like you would with a regular function call.

```
(function() {
  const user = 'John';
  console.log(user); // John
})();

// Outside of IIFE
console.log(user); // Brad
```

When it comes to global scope pollution, if you have a small script with little to no dependencies, this most likely won't be a problem. If you have a larger project, especially if multiple people are working on it, then you may want to use an IIFE because you don't know exactly which variables and functions have been defined.

## Adding Parameters

You can also create an IIFE that takes parameters.

```
(function (name) {  
  console.log('Hello ' + name);  
} )('John');  
  
// Hello John
```

## Using Arrow Functions

We can also use the arrow function syntax to create an IIFE.

```
(( ) => {  
  console.log('Hello from the IIFE!');  
})();
```

## Named IIFE

The examples above are all unnamed functions. We can give IIFEs a name, however, the function name would only be available from within the IIFE. So we could use something called **recursion**, which is when a function calls itself. The function would not be available outside the IIFE. I would not suggest running the function within itself, because it will cause an infinite loop.

```
(function hello() {  
  console.log('Hello from the IIFE!');  
  
  hello() // Causes infinite loop  
} )();  
  
hello(); // Error: hello is not defined
```

IIFEs are also used in specific design patterns such as the **revealing module pattern**, which I will talk about later.

# Function Challenges

---

Now that you know the basics of how functions work, let's try a few challenges.

## Challenge 1

### Instructions:

Create a function called `getCelsius()` that takes a temperature in Fahrenheit as an argument and returns the temperature in celsius.

For bonus points, write it as a one line arrow function

### Expected Result:

```
console.log(getCelsius(32)); // 0
```

You can make the output look prettier by putting it into a string. You can even add `\xB0` (degrees) and a `C` in front of the celsius temperature.

```
console.log(`The temperature is ${getCelsius(32)} \xB0C`);  
// The temperature is 0 °C
```

### Hints:

The formula for Fahrenheit to Celsius is  $(F - 32) * 5 / 9$ . Read more about the formula [here](#) if needed.

► Click For Solution

```
const getcelsius = (fahrenheit) => {  
  const celsius = ((fahrenheit - 32) * 5) / 9;  
  return Math.round(celsius);  
};  
  
console.log(`The temperature is ${getcelsius(32)} \xB0C`); // The temperature  
is 0 °C
```

## Challenge 2

### Instructions:

Create an arrow function called `minMax()` that takes in an array of numbers and returns an object with the minimum and maximum numbers in the array.

### Expected Result:

```
console.log(minMax([1, 2, 3, 4, 5]));
// { min: 1, max: 5 }
```

### Hints:

1. You can use `Math.min()` and `Math.max()` to get the min and max of a list of numbers.
2. You can use the Spread `...` operator to spread the values of an array into separate arguments.

► Click For Solution

```
function minMax(arr) {
  const min = Math.min(...arr);
  const max = Math.max(...arr);

  return {
    min,
    max,
  };
}

console.log(minMax([55, 32, 43, 54, 65, 76, 87, 98, 109]));
// { min: 32, max: 109 }
```

## Challenge 3

Create an IIFE (Immediately Invoked Function Expression) that takes in the length and width of a rectangle outputs it to the console in a message as soon as the page loads.

### Expected Result:

```
// On page load
The area of a rectangle with a length of 10 and a width of 5 is 50.
```

### Hints:

1. The area of a rectangle is `length * width`. These should get passed into the IIFE as arguments.
2. You do not have to return anything from this function, just log to the console.

► Click For Solution

```
((length, width) => {
  const area = length * width;
```

```
const output = `The area of a rectangle with a length of ${length} and a width  
of ${width} is ${area}.`;  
  
console.log(output);  
})(10, 5);
```

# Execution context

---

Inside the browser is a JavaScript engine that handles executing our JavaScript. That engine creates a special environment to handle the transformation and execution of code. This environment is called the **execution context**, and it contains the currently running code and everything that aids in its execution.

When we load a JavaScript file in the browser, the first thing that the engine does is create a new execution context. This is called the **global execution context**. It basically includes all of the code that is in the **global scope**. We talked about scope earlier.

## Visualizing the execution context

To give you a better idea of what the execution context looks like, you can picture it as a box with 2 sides. One side is the **variable environment** which is a memory area for your variables and functions in key-value pairs.

Then on the other side, you have your actual lines of code being run. This is the **thread of execution**. The code is executed one line at a time on a **single thread**, which you can think of as like a **process**. JavaScript is a single-threaded, synchronous language.

MEMORY	EXECUTION (CODE)
<p><b>name: 'John'</b> <b>x: 100</b> <b>y :200</b></p> <p><b>fn: {...}</b></p> <p>This is the <b>variable environment</b> that stores all of your variables and functions as key:value pairs in memory</p>	 <p>This is the <b>thread of execution</b>. Each line of code is executed line by line</p>

## Types of Execution Contexts

So we have a couple different types of execution contexts.

### Global Execution Context

The global execution context is the default or base execution context. This is created when you run a JavaScript file in a browser. It is NOT within a function. When the global execution context is created, it

performs two things. It creates the `global object` which is the `window` object in a browser. In Node.js, it's just called the `global object`. It also sets the value of the `this` keyword equal to the global object.

## Function Execution Context

When a function is invoked, a new execution context is created for that function. Each function has its own execution context. So a new box with the 2 sides is created for the code within the function.

## Eval Execution Context

`eval()` is a function property of the global object. Most JavaScript developers don't use eval, so I'm not even going to get into it here.

So just remember that we have a `global` execution context and when a function is invoked, it has its own `function` execution context.

## Execution Context Phases

When you run a piece of JavaScript code, you create a new execution context and it's created in two phases.

- The first being the **memory creation phase** or sometimes just called the **creation phase**. This is where JavaScript allocates the memory for all of the variables and functions. This happens before any code is executed. Think of it as a first pass where the JavaScript engine just goes over all the code and puts the variables and functions into memory.
- The second is the **execution phase** where the code is executed line by line. This happens after the memory creation phase.

One thing to note is that in the memory creation phase, your variables are all put into memory as `undefined`. The actual values of the variables are set during the execution phase. However, functions are directly stored in memory with all of the code inside of them.

## Execution Context In Action

Let's take the following code and see exactly what happens when it runs:

```
1 var x = 100
2 var y = 50
3 function getSum(n1, n2) {
4     var sum = n1 + n2
5     return sum
6 }
7 var sum1 = getSum(x, y)
8 var sum2 = getSum(10, 5)
```

### 1. Global execution context is created

- Global object is created and assigned to `this`

## 2. Memory creation phase (global execution context)

- **Line 1:** `x` variable is allocated memory and stores `undefined`
- **Line 2:** `y` variable is allocated memory and stores `undefined`
- **Line 3:** `getSum()` function is allocated memory and stores all the code in the function
- **Line 7:** `sum1` variable is allocated memory and stores `undefined`
- **Line 8:** `sum2` variable is allocated memory and stores `undefined`

## 3. Execution phase (global execution context)

- **Line 1:** places the `100` into the `x` variable
- **Line 2:** places the `50` into the `y` variable
- **Line 3:** skips the function because there is nothing to execute. It is a declaration
- **Line 7:** Invokes the `getSum()` function and creates a **function execution context** with a new **variable environment** and **execution thread**. It will also run the **memory phase** and the **execution phase**.

## 4. Memory creation phase (function execution context)

- **Line 3:** parameters `n1` & `n2` variable is allocated memory and stores `undefined`
- **Line 4:** `sum` variable is allocated memory and stores `undefined`

## 5. Execution phase (function execution context)

- **Line 3:** `n1` & `n2` are assigned `100` and `50`
- **Line 4:** The calculation is done and the result (`150`) is put into `sum` variable
- **Line 5:** return tells the **function execution context** to return to the **global execution context** with this value of `sum` (`150`)

**Line 7** - The returned sum value is put into the `sum1` variable in the **global execution context**. The execution context of that function is then **deleted**.

**Line 8** - We repeat the same process and create a new **function execution context**, just with different parameters being passed in.

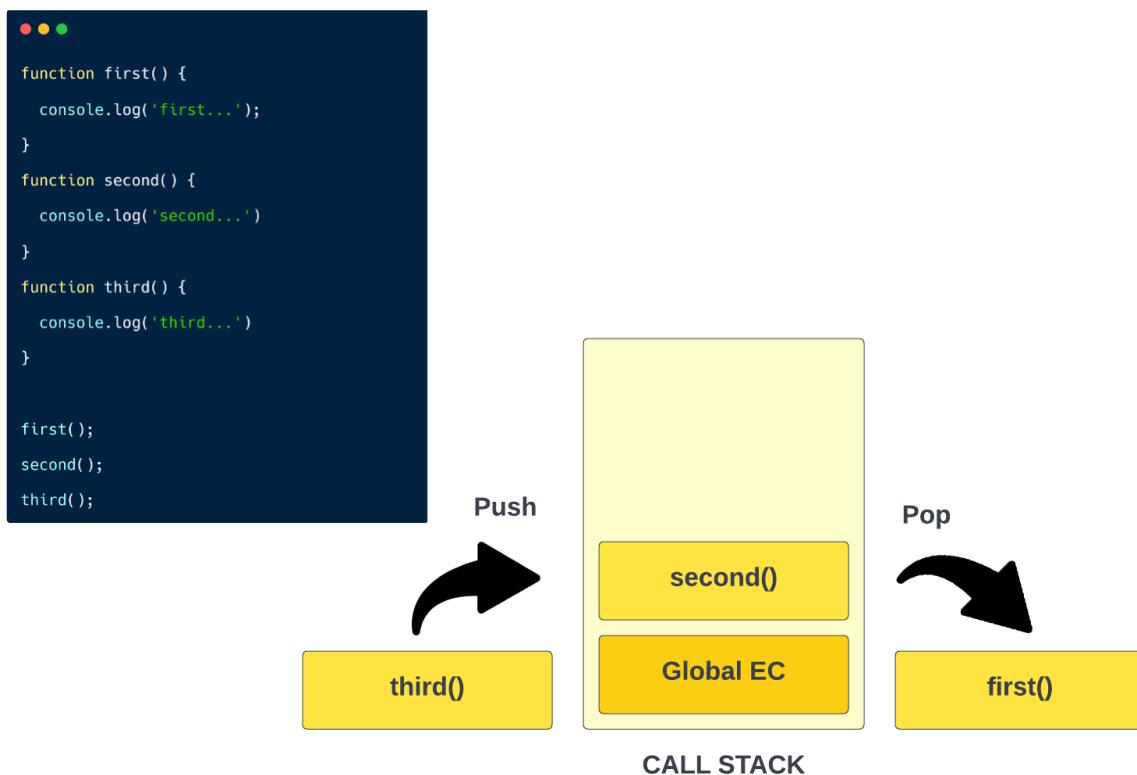
# The Call Stack

So we talked about the **execution context** in the last video and stepped through some code to see exactly what was happening under the hood. We had our global execution context and we had our function execution context. Well every function that was invoked was added to something we call the **call stack**. It's also referred to as the **execution stack** and many other terms. This keeps track of the functions that are currently being executed. You think of it as an execution context manager.

## What is a stack?

In computer science, a **stack** is a data structure. Later in the course, we'll get deeper into stacks. It's a **LIFO (last in, first out)** data structure. What this means is the last thing to be put on top of the stack is the first to come off.

To give you a better picture of this, let's look at some basic code. And I'll actually show you this in the browser as well because you can see the call stack in the sources tab.



Here we have three functions that are all being run in the global scope. When I say global scope I just mean not in a function. So this is how the call stack would work in this case.

First of all, the global execution context is always at the very bottom of the call stack. Then the function named `first` will get put on to the stack because that is the first function run in our code. The technical terminology is **push** when something is placed on the stack and **pop** when something is taken off. So `first()` will be pushed on and executed and popped off. Next, `second()` will be pushed on and executed, then

popped off and finally, third(). Since they're all in the global scope, they don't ever get stacked on top of each other. It's just one after the other.

Now let's look at a different example.

```
● ● ●

function first() {
    console.log('first...');
    second();
}

function second() {
    console.log('second...')
    third();
}

function third() {
    console.log('third...')
}

first();
```



Here, the only function being invoked in the global scope is first(). Then in the first() function, the second() is being invoked, now we're still in the process of executing first(), so that stays on the stack and second() is pushed on top of it. Then in second(), we're executing third(), so first() and second() are still on the stack and third() gets added as well. Once the final console log is done, third() is done and taken off in **last in first out** fashion, then second() is done that gets popped off then first(). Then it's done and the **global execution context** is deleted.

# If Statements

---

In any programming language, there will be times where you need to execute certain blocks of code and commands if a certain condition is true or false. Without the ability to have this type of conditional logic, programming wouldn't really be possible.

The structure of an **if-statement** is as follows:

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

The code block will only execute if the condition is true. If the condition is false, the code block will not execute.

The condition is a boolean expression that evaluates to true or false. In fact, we could put in true or false directly

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

There are something called **truthy** and **falsy** values in JavaScript. I will get more into this in a bit.

## Comparison Operators

A few sections back, we looked at the comparison operators. These operators are used to compare values and return a boolean value.

Operator	Definition	Example	Result
<	Less than	10 < 5	False
>	Greater than	10 > 5	True
<=	Less than or equal to	10 <= 10	True
>=	Greater than or equal to	10 >= 5	True
==	Equal to	10 == '10'	True
!=	Not equal to	10 != 5	True
====	Equal to (including type)	10 === '10'	False
!==	Not equal to (including type)	10 !== '10'	True

We could use these operators to compare values.

```
const x = 10;

if (x > 5) {
  console.log(` ${x} is greater than 5`);
}

if (x === 10) {
  console.log(` ${x} is equal to 10`);
}
```

## if-else Statements

Instead of doing nothing if the condition is false, we can execute different code blocks depending on the condition.

```
const x = 10;

if (x > 5) {
  console.log(` ${x} is greater than 5`);
} else {
  console.log(` ${x} is less than or equal to 5`);
}
```

## Block Scope

We talked about this earlier, but remember that variables defined with `let` or `const` inside of a block are not accessible outside of that block. `var` variables are, but I would not suggest using them.

```
const x = 10;

if (x > 5) {
  const y = 20;
  console.log(`${x} is greater than 5`);
  console.log(` ${y} is greater than 5`);
}

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

## Shorthand If

If the code within your if and else is a single statement and not a block of code, you omit the parentheses. It is usually not recommended and you don't see it that often, but you can do it.

```
if (x > 5) console.log(` ${x} is greater than 5`);

if (x > 5) console.log(` ${x} is greater than 5`);
else console.log(` ${x} is less than 5`);
```

You actually can have multiple statements, but they need to be separated with a comma.

```
if (x > 5)
  console.log(` ${x} is greater than 5`),
  console.log('another line'),
  console.log(` and another`);
```

I would not personally do this.

# Else-if and Nesting

---

There may be cases where you want to execute different code blocks depending on multiple conditions. In this case, you can use an `else if` statement.

Let's make this a bit more interesting and have certain logs during certain hours.

```
const d = new Date();
const hour = d.getHours();
```

```
if (hour < 12) {
  console.log('Good Morning!');
} else if (hour < 18) {
  console.log('Good Afternoon!');
} else {
  console.log('Good Night!');
}
```

## Nesting

There may be cases where you need to have `if statements` inside of `if statements`. In this case, you can use a `nested if statement`.

```
if (hour < 12) {
  console.log('Good Morning!');

  if (hour === 6) {
    console.log('Wake up!');
  }
} else if (hour < 18) {
  console.log('Good Afternoon!');
} else {
  console.log('Good Night!');

  if (hour >= 20) {
    console.log('zzzzzzzzzz!');
  }
}
```

## Multiple conditions

We can test for multiple conditions in the same if statement by using the `&&` (AND) and the `||` (OR) logical operators.

```
if (hour >= 7 && hour < 15) {  
    console.log('It is work time!');  
}  
  
if (hour === 6 || hour === 20) {  
    console.log('Brush your teeth!');  
}
```

# Switches

---

If you find yourself with a lot of else-if statements and you are testing a single value, you may want to consider using a switch statement. The switch evaluates an expression and then executes the first case that matches the value.

The format for a switch statement is as follows:

```
switch (expression) {  
    case value1:  
        statement1;  
        break;  
    case value2:  
        statement2;  
        break;  
    default:  
        statement3;  
}
```

We pass in an expression to evaluate and then we use the case keyword to define a value to match against the expression. We then use the break keyword to exit the switch statement. The default will run if there is no cases that match.

Let's say that we just want to log a message if the month is January, February, or March.

```
const d = new Date(2022, 1, 20, 8, 0, 0);  
const month = d.getMonth();  
  
switch (month) {  
    case 1:  
        console.log('It is January');  
        break;  
    case 2:  
        console.log('It is February');  
        break;  
    case 3:  
        console.log('It is March');  
        break;  
    default:  
        console.log('It is not January, February or March');  
}
```

Switches are best for immediate values. When using ranges, they are slower than `else-if`. To use a switch with ranges, we can do this.

```
const hour = d.getHours();

switch (true) {
  case hour < 12:
    console.log('Good Morning');
    break;
  case hour < 18:
    console.log('Good Afternoon');
    break;
  default:
    console.log('Good Night');
}
```

# Calculator Challenge

---

Now that you know how to write functions and use control structures like if statements and switches, let's try to write a simple calculator.

## Instructions:

Create a function called `calculator` that takes three parameters: `num1`, `num2` and `operator`. The operator can be `+`, `-`, `*` or `/`. The function should return the result of the calculation. If anything other than the four operators is passed in, the function should return an error message.

## Example:

```
calculator(5, 2, '+') // returns 7
calculator(5, 2, '-') // returns 3
calculator(5, 2, '*') // returns 10
calculator(5, 2, '/') // returns 2.5
calculator(5, 2, '%') // returns an error message
```

## Hint:

- You can use an if statement to the operator, but this is a good example for using a switch statement.
- Click For Solution

```
function calculator(num1, num2, operator) {
  let result;
  switch (operator) {
    case '+':
      result = num1 + num2;
      break;
    case '-':
      result = num1 - num2;
      break;
    case '*':
      result = num1 * num2;
      break;
    case '/':
      result = num1 / num2;
      break;
    default:
      result = 'Invalid operator';
  }
  console.log(result);
  return result;
}
```

```
calculator(3, 4, '*'); // returns 12
```

# Truthy & Falsy Values

---

So we looked at some control structures for evaluating some basic expressions such as if something is greater than a value. We can also pass in a single value.

```
const email = 'test@test.com';

if (email) {
  console.log('You passed in an email');
} else {
  console.log('Please enter email');
}
```

This will return true because when you put something in an if statement, it is coerced into a boolean. A string with something in it is what we call a **truthy** value.

## Converting a value to a boolean

We can test truthy and falsy values by converting them to a boolean using the **Boolean** function. We saw this a while back when we looked at data types.

```
console.log(Boolean(name)); // true
```

We can also use the **!!** (double bang) to convert to a boolean.

```
console.log (!!name); // true
```

Let's look at the values in JavaScript that are considered **falsy**.

## falsy Values

- `false` (obviously)
- `0` (also `-0` and `BigInt 0n`)
- `""` (empty string)
- `null`
- `undefined`
- `Nan`

If we test any of these in an if statement, it will evaluate to `false`.

```
const x = 0;
```

```
if (x) {  
  console.log('This is truthy');  
} else {  
  console.log('This is falsy');  
}
```

We talked about type conversion a while ago. If we convert any of these values to a Boolean, they will result in `false`.

```
console.log(Boolean(0)); // false
```

## Truthy Values

Everything that is not falsy will evaluate to `true`, however, Some of these may surprise you.

- Everything else that is not falsy
- `true` (obviously)
- `'0'` (string with 0)
- `'false'` (string with false)
- `''` (space in a string)
- `[]` (empty array)
- `{}` (empty object)
- `function() {}` (empty function)

If we test any of these in an if statement, it will evaluate to `true`.

```
const x = '0';  
  
if (x) {  
  console.log('This is truthy');  
} else {  
  console.log('This is falsy');  
}
```

## Truthy & Falsy Caveats

Let's say we have a variable called `children` that refers to the number of children someone has and we want to check it.

```
let children = 2;  
  
if (children) {  
  console.log(`You have ${children} children`);  
} else {  
  console.log('Please enter the number of children you have');
```

```
}
```

```
// You have 2 children
```

Now that you understand that 0 is falsy, you will understand why the following will not work correctly.

```
let children = 0;
```

```
if (children) {
```

```
    console.log(`You have ${children} children`);
```

```
} else {
```

```
    console.log('Please enter the number of children you have');
```

```
}
```

```
// Please enter the number of children you have
```

In this case, we want 0 to be a valid value for children, but it is falsy, so the `else` block will run and say it is not defined.

This is something that you have to be aware of. There are multiple solutions to this problem. We could check to see if the variable is not `undefined` or `null`.

```
if (children !== undefined) {
```

```
    console.log(`You have ${children} children`);
```

```
} else {
```

```
    console.log('Please enter the number of children you have');
```

```
}
```

You probably want the value to be a number. So you could check if the value is `Nan` (Not a Number) with the `isNaN()` function.

```
if (!isNaN(children)) {
```

```
    console.log(`You have ${children} children`);
```

```
} else {
```

```
    console.log('Please enter the number of children you have');
```

```
}
```

## Checking For Empty Arrays and Objects

Since empty arrays and objects are truthy, we can not simply check for the variable that holds them.

Check for an empty array:

```
const arr = [];

if (arr.length === 0) {
  console.log('The array is empty');
}
```

Check for an empty object:

```
const obj = {};

if (Object.keys(obj).length === 0) {
  console.log('The object is empty');
}
```

## Loose Equality Comparison

As I stated earlier in the course, I prefer to use `==` unless there is a specific reason to use `===`. Using double equals (`==`) can cause some unexpected behavior. Let's take a look at some of the weirdness that can happen when using `==`.

`false`, zero and empty strings are equivalent (when using `==`)

```
false == 0; // true
false == ''; // true
0 == ''; // true
```

# Logical Operators

---

If we use the `&&` and `||` in a conditional, this is how they work:

```
console.log(10 < 20 && 30 > 15 && 40 > 30); // Must all be true
console.log(10 > 20 || 30 < 15); // Only one has to be true
```

There may be cases where you want to conditionally assign a value based on if a value is `truthy` or `falsy`. We can use logical operators and logical assignment for this.

## AND Operator (`&&`)

If we assign a value, using the `&&` operator, it will execute from left to right. If any of the values are `falsy`, that value will be returned; otherwise, the last value will be returned.

```
x = 10 && 20; // 20
x = 10 && 20 && 30; // 30
x = 10 && 0 && 30; // 0 (30 is not evaluated)
x = 10 && false && 30; // false (30 is not evaluated)
```

## OR Operator (`||`)

The `||` operator is the opposite of the `&&` operator. It will return the first value that is `truthy`. This is more common than using the `&&` operator.

```
y = 10 || 20; // 10 (20 is not evaluated)
y = 0 || 20; // 20
y = 10 || 0 || 30; // 10 (0 and 30 are not evaluated)
y = false || false || 30; // 30
```

## Nullish Coalescing Operator (`??`)

The `??` operator will return the right side operand when its left side is either `null` or `undefined`. It is similar to `||` operator except, it doesn't look at all falsey values, only `null` and `undefined`.

```
let c;
c = 10 ?? 20; // 10
c = null ?? 20; // 20
c = undefined ?? 20; // 20
c = null ?? 0 ?? 20; // 0 (it will return 0 because it is not null or
undefined)
```



# Logical Assignment

---

What we looked at in the last video was also technically logical assignment, however what we were evaluation was not necessarily the value that we were assigning. If what we are assigning is what we are evaluating, we can use a shorthand syntax.

## OR Logical Assignment (||=)

We can do something similar with `||=`. This will assigns the right side value only if the left is a falsy value.

Here is the long form of what we are going to do

```
let a = null;  
  
if (!a) {  
  a = 10;  
}
```

If `a` is truthy, it will be left alone. If it is falsy, it will be set to 10.

A shorter way would be to use the logical OR operator like we did in the last video.

```
a = a || 10;
```

An even shorter way would be to use the OR assignment operator

```
a ||= 10;
```

## AND Logical Assignment (&&=)

We can do something similar with the AND assignment operator

First, let's looka at the long version of what we're doing

```
let b = 10;  
  
if (b) {  
  b = 20;  
}
```

If `b` is **truthy** then we are setting it to 20.

A shorter way would be to use the logical AND operator

```
b = b && 20;
```

We can make it even shorter by using the AND assignment operator

```
b &&= 20;
```

This returns 20 because the value of `b` is 10, which is truthy.

If we try this with a falsy value, we get that falsy value.

```
let b = false;
b = b && 20; // false
```

## Nullish Coalescing Assignment (??=)

We can use `??=` to assign a value to a variable if it is null or undefined.

Here is the long version

```
let c = null;

if (c === null || c === undefined) {
  c = 20;
}
```

Using the `??` operator

```
c = c ?? 20;
```

Using the `??=` assignment operator

```
c ??= 20;
```

# Ternary operator

In addition to `if-else` and `switch`, there is a third way to write conditional logic. This is called the **ternary operator**. It is basically a shorthand, one line `if-else` statement. The actual operator is just a question mark. Let's take a look at the syntax

```
condition ? true : false;
```

First, we have the condition to be evaluated as a boolean. Then we have a `?`, which is the **ternary operator** and after that is the expression that we want to happen if the condition is `true`. The `:` is the else. So the expression after that will execute if the condition is `false`.

Let's look at a simple if-else statement:

```
const age = 19;

if (age >= 18) {
  console.log('You can vote!');
} else {
  console.log('You can not vote!');
}

// You can vote!
```

We can do the same thing using the ternary operator:

```
age >= 18 ? console.log('You can vote!') : console.log('You can not vote!');

// You can vote!
```

## Assigning result to a variable

In many cases, we want to store the result of the ternary operator in a variable.

```
const canVote = age >= 18 ? 'You can vote!' : 'You can not vote!';

console.log(canVote); // You can vote!
```

This is much more compact than this:

```
let canVote;

if (age >= 18) {
  canVote = 'You can vote!';
} else {
  canVote = 'You can not vote!';
}

console.log(canVote); // You can vote!
```

## Multiple statements

In most cases, you will just have a single expression in the true/false part of the ternary operator, however you can have Multiple statements by using a comma.

```
const auth = true;

const redirect = auth
? (alert('Welcome To The Dashboard'), '/dashboard')
: (alert('Access Denied'), '/login');

console.log(redirect);
```

In the case above, it will alert and then return the last value in the ternary.

## Multiple Ternary Operators

We can also have multiple ternary operators and conditions. Much like an else-if.

```
const canDrink =
age >= 21
? 'You can drink!'
: age >= 18
? 'You can have 1 beer'
: 'You can not drink';

console.log(canDrink);
```

## Using **&&** as a shorthand

If your else(semi-colon) in a ternary is going to be null or you don't need an else, you can use the **&&** operator instead. Here is an example:

```
auth ? console.log('Welcome to the dashboard') : null;
```

We could just do:

```
auth && console.log('Welcome to the dashboard');
```

This is the same as:

```
if (auth) {  
  console.log('Welcome to the dashboard');  
}
```

Using the `&&` operator as a shorthand is a popular thing to do in React templates. Many times, you will see something like:

```
{!loading && (  
  <Fragment>Some UI</Fragment>  
)}
```

Which is saying if done loading, then load the JSX fragment.

# For Loop

---

A loop is a **control structure**. It provides a way to do **iteration** in programming. Iteration is a process where you repeat something over and over again until a certain condition is met. This can help automate tasks.

There are many different kinds of loops in programming, and one of the most common loop is the **for** loop.

## For Loop Syntax

The syntax for a **for** loop is very similar in many languages. The syntax is:

```
for ([initialExpression]; [conditionExpression]; [incrementExpression])
  statement
```

- The **initial expression** usually initializes a variable/counter
- The **condition expression** is the condition that the loop will continue to run as long as it is met or until the condition is false
- The **increment expression** is the expression that will be executed after each iteration of the loop
- The **statement** is the code that will be executed each time the loop is run. To execute a **block** of code, use the **{}** syntax

It is important to note that we use semi-colons to separate the different parts of the loop and not commas.

Here is a very simple example of a for loop that will print out the string "Number X" where X is the number of the iteration.

```
for (let i = 1; i <= 10; i++) {
  console.log("Number " + i);
}

// Output:
// Number 1
// Number 2
// Number 3
// Number 4
// Number 5
// Number 6
// Number 7
// Number 8
// Number 9
// Number 10
```

The variable `i` is initialized to 1, and the condition is that the loop will continue to run until the value of `i` is greater than 10. The increment expression is that the value of `i` will be incremented by 1 each time the loop is run.

We could change the initial expression to 5 and it will start from 5 instead of 1.

```
for (let i = 5; i <= 10; i++) {  
    console.log("Number " + i);  
}  
  
// Output:  
// Number 5  
// Number 6  
// Number 7  
// Number 8  
// Number 9  
// Number 10
```

If I wanted to count up to 100 by 5s, I could change the initial expression to 0 and the increment expression to 5.

```
for (let i = 0; i <= 100; i += 5) {  
    console.log("Number " + i);  
}  
  
// Output:  
// Number 0  
// Number 5  
// Number 10  
// Number 15  
// ...  
// Number 100
```

## Block scope

Loops are considered a `block`, just like if statements. Remember that variables using `let` and `const` are scoped to the block they are defined in. So if we define a variable in a loop, we can not access it outside of the loop.

```
for (let i = 0; i <= 100; i += 5) {  
    const message = 'Number ' + i;  
    console.log(message);  
}  
  
console.log(message); // ReferenceError: message is not defined
```

However, if we use `var`, we can access the variable outside of the loop. This is not a good practice, but it is possible.

```
for (let i = 0; i <= 100; i += 5) {  
  var message = 'Number ' + i;  
  console.log(message);  
}  
  
console.log(message); // Number 100
```

## Testing Conditions

In many cases, you will be testing for a condition in a loop. Let's look at a simple example:

```
for (let i = 0; i <= 10; i++) {  
  if (i === 7) {  
    console.log('7 is my favorite number');  
  }  
  
  console.log('Number ' + i);  
}
```

In the code above, we are testing the value of `i` to see if it is equal to 7. If it is, we will log the message "7 is my favorite number".

In the next video I will go over skipping an iteration and breaking out of a loop if a certain condition is true.

## Nested For Loops

`i` is a common variable name for a loop counter. Sometimes you will have a loop within a loop. In that case, you need to use a different variable name for each loop. `j` is a common variable name for a nested loop counter.

Here is an example where we loop through 1-10 and then have another loop in each iteration that shows that number multiplied by 1 - 10.

```
for (let i = 1; i <= 10; i++) {  
  console.log('Number ' + i);  
  for (let j = 1; j <= 10; j++) {  
    console.log(i + ' * ' + j + ' = ' + i * j);  
  }  
}
```

## Looping Over Arrays

Arrays have a `forEach()` method that allows you to loop over them. This is the most common way to loop over an array and we will be looking at `forEach()` and other array methods soon, however I do want to show you that we can loop over an array with a `for` loop.

```
const names = ['Brad', 'Sam', 'Sara', 'John', 'Tim'];

for (let i = 0; i < names.length; i++) {
    console.log(names[i]);
}
```

We simply specify the condition expression as long as `i` is less than the length of the array.

If you wanted to find a specific iteration and index, you could do something like this

```
for (let i = 0; i < names.length; i++) {
    if (i === 2) {
        console.log(names[i] + ' is the best');
    } else {
        console.log(names[i]);
    }
}
```

## Infinite Loops

Infinite loops are something that you will probably run into at one point or another. They are loops that will never stop running. One common cause of this is forgetting to increment the counter. Then the condition is always met. This happens more with while loops, because of the way they are formatted.

To purposely create an infinite `for` loop, we could do this:

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

# Break & Continue

---

## Break Statement

We can create conditions inside of a loop to test for different things within each iteration. There may be cases where you want to stop the loop from running if a certain condition is met. In this case, you can use a **break** statement to break out of the loop.

Let's say we have a loop that prints Numbers 1 - 20, but if the number is 15, we want to exit the loop.

```
for (let i = 1; i <= 20; i++) {
  if (i === 15) {
    console.log('Found the number 15!');
    break;
  }

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

## Continue Statement

We can also use a **continue** statement to skip the rest of the code in the current iteration and continue to the next iteration.

```
for (let i = 1; i <= 20; i++) {
  if (i === 13) {
    console.log('Skipping 13!');
    continue;
  }

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

# While Loops & Do While Loops

---

In the last video, we looked at `for` loops, which are used to iterate over a block of code until a condition is met. A `while` and `do while` loop does the same thing using a different syntax. They work a bit different as well.

## While Loop Syntax

The syntax for a while loop is very similar in many languages. The syntax is:

```
while ([conditionExpression]) {  
    statement  
}
```

There are a few differences between the `for` loop, which we have already talked about, and the `while` loop.

In a `while` loop, the variable is initialized before the loop runs and it is initialized outside of the loop.

Let's look at a simple example and print Number 1-10, like we did with the `for` loop.

```
let i = 0;  
  
while (i <= 20) {  
    console.log('Number ' + i);  
    i++;  
}
```

For many cases, you could use a `while` or a `for` loop and achieve the same result, but a rule that many programmers follow is to use a `for` loop when you know the number of times you want to run the loop and use a `while` loop when the number of times the loop will run is unknown. That is not a mandatory convention, but something that a lot of people do.

## Looping Over Arrays

We can loop over arrays with `while` loops as well

```
const arr = [10, 20, 30, 40];  
  
let i = 0;  
  
while (i < arr.length) {  
    console.log(arr[i]);
```

```
i++;  
}
```

## Nested While Loops

We can nest `while` loops to create a loop that runs a loop inside of a loop.

```
let i = 1;  
  
while (i <= 10) {  
    console.log('Number ' + i);  
    let j = 1;  
    while (j <= 10) {  
        console.log(i + ' * ' + j + ' = ' + i * j);  
        j++;  
    }  
    i++;  
}
```

## Do While Loops

The `do while` loop is a little different from the `while` loop. The `do while` loop will always run at least once, even if the condition is false.

So the answer to the question, "when would I want to use a `do while` loop?" is when you **always** want to run the block of code at least once.

The syntax for a `do while` loop is:

```
do {  
    statement  
} while ([conditionExpression]);
```

Let's look at a simple example:

```
let i = 1;  
  
do {  
    console.log('Number ' + i);  
    i++;  
} while (i <= 20);
```

Now let's change the `i` value to 21. This means that the condition expression is never met, however, the `console.log()` will run at least once, even if the condition is false.

```
let i = 21;

do {
  console.log('Number ' + i);
  i++;
} while (i <= 20);
```

You can also use the `break` and `continue` statements with while and do while loops.

# FizzBuzz Challenge

---

When you go to get a job as a web developer, you may be given specific programming challenges. It's good to practice challenges like this not only for job interviews, but it sharpens your skills in general. You'll find that most challenges have some kind of iteration involved. One of the most common challenges is the FizzBuzz challenge. This has to do with loops and conditionals, so I think that this is a good challenge at this point in the course.

## Instructions:

- Print/log the numbers from 1 to 100
- For **multiples of three** print "Fizz" instead of the number
- For **multiples of five** print "Buzz"
- For numbers which are **multiples of both three and five** print "FizzBuzz".

## Hints:

If you have taken the course up to this point, then you know how to write a loop and output/log something for each iteration. You also know how to check for a condition with "if/else/else if". You also know how to get a remainder of a number using the modulus operator (%). This is all you need to know to complete this challenge. Good luck!

► Click For Solution

## Solution 1: For Loop

```
for (let i = 1; i <= 100; i++) {  
    if (i % 15 === 0) {  
        console.log("FizzBuzz");  
    } else if (i % 3 === 0) {  
        console.log("Fizz");  
    } else if (i % 5 === 0) {  
        console.log("Buzz");  
    } else {  
        console.log(i);  
    }  
}
```

In the above code, we set our initialize expression to **1**. We set the condition to **i <= 100**. We set our increment expression to **i++**.

We first checked if **i** was divisible by **15**. Because this means **i** is divisible by both **3** and **5**. Since that is the case, we printed "**FizzBuzz**". Then we checked to see if **i** was divisible by **3**. If so, we printed "**Fizz**". Then we checked to see if **i** was divisible by **5**. If so, we printed "**Buzz**". If **i** was not divisible by either **3** or **5**, we printed **i** (The current number).

## Solution 2: **While** Loop

```
let j = 1;

while(j <= 100) {
    if (j % 15 === 0) {
        console.log("FizzBuzz");
    } else if (j % 3 === 0) {
        console.log("Fizz");
    } else if (j % 5 === 0) {
        console.log("Buzz");
    } else {
        console.log(j);
    }

    j++;
}
```

We did the same thing here, just with a **while** loop

# For Of Loop

---

The `for of` loop is used to loop through iterable objects, such as arrays and strings as well as some things that we have not talked about yet like maps and sets. They can replace `for` loops in many cases.

## For Of Syntax

```
for (variable of iterable) {  
    // do something  
}
```

Let's look at a simple example that loops over an array.

```
const arr = [1, 2, 3, 4, 5];  
  
for (const number of arr) {  
    console.log(number);  
}  
  
// 1 2 3 4 5
```

This is cleaner than a `for` loop because we simply give the value for each iteration a name and then we can use that name to access the value, rather than creating a condition expression using the array length and then using `i` as the index.

## Iterating Over Strings

You may not think of a string as an iterable, but it is. If you needed to loop over every letter in a string, you could.

```
const greet = 'Hello World';  
  
for (const letter of greet) {  
    console.log(letter);  
}  
  
// H E L L O W O R L D
```

## Iterating Over Maps

We haven't talked about Maps yet, but they are very similar to arrays. I will go over Maps later, but just to show you a quick example:

---

```
const map = new Map();
map.set('name', 'John');
map.set('age', 30);

for (const [key, value] of map) {
  console.log(key, value);
}
```

There are other objects like sets and generators that we can use with `for of` loops, but I will go over that stuff later on. I don't want to overwhelm you with stuff that you won't be using very often. At least right now.

# For In Loop

---

The `for in` loop is used to loop through the properties of an object.

## For In Syntax

```
for (let key in object) {  
    // do something  
}
```

Let's look at the object below. It is a set of key/value pairs for colors.

```
const colorObj = {  
    color1: 'red',  
    color2: 'blue',  
    color3: 'green',  
    color4: 'yellow',  
    color5: 'orange',  
    color6: 'purple'  
};
```

If we want to get a list of all the keys in the object, we can use the for in loop.

```
for (let key in colorObj) {  
    console.log(key);  
}  
// color1 color2 color3 color4 color5 color6
```

If we want to get the values, we can simply use the key like so:

```
for (let key in colorObj) {  
    console.log(colorObj[key]);  
}  
// red blue green yellow orange purple
```

## Using For In With Arrays

There are a few ways to loop through arrays. We could use a regular for or while loop as well as an array method called `forEach()`, which I will go over soon, but we can also use a `for in` loop.

```
const colors = ['red', 'blue', 'green', 'yellow', 'orange', 'purple'];

for (let key in colors) {
  console.log(colors[key]);
}

// red blue green yellow orange purple
```

# forEach

## High Order Array Methods

Now we are going to get into **high order array methods**, which are methods that we can use on arrays that take another function as an argument. These functions that we pass in are called **callback functions**. The callback will run once for every element in the array. This gives us access to each element.

If we create an array and then look at the **prototype chain**, we can see all of the available methods

```
const socials = ['Twitter', 'Facebook', 'LinkedIn', 'Instagram'];

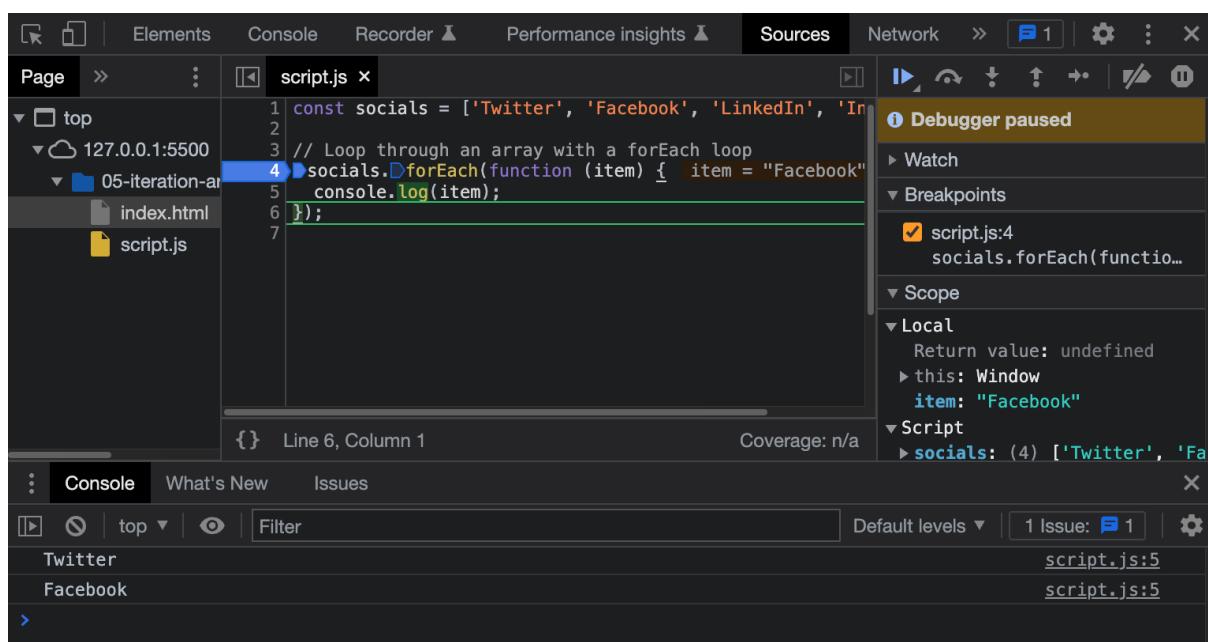
console.log(socials.__proto__);
```

## forEach Method

The `forEach` method is simply a method to loop through an array. It does not return anything, we can just loop through and do whatever we want on each iteration. Let's log all of the social network names.

```
socials.forEach(function(item) {
    console.log(item);
});
```

We learned how to use the browser debugger when we learned about execution context. Let's use it to see each time the callback is run. Place a breakpoint on the same line as the `forEach`. Hit `esc` to bring up the console window so you can see the output. Then hit the step through arrow. You will see it execute and log each item one at a time.



The above function will simply loop through and log the social networks. We don't need to set anything here to be returned, we are just logging each item. The function we pass in will run for every item in the array. The callback function takes in an argument which represents the current item. I called it `item`, but I could have called it anything. I could have called it `social`. It is common to use the singular version of the array name. Then I just logged `item`.

We can use arrow functions to shorten this as well and take away the curly braces, since it is a one liner.

```
socials.forEach((item) => console.log(item));
```

In addition to the item passed into the callback, we can also pass in and get access to the current index (starts at 0) and the entire array itself.

```
socials.forEach((item, index, arr) => {
  console.log(item, index, arr);
});
```

Let's say we want to also console log something if we are on the last iteration of the loop, or the last element in the array

```
socials.forEach((item, index, arr) => {
  if (index === arr.length - 1) {
    console.log('The End');
  }
});
```

You can also use a separate named function as your callback rather than pass in an anonymous one

```
function logSocials(social) {
  console.log(social);
}

socials.forEach(logSocials);
```

Of course, we can use `forEach` on an array of objects. In reality, most of your arrays will probably use objects as items.

```
const socialObjs = [
  { name: 'Twitter', url: 'https://twitter.com' },
  { name: 'Facebook', url: 'https://facebook.com' },
  { name: 'Instagram', url: 'https://instagram.com' },
];
```

```
socialObjs.forEach((item) => console.log(item.url));
```

# Filter() Method

`filter()` is a useful method that filters an array based on a provided function. Unlike `forEach`, it returns a value. That value is an array of items that pass a truth test.

Let's look at a simple example:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const evenNumbers = numbers.filter((number) => number % 2 === 0);

console.log(evenNumbers); // [2, 4, 6, 8, 10]
```

We get back an array of all the even numbers in the original array because the `filter()` method returns all the values that pass the truth test of `number % 2 === 0`.

Again, if the shorthand syntax is messing you up, it is the same as:

```
const evenNumbers2 = numbers.filter(function (number) {
  return number % 2 === 0;
});
```

## Same thing with forEach

We can get the same result with a longer block of code using a `forEach`

```
const evenNumbers = [];
numbers.forEach((number) => {
  if (number % 2 === 0) {
    evenNumbers.push(number);
  }
});

console.log(evenNumbers); // [2, 4, 6, 8, 10]
```

Let's use an array of companies to work with.

```
const companies = [
  { name: 'Company One', category: 'Finance', start: 1981, end: 2004 },
  { name: 'Company Two', category: 'Retail', start: 1992, end: 2008 },
  { name: 'Company Three', category: 'Auto', start: 1999, end: 2007 },
  { name: 'Company Four', category: 'Retail', start: 1989, end: 2010 },
```

```
{ name: 'Company Five', category: 'Technology', start: 2009, end: 2014 },
{ name: 'Company Six', category: 'Finance', start: 1987, end: 2010 },
{ name: 'Company Seven', category: 'Auto', start: 1986, end: 1996 },
{ name: 'Company Eight', category: 'Technology', start: 2011, end: 2016 },
{ name: 'Company Nine', category: 'Retail', start: 1981, end: 1989 },
];
```

Get only the companies that are in the **Retail** category:

```
const retailCompanies = companies.filter(
  (company) => company.category === 'Retail'
);
```

Get companies that started in or after 1980 and ended in or before 2005

```
const earlyCompanies = companies.filter(
  (company) => company.start >= 1980 && company.end <= 2005
);
```

Get companies that lasted 10 years or more

```
const longCompanies = companies.filter(
  (company) => company.end - company.start >= 10
);
```

# map() Method

The `map()` creates a new array populated with the results of calling a provided function on every element in the array.

Let's look at a simple example where we have an array, and we want to create a new array with each number multiplied by 2.

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

const doubledNumbers = numbers.map((number) => number * 2);

console.log(doubledNumbers); // [2, 4, 6, 8, 10]
```

Since the `map()` method returns a new array, we can assign the result to a variable and use it later.

## Using forEach()

Let's do the same thing using the `forEach()` method.

```
const doubledNumbers2 = [];

numbers.forEach((number) => {
  doubledNumbers2.push(number * 2);
});

console.log(doubledNumbers2); // [2, 4, 6, 8, 10]
```

You can see `map()` is cleaner and more concise than `forEach()`.

## Using map() with an array of objects

We can use `map()` to transform an array of objects.

```
const companies = [
  { name: 'Company One', category: 'Finance', start: 1981, end: 2004 },
  { name: 'Company Two', category: 'Retail', start: 1992, end: 2008 },
  { name: 'Company Three', category: 'Auto', start: 1999, end: 2007 },
  { name: 'Company Four', category: 'Retail', start: 1989, end: 2010 },
  { name: 'Company Five', category: 'Technology', start: 2009, end: 2014 },
  { name: 'Company Six', category: 'Finance', start: 1987, end: 2010 },
  { name: 'Company Seven', category: 'Auto', start: 1986, end: 1996 },
  { name: 'Company Eight', category: 'Technology', start: 2011, end: 2016 },
```

```
{ name: 'Company Nine', category: 'Retail', start: 1981, end: 1989 },  
];
```

Let's create an array of company names:

```
const companyNames = companies.map((company) => company.name);  
  
console.log(companyNames);  
// ['Company One', 'Company Two', 'Company Three', 'Company Four', 'Company  
Five', 'Company Six', 'Company Seven', 'Company Eight', 'Company Nine']
```

Let's create an array of new objects with just the name and category properties:

```
const companyInfo = companies.map((company) => {  
  return {  
    name: company.name,  
    category: company.category,  
  };  
});
```

Let's create an array of objects with the name and the length of each company in years:

```
const companyYears = companies.map((company) => {  
  return {  
    name: company.name,  
    length: company.end - company.start + ' years',  
  };  
});
```

## Chaining map Methods

We can chain methods together to create a more complex result.

```
const squareAndDouble = numbers  
.map((number) => Math.sqrt(number))  
.map((number) => number * 2);  
  
console.log(squareAndDouble);  
// [2, 2.8284271247461903, 3.4641016151377544, 4, 4.47213595499958]
```

If the shorthand syntax is confusing you, this is the same as:

```
const squareAndDouble2 = numbers
  .map(function (number) {
    return Math.sqrt(number);
  })
  .map(function (number) {
    return number * 2;
  });

console.log(squareAndDouble2);
// [2, 2.8284271247461903, 3.4641016151377544, 4, 4.47213595499958]
```

## Chaining With Other Methods

We can chain different methods together to create more complex functionality. Let's say that we want to filter the even numbers and then double them, we can chain `map()` and `filter()` together:

```
const evenDouble = numbers
  .filter((number) => number % 2 === 0)
  .map((number) => number * 2);

console.log(evenDouble); // [4, 8, 12, 16, 20]
```

# reduce() Method

The `reduce()` method can be a bit tricky, but it can be very useful. It takes an array and reduces it to a single value. A common use case is to get a total of all the values in an array. This would be useful in a shopping cart application.

## reduce() Syntax

```
reduce(  
  function (previousValue, currentValue, currentIndex) {  
    /* ... */  
  },  
  [initialValue]  
);
```

The callback function that is passed into the `reduce()` method takes three arguments:

- `previousValue` or `accumulator` - The value returned from the last time the callback function was called. This is sometimes called the "accumulator".
- `currentValue` - The value of the current element being processed in the array.
- `currentIndex` - The index of the current element being processed in the array.

You can also pass in an `initialValue` after the callback function. This is the value to use as the first argument to the first call of the callback function. If no initial value is supplied, the first element in the array will be used.

## Simple Example

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
  
const sum = numbers.reduce(function (accumulator, currentValue) {  
  return accumulator + currentValue;  
}, 0);  
  
console.log(sum); // 55
```

We could shorten it up to:

```
const sum2 = numbers.reduce((acc, curr) => acc + curr, 0);  
  
console.log(sum2); // 55
```

If I made the initial `previousValue/accumulator` 10, we would get 65. 10 would be the starting point

```
const sum2 = numbers.reduce((acc, curr) => acc + curr, 10);

console.log(sum2); // 65
```

If we were to do the same thing using a `for of` loop, it would look like this:

```
const sum3 = () => {
  let acc = 0;
  for (const curr of numbers) {
    acc += curr;
  }
  return acc;
};

console.log(sum3()); // 55
```

## Using reduce with objects

You will most likely be working with objects, so let's look at an example of products where we want to get a sum of all of the `price` properties.

```
const cart = [
  { name: 'Product 1', price: 130 },
  { name: 'Product 2', price: 150 },
  { name: 'Product 3', price: 175 },
];

const total = cart.reduce(function (accumulator, product) {
  return accumulator + product.price;
}, 0);

console.log(total); // 455
```

# Array Method Challenges

---

## Challenge 1

### Instructions:

Take the `people` array and create an array called `youngPeople` that stores objects with ONLY `name` and `email` properties of all the people that are 25 and under. The `name` property should have their first and last name.

```
const people = [
  {
    firstName: 'John',
    lastName: 'Doe',
    email: 'john@gmail.com',
    phone: '111-111-1111',
    age: 30,
  },
  {
    firstName: 'Jane',
    lastName: 'Poe',
    email: 'jane@gmail.com',
    phone: '222-222-2222',
    age: 25,
  },
  {
    firstName: 'Bob',
    lastName: 'Foe',
    email: 'bob@gmail.com',
    phone: '333-333-3333',
    age: 45,
  },
  {
    firstName: 'Sara',
    lastName: 'Soe',
    email: 'Sara@gmail.com',
    phone: '444-444-4444',
    age: 19,
  },
  {
    firstName: 'Jose',
    lastName: 'Koe',
    email: 'jose@gmail.com',
    phone: '555-555-5555',
    age: 23,
  },
];
```

### Expected Result:

```
console.log(youngPeople);

[
  {name: 'Jane Poe', email:'jane@gmail.com'},
  {name: 'Sara Soe', email:'sara@gmail.com'},
  {name: 'Jose Koe', email:'jose@gmail.com'}
]
```

► Click For Solution

```
const youngPeople = people
  .filter((person) => person.age <= 25)
  .map((person) => ({
    name: person.firstName + ' ' + person.lastName,
    email: person.email,
  }));

```

## Challenge 2

### Instructions:

Add all of the positive numbers in the array.

### Expected Result:

```
const numbers = [2, -30, 50, 20, -12, -9, 7];

console.log(positiveSum); // 79
```

► Click For Solution

```
const numbers = [2, -30, 50, 20, -12, -9, 7];

const positiveSum = numbers
  .filter((number) => number > 0)
  .reduce((acc, cur) => acc + cur, 0);

console.log(positiveSum);
```

## Challenge 3

### Instructions:

Create a new array called `capitalizedWords` with the words from the `words` array with the first letter of each word capitalized.

### Expected Result:

```
const words = ['coder', 'programmer', 'developer'];
console.log(capitalizedWords); // [ 'Coder', 'Programmer', 'Developer' ]
```

### Hint:

Remember back a few sections, we had a challenge to capitalize the first letter of a string. You are doing the same thing here, but for each word in the array.

► Click For Solution

```
const capitalizedWords = words.map(
  (word) => word[0].toUpperCase() + word.slice(1, word.length)
);
```

# Intro To The DOM

---

Alright, so now we get to get to the fun stuff, working in the actual browser body and not just in the console. I think one of the big mistakes people make is they jump into the browser too quickly without understanding the stuff that we have talked about so far. Now that you understand control flow, functions, array methods, etc, you will be able to do more within the browser.

## The DOM

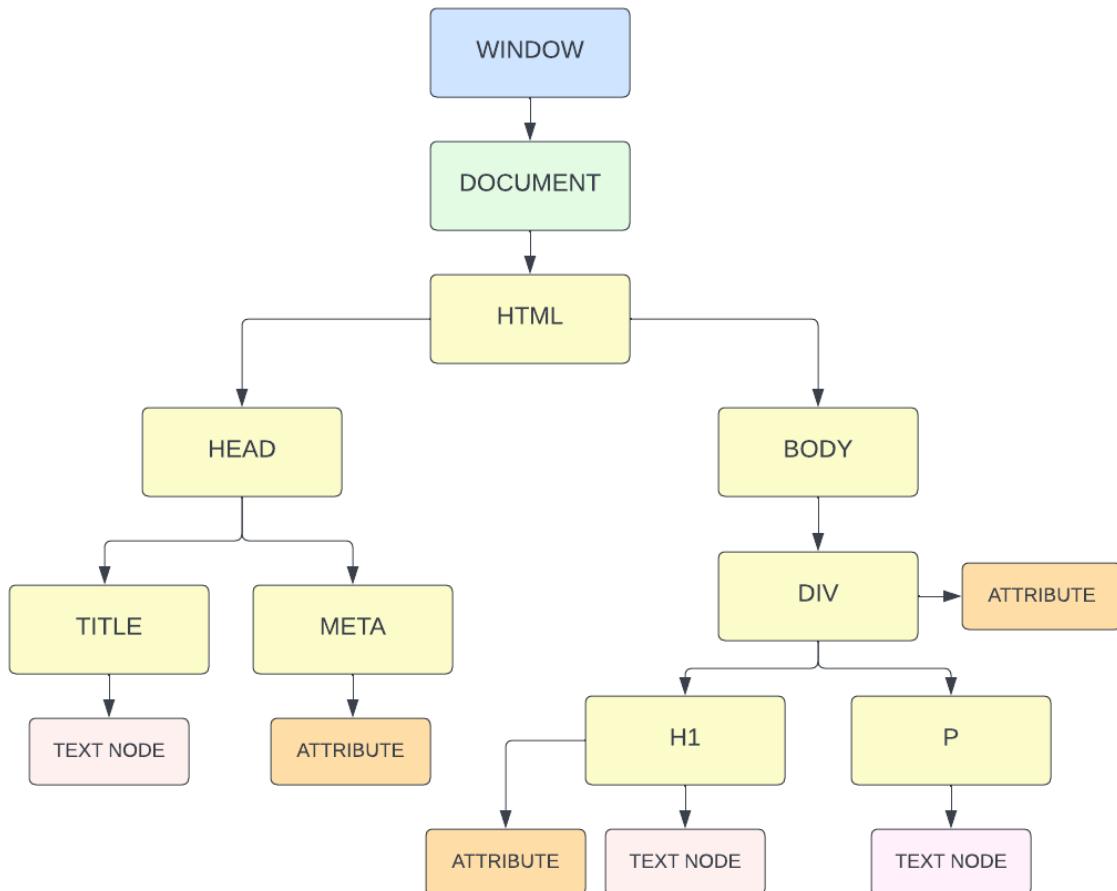
The DOM stands for **Document Object Model**. It is essentially a programming interface for web/HTML documents. We already know that we display the content on web pages using HTML tags with text and we style it using CSS. The DOM is the structure of the web page that we can interact with using JavaScript. It generally includes all of the HTML tags, attributes and the text between the tags called "text nodes". The DOM is usually represented as a tree structure.

Let's look at this very simple page structure:

```
● ○ ●

<html>
  <head>
    <meta charset="UTF-8" />
    <title>Website</title>
  </head>
  <body>
    <div id="main">
      <h1 class="text-lg">
        Hello World
      </h1>
      <p>This is my web page</p>
    </div>
  </body>
</html>
```

The DOM representation of this would look something like this:



I'll talk about the `window` and `document` objects in a minute, but under that you'll see the structure of this document in a tree structure.

We have the `html` element, which holds the `head` and `body` elements. The `head` element holds the `title` element, which has a text node inside of it. In this case that would be the text, 'Website'. The `meta` element includes an attribute of `charset` which is set to `utf-8`. So any attributes are going to be accessible in the DOM as well. The `body` element holds the `h1` element, which has a text node inside of it with the text, 'Hello World'. It also has an `id` attribute. Then we have the `p` element with a text node inside of it.

So this is how your HTML document full of tags, attributes and text nodes looks in the DOM.

## The Window Object

A few sections back, we talked about the global `window` object. The `window` object is the top level object in the browser. It is the root of the browser's object hierarchy. It has properties and methods that are available to all of the JavaScript code in the browser. To see the window object's API (methods and properties), you can type `window` in the console or in your code

```
console.log(window);
```

## The Document Object

---

On that `window` object, we have a property/object called `document`. This is an object that represents the HTML document that we are currently viewing. It has all types of properties and methods that we can use to interact with the HTML document.

```
console.log(window.document);
```

Since `window` is the top-level object in the browser, we do not need to prefix it with `window`

```
console.log(document);
```

Sometimes `console.log()` will show the properties and methods, but sometimes it will show the element/tag itself. To see all the methods and properties, you can use `console.dir`

```
console.dir(document);
```

## Document Properties/Elements

We can access DOM elements directly, such as the `body`

```
console.log(document.body);
```

We can also access elements such as links and images. Let's add a link to the HTML

```
<a href="https://traversymedia.com">Traversy Media</a>
```

We can get all of the links with

```
console.log(document.links);
```

This returns something called an `HTMLCollection`, which is similar to an array. We can access by index

```
console.log(document.links[0]);
```

## Document methods

In addition to properties, the document object has a ton of methods. `document.write()` is a method that will write to the browser body. This is not used very much because you can't really target where you want to output, but it is available

```
document.write('Hello From JS');
```

## Document Selection Methods

Getting elements using direct properties is not usually the way you want to do this. There are special methods available that allow us to directly select elements easily. I will be going over all of these in depth soon, but just to give you an example, let's look at `document.getElementById()`, which does exactly that. It allows us to basically grab an element from the HTML document by its id. Then we could do whatever we want to that specific element including changing the style, removing it, adding an event listener to it and so on. The term **DOM manipulation** just refers to changing the DOM using JavaScript, which again can me adding, removing, changing elements, etc. A lot of the time we change the CSS through JavaScript to make something hide or show based on some user interaction.

Just to give you a very simple example of selecting something from the DOM, let's open an html page and create a new div element with the ID of `main` and just add some text to it.

```
<div id="main">
  <h1>The DOM</h1>

  <a href="https://traversymedia.com">Traversy Media</a>
</div>
```

Now in our JavaScript file, we can use the `document.getElementById()` method to grab the element with the ID of `main`.

```
console.log(document.getElementById('main'));
```

Now let's say we want to add some text into the div element. We can easily do that using the `innerHTML` property.

```
document.getElementById('main').innerHTML = '<h1>Hello World</h1>';
```

This will replace everything in the main element. There are other methods we will get into soon that let us just append or add elements.

So you see, The DOM gives us a lot of power to interact with the HTML document. The code we just wrote is written in JavaScript, but uses the DOM and its API to access the document and its elements. Even when

you start using a front-end framework like React, this is the stuff that it is doing under the hood.

The DOM was created to be independent of any particular language. Yes, JavaScript is definitely the most common language to do this stuff with, but implementations of the DOM can be built for any language, as this Python example demonstrates:

```
# Python DOM example
import xml.dom.minidom as m
doc = m.parse(r"C:\Projects\Py\chap1.xml")
doc.nodeName # DOM property of document object
p_list = doc.getElementsByTagName("para")
```

## DOM Libraries

I've been doing this for about 15 years now and I know people still complain about JavaScript, but I think very few of them, worked with JavaScript ten+ years ago. Before the big ES6 update, JavaScript was not easy to work with, including accessing and manipulating the DOM. So we had libraries like **jQuery** that were built to make it easier to work with the DOM.

A lot of people ask me if they should learn jQuery and my answer to that now is usually no. If you're genuinely interested in learning it, that's fine. Just make sure you learn the fundamentals of vanilla JavaScript first. You may want to learn jQuery to deal with legacy code. There is a chance you'll run into it in the wild. But do you need to learn it? In my opinion, no you don't.

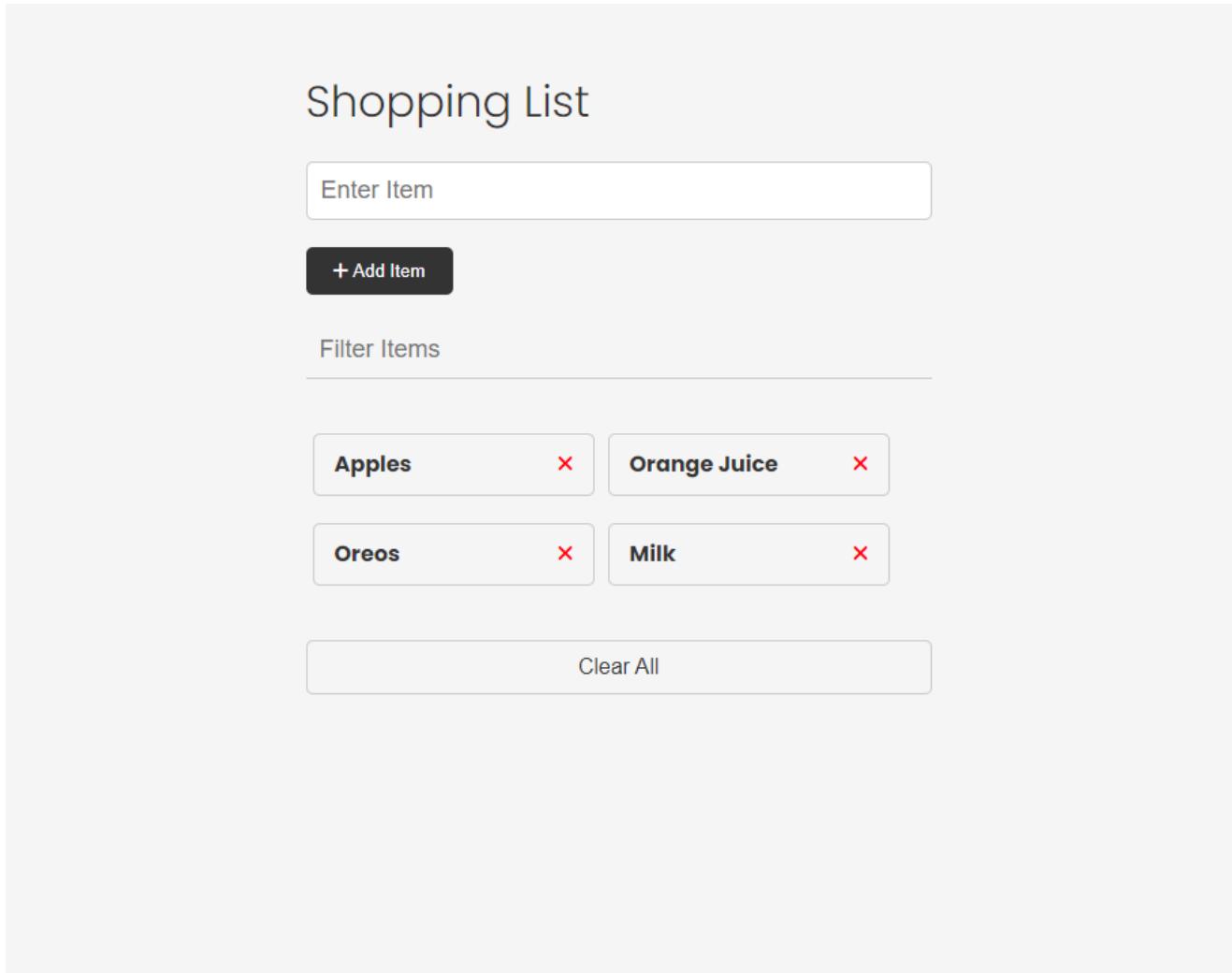
Alright, so in the next video, I want to examine some of the properties on the browser's `document` object itself.

# Examining the Document Object

---

Alright, so we know that the DOM is a tree of objects of elements, nodes and attributes. We also know that the `document` object is the root of the DOM tree. It offers extremely helpful methods for us to use to manipulate the DOM including methods to select elements, create elements, and so on. There's also a ton of properties that we can use to get information about the document and that's what I want to look at in this video.

## Shopping List UI



So what you're looking at right now is the UI or user interface for a simple shopping list application. We will be building this project out, but right now, it's only the HTML and CSS, so it's not actually working because there's no JavaScript. There is a JavaScript file linked, but it's empty. That's what we'll be working in to explore the DOM.

You can download the code resources directly from this section, or from the main sandbox GitHub repository.

Let's quickly just look at the structure of this HTML page, so we know what we're working with.

So we have a container that wraps around everything, a heading, then we have a form that has one single text input and a submit button. Under that, we have a filter input to filter the items. Then we have a list of items with an x icon to remove those items. As I mentioned, the functionality of this will not work because we have no JavaScript. Finally, we have a clear all button that will do just that.

So, it's a pretty simple UI, but it's a good starting point for us to explore the DOM.

## Browser Dev Tools

The browser dev tools will be used extensively throughout this course. We've been using the console quite a bit, but I want to check out the **Elements** tab real quick. This is where you can see all of your elements/tags, attributes, text, etc. In reality, you should know some HTML and CSS and you should have at least a little experience with this tab.

We can see the CSS styling here as well. In many cases, we'll be changing styles and adding and removing classes through JavaScript and we'll be using the dev tools to see what's happening.

## The Document Object

So if we type into the console `window`, we see everything that is available to use in the global object and `document` is one of them. You can see that there's a ton of properties and methods available to us. Anything you see here can be used in your JavaScript code.

Let's start off by just logging the document.

```
console.log(document);
```

```
script.js:1
```

```
▼#document
  <!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">
      <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.1.2/css/all.min.css" integrity="sha512-1sCRPdkRXhBV2PBLUDRb4tMg1w2YPf37qatUFeS7zlBy7jJI8Lf4VHwwfZfpXtYSLy85pkm9GaYVYMfw5BC1A==" crossorigin="anonymous" referrerPolicy="no-referrer">
      <link rel="stylesheet" href="style.css">
      <title>Shopping List</title>
    </head>
    ▶<body>...</body>
  </html>
```

It gives us just that, the entire document including the doctype, html tag and head and body tags with everything in between them.

`document.all`

```
console.log(document.all);
```

```
script.js:5
```

```
HTMLAllCollection(34) [html, head, meta, meta, meta, link, link, title, body, div.container, h1, form#item-form, div.form-control, input#item-input.form-input, div.form-control, button.btn, i.fa-solid.fa-plus, div.filter-tasks, input#filter.form-input-filter, ul#item-list.items, li, button.remove-item.btn-link.text-red, i.fa-solid.fa-xmark, li, ▶button.remove-item.btn-link.text-red, i.fa-solid.fa-xmark, li, button.remove-item.btn-link.text-red, i.fa-solid.fa-xmark, li, button.remove-item.btn-link.text-red, i.fa-solid.fa-xmark, button#clear.btn-clear, script, viewport: meta, item-form: form#item-form, item-input: input#item-input.form-input, filter: input#filter.form-input-filter, item-list: ul#item-list.items, ...]
```

`document.all` gives us everything in the DOM in an array-like object called a `HTMLCollection`. It's not an array, but it is structured like one. We can't use methods like `forEach()` and `map()` on it, but we can

use bracket notation to access the items. We can also use the `length` property to get the number of items in the collection. There's also a method called `item()` that we can use to get a specific item in the collection.

```
console.log(document.all.length);
console.log(document.all[2]);
console.log(document.item(2));
```

`document.all()` is not a very useful method and it's actually deprecated, so we shouldn't use it. It's better to use methods like `document.getElementById()`. I did want to show you that it exists and give you an example of an `HTMLCollection`.

### `document.documentElement`

```
console.log(document.documentElement);
```

This will get you the HTML element and everything in it including both the head and body tags.

### `document.head & document.body`

```
console.log(document.head);
console.log(document.body);
```

These two properties give us the head and body elements. We can get the children of those elements using the `children` property.

```
console.log(document.head.children);
console.log(document.body.children);
```

These will give us an `HTMLCollection` of the direct children of the head and body elements. So in the case of the body, we only get the container div and the script tag because those are the only direct children of the body element.

▶ `HTMLCollection(2) [div.container, script]`

`script.js:10`

If I wanted to get the children of the container div, I would have to do something like this.

```
console.log(document.body.children[0].children);
```

```
script.js:29
```

```
HTMLCollection(5) [h1, form#item-form, div.filter-tasks,  
ul#item-list.items, button#clear.btn-clear, item-form:  
form#item-form, item-list: ul#item-list.items, clear:  
button#clear.btn-clear] i  
▶ 0: h1  
▶ 1: form#item-form  
▶ 2: div.filter-tasks  
▶ 3: ul#item-list.items  
▶ 4: button#clear.btn-clear  
▶ clear: button#clear.btn-clear  
▶ item-form: form#item-form  
▶ item-list: ul#item-list.items  
length: 5  
▶ [[Prototype]]: HTMLCollection
```

So now I have a collection of all of the children of the container div. This is a little bit cumbersome and there are better ways to select elements, but let's say I wanted to change the text of the heading. I could do something like this.

```
document.body.children[0].children[0].textContent = 'Hello World';
```

Hello World

Enter Item

+ Add Item

Filter Items

Apples

X

Orange Juice

X

Oreos

X

Milk

X

Clear All

## Other Page Properties

Here are some other properties on the `document` object that we can access and manipulate.

```
document.doctype;  
document.domain;  
document.URL;  
document.characterSet;  
document.contentType;
```

### `document.forms`

`document.forms`. This will give us an `HTMLCollection` of all of the forms on the page.

```
console.log(document.forms);
```

We can get specific forms in the collection by index or by name.

```
console.log(document.forms[0]);  
console.log(document.forms['item-form']);
```

We can get specific attributes as well

```
console.log(document.forms[0].id);  
console.log(document.forms[0].method);  
console.log(document.forms[0].action);
```

We aren't only limited to reading the attributes. We can also change them.

```
document.forms[0].id = 'new-id';
```

I could add a value to the input like this:

```
document.forms[0].item.value = 'Hello World';
```

`item` being the `name` attribute of the input.

### `document.links`

To get an `HTMLCollection` of links, we can use `document.links`.

```
console.log(document.links);
```

We don't have any links in this specific project, so let's add one just to test it out.

```
<a href="#" class="my-link" id="link">My Link</a>
```

Now we can see that we have a link in the collection.

We can get specific attributes as well.

```
console.log(document.links[0].id);
console.log(document.links[0].className);
console.log(document.links[0].classList);
console.log(document.links[0].href);
```

## document.images

We can also get an `HTMLCollection` of images with `document.images`.

```
output = document.images;
output = document.images[0];
output = document.images[0].src;
```

In the code above, we accessed the attributes directly, but we can also use the `getAttribute()` method.

```
output = document.images[0].getAttribute('src');
```

This will do the same thing as accessing the `src` attribute directly.

## document.scripts

If you want to get an `HTMLCollection` of scripts, you can use `document.scripts`.

```
console.log(document.scripts);
```

So as you can see, everything in our HTML document is available to us in the DOM using the `document` object that the browser creates for us.

This is just the tip of the iceberg. There are so many more properties and methods available. We'll be using them throughout the course.

# DOM Selectors - Single Elements

---

In the last video, we looked at many of the properties available on the `document` object including properties that allowed us to select elements from the DOM in the form of `HTMLCollections`.

In this video, we will look at a much more common way to select elements. With these **single element** selectors, it will only select one element. Even if you have multiple elements that match the selector, it will only select the first one. If you want to select multiple elements, we will look at that in the next video.

We will also look at getting and setting attributes/properties on elements that we select.

## `getElementById()`

The first method we will look at is `document.getElementById()`. This method takes a string as an argument and returns the element with the matching `id` attribute. If no element is found, it returns `null`.

We have a couple IDs in our shopping list, such as on the form and the list itself, but I want to use something simple, so I'm going to use the `id` on the `h1` element of `app-title`.

If we want to select this element, we can use `document.getElementById()` and pass in the ID as a string.

```
console.log(document.getElementById('app-title'));
// <h1 id="app-title">Shopping List</h1>
```

This is a much easier way than using something like `document.all` and trying to find the element by index.

Now that we can select the element, we can get certain properties/attributes from it. For example, we can get the `id`.

```
console.log(document.getElementById('app-title').id);
console.log(document.getElementById('app-title').getAttribute('id'));
```

We can also assign attributes to it. For instance, we can change the `className` property to add a class.

```
document.getElementById('app-title').className = 'title';
```

We can also use the `setAttribute()` method to add an attribute.

```
document.getElementById('app-title').setAttribute('class', 'app-title');
```

It's also very common to assign the result of these selector methods to a variable. This is because we will often want to use the element we selected in multiple places in our code.

```
const title = document.getElementById('app-title');
console.log(title);
```

## Styling Elements From JS

We can also use these selectors to style elements. There is a property called `style` that allows us to access the CSS styles of an element. We can then assign a value to a CSS property.

```
title.style.color = 'red';
title.style.backgroundColor = 'black';
title.style.padding = '10px';
title.style.borderRadius = '10px';
```

Now, usually you wouldn't just style things from JavaScript just for a static site. You would usually have some kind of event and then apply the style. For example you may have a button that when you click it, it applies a style to a specific element. This makes your UI more dynamic. We'll be getting into events soon.

## Adding Content

Another thing we can do is add or change the content of an element. Again, usually you'll do this to add some kind of dynamic functionality using events. There are a few different properties that we can use to modify and access content.

```
// Read
console.log(title.textContent);
// Write
title.textContent = 'Shopping List';
title.innerText = 'My Shopping List';
title.innerHTML = '<strong>Shopping List</strong>';
```

`textContent` and `innerText` are very similar. The main difference is that `innerText` will not return any hidden elements. For example, if we had a `span` element with a class of `hide` and we set the `display` property to `none`, then `innerText` would not return that element. `textContent` will return the element even if it is hidden. `innerHTML` will return the HTML of the element. We can use all of these to update the content of an element.

## querySelector()

Alright, so the next method we will look at is `document.querySelector()`. This method is very similar to `getElementById()`, but it allows us to select elements using CSS selectors. This means we can select elements by their class, tag name, or even by their attributes.

```
// By tag
console.log(document.querySelector('h1'));
// By id
console.log(document.querySelector('#app-title'));
// By class
console.log(document.querySelector('.container'));
// By attribute
console.log(document.querySelector('input[type="text"]'));
// By pseudo-class
console.log(document.querySelector('li:first-child'));
console.log(document.querySelector('li:nth-child(2)'));
```

As you can see, `querySelector` is very powerful and we can select anything we want. This method put JQuery out of business in my opinion, because it pretty much does the same thing in terms of DOM manipulation, without having to use a library.

Let's say that we want to change the color of the second shopping list item:

```
const secondItem = document.querySelector('li:nth-child(2)');
secondItem.style.color = 'red';
```

Everything we talked about with `getElementById()` also applies to `querySelector()`. We can get the content, set the content, style the element, etc.

## Selecting an element within an element

With `document.querySelector()` and `document.querySelectorAll()`, which we will look at in the next video, you are not restricted to selecting elements from the `document` object. You can also select elements from other local elements. For example, if we wanted to select the first `li` element within the `ul` element, we could do the following:

```
const list = document.querySelector('ul');
const firstItem = list.querySelector('li');
firstItem.style.color = 'blue';
```

`document.getElementById()` is only available as a method of the global `document` object. This is because ID values must be unique throughout the entire document, so there is no need for "local" versions of the function.

## Which one to use?

Which one you use is really just preference. I use `getElementById()` quite a bit for `ids` and `querySelector()` for classes and other attributes. You could just as well use `querySelector()` for

everything. I have been doing that in a few recent projects, and I may switch it up. It is just important to be consistent.

# DOM Selectors - Multiple Elements

---

So we looked at `getElementById` and `querySelector`, which will select one element from the DOM so that we can do whatever we want with it. But what if we want to select multiple elements, such as all of the list items in our shopping list? We have a few methods that we can use for this.

## `querySelectorAll()`

So I'm going to start off with the one that you'll use the most, which is `document.querySelectorAll()`. This method takes a string as an argument and returns a NodeList of all the elements that match the selector. If no elements are found, it returns an empty NodeList.

### **NodeList vs HTMLCollection**

So what is a NodeList? Well, it's very similar to an HTMLCollection, which we have already looked at. It's a list of elements that is formatted like an array. The main difference is that a NodeList is not a live collection. This means that if we add or remove elements from the DOM, the NodeList will not update.

Another difference is that we can use methods like `forEach()` on a NodeList, but not on an HTMLCollection. This really comes in handy.

So let's select all of the list items in our shopping list.

```
console.log(document.querySelectorAll('li'));
```

We can pass in absolutely any CSS selector to this method. It could be a tag, id, class, etc. That part works just like `querySelector()`. The difference is instead of getting just one element, it will get all of the elements on the page that match that selector. Even if there is only one item on the page that matches, it will still be returned as a NodeList.

If we select the h1 on the page, let's see what happens.

```
console.log(document.querySelectorAll('h1'));
```

`script.js:5`

▶ `NodeList [h1#app-title]`

Instead of the element itself, it gives us a NodeList with that 1 element in it.

Now, let's say that we want to make all of the items in the list the color red. Your first thought, may be to do something like this.

```
const items = document.querySelectorAll('li');

// This will not work
items.style.color = 'red';
```

This won't work because we can't assign a color to a NodeList.

We can access a specific item using its index, just like an array. We can also manipulate it.

```
console.log(items[0]);
items[0].style.color = 'blue';
```

However, if we want to apply something to all of the elements, we need to loop through the NodeList and apply to all.

```
items.forEach(function (item) {
  item.style.color = 'red';
});
```

We can shorten it up a little bit by using an arrow function.

```
items.forEach((item) => (item.style.color = 'red'));
```

## getElementsByClassName()

The next selector is an older one that you probably won't see very much. As the name implies, it selects elements by their class name. It takes a string as an argument and returns an HTMLCollection of all the elements that match the class name. If no elements are found, it returns an empty HTMLCollection.

The limitations to this method are that it only selects elements by class name and instead of a NodeList, it returns an HTMLCollection, which you can't use methods like `forEach()` on. You would have to first, turn the HTMLCollection into an Array and then loop through it.

Let's look at an example. I put a class of `item` on all of the list items in our shopping list. Let's select them all.

```
console.log(document.getElementsByClassName('item'));
```

```
script.js:23
▶ HTMLCollection(4) [li.item,
  li.item, li.item]
```

As you can see, it returns an `HTMLCollection` of all of the list items as opposed to a `NodeList`.

If we want to turn all of the items blue, we have to first turn the `HTMLCollection` into an array. Then we can use array methods like `forEach()`.

```
const items = document.getElementsByClassName('item');

// Convert into an array
const list = Array.from(items);

list.forEach((item) => (item.style.color = 'green'));
```

## getElementsByTagName()

This method is basically the same thing as `getElementsByClassName()`, except it selects elements by their tag name. It takes a string as an argument and returns an `HTMLCollection` of all the elements that match the tag name. If no elements are found, it returns an empty `HTMLCollection`.

Let's select all of the list items in our shopping list.

```
console.log(document.getElementsByTagName('li'));
```

```
script.js:23
▶ HTMLCollection(4) [li.item,
  li.item, li.item]
```

We get the same result as `getElementsByClassName('item')`.

There really isn't much else to be said on this method, because you can do the same stuff as `getElementsByClassName()`.

I would say 99.9% of the time, you will be using `querySelectorAll()` to select multiple elements. The only time you would use `getElementsByClassName()` or `getElementsByTagName()` is if you are working with an older codebase that uses those methods.

# Traversing The DOM - Elements

---

Now we know how to select elements from the DOM, but many times, we need to select elements within elements or a sibling of an element, so it is important to know how relationships work so that we can traverse or move up and down the DOM.

There are also different types of **nodes** in the DOM. The type we will deal with the most are **Elements**. Any HTML tag on the page is an element. There are specific properties for working with elements and there are properties that will allow us to work with other types of nodes, such as **Text** and **Comment** nodes. In this video, we will strictly be working with **Elements** or **Element Nodes** and in the next video, we will look at other types of nodes.

## Element Relationships

To really understand how to traverse and manipulate the DOM, you need to understand relationships between elements along with some properties that allow you to select elements based on their relationship to other elements.

For this video, I'm not going to use the shopping list example. Instead, I'm going to use a simple HTML document with a few nested elements. This is so you can better understand what I'm doing.

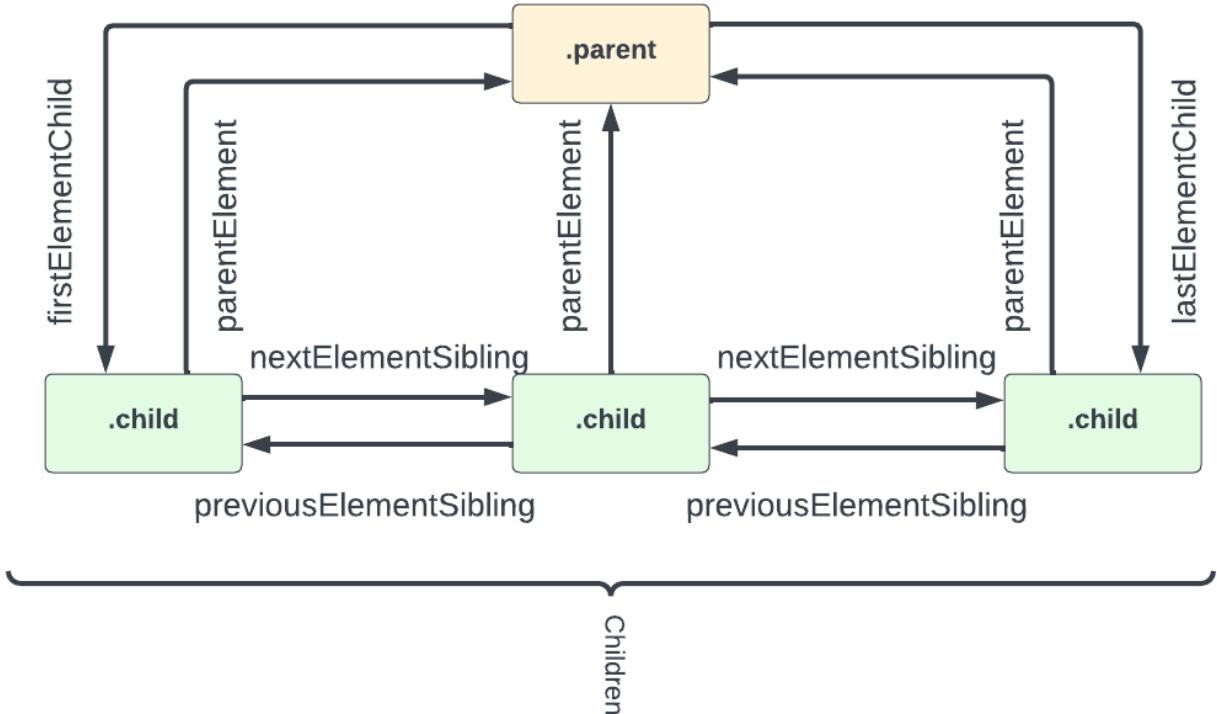
Let's look at the following HTML

```
<div class="parent">
  <!-- Children -->
  <div class="child">Child 1</div>
  <div class="child">Child 2</div>
  <div class="child">Child 3</div>
</div>
```

If we are looking strictly at **element nodes**, it's pretty simple. We have a **div** that is the *parent* of 3 **child divs**. The 3 **child divs** are *siblings* to each other.

## Selecting Related Elements

We have many properties available to select related elements. This diagram shows the properties and a visual example of how they work.



## Child Elements

### children

All of an elements children can be accessed by the `children` property.

```

const parent = document.querySelector('.parent');

console.log(parent.children);
// HTMLCollection [div.child, div.child, div.child]

```

This gives us an `HTMLCollection` of the children. We can access each individual element by index.

```

console.log(parent.children[1]);
// <div class="child">Child 2</div>

// Get individual child properties
console.log(parent.children[1].nodeName);
// 'DIV'
console.log(parent.children[1].className);
// 'child'

// Get the HTML & Text inside the element
console.log(parent.children[1].innerHTML);
// 'Child 2'

// Get the HTML & Text of the entire element

```

```
console.log(parent.children[1].outerHTML);
// '<div class="child">Child 2</div>'
```

We can loop through an `HTMLCollection` using a `for` loop or we can convert it to an array and use a `forEach` loop.

```
for (let i = 0; i < parent.children.length; i++) {
  console.log(parent.children[i].innerHTML);
}
```

We can change attributes, styles and the content of the children of an element

```
// Change the text of the second child
parent.children[1].textContent = 'Child Two';

// Change the color of the third child
parent.children[2].style.color = 'red';
```

### `firstElementChild` and `lastElementChild`

`firstElementChild` and `lastElementChild` are useful for selecting the first or last child of an element.

```
console.log(parent.firstElementChild);
// <div class="child">Child 1</div>

console.log(parent.lastElementChild);
// <div class="child">Child 3</div>

// Change the text of both first and last child
parent.firstElementChild.innerText = 'Child One';
parent.lastElementChild.innerText = 'Child Three';
```

## Parent Element

### `parentElement`

We can use the `parentElement` property to get the parent element of an element.

```
const child = document.querySelector('.child');

console.log(child.parentElement);
// <div class="parent">
```

We can edit the parent element

```
// Give the parent a border and padding  
child.parentElement.style.border = '1px solid #ccc';  
child.parentElement.style.padding = '10px';
```

## Sibling Elements

`nextElementSibling` and `previousElementSibling`

These are the main properties for selecting sibling elements.

```
const secondItem = document.querySelector('.child:nth-child(2)');  
  
secondItem.nextElementSibling.style.color = 'green';  
secondItem.previousElementSibling.style.color = 'orange';
```

As you can see, working with related elements is not that difficult because we have access to these useful properties. In the next video, we will look at properties to work with other types of nodes.

# Traversing The DOM - Nodes

---

Alright, so we looked at some important properties of the DOM that allow us to work with related elements. However, the DOM has more than just element nodes. In fact, there are 12 different types of nodes.

Type	Description	Children
1 Element	Represents an element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
2 Attr	Represents an attribute	Text, EntityReference
3 Text	Represents textual content in an element or attribute	None
4 CDATASection	Represents a CDATA section in a document (text that will NOT be parsed by a parser)	None
5 EntityReference	Represents an entity reference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
6 Entity	Represents an entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
7 ProcessingInstruction	Represents a processing instruction	None
8 Comment	Represents a comment	None
9 Document	Represents the entire document (the root-node of the DOM tree)	Element, ProcessingInstruction, Comment, DocumentType
10 DocumentType	Provides an interface to the entities defined for the document	None
11 DocumentFragment	Represents a "lightweight" Document object, which can hold a portion of a document	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
12 Notation	Represents a notation declared in the DTD	None

Table from W3Schools.com

We don't really need to focus on most of these, but they're good to know.

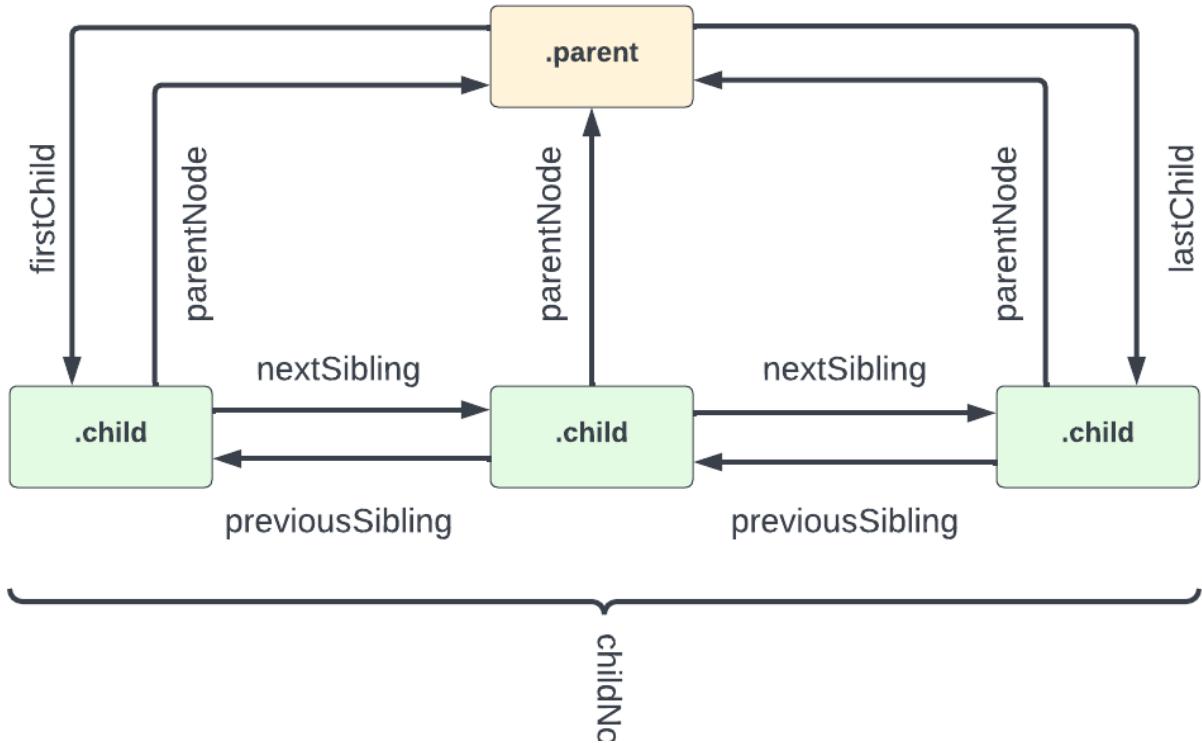
## Properties That Work With Nodes

The properties that we looked at in the last video were to work with related elements. However, there are other properties that we can use to access other types of nodes.

I would say for the most part, you will use the properties that we looked at in the last video, but you should know these as well.

We are going to use the same HTML structure as in the last video:

```
<div class="parent">
  <!-- Children -->
  <div class="child">Child 1</div>
  <div class="child">Child 2</div>
  <div class="child">Child 3</div>
</div>
```



## Child Nodes

### childNodes

This method, does exactly what it says, it gets all of the child nodes of an element. This includes **text nodes** and **comment nodes**.

```
const parent = document.querySelector('.parent');

console.log(parent.childNodes);
```

```
script.js:15
NodeList(9) [text, comment, text,
▼div.child, text, div.child, text,
div.child, text] ⓘ
► 0: text
► 1: comment
► 2: text
► 3: div.child
► 4: text
► 5: div.child
► 6: text
► 7: div.child
► 8: text
length: 9
► [[Prototype]]: NodeList
```

As you can see, we get a NodeList of 9 items. I want to go over what each of these items are...

- 0: White space is a `text node`. Since the first line of HTML within the parent `div` (the comment) is on the next line, that line break is considered a `text node`. If I were to move the comment on to the same line as the parent `div`, then the `text node` would be removed. I know, it's weird.
- 1: The `comment` node. Comments are considered nodes as well.
- 2: Another `text node`. This is the line break between the comment and the first child `div`.
- 3: The first child `div`.
- 4: Another `text node`. This is the line break between the first child `div` and the second child `div`.
- 5: The second child `div`.
- 6: Another `text node`. This is the line break between the second child `div` and the third child `div`.
- 7: The third child `div`.
- 8: Another `text node`. This is the line break between the third child `div` and the closing `div` tag.

We can also access the nodes like this:

```
console.log(parent.childNodes[3]);
// <div class="child">Child 1</div>

// Get the node name
console.log(parent.childNodes[3].nodeName);
// 'DIV'
console.log(parent.childNodes[2].nodeName);
// '#text'

// Edit the first div
parent.childNodes[3].style.color = 'red';
```

The reason I used the index 3 is because 0, 1 and 2 are text and comment nodes. 3 is where the first element node is.

## firstChild and lastChild

Just like we have `firstElementChild` and `lastElementChild`, we have `firstChild` and `lastChild`. the difference is that these two do not work only with element nodes.

```
// Get the first child node
output = parent.firstChild;
// #text

// Get the last child node
output = parent.lastChild;
// #text
```

To edit the value of a text node, we can use the `nodeValue` property.

```
parent.firstChild.nodeValue = 'Hello';
```

This will actually put the text of 'Hello' into the `nodeValue`, `textContent`, `data` and `wholeText` properties.

## Parent Nodes

### parentNode

In many ways, `parentNode` is very similar to `parentElement`. We can use it to do the same type of tasks.

To get the parent node of an element:

```
const child = document.querySelector('.child');

console.log(child.parentNode);
// <div class="parent">
```

The parent of an element will always be and `element`, `document` or `documentFragment` node type.

We looked at `document.documentElement` a few videos back. If we look at the parent node for that, it is a `#document` node

```
console.log(document.documentElement.parentNode);
// #document
```

We can edit the parent node/element, just like we could using the `parentElement` property.

```
child.parentNode.style.border = '1px solid #ccc';
child.parentNode.style.padding = '10px';
```

If you want to get the parent document node of any element, you can use the `ownerDocument` property.

```
console.log(child.ownerDocument);
// #document
```

## Sibling Nodes

`nextSibling` and `previousSibling`

Very similar to `nextElementSibling` and `previousElementSibling`, we have `nextSibling` and `previousSibling`.

```
const secondItem = document.querySelector('.child:nth-child(2)');

// Get next sibling
console.log(secondItem.nextSibling);
// #text

// Get previous sibling
console.log(secondItem.previousSibling);
// #text
```

# Create Elements

---

So we know how to select elements, navigate to related elements, and manipulate them. Now let's look at how to create elements. With JavaScript, we can create any DOM element we want and then insert it into the document.

In many cases, you'll want to create a new element and then insert it into the document on some kind of event. For instance, when we build the final functionality of our shopping list, we're going to want to create a new list item when we click on the *Add Item* button.

We will learn about events soon, but right now, I just want to focus on creating elements via JavaScript.

## `document.createElement()`

This is the main method we'll use to create elements. It takes one argument, which is the tag name of the element we want to create. Let's create a `div` element.

```
const div = document.createElement('div');

console.log(div);
// <div></div>
```

If you do a `console.dir(div)`, you'll see a ton of properties as well as methods in the prototype chain.

We can add any attributes we want to our new `div` element.

```
// Add a class
div.className = 'my-element';

// Add an id
div.id = 'my-element';

// Add an attribute
div.setAttribute('title', 'My Element');
```

## `document.createTextNode()`

If we want to add text, we could technically do it like this:

```
div.innerText = 'Hello World';
```

However, using `innerText` is not the best way to do this when creating a new element. It is really meant to get and change the text of an already existing element.

It's better to create a new text node with `document.createTextNode()` and then append it to the element.

```
// Create a text node
const text = document.createTextNode('Hello World');

// Add the text node to the div
div.appendChild(text);
```

Now if you log the div, you'll see

```
<div class='my-element' id='my-element' title='My Element'>
  Hello World
</div>
```

## Inserting Elements into the Document

Just like we used `appendChild()` to add the text node, we can use it to add it to the document.

```
document.body.appendChild(div);
```

That will place it in the body tag as the last element. We can target any element in the document and insert it into another element.

```
document.querySelector('ul').appendChild(div);
```

# Create Item - innerHTML vs createElement()

---

Now that you know the basics of creating an element within your JavaScript, I want to create a function that we can run to add a new item to the shopping list. There are some really important things I want to show you when it comes to the different ways of doing this. There's a quick and dirty solution, which involves just setting the `innerHTML` to whatever you want and then there's a cleaner and more performant way of creating all of your elements and then inserting them into the DOM. I'm going to show you both ways.

## Quick & Dirty (Using innerHTML)

The first method is to create a `li` element and then simply set the `innerHTML` property to the output that you want, using a template string.

```
function createListItem(item) {
  const li = document.createElement('li');

  li.innerHTML = ` ${item}
<button class="remove-item btn-link text-red">
  <i class="fa-solid fa-xmark"></i>
</button>`;

  document.querySelector('.items').appendChild(li);
}

createListItem('Cereal');
createListItem('Juice');
createListItem('Toothpaste');
```

## Clean & Performant (Creating the elements)

The second way is to actually create all of the elements including the `li` element, text node, button element and icon element and appending each one.

```
function createNewItem(item) {
  const li = document.createElement('li');

  li.appendChild(document.createTextNode(item));

  const button = document.createElement('button');

  button.className = 'remove-item btn-link text-red';

  const icon = document.createElement('i');

  icon.className = 'fa-solid fa-xmark';
```

```
button.appendChild(icon);

li.appendChild(button);

document.querySelector('.items').appendChild(li);
}
```

Now, you could use either method, just know that using innerHTML causes the web browsers to reparse and recreate all the DOM nodes inside the ul element. So this is less efficient than creating a new element and appending to the ul. So the second way is more performant. Also, setting innerHTML will not automatically reattach event handlers to the new elements it creates, so you would have to keep track of them manually. It is just better practice to create your elements, rather than using innerHTML.

# Refactor To Use Multiple Functions

---

In the last lesson, we created a function to add items to our shopping list. In that function, we created multiple elements. We created a list item, a button and an icon. I think that you should generally have functions do one thing. In this case our function did 3 things. Let's refactor this code into multiple functions that we can reuse if needed.

Here is the original code.

```
function createNewItem(item) {
  const li = document.createElement('li');

  li.appendChild(document.createTextNode(item));

  const button = document.createElement('button');

  button.className = 'remove-item btn-link text-red';

  const icon = document.createElement('i');

  icon.className = 'fa-solid fa-xmark';

  button.appendChild(icon);

  li.appendChild(button);

  document.querySelector('.items').appendChild(li);
}
```

Let's create a function to create the button. We will take the classes in as arguments

```
function createButton(classes) {
  const button = document.createElement('button');
  button.className = classes;

  return button;
}
```

We create and return the button. We need to include the icon, so let's create a function to create an icon

```
function createIcon(classes) {
  const icon = document.createElement('i');
  icon.className = classes;
```

```
    return icon;  
}
```

Now we can add the `createIcon()` function to our `createButton()` function

```
function createButton(classes) {  
  const button = document.createElement('button');  
  button.className = classes;  
  
  const icon = createIcon('fa-solid fa-xmark');  
  button.appendChild(icon);  
  
  return button;  
}
```

Now, add the `createButton()` function to our main function

```
function createNewItem(item) {  
  const li = document.createElement('li');  
  li.appendChild(document.createTextNode(item));  
  
  // Add button & icon  
  const button = createButton('remove-item btn-link text-red');  
  li.appendChild(button);  
  document.getElementById('item-list').appendChild(li);  
}
```

Now we have everything broken up into small functions. If something goes wrong, it is much easier to debug and the code is more readable and reusable.

# Insert Elements, Text & HTML

---

We know how to select and create elements. We also learned how to insert an item into the DOM using `appendChild()`. There are other methods that we can use to insert elements, text and HTML into the DOM.

Let's use our shopping list and add some new items in specific positions.

## `insertAdjacentElement()`

`insertAdjacentElement()` as well as the next two methods work in a similar way. We call the method on the current element that we want to insert before or after. Then we pass in a position as the first argument, then our new custom element that we want to insert as the second argument.

The four position options are **beforebegin**, **afterbegin**, **beforeend**, **afterend**. The placement is as follows:

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  ###The content of the initial element###
  <!-- beforeend -->
</p>
<!-- afterend -->
```

Let's say we want to create an `h1` within our JS and insert it before the `filter input`. We will put this code in a custom function called `insertElement()`

```
function insertElement() {
  const filter = document.querySelector('.filter');

  const h1 = document.createElement('h1');
  h1.textContent = 'insertAdjacentElement';

  filter.insertAdjacentElement('beforebegin', h1);
}

insertElement();
```

Now your `h1` is before the filter. You can change the position to play round with it.

## `insertAdjacentText()`

`insertAdjacentText()` works the same way except instead of inserting a custom element, we use it to insert text.

Let's insert some text before the first `li` element

```
function insertText() {  
  const item = document.querySelector('li:first-child');  
  
  item.insertAdjacentText('beforebegin', 'insertAdjacentText');  
}  
  
insertText();
```

## insertAdjacentHTML()

`insertAdjacentHTML()` works the same way, but for straight HTML. It is similar to using `innerHTML`

Let's insert some HTML with an `h2` tag after the clear button

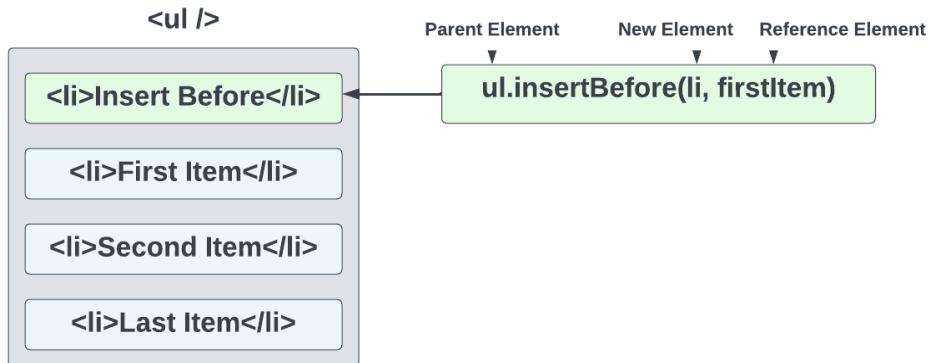
```
function insertHTML() {  
  const clearBtn = document.querySelector('#clear');  
  
  clearBtn.insertAdjacentHTML('afterend', '<h2>insertAdjacentHTML</h2>');  
}  
  
insertHTML();
```

## insertBefore()

`insertBefore()` is called on the parent element and takes two arguments, the new element and the reference element. It will insert the new element before the reference element.

The following will add a new `li` element before the first `li` element

```
const ul = document.querySelector('ul');  
  
const li = document.createElement('li');  
li.textContent = 'Insert Me Before!';  
  
const firstItem = document.querySelector('li:first-child');  
  
// Call on parent element  
ul.insertBefore(li, firstItem);
```



## Custom `insertAfter()`

You would probably think that since there is an `insertBefore()` method, there would be an `insertAfter()` method. Unfortunately, there is not. This is one thing I miss about JQuery that JavaScript has not implemented.

Just because it doesn't exist doesn't mean that we can't create it.

```
function insertAfter(newNode, existingNode) {
  existingNode.parentNode.insertBefore(newNode, existingNode.nextSibling);
}
```

The function above takes in a new node/element and an existing node/element and inserts the new node after the existing node. We could achieve this by using the `parentNode` and `nextSibling` properties of the existing node.

Now we can use this function to insert a new `li` element after the first `li` element.

```
// Parent element
const ul = document.querySelector('ul');
// New element to insert
const li = document.createElement('li');
li.textContent = 'Insert Me After!';

// Reference element to insert after
const firstItem = document.querySelector('li:first-child');

insertAfter(li, firstItem);
```

You do have to be careful with this because, there is no parent, so if your existing node selector matches other elements, you may run into an issue. Just be specific with your selectors.

# Custom insertAfter() Challenge

---

## Instructions

You may think that since there is an `insertBefore()` method, there is also an `insertAfter()`, but there isn't. In this challenge, I want you to create a custom `insertAfter()` function. If you don't want to do it as a challenge, that's fine, just follow along.

- The first param will be `newEl` which will be a new element that you create with `document.createElement()`
- The second param will be `existingEl` which is an element in the DOM that you want to insert your new element after

The function will be called like this:

```
// New element to insert
const li = document.createElement('li');
li.textContent = 'Insert Me After!';

// Existing element to insert after
const firstItem = document.querySelector('li:first-child');

// Our custom function
insertAfter(li, firstItem);
```

## Hint:

Remember the properties to get parent and sibling elements. Use some of those combined with `insertBefore()`.

► Click For Solution

```
function insertAfter(newEl, existingEl) {
    existingEl.parentElement.insertBefore(newEl, existingEl.nextSibling);
}
```

The solution is actually really simple.

- We take the `existingEl` and get the parent with either the `parentElement` or `parentNode` property.
- We call `insertBefore()` on the parent and then pass in `newEl` and the element AFTER `existingEl`. We can get that with the `nextSibling` property

Putting our new element before the existing element's next sibling is the same as putting it after the existing element



# Replacing DOM Elements

---

In the last video we looked at how to insert elements into the DOM. Now let's look at some methods to replace elements.

We are going to be working with our shopping list UI again. We will be replacing some of the `li` elements in the list. There is more than one way to do this, so I will show you a couple.

## Method 1 - `replaceWith()`

Let's replace the first `li` element with a new `li` element using the `replaceWith()` method.

```
// Element to replace
const firstItem = document.querySelector('li:first-child');
// New element
const li = document.createElement('li');
li.textContent = 'Replaced First';

// Replace first element with new element
firstItem.replaceWith(li);
```

We call the `replaceWith()` method on the first `li` element and pass in the new `li` element.

## Method 2 - `outerHTML`

We could also use the `outerHTML` property to replace the entire element. Let's replace the second item this way.

```
// Element to replace
const secondItem = document.querySelector('li:nth-child(2)');
// Replace using outerHTML
secondItem.outerHTML = '<li>Replaced Second</li>';
```

## Replacing All Elements

As you know, using `document.querySelector()` will only select one element even if there are more on the page. Let's use `document.querySelectorAll()` to select all of the `li` elements and then loop through and replace them.

```
// Elements to replace
const lis = document.querySelectorAll('li');
// Loop through and replace all list items
lis.forEach((item) => {
```

```
    item.outerHTML = '<li>Replace All</li>';
});
```

If we wanted to edit one of the elements, we could for example change the text of the first item by checking the index.

```
// Elements to replace
const lis = document.querySelectorAll('li');
// Loop through and replace all list items
lis.forEach((item, index) => {
  item.outerHTML = index === 1 ? '<li>Second Item</li>' : '<li>Item</li>';
});
```

## replaceChild()

Another way that we can replace something is by selecting the parent element and then using the `replaceChild()` method. Let's replace the `h1` element with an `h2`.

```
const h1 = document.querySelector('h1');
const header = document.querySelector('header');
const h2 = document.createElement('h2');
h2.id = 'app-title';
h2.textContent = 'Shopping List';
header.replaceChild(h2, h1);
```

`replaceChild()` is called on the parent element and takes two arguments, the new element and the old element.

# Remove DOM Elements

---

Now let's look at methods to remove elements completely from the DOM.

## remove()

Remove is very simple. You use it on the element you want to remove and it will remove it from the DOM. Let's say that we want to remove the clear button from the page.

```
const clearButton = document.querySelector('#clear');
clearButton.remove();
```

## removeChild()

Remove child is a bit more complicated. You use it on the parent element and it will remove the child element from the parent. Let's say that we want to remove the first `li` in the shopping list.

```
const ul = document.querySelector('ul');
const li = document.querySelector('li:first-child');
ul.removeChild(li);
```

We first select the parent element and then the child element. Then we use `removeChild()` on the parent and pass in the child.

## Removing Specific List Items

Let's create a function to remove a specific list item. I will show you a few ways we can do this.

```
function removeListItem(itemNumber) {
  const ul = document.querySelector('ul');
  const li = document.querySelector(`li:nth-child(${itemNumber})`);
  ul.removeChild(li);
}
```

Here we select the parent element and then use the function argument in the selector to remove the specific item.

Let's look at another:

```
function removeListItem(itemNumber) {
  const ul = document.querySelector('ul');
  const li = document.querySelectorAll('li')[itemNumber - 1];
```

```
    ul.removeChild(li);
}
```

Here, we used the `querySelectorAll()` method to select all the `li` elements and then used the `itemNumber - 1` to select the correct element. We subtracted 1 from the item number because the array/NodeList starts at 0.

One more:

```
function removeListItem(itemNumber) {
  const ul = document.querySelector('ul');
  const li = document.querySelectorAll('li');

  li[itemNumber - 1].remove();
}
```

This one is similar, but we used `remove()` on the element directly.

As you can see, removing elements is pretty easy.

# Event Listeners

---

Alright, so we've learned quite a bit about manipulating the DOM, however, many of the things that we have learned, are actions that you'll want to happen when some kind of event is triggered. So that's what we're going to be talking about for the next few sections.

## What Are Events?

Events are something that happens within the browser. There are many kinds of events. Even the page loading is an event that we can listen for and respond to. Some other examples of events would be..

- Clicking on an element, such as a button
- Typing into a text input field
- Hovering over an element
- Submitting a form
- Closing a window
- Dragging an element
- Resizing an element
- etc.

Like I said, we can listen for events and respond to them. There are a few ways to do this.

## Inline Event Listeners

There are a few ways to listen for an event and react to it. One of the simplest ways is to use the `onclick` attribute. This is NOT recommended as it is not very flexible and can possibly pose a security risk. With that said, you should still know it exists.

Let's add an `onclick` event listener to the clear button in our shopping list. We will set the value to `alert('Cleared')`, which will show a browser alert popup when we click the button.

```
<button onclick="alert('Cleared')" id="clear" class="btn-clear">  
  Clear All  
</button>
```

So as you can see, we can put JavaScript code directly in the HTML attribute. This is not very good practice though. Another thing we could do is use a JavaScript function to handle the event.

```
<button onclick="onClear()" id="clear" class="btn-clear">Clear All</button>
```

Now create the function `onClear()` in the linked JS file.

```
function onClear() {
  console.log('Clear Items');
}
```

Now when you click the button, you'll see the console log show up.

## Event Listeners in JavaScript

Like I mentioned, using inline event listeners is not recommended. Instead, we can use JavaScript to handle the event. There are a couple ways to do this.

One way is to bring in the element and call `onclick` from there.

```
const clearBtn = document.getElementById('clear');

clearBtn.onclick = function () {
  console.log('Clear Items');
};
```

This is equivalent to the inline event we had in the HTML, but it gives you more control over the scope. One drawback to this is you may only have one inline event assigned. Inline events are stored as an attribute/property of the element it can be overwritten.

## addEventListener()

The best way, at least in my opinion, is to use the `addEventListener()` method. It takes in the event type (click, submit, etc) and a callback function. It attaches a listener to an element and will call the callback function when the event is triggered.

Let's look at an example:

```
clearBtn.addEventListener('click', function () {
  console.log('Clear Items');
});
```

We don't have to pass in an anonymous function. We can pass in another defined function, such as the `onClear()` function.

```
function onClear() {
  console.log('Clear Events');
}

clearBtn.addEventListener('click', onClear);
```

This is the method that we will be using in the course and what I would suggest in just about every case when using Vanilla JavaScript.

## removeEventListener()

We can also remove event listeners from elements. If you're removing an element that has an event listener attached to it. It's good practice to first remove the event listener before removing the element. This can prevent memory leaks, especially in older browsers. For the most part, modern browsers will handle this correctly, but it's still good to know how to do this.

Just like we have `addEventListener()`, we can use the `removeEventListener()` method. It takes in the event type and the same callback function that you used on the `addEventListener()`.

What I'm going to do is create a timer using the `setTimeout()` method. This will be used to remove the event listener after a few seconds. I know we have not talked about `setTimeout()` yet. Later on we will learn more about this method.

```
setTimeout(() => {
  clearBtn.removeEventListener('click', onClear);
}, 5000);
```

So now, the event listener is removed after 5 seconds and the button will stop logging when it is clicked.

## Triggering Events in JavaScript

There may be times when you want to call an event from JavaScript without having the user do any kind of interaction. You can easily do this by running the event manually.

I'm going to do another `setTimeout()`, because I want the clear button to click after 5 seconds when the page loads. You should comment out the other `setTimeout()` if you're following along.

```
setTimeout(() => {
  clearBtn.click();
}, 5000);
```

After 5 seconds, the click event fires. We could also use other event types like `submit()` and `mouseover()`. We will talk about other event types soon.

## Clearing The Items

Let's add some code to clear all of the items from the list. I'll give you 3 different ways to make that happen.

```
function onClear() {
  const itemList = document.querySelector('ul');
```

```
const items = itemList.querySelectorAll('li');

// Using innerHTML
itemList.innerHTML = '';

// Using forEach() and remove()
items.forEach((item) => item.remove());

// Using a while loop and firstChild
while (itemList.firstChild) {
    itemList.removeChild(itemList.firstChild);
}

}
```

# Mouse Events

---

You now know how to listen for an event, but we have only looked at the `click` event. There are many more. Everything from mouse events to events that are part of APIs like the web animation API or websockets. In this video, I want to specifically look at mouse events.

Event	Description
click	Click left mouse button
dblclick	Double click left mouse button
contextmenu	Click right mouse button
mousedown	Click down left mouse button
mouseup	Release left mouse button
wheel	Scroll Wheel button
mouseover	Hover over element
mouseout	Leave hovering element
dragstart	Click and start to drag element
drag	Drag element around screen
dragend	Stop dragging

We can test these events with the following code:

```
const logo = document.querySelector('img');

const onClick = () => console.log('click event');
const onDoubleClick = () => console.log('double click event');
const onRightClick = () => console.log('right click event');
const onMouseDown = () => console.log('mouse down event');
const onMouseUp = () => console.log('mouse up event');
const onMouseWheel = () => console.log('mouse wheel event');
const onMouseOver = () => console.log('mouse over event');
const onMouseOut = () => console.log('mouse out event');
const onDragStart = () => console.log('drag start event');
const onDrag = () => console.log('drag event');
const onDragEnd = () => console.log('drag end event');

// Button
logo.addEventListener('click', onClick);
logo.addEventListener('dblclick', onDoubleClick);
logo.addEventListener('contextmenu', onRightClick);
logo.addEventListener('mousedown', onMouseDown);
logo.addEventListener('mouseup', onMouseUp);
```

```
logo.addEventListener('wheel', onWheel);
// Hover
logo.addEventListener('mouseover', onMouseOver);
logo.addEventListener('mouseout', onMouseOut);
// Drag
logo.addEventListener('dragstart', onDragStart);
logo.addEventListener('drag', onDrag);
logo.addEventListener('dragend', onDragEnd);
```

# The Event Object

---

We will look at some other types of events in the next video, but before we do that, I want to talk about the `Event` object. When we run an event handler (The function that runs when the event is triggered), There is an object that is passed in to it that gives us a bunch of information about the element that is attached to that event.

Let's log the `Event` object to the console:

```
logo.addEventListener('click', function (e) {  
  console.log(e);  
});
```

You can also use a named function:

```
function onClick(e) {  
  console.log(e);  
}  
  
logo.addEventListener('click', onClick);
```

It is important to know that the properties on the event object can vary depending on the event and the element. Let's look at some of the properties that are available on the event object:

- `target` - The element that triggered the event
- `currentTarget` - The element that the event listener is attached to (These are the same in this case)
- `type` - The type of event that was triggered
- `timeStamp` - The time that the event was triggered
- `clientX` - The x position of the mouse click relative to the window
- `clientY` - The y position of the mouse click relative to the window
- `offsetX` - The x position of the mouse click relative to the element
- `offsetY` - The y position of the mouse click relative to the element
- `pageX` - The x position of the mouse click relative to the page
- `pageY` - The y position of the mouse click relative to the page
- `screenX` - The x position of the mouse click relative to the screen
- `screenY` - The y position of the mouse click relative to the screen

Try logging some of the properties to the console to see what they are.

```
logo.addEventListener('click', function (e) {  
  console.log(e.target);  
  console.log(e.currentTarget);  
  console.log(e.type);
```

```
    console.log(e.timeStamp);
    console.log(e.clientX, e.clientY);
    console.log(e.offsetX, e.offsetY);
    console.log(e.pageX, e.pageY);
    console.log(e.screenX, e.screenY);
}
```

## target vs currentTarget

Sometimes the element that you click on is different than the element that the event is attached to. In the case of the logo, it's the same because there is nothing else nested in that logo/image. If we add an event listener to the body, the body is the `currentTarget`, but whatever we click on, such as one of the `li` elements is the `target`. Try it out.

```
document.body.addEventListener('click', (e) => {
  console.log(e.target);
  console.log(e.currentTarget);
});
```

## e.preventDefault()

Sometimes we want to prevent the default behavior of an event. For example, if we click on a link, we want to prevent the browser from following the link or we click a form submit button, we want to prevent the form from submitting to an actual file.

Let's add a link around the logo that points to Google:

```
<a href="https://google.com">
  
</a>
```

Right now, it will send us to Google. If we add an event listener to the link, we can prevent the browser from following the link:

```
document.querySelector('header a').addEventListener('click', function (e) {
  e.preventDefault();
  console.log('Link was clicked.');
});
```

## Dynamic Values

Sometimes the values will change as an event fires off. Let's add a drag event to the logo and output the X and Y position from the `h1`.

```
function onDrag(e) {
  document.querySelector('h1').textContent = `X ${e.clientX} Y ${e.clientY}`;
}

logo.addEventListener('drag', onDrag);
```

I would encourage you to play around with this stuff. Even if it is ridiculous UI behavior, you can learn a lot from it.

In this video, we're going to look at events that have to do with keyboards. We usually use these on text input fields.

In the next video, we will look at form and input events, but right now I just want to focus on the actual keyboard button events and getting the buttons that are pressed.

## Keyboard Events

There are 3 main keyboard events that we can listen for:

- `keydown` event is fired when a key is pressed down.
- `keyup` event is fired when a key is released.
- `keypress` event is fired when a key is pressed and released. It does not fire for the left arrow key, home, or end keys. It also fires repeatedly while you hold down the key on the keyboard.

The `keydown` event is usually what I use over the other 2 because it fires for every key and as soon as a key is pressed.

Let's try these out on our input form. We can also get the exact key that was pressed from the `Event` object that we looked at in the previous video.

```
const itemInput = document.getElementById('item-input');

function onKeyDown(e) {
  console.log('keydown:', e.key);
}

function onKeyUp(e) {
  console.log('keyup:', e.key);
}

function onKeyPress(e) {
  console.log('keypress:', e.key);
}

itemInput.addEventListener('keydown', onKeyDown);
itemInput.addEventListener('keyup', onKeyUp);
itemInput.addEventListener('keypress', onKeyPress);
```

Notice that if we hold the key down, the `keydown` and `keypress` event will fire repeatedly.

Also notice that the `keypress` event does not fire for the left arrow key, home, or end keys.

### Getting The Pressed Key

As I showed you above, using the `key` property on the `Event` object, we can get the key that was pressed. This is useful if for instance, you wanted to do something when typing in an input that is not in a form to submit. You could just check for the `enter` key.

```
function onKeyDown(e) {
  if (e.key === 'Enter') {
    alert('Enter was pressed.');
  }
}
```

## Key Codes

The `key` property is not supported in some older browsers. There are a couple other ways to get keys though.

**keyCode is the numeric value of the key.**

Every key on the keyboard has a key code. The key code is a number that represents the key. There is a list of key codes [here](#)

The `enter` key has a key code of `13`. So if we wanted to check for it, we could also check the key code:

```
function onKeyDown(e) {
  if (e.keyCode === 13) {
    alert('Enter was pressed.');
  }
}
```

**code is the name of the key.**

```
if (e.code === 'Digit1') {
  alert('Number 1 key was pressed.');
}
```

## repeat Property

We can also check to see if the key is being held down using the `repeat` property.

```
function onKeyDown(e) {
  console.log('Repeat: ' + e.repeat);
}
```

If you just tap a key, you will get false, but if you hold the key down, you will get true.

## shiftKey, ctrlKey & altKey Property

These properties will be either true or false if you hold down the shift, ctrl, or alt keys.

```
function onKeyDown(e) {  
    console.log('Shift: ' + e.shiftKey);  
    console.log('Ctrl: ' + e.ctrlKey);  
    console.log('Alt: ' + e.altKey);  
  
    if (e.shiftKey && e.key === 'K') {  
        alert('Shift + K was pressed.');//  
    }  
}
```

# Keycode Mini-project

---

In the last video, we learned how to get the key that was pressed using the `key`, `keyCode`, and `code` properties. In this mini-project, we will create a simple keyboard event listener that will display all 3 of these properties on the screen. I think you'll be surprised how easy this will be.

## The HTML

The HTML is really simple:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="style.css" />
    <title>Event KeyCodes</title>
  </head>
  <body>
    <div id="insert">
      <div class="key">
        Press any key to get the keyCode
      </div>
    </div>
    <script src="script.js"></script>
  </body>
</html>
```

We have a wrapper div with an id of `insert`. This is where we will display the information.

## The CSS

Here is some basic styling. We have a class of `key` that is styled like a card or a box. We will wrap each output in a `div` with the class of `key`.

```
@import url('https://fonts.googleapis.com/css?family=Muli&display=swap');

* {
  box-sizing: border-box;
}

body {
  background-color: #e1e1e1;
  font-family: 'Muli', sans-serif;
  display: flex;
  align-items: center;
```

```

justify-content: center;
text-align: center;
height: 100vh;
overflow: hidden;
margin: 0;
}

.key {
  border: 1px solid #999;
  background-color: #eee;
  box-shadow: 1px 1px 3px rgba(0, 0, 0, 0.1);
  display: inline-flex;
  align-items: center;
  font-size: 20px;
  font-weight: bold;
  padding: 20px;
  flex-direction: column;
  margin: 10px;
  min-width: 150px;
  position: relative;
}

.key small {
  position: absolute;
  top: -24px;
  left: 0;
  text-align: center;
  width: 100%;
  color: #555;
  font-size: 14px;
}

```

## The JavaScript

```

const insert = document.getElementById('insert');

window.addEventListener('keydown', (e) => {
  insert.innerHTML = `
    <div class="key">
      ${e.key === ' ' ? 'Space' : e.key}
      <small>e.key</small>
    </div>

    <div class="key">
      ${e.keyCode}
      <small>e.keyCode</small>
    </div>

    <div class="key">
      ${e.code}
    
```

```
<small>event.code</small>
</div>
`;
});
```

As you can see, it's very simple. We add an event listener to the window and insert some HTML into the `insert` div. The HTML includes 3 `divs` with the class of `key`. Each `div` has a `small` that displays the 3 properties.

# Input Events

---

Now let's look at the events that are fired when the user interacts with inputs. These are events we want to happen right when the user interacts with the input, not on a form submission. We will look at submitting forms in the next video.

## Getting Input Values

In the last video, we saw how to use keyboard events and get the specific keys that were pressed, but usually you want to get the entire input value. Every form input has a `value` property associated with it. We can use the keyboard events and get the value of a text input using the `e.target.value` property.

```
const itemInput = document.getElementById('item-input');

function onKeyDown(e) {
  console.log(e.target.value);
}

itemInput.addEventListener('keydown', onKeyDown);
```

If you wanted to bind what you are typing to an element like the `h1`, we could do something like this:

```
function onKeyDown(e) {
  const h1 = document.querySelector('h1');
  h1.innerText = e.target.value;
}
```

## input event

The problem with using the `keydown` event is that it only fires when a key is pressed. Some form inputs do not require you to press a key. For instance, if you have a select list and you want to fire something off as soon as you select an option, you can use the `input` event. You can also use it with text inputs. So I would suggest using the `input` event over `keydown`.

We can achieve the same result of outputting the text into the `h1` element by using the `input` event.

```
const itemInput = document.getElementById('item-input');

function onInput(e) {
  const h1 = document.querySelector('h1');
  h1.innerText = e.target.value;
}

itemInput.addEventListener('input', onInput);
```

Let's try this with a select box. We don't have one in our UI, so we will have to create one. I will add a select box for priority into our form, but you do not necessarily need to have your inputs within a form if you are not submitting the form.

```
<select id="priority-input" name="priority" class="form-input">
  <option value="0">Select Priority</option>
  <option value="1">1</option>
  <option value="2">2</option>
  <option value="3">3</option>
</select>
```

Now we should be able to select the select box and output the value to the `h1` element.

```
const priorityInput = document.getElementById('priority-input');

function onInput(e) {
  const h1 = document.querySelector('h1');
  h1.innerText = e.target.value;
}

priorityInput.addEventListener('input', onInput);
```

## change event

The `change` event is fired when the value of an input changes. You could use the `change` event here and it will do the same exact thing.

```
function onInput(e) {
  const h1 = document.querySelector('h1');
  h1.innerText = e.target.value;
}

priorityInput.addEventListener('change', onInput);
```

There are some limitations to the `change` event. I personally would use the `input` event instead in most cases.

## Checkbox Inputs

Checkbox inputs are a bit different. You can use the `input` or the `change` event, however, if you want to get the value of the checkbox, you should use the `e.target.checked` property. This will show true if the checkbox is checked and false if it is not.

```
<input type="checkbox" id="checkbox" />
```

```
const checkbox = document.getElementById('checkbox');

function onCheck(e) {
  console.log(e.target.checked);
  const h1 = document.querySelector('h1');
  h1.innerText = checkbox.checked ? 'Checked' : 'Unchecked';
}

checkbox.addEventListener('input', onCheck);
```

## focus & blur Events

Text inputs have a `focus` event and a `blur` event. `focus` is fired when we click in the input to make it active and `blur` is fired when we click away from the input. This is useful for adding borders, etc.

Let's make the item input have a green border when we click in it and make it disappear when we click away.

```
function onFocus(e) {
  itemInput.style.outlineStyle = 'solid';
  itemInput.style.outlineWidth = '1px';
  itemInput.style.outlineColor = 'green';
}

function onBlur(e) {
  itemInput.style.outlineStyle = 'none';
}

itemInput.addEventListener('focus', onFocus);
itemInput.addEventListener('blur', onBlur);
```

# Form Submission

---

In the last video we looked at events on specific input fields. Now I want to look at form submission, which includes the `submit` event.

## No Action

If you have any experience with server-side programming with a language such as PHP, then you know that you usually submit a form to a specific page using the `action` attribute. It may look something like this:

```
<form action="/form-handler.php" method="post"></form>
```

When we are dealing with a form submission using front-end JavaScript, we don't have to worry about the `action` attribute. We actually want to prevent that type of submission, so that we can handle everything within JavaScript.

If you do need to submit the form to a backend or an API (to submit to a database, etc), you would do that from your JavaScript code by using the `fetch` API, which I will talk about in another section. So essentially, we are intercepting the form submission and preventing it.

## submit Event

Let's add some JavaScript to select the form and listen for a submit:

```
const form = document.getElementById('item-form');

function onSubmit() {
    console.log('Form submitted');
}

form.addEventListener('submit', onSubmit);
```

If you submit the form, you will see the message in the console, but it will just flash for a second and then disappear. This is because the form submission is being handled by the browser using the `action` attribute by default. If there is no `action` attribute, then the browser will submit the form to the current page.

What we want to do is prevent the default behavior and if you remember, we already used the `preventDefault` method to do that in a past video. Let's do it here.

```
function onSubmit(e) {
    e.preventDefault(); // Add this line
```

```
    console.log('Form submitted');
}
```

Now if you submit, you should see the message in the console, but it will not flash, it should stay there.

## Getting Form Data

There are a couple ways to get your form data. All form inputs have a `value` property that you can get.

Since our form only has a single text input, let's add a priority select input just for this video. Place this in the form:

```
<select id="priority-input" name="priority" class="form-input">
  <option value="1">1</option>
  <option value="2">2</option>
  <option value="3">3</option>
</select>
```

We can get both input values by selecting the element and using the `value` property.

```
function onSubmit(e) {
  e.preventDefault();

  const item = document.getElementById('item-input').value;
  const priority = document.getElementById('priority-input').value;

  console.log(item, priority);
}
```

Whatever we type in the item input will now be in the `item` variable and whatever we select as a priority will be in the `priority` variable. So we could use that to do some validation, submit it to some kind of API, etc.

Let's add a little validation. We don't want to be able to submit an empty item. We also don't want to submit an item with a priority of 0.

```
function onSubmit(e) {
  e.preventDefault();
  const item = document.getElementById('item-input').value;
  const priority = document.getElementById('priority-input').value;

  if (item === '' || priority === '0') {
    alert('Please fill in all fields');
    return;
  }
```

```
    console.log('Adding Item...');  
}
```

If the validation does not pass, we show an alert box and return from the function. So nothing else happens. If it does pass, it will go on to the console log.

## FormData Object

The `FormData` object is a special type of object that is used to collect form data. This is a bit more difficult than the previous method.

Let's create another function called `onSubmit2` and call that on form submission. Then create the new `FormData` object and log it.

```
function onSubmit2(e) {  
  e.preventDefault();  
  
  const formData = new FormData(form);  
  
  console.log(formData);  
}  
  
form.addEventListener('submit', onSubmit2);
```

script.js:20

```
▼ FormData {} ⓘ  
  ▼ [[Prototype]]: FormData  
    ► append: f append()  
    ► delete: f delete()  
    ► entries: f entries()  
    ► forEach: f forEach()  
    ► get: f ()  
    ► getAll: f getAll()  
    ► has: f has()  
    ► keys: f keys()  
    ► set: f ()  
    ► values: f values()  
    ► constructor: f FormData()  
    ► Symbol(Symbol.iterator): f iterator()  
    ► Symbol(Symbol.toStringTag): "Form Data"  
  ▶ [[Prototype]]: Object
```

As you can see, there are a bunch of methods in the prototype. We can use the `get` method to get the value of a specific input.

```
function onSubmit2(e) {  
  e.preventDefault();
```

```
const formData = new FormData(form);
const item = formData.get('item');
const priority = formData.get('priority');

console.log(item, priority);
}
```

We can get all of the form data by using the `entries` method. This will return an array of arrays, where each array is a pair of values. We can see them by using a `for of` loop.

```
function onSubmit2(e) {
  e.preventDefault();

  const formData = new FormData(form);
  const entries = formData.entries();

  for (let entry of entries) {
    console.log(entry);
  }
}
```

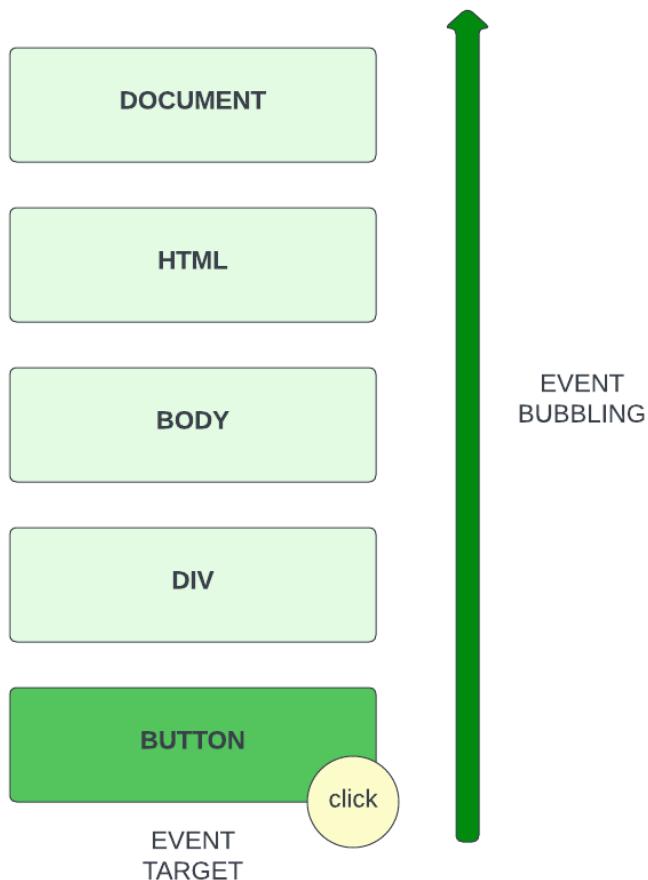
There is a lot more we can do with the `FormData` object, but it is beyond the scope of what we are doing. I just figured I would throw it in here because it is another way to get your form values.

Before we move on to making our shopping list actually work, there are a few more things we need to talk about that have to do with events. In the next video, we'll talk about event bubbling and event delegation.

# Event Bubbling

In this video, we're going to talk about something that is pretty important when it comes to handling events and that's event bubbling.

When we add an event listener to an element, that event moves up the DOM tree and if it finds a parent element that has a listener for that event, it will fire it.



So even if we click on the button, the event will bubble up all the way up the document.

Let me show you exactly what I mean instead of trying to explain it.

Let's use our shopping list example and in the JavaScript, let's select the add item button in the form and add an event listener to it.

```
const button = document.querySelector('form button');

button.addEventListener('click', () => {
  alert('Button was clicked.');
});
```

So we click on the button and we get an alert. No big deal. Now, let's bring in it's parent `div` and add an click event listener to it.

```
```JavaScript
const div = document.querySelector('form div:nth-child(2)');

div.addEventListener('click', (e) => {
  alert('Div was clicked.');
});
```

Now click to the right of the button and you should see **Div was clicked.** as to be expected.

Now click on the button. You'll get the initial button alert, but then you'll get the div alert. This is because the event is bubbling up the DOM tree.

Now let's add a click event on to the parent form.

```
const form = document.querySelector('form');

form.addEventListener('click', (e) => {
  alert('Form was clicked.');
});
```

Now click on the button. You'll get the initial button alert, then the form alert, then the div alert.

Finally, let's ass a click event on to the document body.

```
document.body.addEventListener('click', (e) => {
  alert('Body was clicked.');
});
```

There will be times where you'll need to understand this to be able to achieve certain results.

Now there may be cases where you'll want to stop the event from bubbling up the DOM tree. What if you have a legit click event on the `button` and also on the `div`, but you don't want the `div` to fire the event?

That's where `stopPropagation()` comes in. This is a method on the `Event` object that you can call on an event to stop it from bubbling up the DOM tree.

Let's add it to our `button` event handler.

```
button.addEventListener('click', (e) => {
  alert('Button was clicked.');
```

```
e.stopPropagation();  
});
```

Now if you click on the button, it will not fire off the `div` event or any others above it. You can still click on the `div` and it will fire off. It will also bubble up from there because we didn't call `stopPropagation()`.

You should only call `stopPropagation()` if you have a reason for it. For instance if we actually had a parent and child with the same event listener.

There is also a method called `stopImmediatePropagation()`. This is used if you have multiple handlers on a single event and you want to stop the event for all of them.

Now that you know how bubbling works, in the next video, we'll talk about event delegation.

# Event Delegation

---

Now that you understand how event bubbling works, you'll be able to learn how to use a powerful event handling pattern called event delegation.

## Dealing With Multiple Elements

Where this comes in handy is when we want to have an event handler for multiple elements. Our shopping list is a great example of this. We have a bunch of list items (`li` elements) and we're going to want to be able to click on the x icon to delete a specific item. How would we do this when we have a bunch of the same elements and the number of items is dynamic?

There's actually a couple ways we can do this. One way is to grab the items with `document.querySelectorAll()` and then add an event listener to each one. Let's try that.

I'm not going to target the x icon for this example, we'll just make it so that we can click anywhere on the list item and it will get removed. When we do the project, we'll target the x icon.

```
const listItems = document.querySelectorAll('li');

listItems.forEach((item) => {
  item.addEventListener('click', function (e) {
    e.target.remove();
  });
});
```

This does work. We click on an item and it gets removed. However, we're adding an event listener on every single item. This is inefficient. Especially when we have a large number of items.

## Using Event Delegation

Instead of adding an event listener on every single item, we can use event delegation. We know that events bubble up, so we can use event delegation to listen for events on the parent element and then target the actual element that was clicked on.

Let's grab the list itself (`ul`) and add an event listener to it. Then we can just check to see if the target of the event is a `li` element and if it is, we can remove it.

```
const list = document.querySelector('ul');

list.addEventListener('click', function (e) {
  if (e.target.tagName === 'LI') {
    e.target.remove();
  }
});
```

---

Now we are not creating multiple event listeners. We're just adding one event listener to the list.

Of course you are not bound to checking the tagName. You could check for a class, etc. You also are not bound to removing the item. You can do anything you want, Let's say that I want to change the item to be the color red when I hover over it.

```
list.addEventListener('mouseover', function (e) {
  if (e.target.tagName === 'LI') {
    e.target.style.color = 'red';
  }
});
```

That `e.target` is always going to be the element that the event was fired on.

# Window Events & Page Loading

---

The global `window` object has a number of events that are fired when the window is interacted with. We're just going to use a simple html page with an `h1` and some `p` tags.

## Page Load Events

Back in the day, we used to put `<script>` tags in the head of a webpage. Now, it is suggested that we put it right before the ending `</body>` tag, unless we use `defer`, which I will talk about in a minute.

Putting the JavaScript in the head of the page will run the script before the page is done loading. Therefore, if you try and access the DOM or any page element, you'll get an error. Let's try to run the script for this section in the head and add the following JavaScript

```
document.querySelector('h1').innerHTML = 'Hello';

// Uncaught TypeError: Cannot set properties of null (setting 'innerHTML')
```

It will not work because the H1 is not there yet. What we could do and what we used to do all the time, which was sort of a hack, was use the `load` event to make sure the page was loaded first.

## load Event

The `load` event is fired when the page and all of its resources are finished loading.

```
window.addEventListener('load', () => {
  document.querySelector('h1').innerHTML = 'Hello';
});
```

Now, it works because we waited until the page was loaded.

You could also use `window.onload`, which does the same thing. This code was very common back in the 2000s

```
window.onload = function() {
  document.querySelector('h1').innerHTML = 'Hello';
}
```

## DOMContentLoaded Event

The `DOMContentLoaded` event is fired when the DOM is parsed and loaded. Unlike the `load` event, it does not wait for things like images to load. If you want to make sure the DOM is loaded before you run your

code, you should use this event.

```
window.addEventListener('DOMContentLoaded', () => {
  document.querySelector('h1').innerHTML = 'Hello';
});
```

If we run both blocks of code and then put a global console log under it.

```
console.log('Run me!');
```

You will see that the code in the global space with no event will run first, because it did not wait for anything. Next the code in the `DOMContentLoaded` event will run, because it waited for the DOM to be loaded. And finally, the code in the `load` event will run, because it waited for the page to be fully loaded.

## defer attribute

A very modern way of doing things is putting the script tags in the head, but using the fairly new `defer` attribute. This will defer the JavaScript from loading until the DOM is ready. This is fine to do. I still prefer putting my scripts at the bottom, but if you want to use `defer` and put them in the head, that's fine as well.

Now let's look at a couple other `window` events

## resize Event

The `resize` event is fired when the window is resized. We can get the size of the window using the `innerWidth` and `innerHeight` properties.

Let's display the resized values in the `h1` tag.

```
window.addEventListener('resize', () => {
  document.querySelector(
    'h1'
  ).textContent = `Resized: ${window.innerWidth} x ${window.innerHeight}`;
});
```

## scroll Event

The `scroll` event is fired when the window is scrolled. We can get the scroll position using the `scrollX` and `scrollY` properties.

```
window.addEventListener('scroll', () => {
  console.log(`Scrolled: ${window.scrollX} x ${window.scrollY}`);
});
```

```
if (window.scrollY > 70) {
    document.body.style.backgroundColor = 'black';
    document.body.style.color = 'white';
} else {
    document.body.style.backgroundColor = 'white';
    document.body.style.color = 'black';
}
});
```

There are others as well. You can visit the MDN documentation for more information on `window` events, properties and methods.

You can even use `focus` and `blur` events. The following will turn all the paragraph text blue when you click in the window and back to black when you click outside of it.

```
window.addEventListener('focus', () => {
    document.querySelectorAll('p').forEach((p) => {
        p.style.color = 'blue';
    });
});

window.addEventListener('blur', () => {
    document.querySelectorAll('p').forEach((p) => {
        p.style.color = 'black';
    });
});
```

You now know a lot of different event types, you know how to select elements as well as add, remove and replace them, change their styles. With what we have learned so far, there are a lot of cool little things you can build.

In the next section, we are going to create our shopping list functionality.

# Add Items To List

---

The first piece of functionality we want to add is the ability to add items to the list. We already have the HTML and CSS in place, so let's jump right into the JavaScript.

There will be parts of this project that I refactor as we go along.

First, let's bring in the elements from the DOM that we want to work with.

```
const itemForm = document.getElementById('item-form');
const itemInput = document.getElementById('item-input');
const itemList = document.getElementById('item-list');
```

Let's add an event listener to the form and an `addItem()` function. We want to listen for the `submit` event. We will also do some basic validation to make sure the user has entered something in the input field.

```
function addItem(e) {
  e.preventDefault();

  // Validate input
  if (itemInput.value === '') {
    alert('Please enter an item');
    return;
  }

  console.log('Success');
}

itemForm.addEventListener('submit', addItem);
```

Now, we want to create a new list item and add it to the list. The list item will include a button with a font-awesome icon inside of it.

I am going to create a separate functions for creating the button and the icon. This is just preference. You could do it all in one function if you wanted.

```
function addItem(e) {
  e.preventDefault();

  const newItem = itemInput.value;

  // Validate Input
  if (newItem === '') {
    alert('Please add an item');
```

```
    return;
}

// Create list item
const li = document.createElement('li');
li.appendChild(document.createTextNode(newItem));

const button = createButton('remove-item btn-link text-red');
li.appendChild(button);

itemList.appendChild(li);

itemInput.value = '';
}

function createButton(classes) {
  const button = document.createElement('button');
  button.className = classes;
  const icon = createIcon('fa-solid fa-xmark');
  button.appendChild(icon);
  return button;
}

function createIcon(classes) {
  const icon = document.createElement('i');
  icon.className = classes;
  return icon;
}
```

# Setting Up Git & GitHub

---

Git is a version control system that is used to track changes in code. It is a very powerful tool that is very important to learn as a developer. It is used in many different ways, but we will be using it to track changes in our code as we build our shopping list app.

The basic workflow is:

1. Make changes to your code
2. Add the changes to the staging area
3. Commit the changes to the repository
4. Push the changes to GitHub (or something else like BitBucket)

## Installing Git

Git is a command line tool, so you will need to install it on your computer. You can download it from [git-scm.com](https://git-scm.com). You can use the default settings when installing. If you are on a Mac, you can also install it using Homebrew.

If you are on Windows, you can also install Git Bash, which is a [bash](#) terminal that comes with Git. You can use this instead of the regular Windows command prompt.

## Init Git

Once you install Git, go to the project folder in your command line. VS Code has a built in terminal that you can use, which is what I prefer. Enter the following command:

```
git init
```

This creates a hidden [.git](#) folder in your project folder. This is where all of the Git information is stored.

## Add Files To Git

Now that we have Git initialized, we need to add the files we want to track to Git. We can do this by entering the following command:

```
git add .
```

The files are now in the staging area. We can see what files are in the staging area by entering the following command:

```
git status
```

## Commit Files

Now that we have the files in the staging area, we can commit them to the repository. We can do this by entering the following command:

```
git commit -m "Initial Commit"
```

The `-m` flag is used to add a message to the commit. This is a good practice to get into. It is a good idea to make your commit messages descriptive. This will help you and others understand what changes were made in the commit.

Call it `shopping-list` or whatever you'd like.

## Setup SSH keys for GitHub

Now that we have a remote repo, we need to add an SSH key to GitHub. This will allow us to push our code to GitHub. We can do this by entering the following command:

```
ssh-keygen -t rsa -b 4096 -C "
```

This will create a new SSH key. You can press enter to accept the default file location and the default name of `id_rsa` or you can type in a custom name and location. I like to name my key files for the service I am using it for, so I would do something like

```
/Users/brad/ssh/id_rsa_github
```

You will then be prompted to enter a passphrase. You can leave this blank by pressing enter.

This will create a private key file and a public key file. The public key file will have the same name as the private key file, but with `.pub` at the end. You can view the contents of the public key file by entering the following command:

```
cat ~/.ssh/id_rsa_github.pub
```

## Adding the SSH Key to GitHub

Copy the contents of the public key file. Then go to your GitHub account and click on your profile picture in the top right corner. Then click `Settings`. Then click `SSH and GPG keys`. Then click `New SSH key`. Give it a title and paste the contents of the public key file into the `Key` field. Then click `Add SSH key`.

## Push To GitHub

Next, we need to setup a remote repo at GitHub. We can do this by going to [github.com](https://github.com) and creating a new repository. Login and click the + icon in the top right corner. Then click **New repository**. Give it a name and click **Create repository**.

Now that we have a remote repo, we need to push our files to it. Start copying the **git remote** command that you see on the repo page

```
git remote add origin YOUR_REMOTE_REPO_URL
```

Then specify the branch you want to push to, We will use the main branch.

```
git branch -M main
```

Finally, push the files to the remote repo.

```
git push -u origin main
```

Now, you should see your files in the remote repo.

## Updating the remote repo

Anytime you update your code and you want to push to github, you simply add the files to the staging area, commit them, and push them to the remote repo.

Let's cerate a readme file called `readme.md`. Add the following to the `readme` file:

```
# Shopping List App  
  
This is a shopping list app that I created using HTML, CSS, and JavaScript.
```

This is a **markdown** file. You can learn more about markdown [here](#).

Now push to your remote repo:

```
git add .  
git commit -m "Update"  
git push
```

The `readme` file contents will show up on the repo page.

# Remove & Clear Items

---

Now that we can add items, let's add the ability to remove and clear items.

Let's add an event listener on to the list itself and we will use event delegation to target the button/icon.

```
itemList.addEventListener('click', removeItem);
```

Create the removeItem function

```
function removeItem(e) {
  if (e.target.parentElement.classList.contains('remove-item')) {
    if (confirm('Are you sure?')) {
      e.target.parentElement.parentElement.remove();
    }
  }
}
```

## Clearing all items

We want to make the **Clear All** button function, so let's create an event listener for it.

```
clearBtn.addEventListener('click', clearItems);
```

Create the clearItems function

```
function clearItems() {
  if (confirm('Are you sure?')) {
    while (itemList.firstChild) {
      itemList.removeChild(itemList.firstChild);
    }
  }
}
```

There are many ways to clear the items. you could just set the `innerHTML` to an empty string, but I wanted to use a method that is a bit more performant.

# Clear UI State

---

This is optional, but I want to make the UI a bit more dynamic by not showing the filter input or the clear all button unless there are actual items in the list. If you still have your hardcoded `<li>` items in the HTML, you can remove them now.

We are going to have a function that we can run to check for the list items and if there are not any, we will hide the 2 elements. If they are there, we will show them.

```
function checkUI() {
  const items = itemList.querySelectorAll('li');

  if (items.length === 0) {
    clearBtn.style.display = 'none';
    itemFilter.style.display = 'none';
  } else {
    clearBtn.style.display = 'block';
    itemFilter.style.display = 'block';
  }
}
```

We are going to call this in a few places. first, we will call it at the end of the file in the global scope. This will run when the page loads and check if there are any items in the list.

```
checkUI();
```

We also want to add it when we add, remove and clear items.

```
function addItem() {
  // ...
  checkUI();
}

function removeItem(e) {
  // ...
  checkUI();
}

function clearItems() {
  // ...
  checkUI();
}
```

Now you can test by adding, removing and clearing items. You should see the buttons appear and disappear as you add and remove items.

# Filter Items

---

Now that we can add, remove and clear, we want to be able to type in the filter input and filter the items in the list.

Let's add an event listener to the filter input.

```
itemFilter.addEventListener('input', filterItems);
```

We are going to use the `filter` method to filter the items.

```
function filterItems(e) {
  const items = itemList.querySelectorAll('li');
  const text = e.target.value.toLowerCase();

  items.forEach((item) => {
    const itemName = item.firstChild.textContent.toLowerCase();

    if (itemName.indexOf(text) != -1) {
      item.style.display = 'flex';
    } else {
      item.style.display = 'none';
    }
  });
}
```

In the above code, we get the items and whatever is being typed in the filter input. Then we loop through and use the `indexOf` method to see if the item name contains the text. If it does, we show the item, otherwise we hide it.

# Local Storage Crash Course

---

## What is Local Storage?

Local Storage is a way for web pages to store named key/value pairs locally, within the client web browser. Unlike cookies, the storage limit is far larger (at least 5MB) and information is never transferred to the remote web server.

## Why would you want to use Local Storage?

Local Storage is one of the most commonly used web APIs in modern web development. It is used for client-side storage of persistent data. This means that the data stored in the browser will persist even after the browser window has been closed. Local Storage is ideal for storing simple data. It is used for things like user preferences, shopping cart contents, game scores, etc.

## How do you use Local Storage?

Local Storage is very simple to use. You can set, get, and remove data from Local Storage using the `setItem()`, `getItem()`, and `removeItem()` methods. The `setItem()` method takes a key and a value as parameters. The `getItem()` method takes a key as a parameter and returns the value. The `removeItem()` method takes a key as a parameter and removes the key/value pair from Local Storage.

### Adding an item to Local Storage

```
localStorage.setItem('name', 'John');
```

### Getting an item from Local Storage

```
const name = localStorage.getItem('name');
```

### Removing an item from Local Storage

```
localStorage.removeItem('name');
```

### Clearing all items from Local Storage

```
localStorage.clear();
```

Many times, you will want to save an array or object to Local Storage. You can do this by converting the array or object to a string using `JSON.stringify()` and then saving it to Local Storage. When you want to retrieve the array or object, you can convert it back to an array or object using `JSON.parse()`.

## Saving an array to Local Storage

```
const todos = [
  {
    text: 'Take out trash',
  },
  {
    text: 'Meeting with boss',
  },
  {
    text: 'Dentist appt',
  },
];
localStorage.setItem('todos', JSON.stringify(todos));
```

## Getting an array from Local Storage

```
const todos = JSON.parse(localStorage.getItem('todos'));
```

That is pretty much it. Local Storage is very simple to use.

# Add Items To LocalStorage

---

Now that we have add, remove, clear and filter working in the DOM, we need to be able to persist the items to localStorage in the browser.

I am going to refactor the code a bit to make it easier to work with. Instead of having the submit event call a function called `addItem`, I am going to change the name of that function to `onAddItemSubmit()` and then call that function from the submit event.

```
itemForm.addEventListener('submit', onAddItemSubmit);
```

The part of the function where we add the item to the DOM, I am going to remove and put in its own function called `addItemToDOM()`.

```
function onAddItemSubmit(e) {
  e.preventDefault();

  const newItem = itemInput.value;

  // Validate Input
  if (newItem === '') {
    alert('Please add an item');
    return;
  }

  // Create item DOM element
  addItemToDOM(newItem);

  checkUI();

  itemInput.value = '';
}
```

```
function addItemToDOM(item) {
  // Create list item
  const li = document.createElement('li');
  li.appendChild(document.createTextNode(item));

  const button = createButton('remove-item btn-link text-red');
  li.appendChild(button);

  // Add li to the DOM
  itemList.appendChild(li);
}
```

## Adding items to localStorage

Now we will create a function to add the items to localStorage. We will call it right after we add the item to the DOM.

```
function addItemToStorage(item) {
  let itemsFromStorage;

  if (localStorage.getItem('items') === null) {
    itemsFromStorage = [];
  } else {
    itemsFromStorage = JSON.parse(localStorage.getItem('items'));
  }

  // Add new item to array
  itemsFromStorage.push(item);

  // Convert to JSON string and set to local storage
  localStorage.setItem('items', JSON.stringify(itemsFromStorage));
}
```

```
function onAddItemSubmit(e) {
  // ...

  // Create item DOM element
  addItemToDOM(newItem);

  // Add item to local storage
  addItemToStorage(newItem);
}
```

Now, when you add an item, it will get added to `items` in localStorage. You can check this by going to your application tab in the dev tools and clicking on the `Local Storage` tab.

The screenshot shows a web browser window with a "Shopping List" application open. The application has a logo of a notepad with a pencil and the title "Shopping List". Below it is a search bar with the placeholder "Enter Item" and a button labeled "+ Add Item".

Below the application is the developer tools interface. The "Application" tab is selected. On the left, under "Storage", "LocalStorage" is expanded, showing an entry for "http://127.0.0.1:5500". Under this entry, the "items" key is shown with the value "[\"Juice\"]". A detailed view shows the array with one element at index 0, which is "Juice".

# Display Items From Local Storage

---

We are now able to add items to localStorage. Now we need to display them on the page when the page loads.

## Getting items from localStorage

We will start by creating a function that we can use in multiple places to get the items from localStorage.

```
function getItemsFromStorage() {
  let itemsFromStorage;

  if (localStorage.getItem('items') === null) {
    itemsFromStorage = [];
  } else {
    itemsFromStorage = JSON.parse(localStorage.getItem('items'));
  }

  return itemsFromStorage;
}
```

We are just creating a variable called `itemsFromStorage` and setting it to an empty array. Then we are checking to see if there is anything in localStorage. If there is, we are parsing it and setting it to `itemsFromStorage`. If there isn't, we are just returning an empty array.

We can now replace most of the code in `addItemToStorage()` with this function.

```
function addItemToStorage(item) {
  const itemsFromStorage = getItemsFromStorage();

  // Add new item to array
  itemsFromStorage.push(item);

  // Convert to JSON string and set to local storage
  localStorage.setItem('items', JSON.stringify(itemsFromStorage));
}
```

## Displaying items from localStorage

Next, we will create a function to display the items from localStorage on the page.

```
function displayItems() {
  const itemsFromStorage = getItemsFromStorage();
  itemsFromStorage.forEach((item) => addItemToDOM(item));
```

```
    checkUI();  
}
```

We are getting the items from localStorage and then looping through them. For each item, we are calling `addItemToDOM()` and passing in the item. Finally, we are checking the UI to see if we need to show the clear items button and filter input.

## Adding an `init()` function

Instead of putting all of the event listeners in the global scope, we will create an `init()` function and put them in there.

```
function init() {  
    // Event Listeners  
    itemForm.addEventListener('submit', onAddItemSubmit);  
    itemList.addEventListener('click', removeItem);  
    clearBtn.addEventListener('click', clearItems);  
    itemFilter.addEventListener('input', filterItems);  
    document.addEventListener('DOMContentLoaded', displayItems);  
  
    checkUI();  
}  
  
init();
```

This will work either way, but I prefer to put them in a function rather than the global space

# Remove Items From Local Storage

---

Now we want to be able to click the remove button and remove the item from the DOM and localStorage.

I am going to do the same thing that I did with the 'add' functionality and change the event listener function from `removeItem` to `onClickItem`. One reason I am doing this is because we are also going to have 'edit' functionality and that will also be used on a click on the item.

```
itemList.addEventListener('click', onClickItem);
```

Now we will create the `onClickItem()` function. We will check to see if the target of the click is the remove button. If it is, we will call the `removeItem()` function.

```
function onClickItem(e) {
  if (e.target.parentElement.classList.contains('remove-item')) {
    removeItem(e.target.parentElement.parentElement);
  }
}
```

Let's create the `removeItem()` function. We will pass in the item that we want to remove. We will remove it from the DOM and then remove it from localStorage.

```
function removeItem(item) {
  if (confirm('Are you sure?')) {
    // Remove item from DOM
    item.remove();

    // Remove item from storage
    removeItemFromStorage(item.textContent);

    checkUI();
  }
}
```

Let's create the `removeItemFromStorage()` function. We will pass in the text for the item that we want to remove. We will get the items from localStorage, loop through them, and if the item matches the item that we want to remove, we will remove it from the array. Then we will set the new array to localStorage.

```
function removeItemFromStorage(item) {
  let itemsFromStorage = getItemsFromStorage();

  // Filter out item to be removed
```

```
    itemsFromStorage = itemsFromStorage.filter((i) => i !== item);

    // Re-set to localstorage
    localStorage.setItem('items', JSON.stringify(itemsFromStorage));
}
```

## Clearing all items

This is simple. We just need to remove `items` from local storage in the `clearItems()` function.

```
function clearItems() {
  while (itemList.firstChild) {
    itemList.removeChild(itemList.firstChild);
  }

  // Clear from localStorage
  localStorage.removeItem('items');

  checkUI();
}
```

You could also use `localStorage.clear()` to clear all items from localStorage.

# Set Edit Mode

---

In this section, we will add the ability to click on an item and enable edit mode which will put that item text into the input. We will also make that item in the list a lighter color and change the button from 'Add' to 'Edit'.

Let's start by adding a global variable for the edit state. We will set it to false.

```
let isEditMode = false;
```

Let's go into the `onClickItem()` function and add an else, which will run if we are clicking in the item, but not on the delete icon. If that is true, then we will call a function called `setItemUpEdit()`

```
function onClickItem(e) {
  if (e.target.parentElement.classList.contains('remove-item')) {
    removeItem(e.target.parentElement.parentElement);
  } else {
    setItemUpEdit(e.target);
  }
}
```

Now, let's create the `setItemUpEdit()` function. This function will take in the item element and do the following:

- Set the edit state to true
- Add the `edit-mode` class to the item element
- Change the button text, icon and color
- Set the input value to the item text

```
function setItemUpEdit(item) {
  isEditMode = true;
  item.classList.add('edit-mode');
  formBtn.innerHTML = '<i class="fa-solid fa-pen"></i> Update Item';
  formBtn.style.backgroundColor = '#228B22';
  itemInput.value = item.textContent;
}
```

Now when we click on an item, it does all of those things, but if you click on another item, the 'edit-mode' class is not removed from the previous item. Let's fix that by removing the class from all items before we add it to the new item.

```
function setItemToEdit(item) {
  isEditMode = true;

  itemList
    .querySelectorAll('li')
    .forEach((i) => i.classList.remove('edit-mode'));

  item.classList.add('edit-mode');
  formBtn.innerHTML = '<i class="fa-solid fa-pen"></i> Update Item';
  formBtn.style.backgroundColor = '#228B22';
  itemInput.value = item.textContent;
}
```

# Update Item & Reset State

---

We can now set the app to 'edit mode' by clicking on an item. Now we need to add the ability to update the item.

We can do this by checking to see if we are in edit mode in the `onAddItemSubmit()` function. If we are, then we will update the item, otherwise we will add a new item.

```
function onAddItemSubmit(e) {
  e.preventDefault();

  const newItem = itemInput.value;

  // Validate Input
  if (newItem === '') {
    alert('Please add an item');
    return;
  }

  // Check for edit mode
  if (isEditMode) {
    const itemToEdit = itemList.querySelector('.edit-mode');

    removeItemFromStorage(itemToEdit.textContent);
    itemToEdit.classList.remove('edit-mode');
    itemToEdit.remove();
    isEditMode = false;
  }

  // Create item DOM element
  addItemToDOM(newItem);

  // Add item to local storage
  addItemToStorage(newItem);

  checkUI();

  itemInput.value = '';
}
```

You can not directly update an item in local storage, so we are basically removing the item and adding a new one. We are also removing the `edit-mode` class from the item element and setting the edit state to false.

To make sure that the UI is reset, we will call the `checkUI()` function. I am also going to add the following to the `checkUI()` function:

```
function checkUI() {
    itemInput.value = '';

    const items = itemList.querySelectorAll('li');

    if (items.length === 0) {
        clearBtn.style.display = 'none';
        itemFilter.style.display = 'none';
    } else {
        clearBtn.style.display = 'block';
        itemFilter.style.display = 'block';
    }

    formBtn.innerHTML = '<i class="fa-solid fa-plus"></i> Add Item';
    formBtn.style.backgroundColor = '#333';

    isEditMode = false;
}
```

Just to make sure we set the button back and set edit mode to false whenever we run this function.

# Prevent Duplicate Entries

---

In this lesson, we will add a check to prevent duplicate entries from being added to the shopping list.

Let's create a function that will check if the item already exists in the shopping list.

```
function checkIfItemExists(item) {
  const itemsFromStorage = getItemsFromStorage();
  return itemsFromStorage.includes(item);
}
```

We are using the `includes()` method to check if the item exists in the array of items from local storage.

Now let's use it in the `onAddItemSubmit()` function.

This is what the final function should look like:

```
function onAddItemSubmit(e) {
  e.preventDefault();

  const newItem = itemInput.value;

  // Validate Input
  if (newItem === '') {
    alert('Please add an item');
    return;
  }

  // Check for edit mode
  if (isEditMode) {
    const itemToEdit = itemList.querySelector('.edit-mode');
    removeItemFromStorage(itemToEdit.textContent);
    itemToEdit.classList.remove('edit-mode');
    itemToEdit.remove();
    isEditMode = false;
  } else {
    if (checkIfItemExists(newItem)) {
      alert('That item already exists!');
      return;
    }
  }

  // Create item DOM element
  addItemToDOM(newItem);

  // Add item to local storage
  addItemToStorage(newItem);
```

```
    checkUI();

    itemInput.value = '';
}
```

The reason that we check for edit mode is because if the item is being edited, we want to allow the user to use the same name for the item.

# Deploy To Netlify

---

When it comes to hosting, you have many many choices, especially for a simple HTML, CSS and JavaScript application. There are many services that will even host your project for free up to a limit (traffic and users).

One of my favorite hosting providers for websites and front-end applications is [Netlify](#). It is amazingly simple to deploy your project for free. I am not being sponsored by Netlify in any way. It is just a service that I use and recommend.

## Create a Netlify Account

If you don't already have a Netlify account, you can create one for free. You can use your GitHub account to sign up.

## Create a New Site

There are a few ways to push your project to the server. The easiest way is to simply push to GitHub, which I already showed you how to do. Then, you can connect your GitHub account to Netlify and it will automatically deploy your project.

Just click "Add new site" and select your GitHub project. After a few seconds, it will show you a congratulations message and your project will be live. You will have a strange URL, but you can easily add your own domain by following the instructions in the docs.

## Future deployments

When you make changes to your project, you can simply push to GitHub and Netlify will automatically deploy your project. By default, the `main` branch is used, but you can change that in the settings. Sometimes developers will use a `dev` branch for development and then merge to `main` when they are ready to deploy. Or they may have a branch named `deploy` or `production`. You can change the branch in the settings.

That's it!

# Loan Calculator Project Intro

---

This is another project that I created to get a better understanding of the DOM and how to manipulate it. This project is a loan calculator that will calculate the monthly payment, total payment, and total interest based on the loan amount, interest rate, and the number of years to pay off the loan.

The screenshot shows a web-based loan calculator application. At the top, a dark blue header bar contains the text "Loan Calculator". Below the header, there are three input fields for user input: "Loan Amount \$" with the value "5000", "Loan Interest %" with the value "3", and "Years To Pay" with the value "2". A large, dark blue "Calculate" button is positioned below these inputs. After clicking the button, the results are displayed in a section titled "Results", which includes three items: "Monthly Payment" (\$214.91), "Total Payment" (\$5157.84), and "Total Interest" (\$157.84). The entire application has a light blue background.

## Project Specifications

- Build the UI using Tailwind CSS
- Create a form to take in the loan amount, interest rate, and number of years to pay off the loan
- Calculate the monthly payment, total payment, and total interest
- Show a spinner for 1 second and then display the results
- Show a custom error alert box if there is an error

## Creating The UI

We will start from scratch and create the UI using the Tailwind CSS framework. We won't have to write any custom CSS because Tailwind has very low-level classes for just about any style that you can think of. You can skip the HTML/CSS part if you want and just focus on the JavaScript.

## The Calculation

You do not have to understand the math behind the calculation. I do not fully understand it myself. I just did some research to get the formula that we needed. Here are some links that I mention in the video that you can check out if you want to learn more about the math behind the calculation.

- [Kasasa Website Article](#)
- [Loan Calculator Formula](#)
- [Java GitHub Example](#)

# Tailwind UI

---

It's completely up to you if you want to follow along and type out the HTML and Tailwind classes for this project. If you do not, just copy the HTML below.

It is pretty simple. We are using the `Poppins` font. I added a custom Tailwind object in the head to initialize the font.

We set the `error`, `spinner` and `results` classes to `hidden` by default. We will show them when needed.

```
<!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" />
    <link rel="preconnect" href="https://fonts.googleapis.com" />
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
    <link
      href="https://fonts.googleapis.com/css2?
family=Poppins:wght@300;400;700&display=swap"
      rel="stylesheet"
    />
    <script src="https://cdn.tailwindcss.com"></script>
    <script src="js/script.js" defer></script>
    <script>
      tailwind.config = {
        theme: {
          extend: {
            fontFamily: {
              sans: ['poppins', 'sans-serif'],
            },
          },
        },
      };
    </script>
    <title>Loan Calculator</title>
  </head>
  <body class="bg-blue-100 px-5 relative h-full overflow-hidden">
    <!-- Alert Box -->
    <div
      id="error"
      class="absolute bottom-10 left-1/2 bg-red-600 text-white p-5 rounded-xl
shadow-xl border-2 border-white text-center text-2xl transform -translate-x-1/2
translate-y-[-1000px] transition"
    >
      This is an error
    </div>
```

```
<div class="max-w-3xl m-auto mt-20 bg-white rounded-xl shadow-xl">
  <header class="bg-slate-500 py-10 rounded-t-xl">
    <h1 class="text-3xl text-center text-white">Loan Calculator</h1>
  </header>
  <main class="p-10">
    <form id="loan-form">
      <div class="my-4">
        <label
          for="loan-amount"
          class="inline-block mb-2 text-xl text-gray-500"
          >Loan Amount $</label>
        <br>
        <input
          type="number"
          id="loan-amount"
          class="border-2 rounded-lg p-2 w-full focus:outline-line
focus:ring-2 focus:ring-blue-200 focus:border-transparent"
          placeholder="5000"
        />
      </div>

      <div class="my-4">
        <label
          for="loan-interest"
          class="inline-block mb-2 text-xl text-gray-500"
          >Loan Interest %</label>
        <br>
        <input
          type="number"
          id="loan-interest"
          class="border-2 rounded-lg p-2 w-full focus:outline-line
focus:ring-2 focus:ring-blue-200 focus:border-transparent"
          placeholder="3"
        />
      </div>

      <div class="my-4">
        <label
          for="loan-years"
          class="inline-block mb-2 text-xl text-gray-500"
          >Years To Pay</label>
        <br>
        <input
          type="number"
          id="loan-years"
          class="border-2 rounded-lg p-2 w-full focus:outline-line
focus:ring-2 focus:ring-blue-200 focus:border-transparent"
          placeholder="2"
        />
      </div>
    </form>
  </main>
</div>
```

```
        class="bg-slate-500 text-white px-4 py-3 w-full rounded-lg
hover:bg-slate-600"
      >
    Calculate
  </button>
</div>
</form>

<section>
  

  <div id="loan-results" class="hidden">
    <h2 class="text-2xl text-center text-gray-500 mb-4">Results</h2>

    <div class="flex justify-between mb-4 bg-gray-100 p-3 rounded-lg">
      <span>Monthly Payment</span>
      <span id="monthly-result">$5000</span>
    </div>

    <div class="flex justify-between mb-4 bg-gray-100 p-3 rounded-lg">
      <span>Total Payment</span>
      <span id="payment-result">$500000</span>
    </div>

    <div class="flex justify-between mb-4 bg-gray-100 p-3 rounded-lg">
      <span>Total Interest</span>
      <span id="interest-result">$2000</span>
    </div>
  </div>
</section>
</main>
</div>
</body>
</html>
```

# Get Calculation Values

---

In this lesson, we will bring in the DOM elements that we will be working with and getting the values that we need to do the loan calculation.

Let's start off with selecting what we need from the DOM.

```
const loanForm = document.getElementById('loan-form'),
    amountInput = document.getElementById('loan-amount'),
    interestInput = document.getElementById('loan-interest'),
    yearsInput = document.getElementById('loan-years'),
    loanResults = document.getElementById('loan-results'),
    monthlyResult = document.getElementById('monthly-result'),
    paymentResult = document.getElementById('payment-result'),
    interestResult = document.getElementById('interest-result'),
    errMsg = document.getElementById('error');
```

Let's add an event listener on to the form and call a function and validate the input.

```
loanForm.addEventListener('submit', onLoanFormSubmit);
```

```
function onLoanFormSubmit(e) {
  e.preventDefault();

  // Validate input
  if (
    amountInput.value === '' ||
    interestInput.value === '' ||
    yearsInput.value === ''
  ) {
    alert('Please fill in all fields!');
    return;
  }
}
```

Now let's get the three values that we need to make our calculation. That is the principal, monthly interest and the number of payments (months)

```
function onLoanFormSubmit(e) {
  e.preventDefault();

  // Validate input
  if (
```

```
amountInput.value === '' ||  
interestInput.value === '' ||  
yearsInput.value === ''  
) {  
  alert('Please fill in all fields!');  
  return;  
}  
  
const principal = parseFloat(amountInput.value);  
const monthlyInterest = parseFloat(interestInput.value) / 100 / 12;  
const numberOfPayments = parseFloat(yearsInput.value) * 12;  
  
console.log(principal, monthlyInterest, numberOfPayments);  
}
```

In the next lesson, we will make the calculation.

# Calculate Results

---

In this lesson, we will take the principal, monthly interest and number of payments and calculate the monthly payment and total interest.

Let's create the function that will calculate the results.

```
function calculateResults(principal, monthlyInterest, numberOfPayments) {  
    const x = Math.pow(1 + monthlyInterest, numberOfPayments);  
    // Get monthly payment  
    const monthly = ((principal * x * monthlyInterest) / (x - 1)).toFixed(2);  
    // Get total payment  
    const total = (monthly * numberOfPayments).toFixed(2);  
    // Get total interest  
    const interest = (monthly * numberOfPayments - principal).toFixed(2);  
  
    // console.log(monthly, total, interest);  
  
    if (isNaN(monthly)) {  
        alert('Please check your numbers');  
        return;  
    } else {  
        console.log('Success');  
    }  
}
```

Make sure that you call the function in the `onLoanFormSubmit` function.

```
function onLoanFormSubmit(e) {  
    //...  
  
    calculateResults(principal, monthlyInterest, numberOfPayments); // -----  
    ADD THIS LINE  
}
```

Again, you don't really need to understand the formula. You just need to know where to look to find what you need and then implement it so that you plug the correct values into it.

Here are 2 resources that helped me understand the formula:

- <https://www.kasasa.com/blog/how-to-calculate-loan-payments-in-3-easy-steps>
- <https://gist.github.com/letrongtanbs/d29354da30f12784bc8453af4e4fb6ff>

After calculating the results, we need to display them on the page, we checked the `monthly` value and made sure it was a number. If it is, we are just console logging success. In the next lesson, we will show the spinner and the results on the page.



# Display Spinner and Results

---

Now that we have the results, we need to display them on the page. Let's create a function called `addResultsToDOM()`.

```
function addResultsToDOM(monthly, total, interest) {  
    monthlyResult.innerText = '$' + monthly;  
    paymentResult.innerText = '$' + total;  
    interestResult.innerText = '$' + interest;  
}
```

Now, call the function in the `calculateResults()` function.

```
function calculateResults(principal, monthlyInterest, numberOfPayments) {  
    //...  
  
    if (isNaN(monthly)) {  
        alert('Please check your numbers');  
        return;  
    } else {  
        addResultsToDOM(monthly, total, interest);  
    }  
}
```

This function will put the correct number results where they need to go, but the main `#results` element is still hidden because we have the Tailwind class of `hidden` on it. I want to show a spinner for 1 second and then display the results. So let's create a function called `showSpinnerAndResults()`.

```
function showSpinnerAndResults(seconds) {  
    const spinner = document.getElementById('spinner');  
    spinner.style.display = 'block';  
  
    setTimeout(() => {  
        spinner.style.display = 'none';  
        loanResults.style.display = 'block';  
    }, seconds * 1000);  
}
```

Here, we are taking in the number of seconds to show the spinner and we are using the `setTimeout()` function to show the spinner for that amount of seconds and then hide it and show the results.

Now, let's call this function in the `addResultsToDOM()` function. You could also call it in the `calculateResults()` function if you want.

```
function addResultsToDOM(monthly, total, interest) {  
    //...  
  
    showSpinnerAndResults(1); // ----- ADD THIS LINE  
}
```

At the moment, if we try and use the calculator again, it will still show the results from the previous calculation and the spinner. Let's fix that by adding a `resetUI()` function.

```
function resetUI() {  
    loanResults.style.display = 'none';  
    monthlyResult.innerText = '';  
    paymentResult.innerText = '';  
    interestResult.innerText = '';  
}
```

We will call the `resetUI()` function in the event handler before we do anything

```
function onLoanFormSubmit(e) {  
    e.preventDefault();  
  
    resetUI(); // ----- ADD THIS LINE  
  
    //...  
}
```

# Error Alert Animation

---

Now that we have everything working, I want to make the error handling a little nicer. Instead of using the standard `alert()` function, I want to use a Tailwind CSS animation and custom error box to show the error message. Let's create a new function called `showError()`.

```
function showError(msg) {
  errMsg.style.transform = 'translateY(-450px) translateX(-50%)';
  errMsg.innerText = msg;

  setTimeout(() => {
    errMsg.style.transform = 'translateY(-1000px) translateX(-50%)';
  }, 3000);
}
```

We had already created the error box using Tailwind. We just set the `transform` property to move the box up out of view. Now we are setting the `transform` property to move the box back into view and we are setting the `innerText` property to the error message. We are also using the `setTimeout()` function to move the box back out of view after 3 seconds.

We want to call this function in the event handler if the user forgets to pass in a value and also in the `calculateResults()` function if the user enters invalid numbers.

```
function onLoanFormSubmit(e) {
  e.preventDefault();

  resetUI();

  // Validate input
  if (
    amountInput.value === '' ||
    interestInput.value === '' ||
    yearsInput.value === ''
  ) {
    showError('Please fill in all fields!'); // ----- ADD THIS LINE
    return;
  }

  const principal = parseFloat(amountInput.value);
  const monthlyInterest = parseFloat(interestInput.value) / 100 / 12;
  const numberOfPayments = parseFloat(yearsInput.value) * 12;

  calculateResults(principal, monthlyInterest, numberOfPayments);
}
```

```
function calculateResults(principal, monthlyInterest, numberOfPayments) {  
    const x = Math.pow(1 + monthlyInterest, numberOfPayments);  
    // Get monthly payment  
    const monthly = ((principal * x * monthlyInterest) / (x - 1)).toFixed(2);  
    // Get total payment  
    const total = (monthly * numberOfPayments).toFixed(2);  
    // Get total interest  
    const interest = (monthly * numberOfPayments - principal).toFixed(2);  
  
    // console.log(monthly, total, interest);  
  
    if (isNaN(monthly)) {  
        showError('Please check your numbers'); // ----- ADD THIS LINE  
        return;  
    } else {  
        addResultsToDOM(monthly, total, interest);  
    }  
}
```

# Vercel Deploy

---

Vercel is another incredible hosting service with a very generous free tier. It's also very easy to deploy to Vercel. It is almost an identical process to what we did with Netlify in the last section. You simply select the GitHub repo.

## Push to GitHub

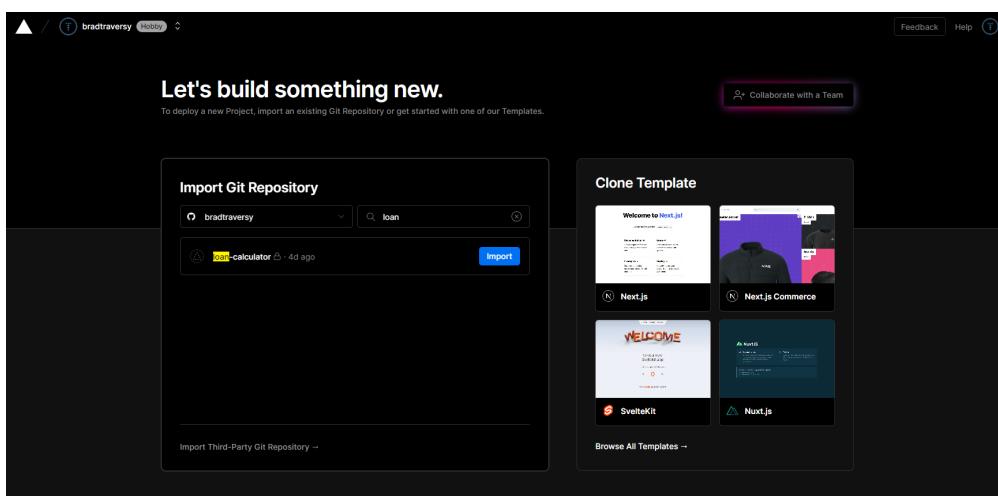
First, we need to push our project to GitHub. If you haven't already, create a new repository on GitHub and push your project to it. We went over how to do this back in the Shopping List project.

## Log In to Vercel

You can log in to Vercel using your GitHub account.

Once you log in, click "Add new" and select "project" from the dropdown.

Simply search for your repo on GitHub and select it.



After a few seconds, your project will be deployed and you will see the URL to access it.

Just do a quick test to make sure everything is working.

You can continue to make changes to your project and push them to GitHub. Vercel will automatically deploy the changes.

You also have the option to connect your project to a custom domain. You can do this by clicking on the "Domains" tab and then clicking "Add Domain".

# Thread Of Execution

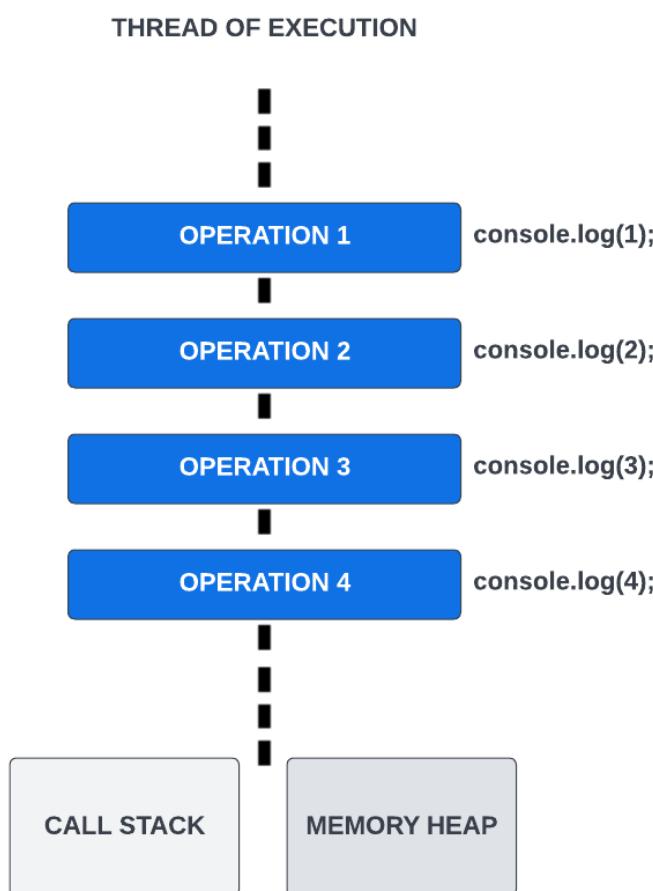
---

## JavaScript is Synchronous

So, we have covered a lot of the fundamentals of JavaScript and how to work with the Document Object Model (DOM). Now we are going to start to get into asynchronous JavaScript. Before we do that though, it is important to understand how JavaScript actually works and to understand that at its core, JavaScript is a **synchronous** and **single-threaded** language. It has **asynchronous** capabilities, which we will be looking at in this section and others, but it is not **asynchronous** by default.

## JavaScript is Single-Threaded

You already have a leg up on this stuff, because you know about the **execution context**. We talked about that earlier in the course. The **execution context** contains the currently running code and everything that aids in its execution. It also runs on what we call a **thread of execution**. The **thread of execution** is a single thread that **runs one line of code at a time**. This is why JavaScript is a **single-threaded** language and is **synchronous**. Some languages have multiple threads that can run code at the same time. JavaScript does not.



If we look at this image, it shows the **thread of execution**, which is a single sequential flow of control and each operation happens one after the other. The second `console.log()` will not run until the first one is finished. The third will not run until the second is finished. This is **synchronous** behavior. The thread also includes the call stack and memory heap, which we've already talked about.

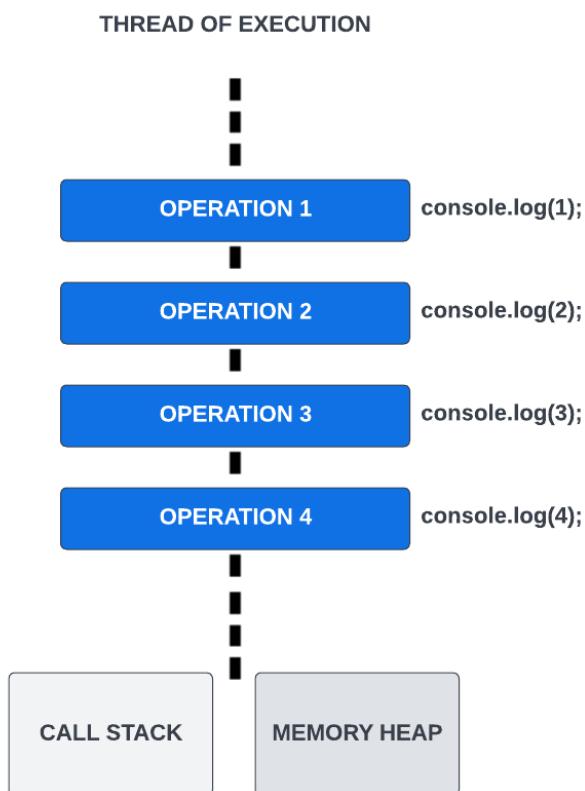
So that's really all I want you to understand up to this point is that everything is executed line by line. In the next video, I'll show you how it is possible to run code asynchronously with the help of web APIs.

# How Asynchronous JavaScript Works: The Task Queue & Event Loop

---

Alright, so I just want to warn you that I'm going to throw a lot of information at you in this video because I'm basically going to explain everything that happens behind the scenes when we create asynchronous code. If this confuses you, don't worry about it. You'll learn more about the behind the scenes stuff as well as the syntax throughout the course.

So you know that JavaScript is a **synchronous** and it's a **single-threaded** language. It runs one line of code at a time. Just to remind you, we can look at the image we talked about in the last video.

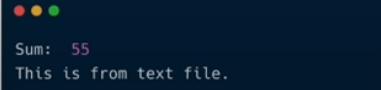


These operations run line by line and each one has to wait for the last one to complete before executing. Where we run into issues is where one of these operations takes a while. Something like fetching data from a server or if you're using Node.js, maybe you're reading from or writing to a file. That can take a while, and by a while, I mean usually a few seconds or even a few milliseconds, but that's a long time in programming.

## Blocking & Non-Blocking Code

When an operation takes a while and basically holds up the line, that is called **blocking** code or operations. It blocks the flow of the program until it completes. **Non-blocking** code refers to code that does not block execution.

Let's look at the following code. Don't worry if you don't understand everything, but both pieces of code are reading a file using Node.js file system methods and then calculating the sum of numbers from 1 to 10.

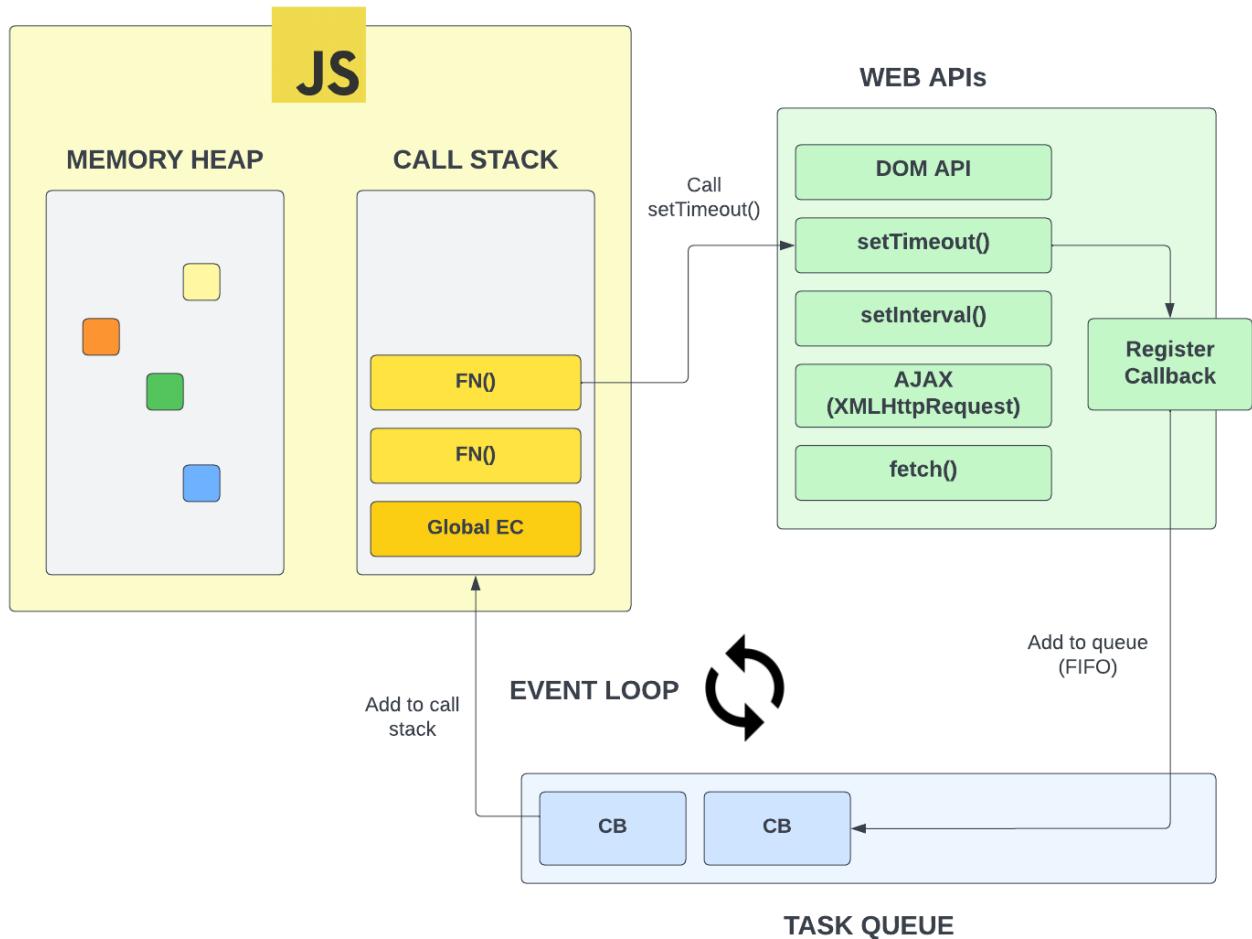
BLOCKING CODE	NON-BLOCKING CODE
<pre>const fs = require('fs'); const filepath = 'text.txt';  // Reads a file in a synchronous and blocking way const data = fs.readFileSync(filepath, {encoding: 'utf8'});  // Prints the content of file console.log(data);  // This section calculates the sum of numbers from 1 to 10 let sum = 0; for(let i=1; i&lt;=10; i++){     sum = sum + i; }  // Prints the sum console.log('Sum: ', sum);</pre>	<pre>const fs = require('fs'); const filepath = 'text.txt';  // Reads a file in a asynchronous and non-blocking way fs.readFile(filepath, {encoding: 'utf8'}, (err, data) =&gt; {     // Prints the content of file     console.log(data); });  // This section calculates the sum of numbers from 1 to 10 let sum = 0; for(let i=1; i&lt;=10; i++){     sum = sum + i; }  // Prints the sum console.log('Sum: ', sum);</pre>
	

On the left, we are using the `readFileSync()` method. This is a blocking method. It will read the file and then move on to calculate the sum. In the console, we see the file contents and then the sum.

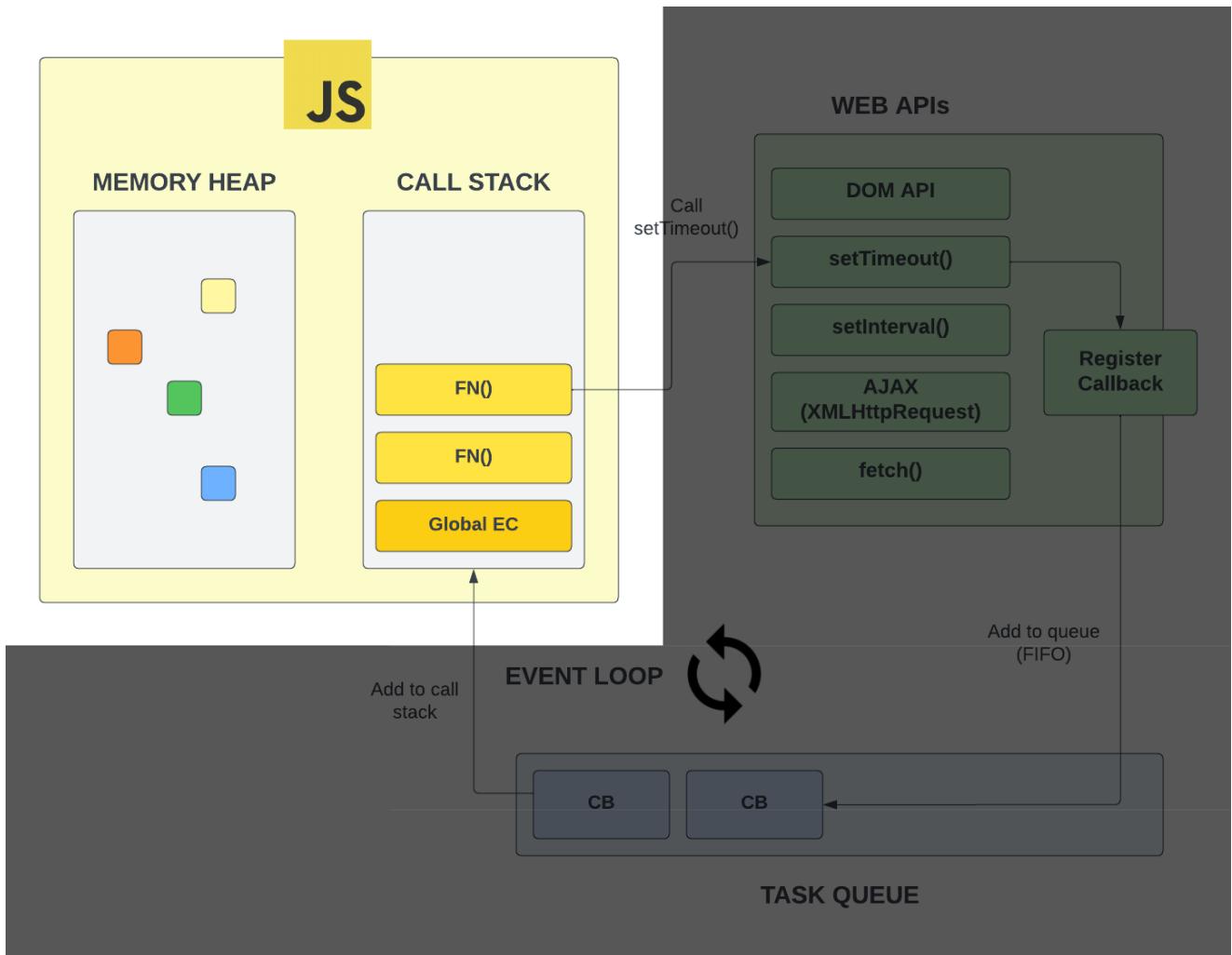
On the right, we are using the `readFile()` method. This is a non-blocking, asynchronous method. It will read the file but it will not block the execution of the program by making it wait. The way it works, is it takes in a callback function and when the file is read, it will execute the callback function. In the console, we see the sum first and then the file contents, because we did not have to wait for the file to be read before calculating the sum.

Now `readFile()` is not available to us in the browser, but there are a lot of APIs that are available that work in a similar asynchronous way.

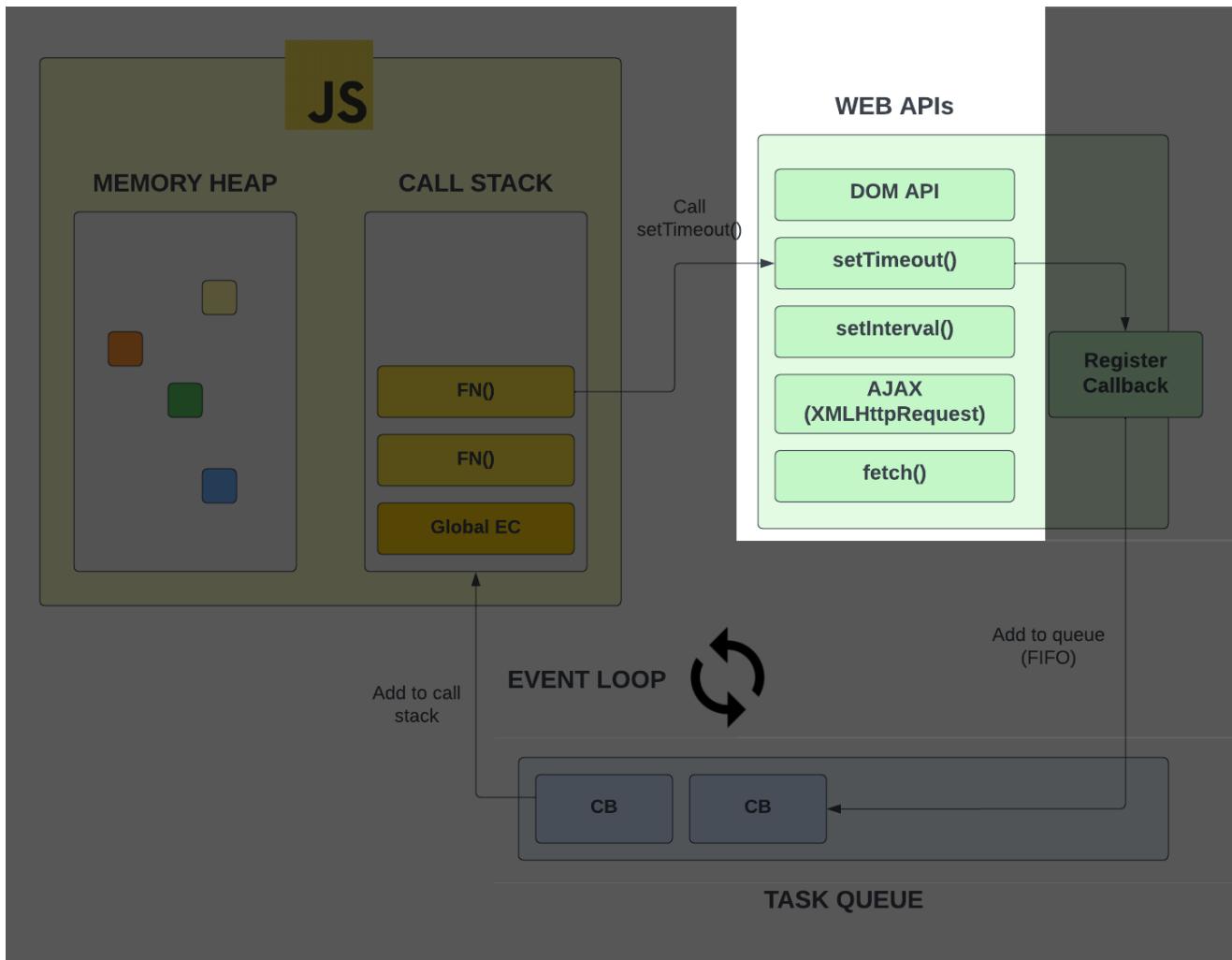
I'm going to show you a diagram to try and explain what happens when we write asynchronous code using these web APIs.



The yellow box represents the **JavaScript engine**. This is the part of the browser that executes our JavaScript code. This is where our **call stack** is that executes all of our functions, etc. This is also where the **memory heap** is, which is where all of our variables and objects are stored.



Outside of that, in the green box, we have a bunch of web APIs that are accessible to us via the browser and the `global object`. Remember, this is created during the *creation phase* of the `global execution context`.



If we go to the browser console and type in `window` and hit enter, you will see `setTimeout()` and `setInterval()` and a bunch of other functions that allow us to do things asynchronously.

```

▶ scroll: f scroll()
▶ scrollBy: f scrollBy()
▶ scrollTo: f scrollTo()
▶ scrollX: 0
▶ scrollY: 0
▶ scrollbars: BarProp {visible: true}
▶ self: Window {window: Window, self: Window, ...
▶ sessionStorage: Storage {length: 0}
▶ setInterval: f setInterval()
▶ setTimeout: f setTimeout()
▶ showDirectoryPicker: f showDirectoryPicker()
▶ showOpenFilePicker: f showOpenFilePicker()
▶ showSaveFilePicker: f showSaveFilePicker()
▶ speechSynthesis: SpeechSynthesis {pending: false}
  status: ""
▶ statusbar: BarProp {visible: true}
▶ stop: f stop()
▶ structuredClone: f structuredClone()
▶ styleMedia: StyleMedia {type: 'screen'}
▶ toolbar: BarProp {visible: true}
▶ top: Window {window: Window, self: Window, ...
▶ trustedTypes: TrustedTypePolicyFactory {empty}
▶ visualViewport: VisualViewport {offsetLeft: 0}
▶ webkitCancelAnimationFrame: f webkitCancelAnimationFrame()
▶ webkitRequestAnimationFrame: f webkitRequestAnimationFrame()
▶ webkitRequestFileSystem: f webkitRequestFileSystem()
▶ webkitResolveLocalFileSystemURL: f webkitResolveLocalFileSystemURL()

```

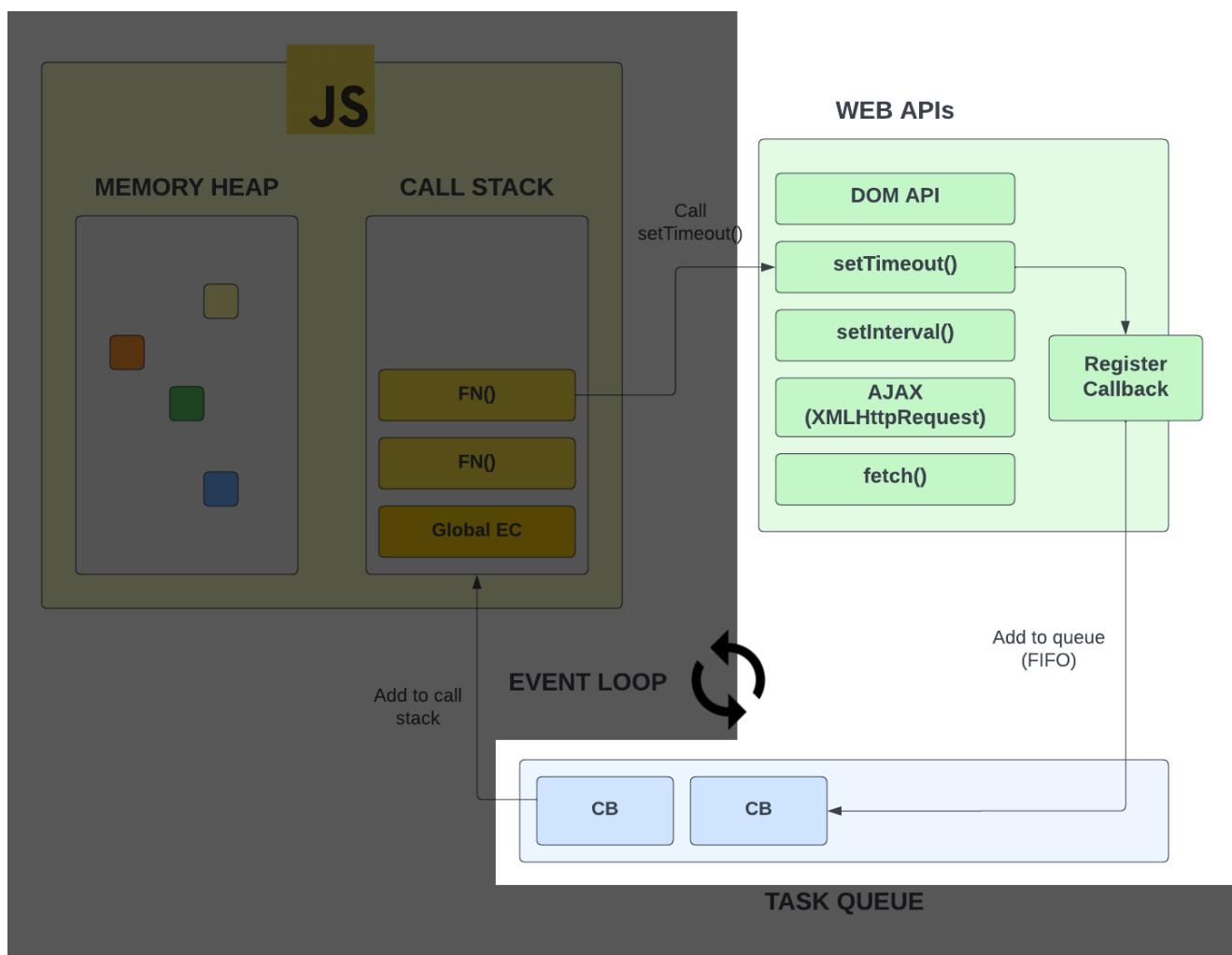
In addition to `setTimeout()` and `setInterval()`, we have the whole DOM API. We select elements and put event listeners on them. That's another API we have available to us. It's not part of JavaScript. When you use Node.js, you don't have access to the document object, because there is no DOM or browser in Node.js. But as you saw in the code examples, in Node, you do have access to the filesystem API, which allows you to read and write files.

`fetch()` is another API that we have access to. It allows us to make HTTP requests. We'll be working with the fetch API quite a bit to send requests to APIs and services. This is something that a front-end developer has to know how to use, and we'll get to that soon enough.

Right now, I just want you to understand that this is stuff provided to us by the browser environment. Now, let's talk about how they work with the JavaScript engine, which is inherently synchronous.

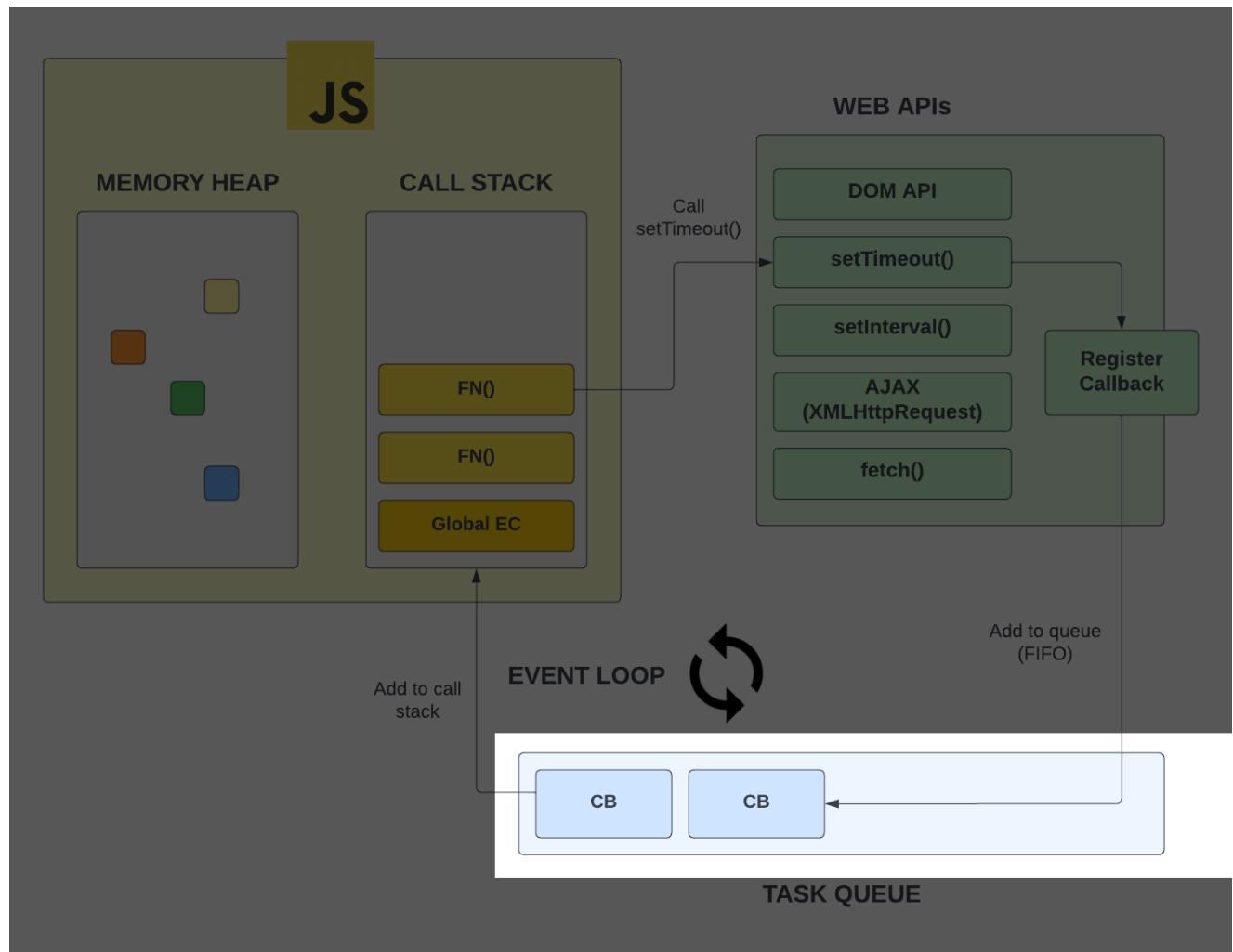
## Task Queue

So we know these APIs are separate from the JavaScript engine. We know that we have the ability to go off and do something while the script continues to execute and when that something is done, we fire off a callback function. This is possible because of something called the `task queue`.



When we call an asynchronous function such as `setTimeout()`. In the diagram it's being called from a function, but it could just as well be from the global scope. When we call it, we added a callback function as an argument. It then registers that callback and it gets added to what we call a `task queue`. This is a queue of callbacks that are waiting to be added to the call stack and executed.

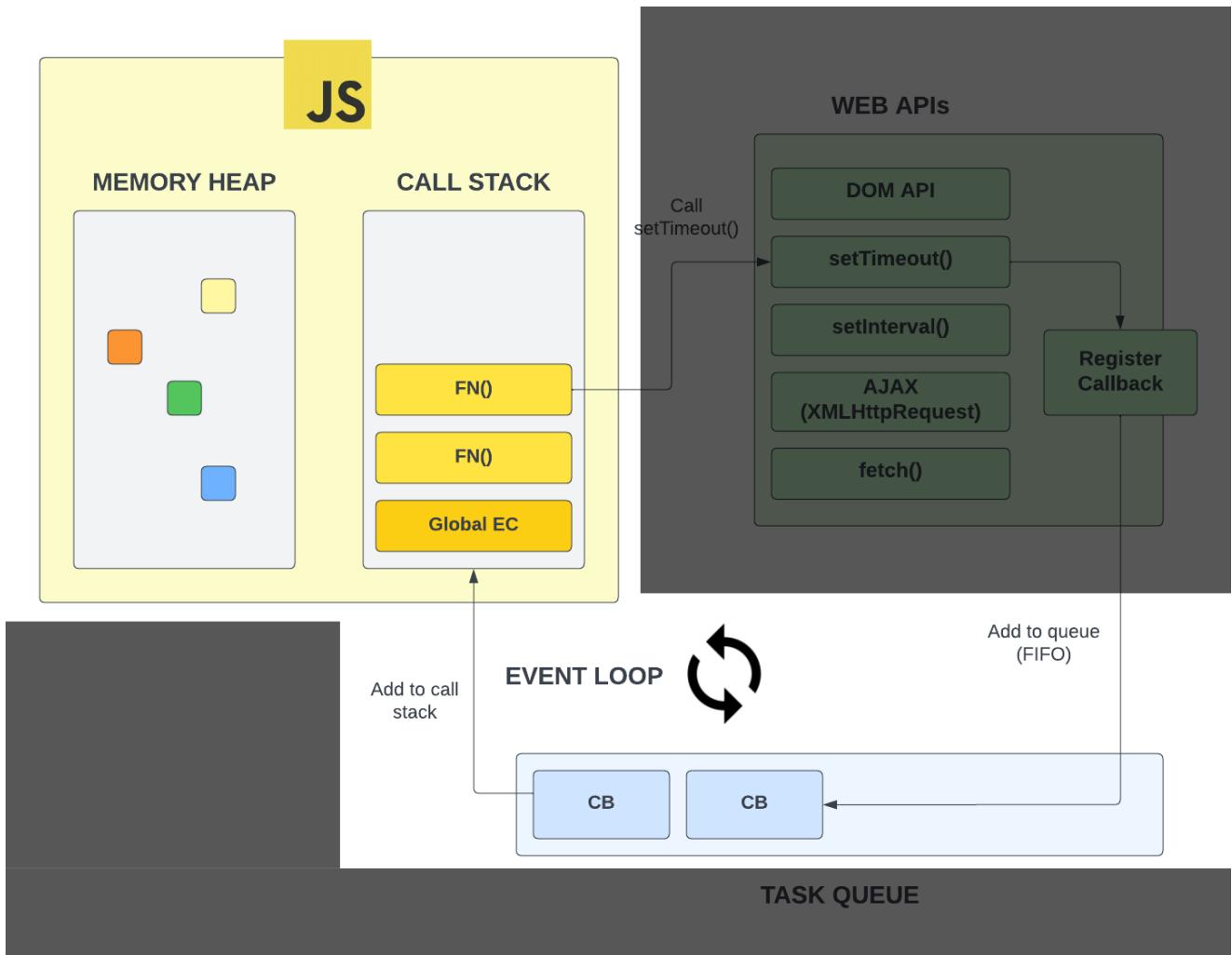
A **queue** is a data structure that follows the **first in, first out** principle. This means that the first item that is added to the queue will be the first item that is removed from the queue. In our case, the first callback that is added to the task queue will be the first callback that is executed.



Remember, we already looked at a **stack**, which follows the **last in, first out** principle. This means that the last item that is added to the stack will be the first item that is removed. The **call stack** is an example. So both **queues** and **stacks** are data structures that are used in programming.

## Event Loop

Any callbacks that are in the queue, have to be put on the stack to be executed. This is where the **event loop** comes in. The event loop is a process that checks the call stack and the task queue. If the call stack is empty, it will take the first callback in the task queue and add it to the call stack to be executed. When we create event listeners with `addEventListener()`, we are also adding callbacks to the task queue.



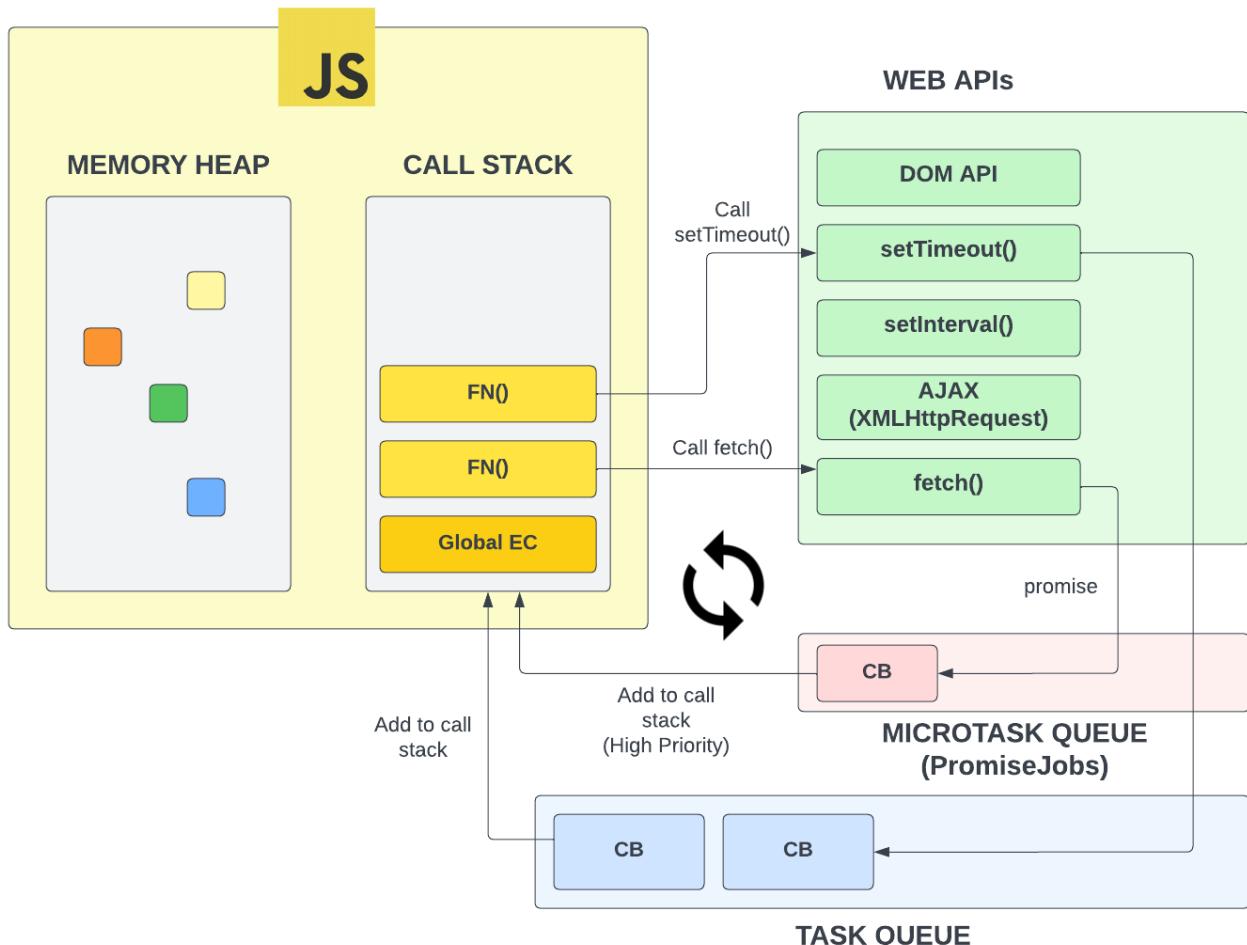
You can think of the event loop like one of those revolving doors at the mall. It's constantly checking to see if the door is open and if it is, it will let people in. If it's not, it will keep spinning until it is. In this case, we're dealing with functions instead of people.

Now this is also how the event loop and the task queue work within Node.js. There are different APIs and functions available, but it all works the same under the hood. Node even uses the same V8 engine that Chrome uses.

## Promises

Now just to confuse you a little bit more, with things like event listeners and `setTimeout()`, callbacks get added to the task queue. When we work with let's say, the `fetch` API, we get a `Promise` object back, which work a little differently.

Promises are objects that represent the eventual completion or failure of an asynchronous operation. They are a way to handle asynchronous code in a more elegant way than using callbacks. We'll be working with promises a lot in the future. The reason I'm mentioning them now is because promises create what are called `PromiseJobs` or the v8 engine calls them `microtasks`.



Microtasks are callbacks that are added to the `microtask queue`. It works in a similar way to the task queue, but it's a separate queue and it's checked before the task queue. It has a higher priority. There are also something called `observers` that are added to the microtask queue. We'll get to those later.

Alright, I know this is confusing as hell, but the truth is, you don't need to understand all of this right now. In fact, I know senior developers that don't know some of this stuff, but I wanted you to get a head start on what is actually happening under the hood.

# setTimeout() and clearTimeout()

In the last video, we looked at a diagram that represented what actually happens under the hood when we use some of the asynchronous APIs that the browser or Node.js offers. In this video, we're going to look at `setTimeout()`, which is really helpful for doing something after a certain amount of time. We're also going to look at `clearTimeout()`, which is a function that we can use to cancel a timeout.

```
setTimeout(function () {
  console.log('Hello from callback');
}, 2000);

console.log('Hello from the top-level code');
```

When we run this code, we see the following output.

```
Hello from the top-level code
Hello from callback
```

The reason for this is because `setTimeout()` is given a number of milliseconds as the second argument and then it waits that amount of time and fires off. It does not block the code though, so we see the top-level `console.log` first and then the callback `console.log`.

Let's go ahead and change the 2000 milliseconds to `0`. What do you think is going to happen?

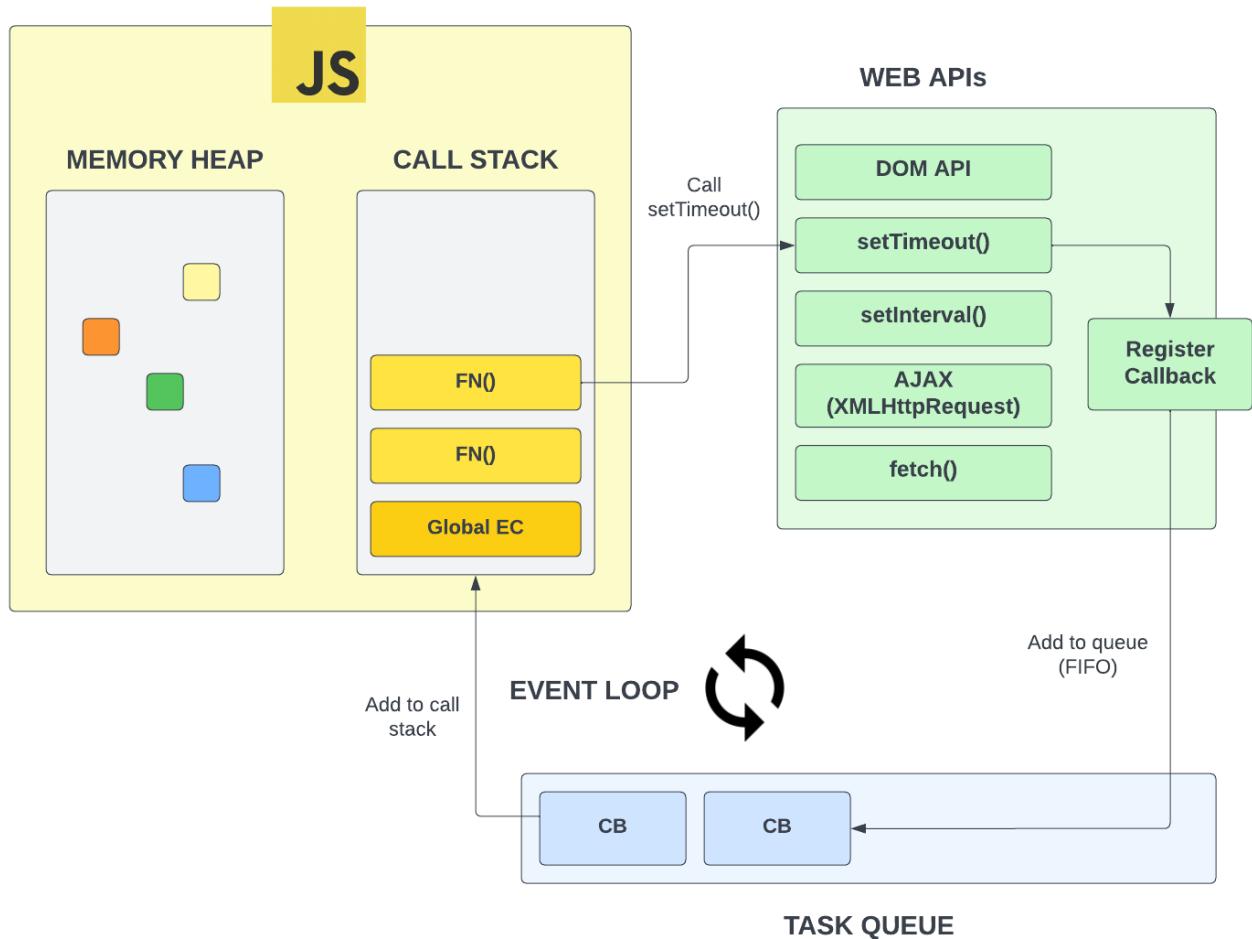
```
setTimeout(function () {
  console.log('Hello from callback');
}, 0);

console.log('Hello from the top-level code');
```

When we run this code, we see the following output.

```
Hello from the top-level code
Hello from callback
```

You may have thought that the callback would execute first since we set the timeout to 0. Remember, that callback gets put on to the `task queue` and then it waits for the call stack to be empty. So, the callback is not going to execute until the call stack is empty, even if we set the timeout to 0. Just to remind you, here is the diagram that we looked at in the last video.



You may want to use this to change something in the DOM after a certain amount of time. We did this in the loan calculator project to show a spinner for 1 second before showing the results.

Let's make the **h1** tag change after a few seconds.

```
setTimeout(() => {
  document.querySelector('h1').textContent = 'Hello from callback';
}, 3000);
```

We could also put that in a separate function and then call it.

```
function changeText() {
  document.querySelector('h1').textContent = 'Hello from callback';
}

setTimeout(changeText, 3000);
```

**clearTimeout()**

In addition to `setTimeout()`, we also have `clearTimeout()`. This is a function that we can use to cancel a timeout.

Let's create a button that will cancel the timeout to change the text.

```
<button id="cancel">Cancel Text Change</button>
```

In order to know which timeout to cancel, we need to store the id that is returned from `setTimeout()`.

```
const timerId = setTimeout(changeText, 3000);
```

Now, we can create an event listener for the button that will call `clearTimeout()`.

```
document.querySelector('#cancel').addEventListener('click', () => {
  clearTimeout(timerId);
  console.log('Timer Cancelled');
});
```

# setInterval() and clearInterval()

`setInterval()` is used to run a specific callback function and repeat it at a set interval. The number of milliseconds passed to the function is the amount of time to wait between each function call. Let's look at a simple example

```
const intervalID = setInterval(myCallback, 1000);

function myCallback() {
  console.log(a, Date.now());
}
```

This will log the timestamp every second.

We can also pass in parameters

```
const intervalID = setInterval(myCallback, 1000, 'Hello');

function myCallback(a) {
  console.log(a, Date.now());
}
```

## clearInterval()

To clear or stop the interval, we can use `clearInterval()` and pass in the interval ID

```
clearInterval(intervalID);
```

Let's create a script to change the body background color every second. We will have buttons to start and stop it.

```
let intervalID;

function startChange() {
  if (!intervalID) {
    intervalID = setInterval(changeBackground, 1000);
  }
}

function changeColor() {
  if (document.body.style.backgroundColor !== 'black') {
    document.body.style.backgroundColor = 'black';
    document.body.style.color = 'white';
  }
}
```

```
    } else {
        document.body.style.backgroundColor = 'white';
        document.body.style.color = 'black';
    }
}

function stopChange() {
    clearInterval(intervalID);
}

document
    .getElementById('start')
    .addEventListener('click', startChange);
document.getElementById('stop').addEventListener('click', stopChange);
```

We could make it a random color by generating a hex value

```
function changeRandomColor() {
    const randomColor = Math.floor(Math.random() * 16777215).toString(16);
    document.body.style.backgroundColor = `#${randomColor}`;
}
```

# Callbacks

---

Let's touch on callback functions a bit more. A callback is simply a function that is passed into another function as an argument and executed within the function that it was passed into.

We have already used callbacks quite a few times in this course. For example, we've used them with `addEventListener()` and `setTimeout()`.

Just because a function takes in a callback does not mean that it is asynchronous. It is a way to handle asynchronous code, such as we saw with `setTimeout()`, where the callback is placed in the `task queue` and then it waits for the call stack to be empty before it is executed. But, we also used callbacks with high order array methods like `forEach()` and `map()`. These are not asynchronous. The callbacks are executed immediately in this case.

## Callback Recap

`addEventListener()` is a good example of a function that takes in a callback. Let's look at the following code.

```
function toggle(e) {
  const bgColor = e.target.style.backgroundColor;

  if (bgColor === 'red') {
    e.target.style.backgroundColor = '#333';
  } else {
    e.target.style.backgroundColor = 'red';
  }
}

document.querySelector('button').addEventListener('click', toggle);
```

Notice, when we pass in a callback, we do not use parentheses. We just pass in the name of the function. Parentheses are used when we want to execute the function. The function is executed within the `addEventListener()` function at a later time (when the event occurs).

If we were to add parentheses, the function would execute immediately and the callback would not be passed into the `addEventListener()` function.

```
function toggle(e) {
  // Add this
  console.log('toggle ran...');

  const bgColor = e.target.style.backgroundColor;

  if (bgColor === 'red') {
    e.target.style.backgroundColor = '#333';
```

```

} else {
  e.target.style.backgroundColor = 'red';
}
}

document.querySelector('button').addEventListener('click', toggle());

```

If you run the code above, you will see the console log right away. You will also get an error, because it can't read the event object.

## Implementing Callbacks

Until you are writing advanced JavaScript, you probably will not have too many times where you will actually create a function that takes in a callback, but let's try it, just to see how it works.

Let's create a couple posts inside of an array:

```

const posts = [
  { title: 'Post One', body: 'This is post one' },
  { title: 'Post Two', body: 'This is post two' },
];

```

Now I am going to create two functions. One to create a new post and one to get all posts. The `createPost()` function is going to create after two seconds and the `getPosts()` function is going to get all posts after one second.

```

function createPost(post) {
  setTimeout(() => {
    posts.push(post);
  }, 2000);
}

function getPosts() {
  setTimeout(() => {
    posts.forEach(function (post) {
      const div = document.createElement('div');
      div.innerHTML = `${post.title} - ${post.body}`;
      document.querySelector('#posts').appendChild(div);
    });
  }, 1000);
}

createPost({ title: 'Post Three', body: 'This is post three' });

getPosts();

```

We ran both functions, yet we only see the initial two posts. The third never shows up because the posts already showed up after one second then the `createPost()` function ran after two seconds. We need to use a callback to fix this.

We can use a callback to fix this. We can pass in a callback to the `createPost()` function and then call the `getPosts()` function inside of the callback.

```
function createPost(post, cb) {
  setTimeout(() => {
    posts.push(post);
    cb();
  }, 2000);
}

function getPosts() {
  setTimeout(() => {
    posts.forEach(function (post) {
      const div = document.createElement('div');
      div.innerHTML = `<strong>${post.title}</strong> - ${post.body}`;
      document.querySelector('#posts').appendChild(div);
    });
  }, 1000);
}

createPost({ title: 'Post Three', body: 'This is post three' }, getPosts);
```

Now, when we run the code, we see all three posts. The `createPost()` function is executed and then the callback is executed.

Where we can get in trouble with callbacks is when we have multiple callbacks nested within each other. This is called callback hell. To address this, we can instead use something called `promises`, which we will get into a little later.

In the next lesson, I want to get into HTTP requests. Making HTTP requests will give us more realistic examples of asynchronous code, rather than just using `setTimeout()`.

# HTTP Requests

---

Alright, so now we're starting to get to the fun stuff. The JavaScript code that we have been writing in this course has all been **front-end** code, meaning it is being run in the browser on the client machine as opposed to on the server or the **back-end**. You can write server-side JavaScript with Node.js, but that's way beyond what we're learning now. However, there will be times when you'll want to fetch some data from some kind of API on a server, or send some data to a server from a form. This is where **HTTP** requests come in. The server could be either your own back-end that you wrote with Node.js or another language like Python or Ruby, or it could be a public third-party API. We'll be working with a couple third-party APIs in this course. We interact with APIs by making HTTP requests.

## What is an HTTP request?

**HTTP** stands for **HyperText Transfer Protocol**. It is the protocol that is used to send and receive data on the web. **HTTP** is a client-server protocol, which means requests are initiated by the client, usually the web browser, and the server will respond with a response. Every time that you go to a web page, your browser is making a request to a server. The server is responding with things like HTML, CSS, and JavaScript files. It may also respond with images, videos, and other media as well as data formatted as JSON or XML. When we fetch data using JavaScript, it is usually JSON data. If you go to your browser's developer tools **network** tab, you can see the requests that are being made to the server. You can also see the responses that are being sent back.

## Making HTTP requests

You can make HTTP requests right from your JavaScript code without having to refresh the page. There are a few ways to do this. The old way of doing this was to use **AJAX**, which stands for **Asynchronous JavaScript and XML**. You can still use **AJAX** to make HTTP requests, but it's not the most common way to do it anymore. The most common way to make HTTP requests is to use the **fetch()** API. The **fetch()** API is a promise-based API, which means that it returns a promise that will resolve with the response from the server. We're going to get into **fetch()** and **promises**, but first, I'll show you how to make an HTTP request with **AJAX** and the **XMLHttpRequest** object. You probably won't use this method much, but it's good to know how it works.

## JSON responses

When you make an HTTP request from your JavaScript code, the server will respond with some data. The data can be in a variety of formats, but the most common format is **JSON**. We talked about **JSON** earlier, but let's have a little refresher. **JSON** stands for **JavaScript Object Notation**. It is a way to represent data in a format that is easy for both humans and machines to read and write. We used to use XML (Which is why the X in **AJAX** stands for XML), but it's become more common to use **JSON** because it's easier to work with. We'll be fetching JSON data from the APIs that we'll work with. We can then use that data in our web page.

## HTTP Methods

When you make an HTTP request, you need to specify what kind of request you're making. There are a few different types of requests, but the most common ones are **GET**, **POST**, **PUT**, and **DELETE**. We'll mostly be using **GET** in this course.

- **GET** requests are used to retrieve data from the server. In fact, every time you visit a webpage, you are making a **GET** request.
- **POST** requests are used to send data to the server. Usually when you submit a website form, it makes a **POST** request.
- **PUT** requests are used to update data on the server. There's also a **PATCH** method that can be used to update.
- **DELETE** requests are used to delete data on the server.

There are other less common methods, but these are the main four. When you use the **fetch** API or **AJAX**, you can choose which type of request you want to make.

## HTTP Status Codes

I'm going to mention HTTP status codes here, but this is not something you really need to remember right now. You'll learn them in time. When you make an HTTP request, the server will respond with a status code. The status code will tell you if the request was successful or not. Status code ranges have specific meanings.

- **100** range is just continue or processing. It means the request is not done yet, but is ok up to that point. You don't work with 100 range responses very often.
- **200** range means **success** and everything went as it should. 200 is what you want.
- **300** range means some kind of redirect.
- **400** range means there was a client error. So basically, it's your fault whether that is a missing field in a form or if you're unauthorized to make that request.
- **500** range is a server error. So it's not the client's fault something went wrong on the server in the back-end code.

### Common Codes

Here are some very common codes. These are the main codes you will run into:

- 200: Success
- 201: Success and something was created
- 204: Success but no content returned
- 301: Permanent redirect
- 400: Bad request
- 401: Unauthorized
- 403: Forbidden
- 404: Resource not found

- 500: Something wrong on server

Read more about status codes [here](#).

## An Example Request

Let's make a request to an API right from our browser. We'll use the public GitHub API. APIs have different endpoints, which are just URLs that you can make requests to using certain HTTP methods to achieve certain results. The GitHub API has a few different endpoints, but we'll use the `users` endpoint. The `users` endpoint will return information about a GitHub user. Later, we'll do this from our code, but for now, let's go to the following URL in the browser:

```
https://api.github.com/users/bradtraversy
```

You can replace my username with your own if you want. This will return a JSON object with information about the user. The data will show right in the browser. You can also see the response in the browser's developer tools `network` tab along with the headers and any other information including the status code of 200, which means successful.

The screenshot shows a browser window with the URL `https://api.github.com/users/bradtraversy`. The left pane displays the raw JSON response, which includes various user details like login, id, node\_id, avatar\_url, url, html\_url, followers\_url, following\_url, gists\_url, starred\_url, subscriptions\_url, organizations\_url, repos\_url, events\_url, received\_events\_url, type, site\_admin, name, company, blog, location, email, bio, hireable, and created\_at/updated\_at timestamps. The right pane shows the Network tab of the developer tools, specifically the Headers section. It lists the Request URL (`https://api.github.com/users/bradtraversy`), Request Method (GET), Status Code (200), Remote Address (140.82.114.6:443), and Referrer Policy (strict-origin-when-cross-origin). The Headers listed include accept-ranges, access-control-allow-origin, access-control-expose-headers, cache-control, content-encoding, and content-length.

If we made this request from our code, we could then use any of this information in our web page. We'll do this later on. For now, I just wanted to get you familiar with making HTTP requests and seeing the response.

# Sample HTTP Requests & The Network Tab

---

Okay, so I want to show you how to use the Network tab in your browser's developer tools. This is a really useful tool for debugging your code. It allows you to see what requests are being made to your server and what data is being sent back.

I'm going to use my own website as an example, so I'll go to <https://traversymedia.com> and go to the developer tools. I'll click on the Network tab and then refresh the page.

This will show us every single resource that was sent with the response from the **GET** request that was made to the server by the browser. You can see that there are a lot of resources here. There are images, CSS files, JavaScript files, and more. If you click on one of these, you can see the details of the request and response.

If I click on one of the resources such as one of the CSS files, we can see the status code is **200**, which means **success**. We can also see the size of the file, the type of file and much more. If I click on **response**, we can see the actual CSS file that was sent back to the browser.

## Getting Data

So that's how you can use the Network tab to see what resources are being sent back to the browser. Now, let's look at a data resource that we would typically use JavaScript to fetch data from. I'm going to use the GitHub API as an example.

In your browser, you can go to <https://api.github.com/users/bradtraversy>. This is the API endpoint for my GitHub profile.

The data will display right in the browser. I have a Chrome extension called **JSON Viewer** that makes it easier to read. If you don't have this extension, you can install it from the Chrome Web Store.

If we click on the Network tab and refresh the page, we can see that the request was made to the server and the response was sent back. If we click on the response, we can see the data that was sent back.

Ultimately, we would want to use JavaScript to fetch this data and use it within our project. We can do this with the **fetch API**. Before we get to fetch, I want to show you how we can do it using the **XMLHttpRequest** object. This will give us more opportunities to work with callbacks.

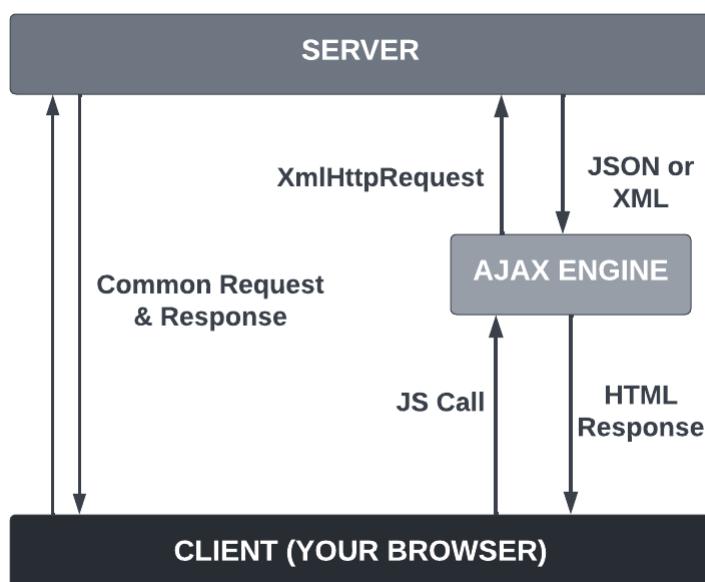
# AJAX & XHR (XMLHttpRequest)

So when we make a request to a server or some kind of data API, we will usually use the `fetch` API. Any project we do in this course, that's what we'll be doing. But it's important to know that there's another way to do it, and that's with the `XMLHttpRequest` object. This is the older way of doing it, but you should at least know the basics just in case you run into it. It will also help you understand more about callbacks, etc.

## XMLHttpRequest

The `XMLHttpRequest` object is a built-in browser object that allows us to make HTTP requests. It's a little more complicated than the `fetch` API, but it's still pretty straightforward. You may notice `XML` is included in the name of the object. AJAX also stands for "Asynchronous JavaScript and XML". This is because years ago when this was created, XML was the common data format we used. These days you almost never see XML, instead we work with JSON, which is much simpler and cleaner.

Let's look at the following diagram to see how this works:



So we have our client, which is our browser, and we have the server or third-party API that we want to communicate with via HTTP request. This could also just be a file on your file system.

Like I said before, when we visit a webpage, we are making an HTTP request. That is what is demonstrated on the left.

What AJAX and the XHR object allow us to do is make those same types of requests via JavaScript. So from our code we make a call using the AJAX engine and XHR object. It sends a request behind the scenes without refreshing the page. Then we get back JSON or XML data in a response and we can then update the dom with that data. So all this happens behind the scenes without the user having to refresh the page.

Let's look at an example of how we do this within our code:

```
const xhr = new XMLHttpRequest();

xhr.open('GET', 'https://api.github.com/users/bradtraversy');

xhr.onreadystatechange = function () {
  if (this.readyState === 4 && this.status === 200) {
    console.log(JSON.parse(this.responseText));
  }
};

xhr.send();
```

Let's go through bit-by-bit.

First, we create a new `XMLHttpRequest` object. This is the object that we'll use to make the request.

```
const xhr = new XMLHttpRequest();
```

Next, we use the `open()` method to initialize the request. We pass in the type of request we want to make, and the URL we want to make the request to. You can also pass in a third parameter to specify whether the request should be made asynchronously or not. This is `true` by default, but we can set it to `false` if we want to make a synchronous request, which will block the rest of the code from running until the request is complete. We won't be doing this in this course, but it's good to know that it's possible.

```
xhr.open('GET', 'https://api.github.com/users/bradtraversy');
```

Then we use the `onreadystatechange` event handler to listen for the `readystatechange` event. This event will fire every time the `readyState` changes. We can use this to check the status of the request. You can also use `addEventListener()` to listen for the event.

`readyState` has 5 possible values:

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4: request finished and response is ready

We want to check that the `readyState` is 4, and that the `status` is 200, which means everything is OK. If the `status` is not 200, then there was an error. We use the `this` keyword to refer to the `XMLHttpRequest` object.

We can see what the values are every time the status changes:

```
console.log(this.readyState, this.status);
```

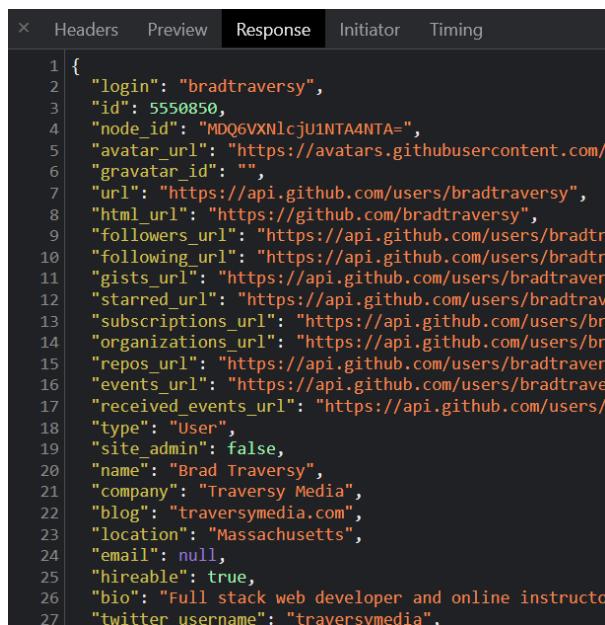
The data that we get back will be in the `responseText` property. We can use `JSON.parse()` to convert it to a JavaScript object.

```
xhr.onreadystatechange = function () {
  if (this.readyState === 4 && this.status === 200) {
    console.log(JSON.parse(this.responseText));
  }
};
```

Finally, we use the `send()` method to send the request.

```
xhr.send();
```

We will see the data in the console, but you can also look in your devtools `network` tab. You should see a request to the GitHub API. Click on it and you should see the response.



```
1 {
2   "login": "bradtraversy",
3   "id": 5550850,
4   "node_id": "MDQ6VXNlcjU1NTA4NTA=",
5   "avatar_url": "https://avatars.githubusercontent.com/
6   "gravatar_id": "",
7   "url": "https://api.github.com/users/bradtraversy",
8   "html_url": "https://github.com/bradtraversy",
9   "followers_url": "https://api.github.com/users/bradtr
10  "following_url": "https://api.github.com/users/bradtr
11  "gists_url": "https://api.github.com/users/bradtraver
12  "starred_url": "https://api.github.com/users/bradtrav
13  "subscriptions_url": "https://api.github.com/users/br
14  "organizations_url": "https://api.github.com/users/br
15  "repos_url": "https://api.github.com/users/bradtraver
16  "events_url": "https://api.github.com/users/bradtrave
17  "received_events_url": "https://api.github.com/users/
18  "type": "User",
19  "site_admin": false,
20  "name": "Brad Traversy",
21  "company": "Traversy Media",
22  "blog": "traversymedia.com",
23  "location": "Massachusetts",
24  "email": null,
25  "hireable": true,
26  "bio": "Full stack web developer and online instructo
27  "twitter_username": "traversymedia",
```

If you want to try getting all of the GitHub repos for a user, you can use the following URL:

```
xhr.open('GET', 'https://api.github.com/users/bradtraversy/repos');
```

Instead of getting back an object, you will get back an array of objects.

Let's output the data to the page. We can use the `forEach()` method to loop through the array of repos. We can then create a list item for each repo and append it to the page.

```
xhr.onreadystatechange = function () {
  if (this.readyState === 4 && this.status === 200) {
    const data = JSON.parse(this.responseText);

    data.forEach((repo) => {
      const li = document.createElement('li');
      li.innerHTML = `${repo.name} - ${repo.description}`;
      document.querySelector('ul').appendChild(li);
    });
  }
};
```

It is important to understand that whatever you get back from an API is up to the server-side developer. So you may get back a different format of data, or different properties, etc. What they chose to send back will be accessible in the `responseText` property.

So now that we know how to make a simple request, in the next video, we're going to create a little joke generator app using the Chuck Norris Jokes API.

# Joke Generator Project

---

In the last video, we saw how to make a request to an API using the `XMLHttpRequest` object. In this video, we'll use the same technique to make a request to the <https://api.chucknorris.io/> to get a random Chuck Norris joke and put it into the page. We will also create a button that will allow us to get a new joke.

Before I move on, I would like to challenge you to create this yourself without having to follow along. We went over everything that you need to know already. In the resources for this video, you'll find the 'chuck-joke-generator' folder with the HTML and CSS along with an empty `script.js` file. You just need to do the following:

- Add an event listener for the button
- Create a function that will make a request to `https://api.chucknorris.io/jokes/random` using the `XMLHttpRequest` object
- Get the data that is sent back (`this.responseText`), parse the JSON and get the joke from it
- Display the joke in the page (you can use the `innerHTML` property)

## ► Click For Solution

First I will bring in the button and the id of where I want the joke to be displayed:

```
const jokeEl = document.getElementById('joke');
const jokeBtn = document.getElementById('jokeBtn');
```

Next, I will add an event listener for the button:

```
jokeBtn.addEventListener('click', generateJoke);
```

Now, I will create the `generateJoke` function. This function will make the request to the API and get the joke:

```
function generateJoke() {
    const xhr = new XMLHttpRequest();

    xhr.open('GET', 'https://api.chucknorris.io/jokes/random');

    xhr.onreadystatechange = function () {
        if (this.readyState === 4) {
            if (this.status === 200) {
                jokeEl.innerHTML = JSON.parse(this.responseText).value;
            } else {
                jokeEl.innerHTML = 'Something went wrong (Not Funny)';
            }
        }
    }
}
```

```
};

xhr.send();
}
```

I am going to first check for the `readystate` to be `4` and then check for the `status` to be `200`. If the status is anything but `200`, then I will put an error message into the joke element.

Remember, the server decides how to format the response. In this case, we get a JSON object with a `value` property. So, we need to parse the JSON and get the joke from it. Then we add it to the page.

We also want this to run right away, so let's use the `DOMContentLoaded` event:

```
document.addEventListener('DOMContentLoaded', generateJoke);
```

And that's it. We now have a Chuck Norris joke generator. You can see the final code in the [chuck-joke-generator-final](#) folder in the resources for this video.

# Callback Hell

---

So, we have seen a bunch of examples of callback functions. They come in handy when we want to make sure that some code is executed after another piece of code has finished executing. But what if we want to make sure that some code is executed after multiple pieces of code have finished executing? This can sometimes result in a lot of nested callbacks, which is called callback hell. There are something called **promises** that can help us with this. We will look at **promises** in the next video, but let's create a situation where we have to have multiple callback functions nested within each other.

I want to create a function called `getData()` that we can use to pass in an endpoint, whether a URL or a file path, and then it will make a request to that endpoint and return the data. We will use the `XMLHttpRequest` object to make the request. Right now, we will just log the data from the function. No callback is being passed in. Let's go ahead and do that.

```
function getData(endpoint) {
  const xhr = new XMLHttpRequest();
  xhr.open('GET', endpoint);

  xhr.onreadystatechange = function () {
    if (this.readyState === 4 && this.status === 200) {
      console.log(JSON.parse(this.responseText));
    }
  };

  setTimeout(() => {
    xhr.send();
  }, Math.floor(Math.random() * 3000) + 1000);
}
```

Now, you never know how long a request will take, so in addition to the request, I am adding a `setTimeout()` that will return the response within 1-3 seconds.

Now Let's create some .json files to fetch. These will just be local files, but it could just as well be a URL endpoint.

## File 1 - movies.json

```
[  
  {  
    "title": "Scarface",  
    "release_year": "1983"  
  },  
  {  
    "title": "The Godfather",  
    "release_year": "1972"  
  },
```

```
[  
  {  
    "title": "Goodfellas",  
    "release_year": "1990"  
  },  
  {  
    "title": "A Bronx Tale",  
    "release_year": "1993"  
  }  
]
```

### File 2 - actors.json

```
[  
  {  
    "name": "Al Pacino",  
    "age": "78"  
  },  
  {  
    "name": "Robert De Niro",  
    "age": "76"  
  },  
  {  
    "name": "Joe Pesci",  
    "age": "77"  
  },  
  {  
    "name": "Chazz Palminteri",  
    "age": "62"  
  }  
]
```

### File 3 - directors.json

```
[  
  {  
    "name": "Brian De Palma",  
    "age": "78"  
  },  
  {  
    "name": "Francis Ford Coppola",  
    "age": "82"  
  },  
  {  
    "name": "Martin Scorsese",  
    "age": "76"  
  },  
  {  
    "name": "Robert De Niro",  
    "age": "76"  
  }  
]
```

```
        "age": "76"
    }
]
```

Now, let's say that we want to get the data from all 3 files. Let's do that.

```
getData('movies.json');
getData('actors.json');
getData('directors.json');
```

So, the way that we are doing it now, you'll notice that the order that we get the data is not the same order that we are requesting the data. This is because the `setTimeout()` is randomizing the order that the data is returned.

If we want to make sure that the data is returned in the order that we requested it. We can do that by passing in a callback function. Let's change the `getData()` function to accept and run a callback function.

```
function getData(endpoint, cb) {
  const xhr = new XMLHttpRequest();
  xhr.open('GET', endpoint);

  xhr.onreadystatechange = function () {
    if (this.readyState === 4 && this.status === 200) {
      cb(JSON.parse(this.responseText));
    }
  };

  setTimeout(() => {
    xhr.send();
  }, Math.floor(Math.random() * 3000) + 1000);
}
```

Now, we can pass in a callback function when we call the `getData()` function.

```
getData('./movies.json', (data) => {
  console.log(data);
  getData('./actors.json', (data) => {
    console.log(data);
    getData('./directors.json', (data) => {
      console.log(data);
    });
  });
});
```

So you can see the issue here. We have nested 3 callback functions within each other. This is called callback hell. It is not very readable and it can get very messy very quickly. However it does work, it gets the data in the correct order.

In the next video, we will look at [promises](#), which gives us a more elegant solution. We will first look at how they work and then in the video after that, we will address this code using promises.

# Promises

---

Alright, so now we are going to learn about promises. A promise is an object that represents the eventual completion or failure of an asynchronous operation. The concept is that a promise is made to complete some kind of task or operation, such as fetching data from a server. Meanwhile, the rest of the code continues to execute. So it's asynchronous and non-blocking. When the task is complete, the promise is either fulfilled or rejected. It also prevents **callback hell**, which are multiple nested callbacks, as we saw in the previous video.

Most of the time, until you get into more advanced JavaScript, you will be dealing with the response from promises, not writing them. For instance, using the **fetch API** will return a promise. So you will need to know what to do with it. In this video, I will show you how to deal with them but also how to create them with the **Promise** constructor.

I'm going to show you how to refactor our posts code from callbacks to promises in the next video, but first I want to give you a super simple example of creating and dealing with promises, so that you can understand the syntax and the concept in general.

## Creating a Promise

We use the **Promise** constructor to create a new promise. The **Promise** constructor takes in a function that has two parameters, **resolve** and **reject**. The **resolve** function is called when the promise is successful and the **reject** function is called when the promise is not successful.

Let's create a simple promise:

```
const promise = new Promise(function (resolve, reject) {
  // Do an async task
  setTimeout(function () {
    console.log('Async task complete');
    resolve();
  }, 1000);
});
```

So we used the **Promise** constructor to create a new promise and passed in the function with the **resolve** and **reject** parameters. We then used the **setTimeout** function to simulate an asynchronous task. After 1 second, we called the **resolve** function. This will "resolve" the promise.

## Consuming/Handling a Promise

If we run this code, nothing will happen because we haven't dealt with the promise yet. To do that, we use the **then** method. The **then** method takes in a function that will be called when the promise is resolved. I do want to mention that there is an alternate way to handle promises and that is with something called **Async/Await**, which we will be learning in a little bit. For now, we will use the **then** method.

```
promise.then(function () {
  console.log('Promise consumed');
});
```

You also don't have to put the promise into a variable. You could just do this:

```
new Promise(function (resolve, reject) {
  // Do an async task
  setTimeout(function () {
    console.log('Async task complete');
    resolve();
  }, 1000);
}).then(function () {
  console.log('Promise consumed');
});
```

If we look in the console, we can see that the `Async task complete` message is logged first and then the `Promise consumed` message is logged. To show you that this is asynchronous, I will add a global console log to the bottom of the file.

```
console.log('Global console log');
```

Now if we run it, we will see the global console log first because the code is not blocked by the promise. The promise is asynchronous and non-blocking.

## Returning Data from a Promise

To return data from a promise, we simply pass it in the `resolve` function. Let's say we want to return a user object from a promise. We can do that like this:

```
const promise = new Promise(function (resolve, reject) {
  // Do an async task
  setTimeout(function () {
    resolve({ name: 'John', age: 30 });
  }, 1000);
});
```

The `then` method takes in a function that has a parameter for the data that is returned from the promise. We can call it whatever we want, but I will call it `user`.

```
promise.then(function (user) {
  console.log(user);
```

## Handling Errors

Remember, we also have a `reject` function that we can call when the promise is not successful, meaning there is some kind of error.

Let's create a variable that represents an error and then check for it and call the `reject` function if it exists. We will also pass in an error message.

```
const promise = new Promise(function (resolve, reject) {
  // Do an async task
  setTimeout(function () {
    let error = false;

    if (!error) {
      resolve({ name: 'John', age: 30 });
    } else {
      reject('Error: Something went wrong');
    }
  }, 1000);
});
```

If `error` is set to false, we get the same result as before. But let's set it to `true` and see what happens.

```
let error = true;
```

So, we do see our error message, but notice it also says `Uncaught (in promise)`. This is because we are not handling the error. We can handle the error by using the `catch` method. The `catch` method takes in a function that has a parameter for the error message. We can call it whatever we want, but I will call it `error`.

```
promise
  .then(function (user) {
    console.log(user);
  })
  .catch(function (error) {
    console.log(error);
});
```

Now, we are handling the error and we can see the error message in the console.

We can shorten this with arrow functions and implicit returns.

```
promise.then((user) => console.log(user)).catch((error) => console.log(error));
```

## finally Method

The `finally` method is used to execute code after the promise is resolved or rejected. It will run no matter what. I personally have not had too many instances where I needed to add a `finally` block, but you should know it exists. Let's add a `finally` method to our promise.

```
promise
  .then((user) => {
    console.log(user);
    return user.name;
  })
  .then((name) => console.log(name))
  .catch((error) => console.log(error))
  .finally(() => console.log('The promise has been resolved or rejected'));
```

Now that you know the basic syntax of a promise, in the next lesson, we will re-factor our posts code from callback to promise.

# Callback To Promise Refactor

---

In the previous lesson, we saw how to use promises to handle asynchronous code. In this lesson, we will refactor our callback code to use promises.

Here is the original code:

```
function createPost(post, cb) {
  setTimeout(() => {
    posts.push(post);
    cb();
  }, 2000);
}

function getPosts() {
  setTimeout(() => {
    posts.forEach(function (post) {
      const div = document.createElement('div');
      div.innerHTML = `<strong>${post.title}</strong> - ${post.body}`;
      document.querySelector('#posts').appendChild(div);
    });
  }, 1000);
}

createPost({ title: 'Post Three', body: 'This is post three' }, getPosts);
```

We call the `createPost` function and pass in the `getPosts` function as the callback. This is the function that will be called when the `createPost` function is done. The `getPosts` function will get all the posts and display them on the page.

## Creating a Promise

To refactor this, let's get rid of the `cb` parameter and create a `promise` instead. We will use the `Promise` constructor to create a new promise. The `Promise` constructor takes in a function that has two parameters, `resolve` and `reject`. The `resolve` function is called when the promise is successful and the `reject` function is called when the promise is not successful.

We will create a variable that represents an error and then check for that error. If it is true, then we will call the `reject` function. If it is false, then we will call the `resolve` function.

```
function createPost(post) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      posts.push(post);

      let error = true;
```

```
        if (!error) {
            resolve();
        } else {
            reject('Error: Something went wrong');
        }
    }, 2000);
});
}
```

The `getPosts` function will stay the same.

```
function getPosts() {
    setTimeout(() => {
        posts.forEach(function (post) {
            const div = document.createElement('div');
            div.innerHTML = `<strong>${post.title}</strong> - ${post.body}`;
            document.querySelector('#posts').appendChild(div);
        });
    }, 1000);
}
```

## Handling the Promise

To handle promises, we use the `then` method, which takes in a function that will be called when the promise is resolved. We will simply pass in the `getPosts` function as the `then` method.

```
createPost({ title: 'Post Three', body: 'This is post three' }).then(getPosts);
```

This way, we don't have to pass any callback into the `createPost` function. The `then` method will be called when the promise is resolved.

## Handling The Error

We can also add a `catch` method to handle the error. The `catch` method takes in a function that will be called when the promise is rejected.

We will create a `showError` function that will display the error on the page.

```
function showError(error) {
    const h1 = document.createElement('h1');
    h1.innerHTML = `<strong>${error}</strong>`;
    document.querySelector('#posts').appendChild(h1);
}
```

```
createPost({ title: 'Post Three', body: 'This is post three' })
  .then(getPosts)
  .catch(showError);
```

There we go. We have successfully refactored our callback code to use a promise.

# Promise Chaining

---

You may have a case where you have a promise that returns another promise. In this case, you can chain the promises together. This is called promise chaining. You would do this if you had a promise that returned a value that you needed to use in another promise or had a sequence of asynchronous tasks that you needed to complete. Let's take our example from the previous lesson:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    let error = false;

    if (!error) {
      resolve({ name: 'John', age: 30 });
    } else {
      reject('Error: Something went wrong');
    }
  }, 1000);
});

promise
  .then((user) => {
    console.log(user);
  })
  .catch((error) => console.log(error));
```

So we have a promise that resolves with a `user` object after 1 second. We're then logging the `user` object. Whatever we return from the `then` callback will be passed to the next `then` callback. So let's return the user's `name` property. We can then log the name in the next `then` callback.

```
promise
  .then((user) => {
    console.log(user);
    return user.name;
  })
  .then((name) => {
    console.log(name);
  })
  .catch((error) => console.log(error));
```

We can chain as many `then` callbacks as we want. Let's say we wanted to get the user's `name length` property. We can do that by returning it and chaining another `then` callback.

```
promise
  .then((user) => {
```

```

        console.log(user);
        return user.name;
    })
    .then((name) => {
        console.log(name);
        return name.length;
    })
    .then((nameLength) => console.log(nameLength))
    .catch((error) => console.log(error));

```

We can go on and on like this. I think you'll already see how this is a good way to avoid `callback hell`, but I'll show you a direct comparison soon.

It's also good to know that if you have a `.then()` after a `.catch()`, the `.then()` will still run. This is because the `.catch()` only handles errors in the previous `.then()`. Let's change the `error` flag to `true` in the promise and add a `.then()` after the `.catch()`.

```

const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        let error = true; // Change this to true

        if (!error) {
            resolve({ name: 'John', age: 30 });
        } else {
            reject('Error: Something went wrong');
        }
    }, 1000);
});

promise
    .then((user) => {
        console.log(user);
        return user.name;
    })
    .then((name) => {
        console.log(name);
        return name.length;
    })
    .then((nameLength) => console.log(nameLength))
    .catch((error) => console.log(error))
    .then((x) => console.log('This will run no matter what'));

```

We can even return something from the `.catch()` callback. This will be passed to the next `.then()` callback. Let's return a string.

```

promise
    .then((user) => {
        console.log(user);

```

```
    return user.name;
  })
  .then((name) => {
    console.log(name);
    return name.length;
})
.then((nameLength) => console.log(nameLength))
.catch((error) => {
  console.log(error);
  return 123;
})
.then((x) => console.log('This will run no matter what', x));
```

Now that you know how to chain promises, let's look at how this compares to [callback hell](#).

# Promises vs Callback Hell

---

So a couple videos back, I showed you an example of what we call "callback hell". This is when you have a bunch of callbacks nested inside of each other. It can get really messy and creates a pyramid of code that is hard to follow. It's also hard to debug.

What I want to do now is create that same `getData` function to get our movies, actors and directors, but I want to use promises instead of callbacks. To get our data, we are still using the XMLHttpRequest (XHR) object. We will be switching to the `fetch API` very soon.

Let's first create the function:

```
function getData(endpoint) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', endpoint);

    xhr.onreadystatechange = function () {
      if (this.readyState === 4) {
        if (this.status === 200) {
          resolve(JSON.parse(this.responseText));
        } else {
          reject('Error: Something went wrong');
        }
      }
    };

    setTimeout(() => {
      xhr.send();
    }, Math.floor(Math.random() * 3000) + 1000);
  });
}
```

As you can see, we created a new promise and if everything goes as planned and we get a `200` status code, then we will resolve the promise. If we get an error, then we will reject the promise.

## Handling Multiple Functions

---

We will call the the first `getData` function to get the movies and we are going to use the `then` method to handle the response from the promise.

```
getData('./movies.json').then((movies) => {
  console.log(movies);
});
```

Now, we want to call `getData` again to get the actors and directors. We can do this by chaining another `then` method. What's great about this is whatever we return from the first `then` method, we can use that in the second `then` method. So let's return the next `getData` function and pass in the actors endpoint. Then we will do it again for the directors.

```
getData('./movies.json')
  .then((movies) => {
    console.log(movies);
    return getData('./actors.json');
  })
  .then((actors) => {
    console.log(actors);
    return getData('./directors.json');
  })
  .then((directors) => {
    console.log(directors);
  });
});
```

So, as you can see, we are able to chain multiple `then` methods together just like we did in the last lesson. This is a lot cleaner than having a pyramid of code. It's also easier to debug because we can see exactly where the error is happening.

We can also use the `catch` method to handle any errors.

```
getData('./movies.json')
  .then((movies) => {
    console.log(movies);
    return getData('./actors.json');
  })
  .then((actors) => {
    console.log(actors);
    return getData('./directors.json');
  })
  .then((directors) => {
    console.log(directors);
  })
  .catch((error) => console.log(error));
```

So more importantly than creating promises, right now, I want you to understand how to handle them with `then` and `catch`.

In the next video, I'm going to show you how to handle multiple promises with `Promise.all`.

# Handle Multiple Promises with `Promise.all()`

---

There may be some instances where you want to handle multiple promises at the same time. For example, you may want to get data from multiple endpoints at the same time. We did this in the last video by chaining multiple `then` methods together. However, there is another way to handle multiple promises at the same time and that is with `Promise.all()`.

Let's take our `getData` function from the last video and use `Promise.all()` to handle multiple promises at the same time.

```
function getData(endpoint) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', endpoint);

    xhr.onreadystatechange = function () {
      if (this.readyState === 4) {
        if (this.status === 200) {
          resolve(JSON.parse(this.responseText));
        } else {
          reject('Error: Something went wrong');
        }
      }
    };

    setTimeout(() => {
      xhr.send();
    }, Math.floor(Math.random() * 3000) + 1000);
  });
}
```

Now, let's store each promise response in a variable:

```
const moviesPromise = getData('./movies.json');
const actorsPromise = getData('./actors.json');
const directorsPromise = getData('./directors.json');
```

Instead of chaining multiple `then` methods, we can use `Promise.all()` to handle all of the promises at the same time. We will pass in an array of promises and then we can use the `then` method to handle the response.

```
Promise.all([moviesPromise, actorsPromise, directorsPromise])
  .then((data) => {
    console.log(data);
```

```
)  
.catch((error) => console.log(error));
```

It's as easy as that. If I wanted to add another promise to the mix, I would just add it to the array.

```
const moviesPromise = getData('./movies.json');  
const actorsPromise = getData('./actors.json');  
const directorsPromise = getData('./directors.json');  
const dummyPromise = new Promise((resolve, reject) => {  
    setTimeout(() => {  
        resolve('Dummy data');  
    }, 2000);  
});  
  
Promise.all([moviesPromise, actorsPromise, directorsPromise, dummyPromise])  
.then((data) => {  
    console.log(data);  
})  
.catch((error) => console.log(error));
```

So now that you know how to work with promises, we can start to learn the [fetch API](#), which is a much more modern and cleaner way to make HTTP requests than the [XMLHttpRequest](#) object.

# Fetch API

---

So you saw how we can use the `XMLHttpRequest` object to make HTTP requests. But there's a more modern way that's been added to the browser to make HTTP requests, called the `Fetch API`. It's a little bit easier to use than the `XMLHttpRequest` object. It's also more powerful. Fetch also provides a single logical place to define other HTTP-related concepts such as CORS and extensions to HTTP. We'll talk about that later, but right now, we're just going to learn how to make simple requests.

## How Fetch Works

`fetch()` is a method that only requires one argument, which is the URL or file path that you want to make the request to. It returns a `promise` that resolves to a `Response` object. The `Response` object contains the response from the server.

Let's make a request to get the data in the `movies.json` file. You know how to handle `promises` now, so you can use the `then` method to handle the response.

```
fetch('./movies.json').then(function (response) {
  console.log(response);
});
```

You can use this syntax or you can use the arrow function syntax, which is what I prefer and probably what I will be using for the rest of the course.

It is important to note that this does not directly return the data or the JSON response body (in this case, movies) but instead returns a promise that resolves with a `Response` object with a bunch of properties. It includes stuff like the `status` of `200`, `statusText` of `ok`. It has a `body` property that contains a `ReadableStream` object. To extract the JSON data from the `Response` object, we return the `json()` method, which returns a second promise that resolves with the data that we're looking for.

Remember, when we chain `then` methods, the return value of the first `then` method is passed as an argument to the second `then` method. We can call that argument whatever we want, but in this case, we'll call it `data`.

```
fetch('./movies.json')
  .then(function (response) {
    return response.json();
})
  .then(function (data) {
    console.log(data);
});
```

We can shorten this up using arrow functions and implicit returns. This is commonly what you'll see in the wild.

```
fetch('./movies.json')
  .then((response) => response.json())
  .then((data) => console.log(data));
```

## Fetching Text

Usually we want to fetch JSON data, but just to show you it is possible, we can fetch plain text. Let's create a file named `test.txt` and just add some random text in it. We can use the `text()` method to get the text from the `Response` object.

```
fetch('./test.txt')
  .then((response) => response.text())
  .then((data) => console.log(data));
```

## Fetching Data from an API

Now that we know how to make a request to a local file, let's make a request to an API. There are a ton of public APIs that we can use. Some of them do not require any type of authentication, but some of them do. You may need to register what we can an API key, and send that with your request. We'll talk about that later.

One of my favorite resources on the Web is this GitHub repo called [Public APIs](#). It's a list of public APIs that you can use in your projects. There are all kinds of categories and it tells you if there is any authentication required, if HTTPS is required and if there is a CORS policy, which we will talk about later.

Most public APIs will have documentation showing you which endpoints are available and what kind of data you can expect back as well as any operations that the API lets you run.

Let's try getting data from a couple APIs that don't require any authentication.

We've already looked at the public GitHub API in the last section. Let's make a request to the GitHub API to get information about a user. Feel free to replace my username with your own.

```
fetch('https://api.github.com/users/bradtraversy')
  .then((response) => response.json())
  .then((data) => console.log(data));
```

Here is the response that we get back:

```
{
  "login": "bradtraversy",
  "id": 1198226,
  "node_id": "MDQ6VXNlcjExOTgyMjY=",
```

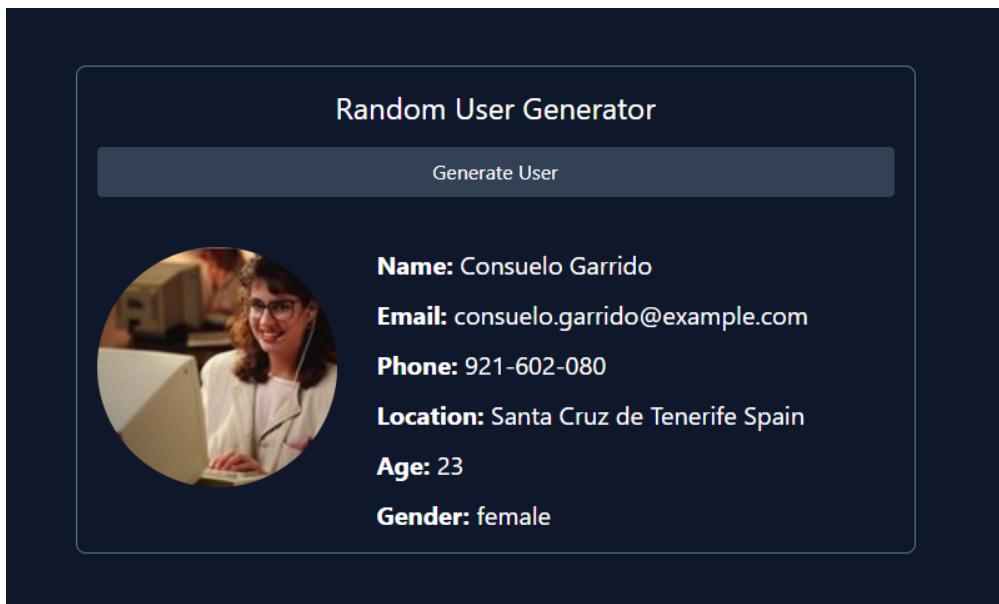
```
"avatar_url": "https://avatars.githubusercontent.com/u/1198226?v=4",
"gravatar_id": "",
"url": "https://api.github.com/users/bradtraversy",
"html_url": "",
"followers_url": "https://api.github.com/users/bradtraversy/followers",
"following_url":
  "https://api.github.com/users/bradtraversy/following{/other_user}",
"gists_url": "https://api.github.com/users/bradtraversy/gists{/gist_id}",
"starred_url": "https://api.github.com/users/bradtraversy/starred{/owner}{/repo}",
"subscriptions_url":
  "https://api.github.com/users/bradtraversy/subscriptions",
"organizations_url": "https://api.github.com/users/bradtraversy/orgs",
"repos_url": "https://api.github.com/users/bradtraversy/repos",
"events_url": "https://api.github.com/users/bradtraversy/events{/privacy}",
"received_events_url":
  "https://api.github.com/users/bradtraversy/received_events",
"type": "User",
"site_admin": false,
"name": "Brad Traversy",
"company": "Traversy Media",
"blog": "http://www.traversymedia.com",
"location": "Charleston, SC",
"email": null,
"hireable": null,
"bio": "Full Stack Web Developer & Instructor",
"twitter_username": "traversymedia",
"public_repos": 185,
"public_gists": 32,
"followers": 10000,
"following": 0,
"created_at": "2012-09-30T15:36:51Z",
"updated_at": "2021-03-31T20:19:57Z"
}
```

So we could use any of this data to display on our page.

# Random User Generator Challenge

---

Before we move on to fetch options and making other types of requests, I want to do a mini project using the [Random User API](#). We are going to be able to click a button and fetch a random user and put their info on the page including their picture, name, email, phone, location, age and gender.



If you think you are ready to try this on your own, We already have the HTML and CSS for the project. You just need to add the JavaScript to add an event listener to the button, fetch the user and display them on the page. We'll be using the same HTML structure as the hardcoded HTML, we will just replace the values with data coming from the API.

Alright, let's get started!

## Get Elements & Add Event Listener

First, we will get the generate button, the user result element and the loader element. Then we will add an event listener to the button.

```
const generateBtn = document.querySelector('#generate');
const userEl = document.querySelector('#user');
const loadingEl = document.querySelector('#loading');

generateBtn.addEventListener('click', () => {
  console.log('Fetch User...');
});
```

Make sure that works by clicking the button and checking the console. You should see the message.

Now, let's make the request to the API endpoint, which is <https://randomuser.me/api/>.

---

```
generateBtn.addEventListener('click', () => {
  fetch('https://randomuser.me/api/')
    .then((res) => res.json())
    .then((data) => {
      // console.log(data);
    });
});
```

You should see the data in the console. It is an array with a single object. We need to get the object out of the array. We can do that by using [0]:

```
generateBtn.addEventListener('click', () => {
  fetch('https://randomuser.me/api/')
    .then((res) => res.json())
    .then((data) => {
      const user = data.results[0];
      console.log(user);
    });
});
```

Now, we can start displaying the user data on the page. You can basically copy everything that is in the `#user` div in the HTML and then add it to the `div` via JavaScript with the values from the API response:

```
generateBtn.addEventListener('click', () => {
  fetch('https://randomuser.me/api/')
    .then((res) => res.json())
    .then((data) => {
      const user = data.results[0];

      userEl.innerHTML = `
        <div class="flex justify-between">
          <div class="flex">
            
            <div class="space-y-3">
              <p class="text-xl">
                <span class="font-bold">Name: </span> ${user.name.first}
                ${user.name.last}
              </p>
              <p class="text-xl">
                <span class="font-bold">Email: </span> ${user.email}
              </p>
              <p class="text-xl">
                <span class="font-bold">Phone: </span> ${user.phone}
              </p>
            </div>
        </div>
      `;
```

```

        <p class="text-xl">
            <span class="font-bold">Location: </span> ${user.location.city}
${user.location.country}
        </p>
        <p class="text-xl"><span class="font-bold">Age: </span>
${user.dob.age}</p>
        <p class="text-xl">
            <span class="font-bold">Gender: </span> ${user.gender}
        </p>
        </div>
    </div>
    `;
});
);

```

It's up to you if you want to remove the initial user from the HTML code. I left it, but commented it out.

## Show Loading Spinner

If you want to show the spinner, add a link to the stylesheet in the `head` tag right above the Tailwind CDN. Then add a function to show and hide the spinner. It is set to `display: none` in the CSS, so we will just change that to `display: block` when we want to show it:

```

function showSpinner() {
    loadingEl.style.display = 'block';
}

function hideSpinner() {
    loadingEl.style.display = 'none';
}

```

Now, add it to the event listener:

```

generateBtn.addEventListener('click', () => {
    showSpinner();

    fetch('https://randomuser.me/api/')
        .then((res) => res.json())
        .then((data) => {
            hideSpinner();
            const user = data.results[0];
            // console.log(data);

            userEl.innerHTML = `
                <div class="flex justify-between">
                    <div class="flex">
                        
    <div class="space-y-3">
        <p class="text-xl">
            <span class="font-bold">Name: </span> ${user.name.first}
${user.name.last}
        </p>
        <p class="text-xl">
            <span class="font-bold">Email: </span> ${user.email}
        </p>
        <p class="text-xl">
            <span class="font-bold">Phone: </span> ${user.phone}
        </p>
        <p class="text-xl">
            <span class="font-bold">Location: </span> ${user.location.city}
${user.location.country}
        </p>
        <p class="text-xl"><span class="font-bold">Age: </span>
${user.dob.age}</p>
        <p class="text-xl">
            <span class="font-bold">Gender: </span> ${user.gender}
        </p>
    </div>
</div>
`;
});
});

```

I prefer to break my code up into smaller functions. This is optional, but I'm going to create a function to fetch the user and another to display the user. I also like to put my event listeners at the bottom. Here is the final result:

```

const generateBtn = document.querySelector('#generate');
const userEl = document.querySelector('#user');
const loadingEl = document.querySelector('#loading');

function fetchUser() {
    showSpinner();
    fetch('https://randomuser.me/api/')
        .then((res) => res.json())
        .then((data) => {
            hideSpinner();
            const user = data.results[0];
            displayUser(user);
        });
}

function displayUser(user) {

```

```

userEl.innerHTML = `

<div class="flex justify-between">
<div class="flex">
  
  <div class="space-y-3">
    <p class="text-xl">
      <span class="font-bold">Name: </span> ${user.name.first}
      ${user.name.last}
    </p>
    <p class="text-xl">
      <span class="font-bold">Email: </span> ${user.email}
    </p>
    <p class="text-xl">
      <span class="font-bold">Phone: </span> ${user.phone}
    </p>
    <p class="text-xl">
      <span class="font-bold">Location: </span> ${user.location.city}
      ${user.location.country}
    </p>
    <p class="text-xl"><span class="font-bold">Age: </span>
      ${user.dob.age}</p>
    <p class="text-xl">
      <span class="font-bold">Gender: </span> ${user.gender}
    </p>
  </div>
</div>
</div>
`;

}

function showSpinner() {
  loadingEl.style.display = 'block';
}

function hideSpinner() {
  loadingEl.style.display = 'none';
}

generateBtn.addEventListener('click', fetchUser);

```

That's it! Now we have a random user generator.

What I want to talk about next is error handling with the Fetch API.

# Fetch Options

---

When we make a request with the fetch API, the first parameter is the URL, but we can also pass in an object as the second parameter to add options to the request.

Some of the available options are:

- **method**: The HTTP method to use, such as `GET`, `POST`, `PUT`, `DELETE`, etc. By default it is `GET`.
- **headers**: An object containing any custom headers that you want to add to the request.
- **body**: The data to send with the request. If you are submitting form data for instance, you can send that data here. It can be a string, a `Blob`, a `BufferSource`, a `FormData`, a `URLSearchParams`, or a `USVString`.

There are others as well such as `mode`, `credentials`, `cache`, `redirect`, `referrer`, `referrerPolicy`, `integrity`, and `keepalive`. You can read more about them [here](#).

## JSONPlaceholder API

For this video and the next project, we will be using the [JSON Placeholder Fake REST API](#). This API gives us access to a bunch of different endpoints for resources like posts, users, todos, etc.

It is a REST API, which adheres to specific standards when it comes to HTTP methods and how the endpoint URLs are structured. We will be using the blog post resource for this. These are the REST endpoints

- GET /posts **Fetch all posts**
- GET posts/1 **Fetch a post with the ID of 1**
- POST /posts **Add a new post**
- PUT /posts/1 **Update a post with the id of 1**
- DELETE /posts/1 **Delete a post with the id of 1**

Now, we can make the requests, just know that the data does not actually persist into any database. So for instance, when we make a post request, we get a successful response, however the data does not stick because it is not actually saved in the database. They can't let the public add any data they want. There is a tool called [JSON Server](#) from this developer that we can use to store data locally. We will look at that later.

## Making a POST Request

By default, when we pass a URL into `fetch()` it will make a `GET` request. This is what we use if we want to retrieve data from a server. But what if we want to send data to the server? We can do that by passing in an object as the second parameter to `fetch()`. This object can contain options for the request, including the `method` option.

Let's make a POST request to the [JSON Placeholder API](#).

Usually, when we make a `POST` request, we send some form data to the server to create some kind of resource like a blog post, user, etc.

JSONPlaceholder has endpoints for creating posts, users, and comments. Let's make a `POST` request to create a new post.

```
function createPost({ title, body }) {
  fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify({
      title,
      body,
    }),
    headers: {
      'Content-Type': 'application/json',
    },
  })
  .then((response) => response.json())
  .then((data) => console.log(data));
}

createPost({ title: 'My Post', body: 'This is my post' });
```

The `body` option is where we send the data to the server. We can send any kind of data here, but we need to make sure that the server knows what kind of data we are sending. We do this by setting the `Content-Type` header to the appropriate value. In this case, we are sending JSON data, so we set the `Content-Type` header to `application/json`. We also need to make sure that the data is in the correct format. In this case, we need to convert the data to a string using `JSON.stringify()`.

We can also add any custom headers that we wanted. For instance, the API may require you to authenticate and get a token and then send that token with a request to access a protected route.

```
headers: {
  'Content-type': 'application/json; charset=UTF-8',
  'token': 'abc123'
}
```

The server will respond with the data that we sent, but it will also add an `id` property to the object. This is the ID of the new post that was created.

If you open your DevTools and go to the `Network` tab, you will see all of the info about the request and response including the headers, payload, status and response.

# Typicode Todos Mini-Project

---

Ok, so in the last lesson, I showed you how to add options to your fetch request including the method, headers and body. I want to take that a little further with another mini project. We're going to use the [JSONPlaceholder API](#). Here are the specs...

- Fetch the todos from <https://jsonplaceholder.typicode.com/todos> and put them on the page. We already have the HTML and CSS. I also want to be able to add a todo by making a POST request to the api and then adding the todo to the DOM. Now it's important to understand that this data will NOT persist. The API does not actually let us store data, so it will not stick. If we reload the page, any new todos that we added will go away. Later on I will show you another tool by this API developer called [JSON Server](#) which we can run locally and actually save the data.
- All of the todos that have the property `completed: true`, will have a class of `.done`, which will give it a light gray background.
- We click on the todo and it toggles the update value. This means we need to toggle the class in the DOM and make a PUT request to the API to update the `completed` value. Again, this will not stick because we can not actually update their data. It will give us the response though.
- We doouble click and send a DELETE request then delete the todo from the DOM.

## Getting and Displaying the Todos

Let's start by getting and displaying the todos

I am going to use an `init()` function for our event listeners

```
const apiUrl = 'https://jsonplaceholder.typicode.com/todos';

function getTodos() {
  fetch(`${apiUrl}?_limit=5`)
    .then(res => res.json())
    .then(data => {
      data.forEach(todo => {
        const div = document.createElement('div');
        div.classList.add('todo');
        div.appendChild(document.createTextNode(todo.title));
        div.setAttribute('data-id', todo.id);

        document.getElementById('todo-list').appendChild(div);
      });
    });
}

function init() {
  document.addEventListener('DOMContentLoaded', getTodos);
}
```

```
init();
```

The code above will fetch the data and then loop through it and create a new `div` for each todo and add it to the list.

I added `?_limit=5` on to the end of our apiUrl. JSONPlaceholder allows this to limit the number of resources. I only want to get 5.

I added a custom `data` attribute called `data-id`. That is because we will need the id later. This will let us access it. When you create an attribute that does not exist in HTML, you should prefix it with `data-`. We can then access it later using the 'dataset' object.

## Add background for completed todos

To add a gray background, just check for the completed value and add the `done` class:

```
if (todo.completed) {
    div.classList.add('done');
}
```

## Create separate function to add todo to DOM

When we create a todo, we will also add it to the DOM, so let's create a function for that

```
function addTodoToDOM(todo) {
    const div = document.createElement('div');
    div.classList.add('todo');
    div.appendChild(document.createTextNode(todo.title));
    div.setAttribute('data-id', todo.id);

    if (todo.completed) {
        div.classList.add('done');
    }

    document.getElementById('todo-list').appendChild(div);
}
```

Now add it to the foreach in `getTodos()`

```
function getTodos() {
    fetch(`${apiUrl}?_limit=5`)
        .then((res) => res.json())
        .then((data) => {
            data.forEach((todo) => addTodoToDOM(todo));
```

```
});  
}
```

## Create a todo

Now, let's add the event listener and the `createTodo()` function

```
function createTodo(e) {  
  e.preventDefault();  
  
  const newTodo = {  
    title: e.target.firstChild.value,  
    completed: false,  
  };  
  fetch(apiUrl, {  
    method: 'POST',  
    body: JSON.stringify(newTodo),  
    headers: {  
      'Content-Type': 'application/json',  
      token: 'abc123',  
    },  
  })  
    .then((response) => response.json())  
    .then((data) => addTodoToDOM(data));  
}  
  
function init() {  
  document.addEventListener('DOMContentLoaded', getTodos);  
  document.getElementById('todo-form').addEventListener('submit', createPost);  
}  
  
init();
```

## Update completed field

Let's make it so we can click on a todo and update the `completed` field. Again, this will go away on refresh because it does not persist to the database of the API.

```
function toggleComplete(e) {  
  if (e.target.classList.contains('todo')) {  
    e.target.classList.toggle('done');  
    updateTodo(e.target.dataset.id, e.target.classList.contains('done'));  
  }  
}  
  
function updateTodo(id, completed) {  
  fetch(`${apiUrl}/${id}`, {  
    method: 'PUT',  
  })  
    .then((response) => response.json())  
    .then((data) => addTodoToDOM(data));  
}
```

```

        body: JSON.stringify({ completed }),
        headers: {
          'Content-Type': 'application/json',
        },
      })
      .then((res) => res.json())
      .then((data) => console.log(data));
    }

    function init() {
      document.addEventListener('DOMContentLoaded', getTodos);
      document.getElementById('todo-form').addEventListener('submit', createTodo);
      document
        .getElementById('todo-list')
        .addEventListener('click', toggleComplete);
    }

    init();

```

## Delete a todo

Let's add a double click event listener and handler to remove the Todo. We will make a DELETE request and get a response, but it will not stick.

```

function deleteTodo(e) {
  if (e.target.classList.contains('todo')) {
    const id = e.target.dataset.id;
    fetch(`${apiUrl}/${id}`, {
      method: 'DELETE',
    })
    .then((res) => res.json())
    .then(() => e.target.remove());
  }
}

function init() {
  document.addEventListener('DOMContentLoaded', getTodos);
  document.getElementById('todo-form').addEventListener('submit', createTodo);
  document
    .getElementById('todo-list')
    .addEventListener('click', toggleComplete);
  document.getElementById('todo-list').addEventListener('dblclick', deleteTodo);
}

init();

```

Here is the final code:

```
const apiUrl = 'https://jsonplaceholder.typicode.com/todos';

function getTodos() {
  fetch(` ${apiUrl}?_limit=5`)
    .then((res) => res.json())
    .then((data) => {
      data.forEach((todo) => addTodoToDOM(todo));
    });
}

function createTodo(e) {
  e.preventDefault();

  const newTodo = {
    title: e.target.firstChild.value,
    completed: false,
  };
  fetch(apiUrl, {
    method: 'POST',
    body: JSON.stringify(newTodo),
    headers: {
      'Content-Type': 'application/json',
      token: 'abc123',
    },
  })
    .then((response) => response.json())
    .then((data) => addTodoToDOM(data));
}

function addTodoToDOM(todo) {
  const div = document.createElement('div');
  div.appendChild(document.createTextNode(todo.title));
  div.setAttribute('data-id', todo.id);

  if (todo.completed) {
    div.classList.add('done');
  }

  document.getElementById('todo-list').appendChild(div);
}

function toggleComplete(e) {
  e.target.classList.toggle('done');
  updateTodo(e.target.dataset.id, e.target.classList.contains('done'));
}

function updateTodo(id, completed) {
  fetch(` ${apiUrl}/${id}`, {
    method: 'PUT',
    body: JSON.stringify({ completed }),
  })
    .then((response) => response.json())
    .then((data) => {
      const todo = document.querySelector(`[data-id='${data.id}']");
      todo.classList.toggle('done');
    });
}
```

```
headers: {
  'Content-Type': 'application/json',
},
})
.then((res) => res.json())
.then((data) => console.log(data));
}

function deleteTodo(e) {
  const id = e.target.dataset.id;
  fetch(` ${apiUrl}/${id}` , {
    method: 'DELETE',
  })
  .then((res) => res.json())
  .then(() => e.target.remove());
}

function init() {
  document.addEventListener('DOMContentLoaded', getTodos);
  document.getElementById('todo-form').addEventListener('submit', createTodo);
  document
    .getElementById('todo-list')
    .addEventListener('click', toggleComplete);
  document.getElementById('todo-list').addEventListener('dblclick', deleteTodo);
}

init();
```

# Fetch API Error Handling

---

In this lesson, I want to talk about error handling in the Fetch API. There are a couple things that you may not expect. It is important to understand the main HTTP status codes and what they mean, because it is important that you understand your API's response. So let's recap on the most common ones.

The **200** range is for successful responses.

- 200 - OK
- 201 - Created
- 204 - No Content

The **400** range is for client errors. This means that the request was not understood by the server.

- 400 - Bad Request
- 401 - Unauthorized
- 403 - Forbidden
- 404 - Not Found

The **500** range is for server errors. This means that the server encountered an error while processing the request.

- 500 - Internal Server Error

The most common of the common will be **200**, **404** and **500**. So let's look at how we can handle these in the Fetch API.

We can use the website [httpstat.us](http://httpstat.us) to send requests to and get a specific status code. Let's try it out.

```
fetch('http://httpstat.us/200')
  .then((response) => {
    return response;
})
  .then((data) => {
    console.log('Success');
});
```

If you run this, you should see 'Success' logged to the console. Let's try a **404** now.

```
fetch('http://httpstat.us/404')
  .then((response) => {
    return response;
})
  .then((data) => {
    console.log('Success');
});
```

---

A **404** is not a successful response. It means what we are requesting is not there. However, we are still seeing 'Success' logged to the console.

Let's try adding a **catch** to the end of the chain. Remember, a **catch** will catch any errors that occur in the chain. So let's try it.

```
fetch('http://httpstat.us/404')
  .then((response) => {
    return response;
  })
  .then((data) => {
    console.log('Success');
  })
  .catch((error) => {
    console.log('Error:', error);
 });
```

Nothing happens. The **catch** is not called. This is because the **fetch** request itself was successful. The **catch** will only be called if there is a network error. So if we make a request to a URL that doesn't exist, we will see the **catch** called. let's try this:

```
fetch('http://hello123.net')
  .then((response) => {
    return response;
  })
  .then((data) => {
    console.log('Success');
  })
  .catch((error) => {
    console.log('Error:', error);
 });
```

This does result with the **catch** being called. So we can use the **catch** to handle network errors. But what about the **404** or any other unsuccessful response? We want to handle that as well. We can do this by checking the **status** of the response and then throwing an error if we need to. Let's try it out.

```
fetch('http://httpstat.us/404')
  .then((response) => {
    if (response.status !== 200) {
      throw new Error('Not 200 response');
    }
    return response;
  })
  .then((data) => {
    console.log('Success');
```

```
})
.catch((error) => {
  console.log('Error:', error);
});
```

So we looked at the `status` of the response and if it is not `200`, we throw an error. `throw` is a keyword that will stop the chain and call the `catch` with the error that we throw. So now we are seeing the `catch` called with the error that we threw. Try a `200` response and you will see the `then` called. If you try a `500` response or anything but `200`, you will see the `catch` called.

If you want to check for specific status codes, you can use `else if` statements:

```
fetch('http://httpstat.us/400')
.then((response) => {
  if (response.status === 404) {
    throw new Error('Not found');
  } else if (response.status === 400) {
    throw new Error('Bad request');
  } else if (response.status === 500) {
    throw new Error('Server error');
  } else if (response.status !== 200) {
    throw new Error('Not 200 response');
  }
  return response;
})
.then((data) => {
  console.log('Success');
})
.catch((error) => {
  console.log('Error:', error);
});
```

There are other `success` status codes that you may want to handle. For example, `201` is a successful response that means a resource was created. You may want to handle this differently than a `200` response. You can do this by adding another `else if` statement. Let's try it out.

```
fetch('http://httpstat.us/404')
.then((response) => {
  if (response.status >= 200 && response.status <= 299) {
    throw new Error('Not 200 response');
  }
  return response;
})
.then((data) => {
  console.log('Success');
})
.catch((error) => {
```

```
    console.log('Error:', error);
});
```

There is also an `ok` property on the response object. This is a boolean that is `true` if the status is in the `200` range. So we can use this as well:

```
fetch('http://httpstat.us/400')
  .then((response) => {
    if (!response.ok) {
      throw new Error('Something went wrong');
    }
    return response;
})
  .then((data) => {
    console.log('Success');
})
  .catch((error) => {
    console.log('Error:', error);
});
```

## Add error handling to project

The random user API is pretty simple. We most likely won't get a 404 or 500 error. But we can still add some error handling. Let's add a `catch` to the end of the chain and show a message if there is an error.

```
function fetchUser() {
  showSpinner();
  fetch('https://randomuser.me/api/')
    .then((res) => {
      if (res.status !== 200) {
        throw new Error('Something went wrong');
      }
      return res.json();
    })
    .then((data) => {
      hideSpinner();
      const user = data.results[0];
      displayUser(user);
    })
    .catch((error) => {
      hideSpinner();
      userEl.innerHTML = `<p class="text-xl text-center text-red-500 mb-5">${error}</p>`;
    });
}
```

Now try and change the URL to <https://randomuser.me/api1/>. Now this would still run the catch even without the check because it doesn't exist, but the check is there to show for specific status codes and makes your code more less error prone.

# Async / Await

---

Now that we know how about promises, I want to show you an alternate way to consume them called **Async & Await**. We already saw that we can use the `.then()` method to handle the promise. That method is cleaner than the callback syntax, but it can get a little messy when you have a lot of nested `.then()` methods. That's where **Async & Await** comes in. It allows us to write asynchronous code that looks synchronous. It's also a lot easier to read and understand.

Let's take the first promise that we created a while ago and convert it to use **Async & Await**.

```
const promise = new Promise(function (resolve, reject) {
  // Do an async task
  setTimeout(function () {
    let error = false;

    if (!error) {
      resolve({ name: 'John', age: 30 });
    } else {
      reject('Error: Something went wrong');
    }
  }, 1000);
});
```

So this code will create a promise that resolves after 1 second. I'll use the `.then()` syntax for comparison:

```
promise.then((data) => {
  console.log(data);
});
```

When we use **Async & Await**, we create a function that is prefixed with the `async` keyword. We can then use the `await` keyword in front of any promise. This will pause the execution of the function until the promise is resolved. Once the promise is resolved, we can store the result in a variable.

```
async function getPromise() {
  const response = await promise;
  console.log(response);
}

getPromise();
```

As you can see, this does the same exact thing, but it's a lot easier to read and understand. It reads like a normal synchronous function. We simply add the `async` keyword to the function, and then we use the

`await` keyword in front of the promise. It is important to stress that you will not be able to use the `await` keyword outside of an `async` function. If you try it, you will get an error.

## Async / Await with Fetch

Using Async/Await with the Fetch API is very similar. We'll create a function that is prefixed with the `async` keyword. We'll then use the `await` keyword in front of the `fetch()` method. This will pause the execution of the function until the promise is resolved. Once the promise is resolved, we can store the result in a variable.

You can use any API, but I'm going to use the `users` endpoint from <https://jsonplaceholder.typicode.com>.

```
async function getUsers() {  
  const response = await fetch('https://jsonplaceholder.typicode.com/users');  
  const data = await response.json();  
  
  console.log(data);  
}  
  
getUsers();
```

Now, remember the Fetch API returns the initial `Response` object. We can't get the JSON data from that object directly. We need to use the `.json()` method on the response object. This will return another promise, so we need to use the `await` keyword again. Once the promise is resolved, we can store the result in a variable.

Let's make it a bit more interesting and show the users on the page. I know this is not a DOM lesson, but I want to make sure I keep you on your toes.

Add a `div` with an `id` of `output` to the `index.html` file.

```
<div id="output"></div>
```

Then edit your `getUsers()` function

```
async function getUsers() {  
  const response = await fetch('https://jsonplaceholder.typicode.com/users');  
  const data = await response.json();  
  
  console.log(data);  
}  
  
getUsers();
```

## Using Async / Await with Arrow Functions

When we use arrow functions, the `async` keyword goes before the arrow. Here is the same function as above, but using an arrow function.

```
const getUsers2 = async () => {
  const response = await fetch('https://jsonplaceholder.typicode.com/users');
  const data = await response.json();

  console.log(data);
};

getUsers2();
```

## Do I need to use Async/Await?

No, you don't need to use Async/Await. It's just another way to consume promises. It's a lot easier to read and understand, but it's not required. You can use the `.then()` method if you want. I almost never use the `.then()` method. I think it's a lot easier to read and understand the `Async & Await` syntax. We'll mostly be using the `Async & Await` syntax for the rest of the course. If you'd like to use the `.then()` method, you can. It's just a matter of preference.

Next, let's look at how to handle errors using `try/catch`.

# Try...Catch

---

A try...catch statement is used to handle errors in JavaScript. It is a syntax that allows you to write code that will run if an error occurs, and code that will run if no error occurs. The reason that I am choosing to cover this now is because it is how we usually handle errors with Async & Await. With the `.then()` syntax, we can use the `.catch()` method, which you could use with Async/Await if you want, but usually to handle errors. With Async & Await, we use a try...catch statement. I will get into using it with Async & Await in the next lesson.

Let's start off by just logging a variable that does not exist:

```
console.log(x);
```

If we run this code, we will get an error in the console:

```
Uncaught ReferenceError: x is not defined
```

Notice how it says `uncaught`? Let's wrap this code in a try...catch statement:

The basic syntax is as follows:

```
try {  
    // Try to execute this code  
} catch (error) {  
    // If an error occurs, execute this code  
}
```

Let's add the `console.log` and also log any errors:

```
try {  
    console.log(x);  
} catch (error) {  
    console.log('Error:', error);  
}
```

Now, if we run this code, we will see the error in the console:

```
Error: ReferenceError: x is not defined
```

So we still get the same message, but it does not say `uncaught`. This is because we are handling the error.

## When To Use Try...Catch

There is no reason to fill your code with try...catch statements. You should use a try...catch statement when you are doing something that may result in an error and you plan on handling that error. For example, if you are making an API request and you are not sure if the request will succeed, you can use a try...catch statement to handle the error if it occurs. We can then do something with the error, like display it on the page:

```
try {
  console.log(x);
} catch (error) {
  document.body.append(error);
}
```

There may be times where you do some kind of check and then want to throw an error if the check fails. For example, let's say we have a function that takes in a number and doubles it. We want to make sure that the argument is a number.

```
function double(number) {
  if (isNaN(number)) {
    throw new Error(number + ' is not a number');
  }

  return number * 2;
}
```

Let's try to call this function with a string inside a try...catch statement:

```
try {
  const y = double('a');
} catch (error) {
  console.log(error);
}
```

We get the following error, which we can now do whatever we want with:

```
Error: a is not a number
```

## Finally Block

The finally block is a block of code that will always run, regardless of whether an error occurs or not. It is very similar to the `.finally()` method that we can use with `.then()`.

Let's say we want to log a message after the try...catch statement runs. We can do this with the finally block:

```
try {
  const y = double('a');
} catch (error) {
  console.log(error);
} finally {
  console.log('This will always run');
}
```

In the next video, we will look at how to use try...catch with Async & Await.

# Async/Await Error Handling

---

In the last lesson, we looked at how to use try...catch statements. In this lesson, we will look at how to use try...catch with Async & Await.

Let's use the same `getUsers()` example from the other lesson. I'll use the arrow function version:

```
const getUsers = async () => {
  const response = await fetch('https://jsonplaceholder.typicode.com/users');
  const data = await response.json();

  document.querySelector('#output').innerHTML = data
    .map(
      (user) =>
        <li>${user.name}</li>
    )
    .join(' ');
};

getUsers();
```

Remember, with the fetch API, if we want to handle status codes like 404, we need to check the `status` or `ok` property on the response object.

In that case, we will throw an error:

```
const getUsers = async () => {
  const response = await fetch('http://jsonplaceholder.typicode.com/users');

  // Add this
  if (response.status !== 200) {
    throw new Error('Something went wrong');
  }

  const data = await response.json();

  document.querySelector('#output').innerHTML = data
    .map(
      (user) =>
        <li>${user.name}</li>
    )
    .join(' ');
};

getUsers();
```

In order to catch and use that error, we can use a try...catch statement. Let's also change the URL to something that will return a 404 error.

```
const getUsers = async () => {
  try {
    // const response = await
    fetch('http://jsonplaceholder.typicode.com/users');
    const response = await fetch('http://httpstat.us/404');

    if (response.status !== 200) {
      throw new Error('Something went wrong');
    }

    const data = await response.json();

    document.querySelector('#output').innerHTML = data
      .map(
        (user) => `
          <li>${user.name}</li>
        `
      )
      .join(' ');
  } catch (error) {
    document.querySelector('#output').innerHTML = error;
  }
};

getUsers();
```

And remember from the earlier lesson, you can check for specific responses and give specific error messages if you want.

Again, you do not always have to use `try...catch`, just like you don't always need to use a `then...catch`. It depends on your use case.

# Async / Await With Multiple Promises

So we know that we can chain promises together using multiple `.then()` methods. We can also use `promise.all` to wait for multiple promises to resolve. But what about using `async / await` to wait for multiple promises to resolve?

Let's look at our `getData()` function from a past lesson.

```
function getData(endpoint) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', endpoint);

    xhr.onreadystatechange = function () {
      if (this.readyState === 4) {
        if (this.status === 200) {
          resolve(JSON.parse(this.responseText));
        } else {
          reject('Error: Something went wrong');
        }
      }
    };
    setTimeout(() => {
      xhr.send();
    }, Math.floor(Math.random() * 3000) + 1000);
  });
}
```

It fetches data from an endpoint and returns a promise. It can take anywhere from 1 to 4 seconds to resolve. I first showed you how to do this with callbacks, then with promises and chaining `.then()` methods, and now we're going to do it with `async / await`.

Just for reference, let's look at how we did this with promise chaining.

```
getData('./movies.json')
  .then((movies) => {
    console.log(movies);
    return getData('./actors.json');
  })
  .then((actors) => {
    console.log(actors);
    return getData('./directors.json');
  })
  .then((directors) => {
    console.log(directors);
  });
```

We returned a value from each `.then()` method, and that value was passed to the next `.then()` method. This is ok. It looks better than callbacks, but it's still a little messy. We can do better.

```
async function getAllData() {  
  const movies = await getData('./movies.json');  
  console.log(movies);  
  const actors = await getData('./actors.json');  
  console.log(actors);  
  const directors = await getData('./directors.json');  
  console.log(directors);  
}  
  
getAllData();
```

Or even better:

```
async function getAllData() {  
  const movies = await getData('./movies.json');  
  const actors = await getData('./actors.json');  
  const directors = await getData('./directors.json');  
  console.log(movies, actors, directors);  
}  
  
getAllData();
```

We were able to use `async / await` to wait for multiple promises to resolve and it looks much nicer. We also have access to the data from each promise in the same scope.

## Using `fetch` instead of `XMLHttpRequest`

In the `getData()` function, we used the `XMLHttpRequest` object to fetch data from an endpoint. We could also use the `fetch()` API to do the same thing and get rid of the `getData()` function completely.

```
async function getAllDataWithFetch() {  
  const moviesRes = await fetch('./movies.json');  
  const movies = await moviesRes.json();  
  
  const actorsRes = await fetch('./actors.json');  
  const actors = await actorsRes.json();  
  
  const directorsRes = await fetch('./directors.json');  
  const directors = await directorsRes.json();  
  
  console.log(movies, actors, directors);  
}
```

```
getAllDataWithFetch();
```

There is no returning from `.then()` methods. The code looks synchronous, but it's not. It's still asynchronous. We're just using `async / await` to make it look synchronous.

## Promise.all()

We can also still use `Promise.all()` to wait for multiple promises to resolve.

```
async function getAllDataWithPromiseAll() {
  const [moviesRes, actorsRes, directorsRes] = await Promise.all([
    fetch('./movies.json'),
    fetch('./actors.json'),
    fetch('./directors.json'),
  ]);

  const movies = await moviesRes.json();
  const actors = await actorsRes.json();
  const directors = await directorsRes.json();

  console.log(movies, actors, directors);
}

getAllDataWithPromiseAll();
```

If you want, you can still use `.then()` on the response promise to get the data:

```
async function getAllDataWithPromiseAll2() {
  const [movies, actors, directors] = await Promise.all([
    fetch('./movies.json').then((response) => response.json()),
    fetch('./actors.json').then((response) => response.json()),
    fetch('./directors.json').then((response) => response.json()),
  ]);

  console.log(movies, actors, directors);
}

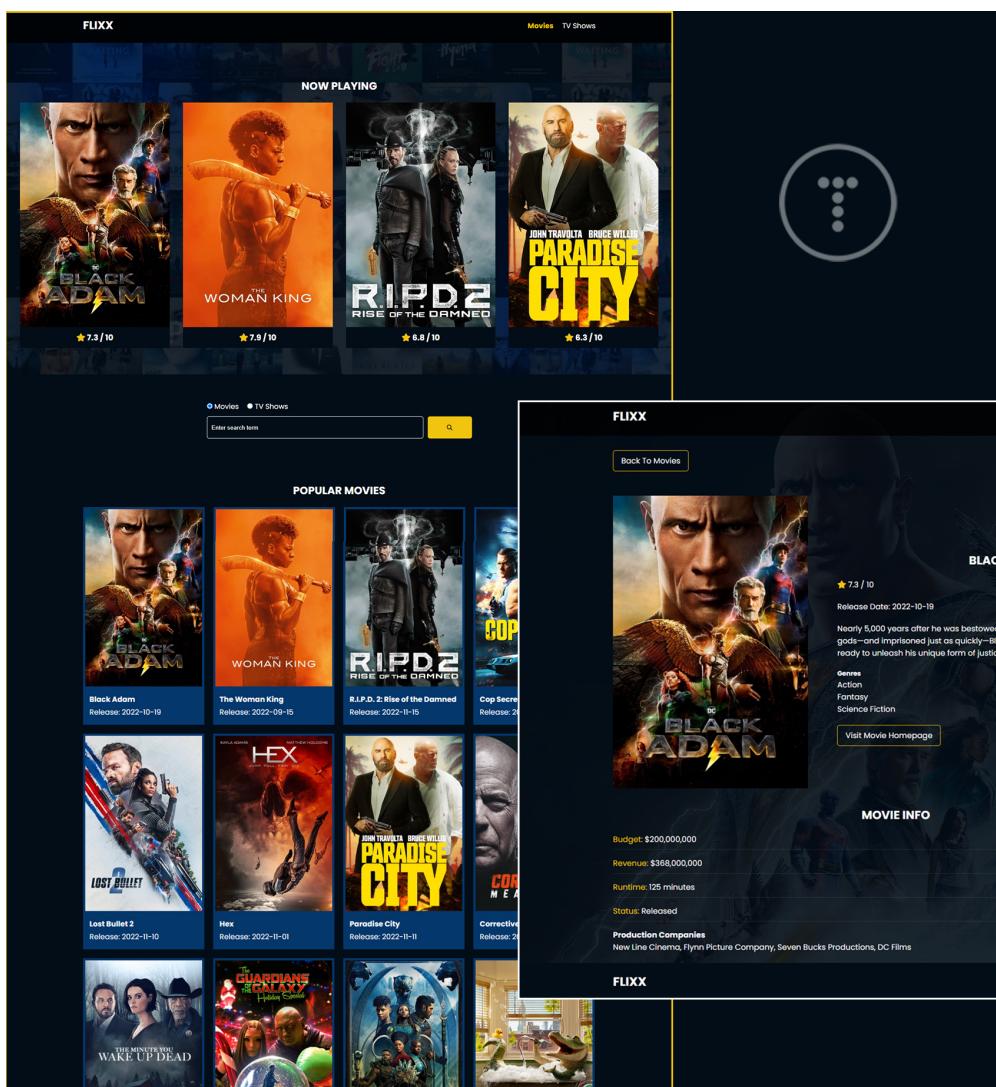
getAllDataWithPromiseAll2();
```

# Flixx App Project Intro

In this project, we are going to build a movie info application that will fetch data from the [TMDB API](#) and display it in a nice way. The app will have the following features:

- Display the most popular movies
- Display the most popular TV shows
- Show specific movie details
- Show specific TV show details
- Display the 'Now Playing' movies in a slider
- Allow users to search for movies and TV shows
- Have pagination for search results

We will be working with the fetch api, async/await, the DOM and pretty much everything that we have talked about up until this point.



# Theme Overview & Prep

---

I am going to include an initial theme (`flix-app-theme`) that includes the HTML, CSS and any assets that the website uses. This will allow you to get started quickly and focus on the JavaScript.

The theme includes the following:

- HTML: All of the html pages including `index.html`, `shows.html`, `movie-details.html`, `show-details.html` and `search.html`.
- CSS: The main stylesheet (`style.css`) and the `spinner.css` stylesheet
- Images: There is a `no-image.jpg` that we will use for when there is no image available for a movie or TV show. There is also a showcase/slider background image
- Libraries: There is a `lib` folder which includes the CSS/JS files for the Swiper Slider that we will use. There is also a font-awesome CSS file as well as a `webfonts` folder that contains the font files for font-awesome.

## Getting Started

Simply rename the `flix-app-theme` folder to `flix-app` and open it in your code editor. Then open the `index.html` file in VS Code and run it with `Live Server`. Don't just open the `index.html` on your filesystem because the CSS and JS files may not load. You should have some kind of server running that will serve the files.

You may want to initialize your git repo at this point and commit the files. You can then create a new repo on GitHub and push the files to it.

# API Overview & API Key

---

We will be using the [The Movie Database \(TMDb\) API](#) to get the data for our app. You will need to create an account and get an API key. The API key is free and you can get one by following these steps:

1. Go to the [TMDb website](#)
2. Click on the [Sign Up](#) button in the top right corner
3. Fill out the form and click [Sign Up](#)
4. Once you are logged in, click on your avatar in the top right corner and select [Settings](#)
5. Click on the [API](#) tab
6. Click on the [Create](#) button
7. Enter a name for your API key and click [Save](#)
8. Copy the API key and paste it into a text file. You will need it later

All of the API documentation at the <https://developers.themoviedb.org/3>. We are using version 3 of the API. The API documentation is very good and you should read through it to get a better understanding of how the API works.

Here are the endpoints that we will need to hit to get the data for the app. You will need to send your API key with every request.

- [Get Popular Movies](#)
- [Get Popular TV Shows](#)
- [Get Movie Details](#)
- [Get TV Show Details](#)
- [Get Now Playing Movies](#)
- [Search Movies](#)
- [Search TV Shows](#)

# Page Router & Active Link

---

Since we will have multiple HTML pages in the project and we need to run different JavaScript code on each page, we will be creating a very simple page router using a `switch` statement. We will also be creating a function to add an `active` class to the active link in the navigation.

## App State

At the very top of the JS file, we will have a `global` object for our global state. This will include any values that we will need in multiple places. We will create a `currentPage` property and set it to `window.location.pathname`. This will give us the current page path.

```
const global = {
  currentPage: window.location.pathname,
};
```

## Init() and Router

We will have an `init()` function that will run when the DOM loads and this is where we will add the router code. Right now, we will just add a `console.log()` statement for each page. We will add the code to the `init()` function later.

```
// Init App
function init() {
  switch (global.currentPage) {
    case '/':
    case '/index.html':
      console.log('Home');
      break;
    case '/shows.html':
      console.log('Shows');
      break;
    case '/movie-details.html':
      console.log('Movie Details');
      break;
    case '/tv-details.html':
      console.log('TV Details');
      break;
    case '/search.html':
      console.log('Search');
      break;
  }
}

document.addEventListener('DOMContentLoaded', init);
```

## Highlight Active Link

Let's create a function to highlight the nav link for the current page. We will add a `active` class to the link. We will also remove the `active` class from any other links. We will call the `highlightActiveLink()` function in the `init()` function.

```
// Highlight active link
function highlightActiveLink() {
  const links = document.querySelectorAll('.nav-link');
  links.forEach((link) => {
    if (link.getAttribute('href') === global.currentPage) {
      link.classList.add('active');
    }
  });
}
```

```
// Init App
function init() {
  // ...
  highlightActiveLink();
}
```

Now you should see the link highlight and the `console.log()` for each page.

# Display Popular Movies

---

In this section, we will be displaying the popular movies on the home screen of the app. We will be using the [Get Popular Movies](#) endpoint to get the data. The endpoint returns a list of movies sorted by popularity.

## fetchAPIData() function

We are going to create a function called `fetchAPIData()` that will be used to fetch data from the API. We will pass in the URL for the endpoint as a parameter.

```
async function fetchAPIData(endpoint) {
  const API_KEY = ''; // Put your api key here
  const API_URL = 'https://api.themoviedb.org/3/';

  const response = await fetch(
    `${API_URL}${endpoint}?api_key=${API_KEY}&language=en-US`
  );

  const data = await response.json();

  return data;
}
```

This function will reach out to the server and get the data that we need using the endpoint that is passed in. It will then return the data as a JavaScript object.

It is important to understand that the API Key is public in this case. This is only for development and small projects. For real production projects, you should not store your API Key in your code. You should store it in a server and make requests from there. That is beyond what we have covered so far. I may do that later on in the course.

## displayPopularMovies() function

Now we can create a function to display the popular movies. We will call the `fetchAPIData()` function and pass in the endpoint. We will then loop through the results and display the data. We will add a link to the movie details page as well. This will include the id of the movie.

```
async function displayPopularMovies() {
  const { results } = await fetchAPIData('movie/popular');

  results.forEach((movie) => {
    const div = document.createElement('div');
    div.classList.add('card');
    div.innerHTML = `
```

```

<a href="movie-details.html?id=${movie.id}">
${{
  movie.poster_path
    ? ``
    : ``
  }
</a>
<div class="card-body">
  <h5 class="card-title">${movie.title}</h5>
  <p class="card-text">
    <small class="text-muted">Release: ${movie.release_date}</small>
  </p>
</div>
`;

document.querySelector('#popular-movies').appendChild(div);
});
}

```

## Remove Hardcoded Movies

In the `index.html` file, we have some hardcoded movies. We can remove those now. Every div that has the class of `card` can be removed. So the div with the id of `popular-movies` should be empty.

We will call the `displayPopularMovies()` function in the `switch` statement.

```

switch (global.currentPage) {
  case '/':
  case '/index.html':
    displayPopularMovies();
    break;
  // ...
}

```

Now you should see 20 popular movies on the page

# Spinner and Display Popular Shows

---

One thing I want to do before we fetch the TV shows, is add a spinner to display while data is being fetched. This will give the user some feedback that the app is working.

In the HTML, there is

```
<div class="spinner"></div>
```

If you add a class of `show` to that div, it will display the spinner. We will add this class when we are fetching the data. Let's create a function to show and hide the spinner.

```
function showSpinner() {
  document.querySelector('.spinner').classList.add('show');
}

function hideSpinner() {
  document.querySelector('.spinner').classList.remove('show');
}
```

Now in the `fetchAPIData()` function, we can call the `showSpinner()` function before we fetch the data. We can also call the `hideSpinner()` function after the data is fetched.

```
async function fetchAPIData(endpoint) {
  const API_KEY = '';
  const API_URL = 'https://api.themoviedb.org/3/';

  showSpinner();

  const response = await fetch(
    `${API_URL}${endpoint}?api_key=${API_KEY}&language=en-US`
  );

  const data = await response.json();

  hideSpinner();

  return data;
}
```

## displayPopularShows() function

Now we can create a function to display the popular shows. We will call the `fetchAPIData()` function and pass in the endpoint. We will then loop through the results and display the data. We will add a link to the show details page as well. This will include the id of the show.

```
async function displayPopularShows() {
  const { results } = await fetchAPIData('tv/popular');

  results.forEach((show) => {
    const div = document.createElement('div');
    div.classList.add('card');
    div.innerHTML = `
      <a href="tv-details.html?id=${show.id}">
        ${
          show.poster_path
            ? `
            />`
            : `
            />`
        }
      </a>
      <div class="card-body">
        <h5 class="card-title">${show.name}</h5>
        <p class="card-text">
          <small class="text-muted">Air Date: ${show.first_air_date}</small>
        </p>
      </div>
    `;
    document.querySelector('#popular-shows').appendChild(div);
  });
}
```

Remove all of the hardcoded show divs with the class of `card` in the `shows.html` file. You should now see a list of popular TV shows.

Now add the `displayPopularShows()` function to `switch` statement

```
// Init App
function init() {
  switch (global.currentPage) {
    // ...
    case '/shows.html':
```

```
    displayPopularShows();
    break;
    // ...
}
}
```

# Movie Details Page

---

Right now, when we click a movie on the homepage, it brings us to a url like this:

/movie-details.html?id=550988

We need to create a function to get the movie id from the url and then fetch the movie data from the API. We will call it `displayMovieDetails()`.

We will use the `window.location.search` property to get the query string from the url. The query string will look like this: `?id=550988`. We will use the `split()` method to split the string at the `=` sign and then get the second item in the array, which will be the movie id.

We then use the `fetchAPIData()` function to fetch the movie data from the API. We pass in the `movie/${movieId}` as the endpoint. Then we add the movie data to the DOM.

```
async function displayMovieDetails() {
  const movieId = window.location.search.split('=')[1];

  const movie = await fetchAPIData(`movie/${movieId}`);

  const div = document.createElement('div');

  div.innerHTML = `
    <div class="details-top">
      <div>
        ${
          movie.poster_path
            ? ``
            : ``
        }
      </div>
      <div>
        <h2>${movie.title}</h2>
        <p>
          <i class="fas fa-star text-primary"></i>
          ${movie.vote_average.toFixed(1)} / 10
        </p>
        <p class="text-muted">Release Date: ${movie.release_date}</p>
        <p>
          ${movie.overview}
        </p>
      </div>
    </div>
  `;
```

```

        </p>
        <h5>Genres</h5>
        <ul class="list-group">
            ${movie.genres.map((genre) => `<li>${genre.name}</li>`).join('')}
        </ul>
        <a href="${{
            movie.homepage
        }" target="_blank" class="btn">Visit Movie Homepage</a>
    </div>
</div>
<div class="details-bottom">
    <h2>Movie Info</h2>
    <ul>
        <li><span class="text-secondary">Budget:</span> ${addCommasToNumber(
            movie.budget
        )}</li>
        <li><span class="text-secondary">Revenue:</span> ${addCommasToNumber(
            movie.revenue
        )}</li>
        <li><span class="text-secondary">Runtime:</span> ${{
            movie.runtime
        }} minutes</li>
        <li><span class="text-secondary">Status:</span> ${movie.status}</li>
    </ul>
    <h4>Production Companies</h4>
    <div class="list-group">
        ${movie.production_companies
            .map((company) => `<span>${company.name}</span>`)
            .join(', ')}
    </div>
</div>
`;

document.querySelector('#movie-details').appendChild(div);
}

```

Notice that we used a function called `addCommasToNumber()` to add commas to the budget and revenue numbers. Let's create that now.

```

function addCommasToNumber(number) {
    return number.toString().replace(/\B(?=(\d{3})+(?!\d))/g, ',');
}

```

This function uses the `replace()` method to replace every three digits with a comma. It takes in a regular expression. We have not gone over regular expressions yet, but it is something that we will get to soon. For now, just know that it is a way to match patterns in strings.

Call the `displayMovieDetails()` function in the `switch` statement:

```
// Init App
function init() {
  switch (global.currentPage) {
    // ...
    case '/movie-details.html':
      displayMovieDetails();
      break;
    // ...
  }
}
```

# Details Page Backdrop

The API that we are working with gives us a link to a really cool backdrop image of the movie or show that we are fetching. We can use this image to create a nice background for our details page.

Let's create a function called `displayBackgroundImage` that will take in a type and a path. The type will be either `movie` or `tv` and the path will be the backdrop path that we get from the API.

```
// Display Backdrop On Details Pages
function displayBackgroundImage(type, outputPath) {
  const overlayDiv = document.createElement('div');
  overlayDiv.style.backgroundImage =
`url(${https://image.tmdb.org/t/p/original/${backdropPath}})`;
  overlayDiv.style.backgroundSize = 'cover';
  overlayDiv.style.backgroundPosition = 'center';
  overlayDiv.style.backgroundRepeat = 'no-repeat';
  overlayDiv.style.height = '100vh';
  overlayDiv.style.width = '100vw';
  overlayDiv.style.position = 'absolute';
  overlayDiv.style.top = '0';
  overlayDiv.style.left = '0';
  overlayDiv.style.zIndex = '-1';
  overlayDiv.style.opacity = '0.1';

  if (type === 'movie') {
    document.querySelector('#movie-details').appendChild(overlayDiv);
  } else {
    document.querySelector('#show-details').appendChild(overlayDiv);
  }
}
```

So what we are doing here is taking in the type, which will be tv or movie and then the path, which will come from the api. Then we are just setting the image and a bunch of background styles to the overlay div. We are also appending it to the movie details or show details div depending on the type.

We can then call this function in our `displayMovieDetails` and `displayShowDetails` functions.

```
async function displayMovieDetails() {
  const movieId = window.location.search.split('=')[1];

  const movie = await fetchAPIData(`movie/${movieId}`);

  // Overlay for background image
  displayBackgroundImage('movie', movie.backdrop_path);

  // ...
}
```

```
async function displayShowDetails() {
  const showId = window.location.search.split('=')[1];

  const show = await fetchAPIData(`tv/${showId}`);

  // Overlay for background image
  displayBackgroundImage('tv', show.backdrop_path);

  // ...
}
```

# Show Details Page

---

Just like we did with movies, we need to create a page to show the details of a TV show. This will be on the `tv-details.html` page. We will create a function called `displayShowDetails()` to get the show id from the url and then fetch the show data from the API.

```
// Display Show Details
async function displayShowDetails() {
  const showId = window.location.search.split('=')[1];

  const show = await fetchAPIData(`tv/${showId}`);

  // Overlay for background image
  displayBackgroundImage('tv', show.backdrop_path);

  const div = document.createElement('div');

  div.innerHTML = `
    <div class="details-top">
      <div>
        ${{
          show.poster_path
            ? `
            `
            : `
            `
        }}
      </div>
    </div>
    <div>
      <h2>${show.name}</h2>
      <p>
        <i class="fas fa-star text-primary"></i>
        ${show.vote_average.toFixed(1)} / 10
      </p>
      <p class="text-muted">Last Air Date: ${show.last_air_date}</p>
      <p>
        ${show.overview}
      </p>
      <h5>Genres</h5>
      <ul class="list-group">
        ${show.genres.map((genre) => `<li>${genre.name}</li>`).join('')}
      </ul>
      <a href="#{
        show.id
      }">View Details</a>
    </div>
  `;

  document.body.appendChild(div);
}
```

```

        show.homepage
    }" target="_blank" class="btn">Visit show Homepage</a>

```

```

</div>
</div>
<div class="details-bottom">
    <h2>Show Info</h2>
    <ul>
        <li><span class="text-secondary">Number of Episodes:</span> ${show.number_of_episodes}</li>
        <li><span class="text-secondary">Last Episode To Air:</span> ${show.last_episode_to_air.name}</li>
        <li><span class="text-secondary">Status:</span> ${show.status}</li>
    </ul>
    <h4>Production Companies</h4>
    <div class="list-group">
        ${show.production_companies
            .map((company) => `<span>${company.name}</span>`)
            .join(', ')}

    </div>
</div>
`;

document.querySelector('#show-details').appendChild(div);
}

```

Call the function in the `switch` statement:

```

// Init App
function init() {
    switch (global.currentPage) {
        // ...
        case '/tv-details.html':
            displayShowDetails();
            break;
        // ...
    }
}

```

# Search Function & Custom Alert

---

In this lesson, we will start to add our search functionality. We will also create a custom alert function that will display a message to the user.

## Search Info In Global State

We are going to need to access certain search info in multiple places in our app, so I am going to add a search object to the `global` object. I am also going to move the api key and api url into the global object because we will have a separate function for the search requests and we need those pieces of info in that function.

```
const global = {
  currentPage: window.location.pathname,
  search: {
    term: '',
    type: '',
    page: 1,
    totalPages: 1,
  },
  api: {
    apiKey: '',
    apiUrl: 'https://api.themoviedb.org/3/',
  },
};
```

The page stuff is for pagination. We will add that later. We are concerned with the search term and the search type right now.

## search() Function

Let's create the `search()` function that will fetch the search results from the API and display them. We run a function called `searchAPIData()` to fetch the data from the API. We will create that function next. For now, we are just logging the data.

We also call a function called `showAlert()` to display a message to the user if something went wrong. We will create that function as well.

```
async function search() {
  const queryString = window.location.search;
  const urlParams = new URLSearchParams(queryString);

  // Add the type and term to the global object
  global.search.type = urlParams.get('type');
  global.search.term = urlParams.get('search-term');
```

```

if (global.search.term !== '' && global.search.term !== null) {
  const results = await searchAPIData();
  console.log(results);
} else {
  showAlert('Please enter a search term');
}

```

## searchAPIData() Function

Notice we used a different function for getting data. We are going to create a function called `searchAPIData()` because the request is a bit different than the other requests we have made.

```

// Make Request To Search
async function searchAPIData() {
  const API_KEY = global.api.apiKey;
  const API_URL = global.api.apiUrl;

  showSpinner();

  const response = await fetch(
    `${API_URL}search/${global.search.type}?api_key=${API_KEY}&language=en-US&query=${global.search.term}`
  );

  const data = await response.json();

  hideSpinner();

  return data;
}

```

Also, change the api key and url lines in the `fetchAPIData()` function to use the global object.

```

async function fetchAPIData() {
  const API_KEY = global.api.apiKey;
  const API_URL = global.api.apiUrl;

  // ...
}

```

Our form in the html has an `action` attribute that submits to the `search.html` page. We will call the `search()` function within the `switch` statement on the search page.

```

switch (global.currentPage) {
  case '/search.html':

```

```
    search();
    break;
    // ...
}
```

## showAlert() Function

We are going to create a function that will display a message to the user. It will take in a message and a class name. The class name will be used to style the alert. It will use the `alert` class and the class name that is passed in. The class name will be `error` by default.

```
// Show Alert
function showAlert(message, className = 'error') {
  const alertEl = document.createElement('div');
  alertEl.classList.add('alert', className);
  alertEl.appendChild(document.createTextNode(message));
  document.querySelector('#alert').appendChild(alertEl);

  setTimeout(() => alertEl.remove(), 3000);
}
```

# Display Search Results

---

In the last lesson, we made it so that we could search for a term and get the data and log it. Now we will display the results on the page. Let's add to the `search()` function.

```
async function search() {
  const queryString = window.location.search;
  const urlParams = new URLSearchParams(queryString);

  // Add the type and term to the global object
  global.search.type = urlParams.get('type');
  global.search.term = urlParams.get('search-term');

  if (global.search.term !== '' && global.search.term !== null) {
    const { results, total_pages, page } = await searchAPIData();

    if (results.length === 0) {
      showAlert('No results found');
      return;
    }

    displaySearchResults(results);

    document.querySelector('#search-term').value = '';
  } else {
    showAlert('Please enter a search term');
  }
}
```

Now we are destructuring the results from the `searchAPIData()` function. We are also getting the `total_pages` and `page` from the API. We will use these to display the pagination. For now, we only need the `results` array. We will pass that into the `displaySearchResults()` function. Let's create that function now.

```
function displaySearchResults(results) {
  results.forEach((result) => {
    const div = document.createElement('div');
    div.classList.add('card');
    div.innerHTML =
      `
        \${
          result.poster\_path
            ? `
        
      
```

```

        }"
      />`  

        : ``  

    }  

</a>  

<div class="card-body">  

<h5 class="card-title">${  

    global.search.type === 'movie' ? result.title : result.name  

}</h5>  

<p class="card-text">  

<small class="text-muted">Release: ${  

    global.search.type === 'movie'  

? result.release_date  

: result.first_air_date  

}</small>  

</p>  

</div>  

`;
`;  

document.querySelector('#search-results').appendChild(div);
});  

}

```

Now we are showing the results on the page in cards. You can remove any hardcoded data from the HTML and CSS. In the next lesson, we will implement the pagination.

# Search Pagination

---

We are now able to search for movies and TV shows, but we only display 20 at a time. We need to add pagination to the search results.

## Add Dynamic Heading

One thing I want to do before that is show a heading that says "X of Y results found for MOVIE NAME". Let's start by adding `totalResults` to the `global` object:

```
const global = {
  // ...
  search: {
    term: '',
    type: '',
    page: 1,
    totalPages: 1,
    totalResults: 0, // Add this
  },
  // ...
};
```

In the `search()` function, we will add `totalResults` to the `global` object:

```
// ...
const { results, total_pages, page, total_results } = await searchAPIData();
global.search.totalResults = total_results;
// ...
```

Now at the bottom of `displaySearchResults()`, right above where we add the results to the DOM, add the following code:

```
document.querySelector('#search-results-heading').innerHTML = `
  <h2>${results.length} of ${global.search.totalResults} Results
  for ${global.search.term}</h2>
`;
```

## Add Pagination

The pagination HTML is in the `search.html` file. We need to cut the following code:

```

<div class="pagination">
  <button class="btn btn-primary" id="prev">Prev</button>
  <button class="btn btn-primary" id="next">Next</button>
  <div class="page-counter">Page 1 of 5</div>
</div>

```

Keep the div with the `id` of pagination. The class and everything within it can be cut because we will insert it with JavaScript and make it dynamic.

## displayPagination() Function

This function will display the HTML and include a `prev` and `next` event listener. When the user clicks on the `prev` or `next` button, it will change the page number and call the `searchAPIData()` function again with the page number needed. Then we will call `displaySearchResults()` with the new results.

```

// Create & Display Pagination For Search
function displayPagination() {
  const div = document.createElement('div');
  div.classList.add('pagination');
  div.innerHTML =
    `<button class="btn btn-primary" id="prev">Prev</button>
     <button class="btn btn-primary" id="next">Next</button>
     <div class="page-counter">Page ${global.search.page} of
      ${global.search.totalPages}</div>
    `;
  document.querySelector('#pagination').appendChild(div);

  // Disable prev button if on first page
  if (global.search.page === 1) {
    document.querySelector('#prev').disabled = true;
  }

  // Disable next button if on last page
  if (global.search.page === global.search.totalPages) {
    document.querySelector('#next').disabled = true;
  }

  // Next page
  document.querySelector('#next').addEventListener('click', async () => {
    global.search.page++;
    const { results, total_pages } = await searchAPIData();
    displaySearchResults(results);
  });

  // Prev page
  document.querySelector('#prev').addEventListener('click', async () => {
    global.search.page--;
    const { results, total_pages } = await searchAPIData();
  });
}

```

```
        displaySearchResults(results);
    });
}
```

We do need to update the `searchAPIData()` function to include the page number:

```
const response = await fetch(
  `${API_URL}search/${global.search.type}?api_key=${API_KEY}&language=en-US&query=${global.search.term}&page=${global.search.page}`
);
```

That's it! Our application is complete! We can now search for movies and TV shows, and we can paginate through the results.

# What Are Web/Browser APIs?

---

Web APIs, also known as browser APIs are a set of tools that allow developers to interact with the browser and its features. They are not part of the core JavaScript language, but they are built into the browser and can be accessed using JavaScript. We don't need to use any external libraries to use them, they are already built into the browser.

We have already used a few Web APIs in previous chapters, such as the DOM API, the Fetch API and the localStorage API. In this section, we are going to dive into some others and create some small projects using them.

The web APIs that we will be going over include:

- The Geolocation API
- The Canvas API
- The Web Audio API
- The Web Video API
- The WebRTC API
- The Web Workers API
- The WebSockets API
- The Web Animation API
- The Web Speech API
- The Web Bluetooth API

# Geolocation API

---

The Geolocation API allows us to access the user's location. It is part of the HTML5 specification and is supported by all modern browsers. For privacy reasons, the user is asked for permission to report location information. If the user grants permission, the location information is made available to the web page. I'm sure most of you have seen this when the webpage asks you if it can access your location.

## Why use the Geolocation API?

The Geolocation API is useful for a variety of applications. For example, if you are building a weather app, you can use the Geolocation API to get the user's location and then use that information to display the weather for that location. You can also use the Geolocation API to build a map app that shows the user's location on a map.

There are 2 main methods that we can use to access the user's location:

- `navigator.geolocation.getCurrentPosition()` - This method is used to get the user's current position.
- `navigator.geolocation.watchPosition()` - This method is used to watch the user's position for changes.

## Getting the user's current position

The `navigator.geolocation.getCurrentPosition()` method is used to get the user's current position. It takes 2 callback functions as parameters as well as a third optional parameter that is an object containing options.

- `success` - This callback function is called if the user's location is successfully retrieved.
- `error` - This callback function is called if there is an error getting the user's location.
- `options` - Options about how to get the user's location. This is an optional parameter.

Let's call the method, with named functions as the callbacks:

```
navigator.geolocation.getCurrentPosition(curSuccess, curError, curOptions);
```

Let's create the success function:

```
function curSuccess(pos) {  
    const coords = pos.coords;  
  
    console.log('Current position is:');  
    console.log(`Latitude : ${coords.latitude}`);  
    console.log(`Longitude: ${coords.longitude}`);  
    console.log(`More or less ${coords.accuracy} meters.`);  
}
```

Let's create the error function:

```
function curError(err) {
  console.warn(`ERROR(${err.code}): ${err.message}`);
}
```

Let's create the options object:

```
const curOptions = {
  enableHighAccuracy: true, // use GPS if available
  timeout: 5000, // wait 5 seconds before giving up
  maximumAge: 0, // do not use a cached position
};
```

As you can see, it will show your latitude and longitude as well as the accuracy of the location. The accuracy is the radius of a 95% confidence interval. In other words, it is the radius of a circle centered at the given position, where the probability of the device's position being within the circle is 95%.

My location is based on my VPN so it is not accurate. If you are using a VPN, make sure to disable it before testing this.

## Watching the user's position

Let's call the `navigator.geolocation.watchPosition()` method, with named functions as the callbacks:

```
navigator.geolocation.watchPosition(watchSuccess, watchError, watchOptions);
```

Let's create the success function. We can also create a target that when that target is reached, it can do something. In this case, we will just log a message to the console when the target is reached. We won't see the message, because obviously I am not moving, while I'm sitting here creating a tutorial. But if you were to move and reach the target, you would see the message in the console.

```
let target = {
  latitude: 41.7568588,
  longitude: -71.6789246,
};
```

```
function watchSuccess(pos) {
  const coords = pos.coords;
```

```
if (
  target.latitude === coords.latitude &&
  target.longitude === coords.longitude
) {
  console.log('Congratulations, you reached the target');
  navigator.geolocation.clearWatch(id);
}
}
```

Let's create the error function:

```
function watchError(err) {
  console.warn(`ERROR(${err.code}): ${err.message}`);
}
```

And the options:

```
const watchOptions = {
  enableHighAccuracy: false, // use GPS if available
  timeout: 5000, // wait 5 seconds before giving up
  maximumAge: 0, // do not use a cached position
};
```

This method returns an ID that we can use to stop watching the user's position. We used this in the success function with the `navigator.geolocation.clearWatch()` method to stop watching the user's position.

```
const id = navigator.geolocation.watchPosition(
  watchSuccess,
  watchError,
  watchOptions
);
console.log(id);
```

# Plotting Your Location on a Map

---

In this lesson, you will learn how to use the Geolocation API along with a map library called [Leaflet](#) to plot your location on a map.

The Leaflet website is located at <https://leafletjs.com/>. If you go to [Tutorials->Get Started](#), you will see the code to add Leaflet to your project. We can include the CDN in our HTML. Your HTML page should look like this:

```
<!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" />
    <link
      rel="stylesheet"
      href="https://unpkg.com/leaflet@1.9.3/dist/leaflet.css"
      integrity="sha256-kLaT2GOSpHechhsOzzB+fInD+zUyjE2L1fWPgU04xyI="
      crossorigin=""
    />
    <title>My Location</title>

    <style>
      #map {
        height: 600px;
        width: 600px;
      }
    </style>
  </head>
  <body>
    <h1>My Location</h1>

    <div id="map"></div>
    <script
      src="https://unpkg.com/leaflet@1.9.3/dist/leaflet.js"
      integrity="sha256-WBkoX0wTeyKclOHuWtc+i2uENFpDZ9YPdf5Hf+D7ewM="
      crossorigin=""
    ></script>
    <script src="script.js"></script>
  </body>
</html>
```

We brought in the CSS and JS and added a div for the map to display. I also set the size to 600x600. Now we need to add the JavaScript to plot our location on the map.

Let's start by displaying the map:

```
// Initialize the map
const map = L.map('map').setView([0, 0], 2);

// Add a tile layer to the map
L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
  maxZoom: 19,
  attribution:
    '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>',
}).addTo(map);
```

This is just the basic code from the docs to display a map. It will show the whole world. Now let's add a marker to the map:

```
// Add a marker to the map
const marker = L.marker([0, 0]).addTo(map);
```

Right now, it will just display a marker in the middle of the map. We need to get the user's location and update the marker's position. We can do this using the Geolocation API:

```
// Use the HTML5 geolocation API to get the current location
navigator.geolocation.getCurrentPosition(function (position) {
  // Get the coordinates of the current location
  const lat = position.coords.latitude;
  const lng = position.coords.longitude;

  // Set the marker to the current location and zoom the map
  marker.setLatLng([lat, lng]).update();
  map.setView([lat, lng], 13);

  // Add a popup to the marker
  marker.bindPopup('<b>Hello world!</b><br>This is my current location');
});
```

So we are just using the `getCurrentPosition` method to get the current location. We then get the latitude and longitude from the `position` object. We then set the marker's position to the current location and zoom the map to 13.

We can also add a popup:

```
navigator.geolocation.getCurrentPosition(function (position) {
  // ...

  // Add a popup to the marker
```

```
marker.bindPopup('<b>Hello world!</b><br>This is my current location');
});
```

That's it. We can now see our location on a map.

# Canvas API

---

The Canvas API is used to draw graphics, on the fly, via JavaScript. It is part of the HTML5 specification and is supported by all modern browsers.

## Why use the Canvas API?

The Canvas API is useful for a variety of applications. It's used a lot for gaming and animation, data visualization, photo manipulation and more.

## How does it work?

The Canvas API is based on a 2D rendering context. This context is created by the `<canvas>` element. The `<canvas>` element is a container for graphics, similar to an `<img>` element. The `<canvas>` element has a `width` and `height` attribute, just like an `<img>` element. The `width` and `height` attributes define the size of the canvas, in pixels.

Let's create a simple example. We need to add the `<canvas>` element to our HTML page. You can add a width and a height. Also, add an ID, so that we can access it from our JavaScript. I am also going to put a border on the canvas, so that we can see it.

```
<!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" />
    <script src="script.js" defer></script>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
    <title>Canvas</title>
  </head>
  <body>
    <h1>Canvas</h1>

    <canvas id="my-canvas" width="600" height="600"></canvas>
  </body>
</html>
```

## Creating the rendering context

Now, we need to create the rendering context. We can do this in our JavaScript file. We need to get a reference to the `<canvas>` element, and then call the `getContext()` method. The `getContext()` method

takes a string argument, which is the type of context we want. In this case, we want a 2D context, so we pass in the string "`2d`". This will return a 2D rendering context object, which we can store in a variable.

```
const canvas = document.getElementById('my-canvas');
// Create a 2D context
const ctx = canvas.getContext('2d');
```

## Drawing on the canvas

Let's start by drawing a rectangle on the canvas:

```
ctx.fillStyle = 'green';
ctx.fillRect(10, 10, 150, 100);
```

Here we used the context object to set the fill style to green. We then used the `fillRect()` method to draw a rectangle. The `fillRect()` method takes four arguments: the x and y coordinates of the top-left corner, and the width and height of the rectangle.

## Drawing a circle

Let's draw a circle on the canvas. We can do this by using the `arc()` method. The `arc()` method takes five arguments: the x and y coordinates of the center of the circle, the radius, the start angle and the end angle. The start angle and end angle are measured in radians. We can use the `Math.PI` constant to convert degrees to radians. For example, `Math.PI / 2` is 90 degrees, and `Math.PI` is 180 degrees.

```
ctx.beginPath();
ctx.arc(300, 300, 100, 0, Math.PI * 2);
ctx.fillStyle = 'red';
ctx.fill();
```

## Drawing text

We can draw text on the canvas using the `fillText()` method. The `fillText()` method takes three arguments: the text to draw, the x and y coordinates of the top-left corner of the text.

```
ctx.font = '30px Arial';
ctx.fillStyle = 'blue';
ctx.fillText('Hello World', 10, 300);
ctx.strokeText('Hello World', 10, 300);
```

## Drawing lines

We can draw lines from one point to another using the `moveTo()` and `lineTo()` methods. The `moveTo()` method takes two arguments: the x and y coordinates of the starting point. The `lineTo()` method takes two arguments: the x and y coordinates of the ending point.

```
ctx.beginPath();
ctx.moveTo(10, 10);
ctx.lineTo(300, 300);
ctx.lineTo(10, 300);
ctx.lineTo(300, 10);
ctx.strokeStyle = 'orange';
ctx.stroke();
```

`stroke()` is used to draw the line.

## Drawing an image

We can draw an image on the canvas using the `drawImage()` method. The `drawImage()` method takes five arguments: the image to draw, the x and y coordinates of the top-left corner of the image, and the width and height of the image. The image can be an `<img>` element, a `<video>` element or a `<canvas>` element.

Let's add an image on the page. I have a `ball.png` image in the sandbox files.

```

```

```
const image = document.querySelector('img');
image.style.display = 'none';

image.addEventListener('load', () => {
  ctx.drawImage(image, 270, 270, 50, 50);
});
```

We brought in the image, and set the display to `none`, so that it is not visible on the page. We then added an event listener to the image, so that we can draw it on the canvas once it has loaded. We used the `drawImage()` method to draw the image on the canvas.

## Draw quadratic curves

We can draw quadratic curves using the `quadraticCurveTo()` method. This method takes four arguments: the x and y coordinates of the control point, and the x and y coordinates of the end point.

```
ctx.beginPath();
ctx.moveTo(10, 10);
```

```
ctx.quadraticCurveTo(300, 300, 10, 300); // (cp1x, cp1y, x, y)
ctx.strokeStyle = 'purple';
ctx.stroke();
```

There are other types of curves that we can draw, such as bezier curves and arcs. You can find more information about these in the [MDN documentation](#).

This is all very basic and may leave you saying, what is this actually good for? It's actually very powerful, especially when you introduce animation, which we will do in the next lesson.

# requestAnimationFrame

---

The `requestAnimationFrame()` method tells the browser that you wish to perform an animation and requests that the browser call a specified function to update an animation before the next repaint. The method takes as an argument a callback to be invoked before the repaint.

Let's go through this very slowly so that I can explain everything in an understandable way.

We are going to create a function called `step` to be called before the next repaint. For now, we will just log 'hello' to the console.

```
function step() {  
  console.log('Hello');  
}
```

Now, I am going to call the `requestAnimationFrame()` method, passing in the `step` function as an argument. This will tell the browser to call the `step` function before the next repaint.

```
requestAnimationFrame(step);
```

No big deal, we just see 'Hello' logged to the console once. That was just a single call to the `requestAnimationFrame()` method. Let's make it a little more interesting. Let's call the `requestAnimationFrame()` method inside the `step` function. This will tell the browser to call the `step` function before the next repaint, and then call the `step` function again before the next repaint, again and again, creating a loop.

```
function step() {  
  console.log('Hello');  
  requestAnimationFrame(step);  
}  
  
requestAnimationFrame(step);
```

Now, you're seeing 'Hello' logged to the console over and over again, in a never-ending loop.

We can also pass in a timestamp to the `step` function. This is the number of milliseconds since the page was loaded. We can use this to calculate the time that has passed since the last repaint. Let's log this to the console.

```
function step(timestamp) {  
  console.log(timestamp);
```

```
    requestAnimationFrame(step);
}
```

Now, you're seeing a number logged to the console over and over again, in a never-ending loop. This number is the number of milliseconds since the page was loaded. We can use this to calculate the time that has passed since the last repaint.

```
let start;
let done = false;

function step(timestamp) {
  if (start === undefined) {
    start = timestamp;
  }

  const elapsed = timestamp - start;
  console.log(elapsed);

  requestAnimationFrame(step);
}
```

Here, we created some variables and we are using the `start` variable to store the timestamp of the first repaint. We are then using the `elapsed` variable to store the number of milliseconds that have passed since the first repaint. We are then logging this to the console.

Let's make the animation stop after 2 seconds. We can do this by using the `done` variable. We are going to set the `done` variable to `true` after 2 seconds have passed. We are then going to check the `done` variable inside the `step` function. If the `done` variable is `true`, we are going to return from the function. This will stop the animation.

```
function step(timestamp) {
  if (start === undefined) {
    start = timestamp;
  }

  const elapsed = timestamp - start;

  if (elapsed > 2000) {
    done = true;
  }
  if (done) {
    return;
  }
  requestAnimationFrame(step);
}
```

Now let's actually have something happen. Let's move the soccer ball image. Or, I'm sorry, the "football", image for my non-American friends. We will do this by changing the transform property of the image. I am also going to change the animation to 5 seconds.

```
function step(timestamp) {
  if (start === undefined) {
    start = timestamp;
  }

  const elapsed = timestamp - start;

  if (elapsed > 5000) {
    done = true;
  }
  if (done) {
    return;
  }
  image.style.transform = `translateX(${elapsed / 20}px)`;
  requestAnimationFrame(step);
}
```

Let's rotate the ball as well. Just add `rotate` to the transform property.

```
image.style.transform = `translateX(${elapsed / 20}px) rotate(${{
  elapsed / 20
}deg)`;
```

Now that you know how the `canvas` API works as well as the `requestAnimationFrame()` method, in the next lesson, we will create an animated clock project.

# Animated Clock Project - Part 1

---

In this project, we will use the `canvas` api along with the `requestAnimationFrame` method to create an animated clock. This will be 2 parts. In this part, we will create the clock and animate it. In the next part, we will add a form to change the clocks colors and save the clock as an image.

## The HTML

The HTML is very simple. We just have a `canvas` element. In part 2, we will have a form to change the look of the clock, but for now, we just have the clock on the canvas.

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

    <title>Animated Clock</title>
  </head>
  <body>
    <canvas id="canvas" width="500" height="500"></canvas>
  </body>
</html>
```

Let's start by creating our `clock()` function. This function will be called by the `requestAnimationFrame` method. For now, let's get the current time and create a new canvas context

```
function clock() {
  const now = new Date();
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');
}

clock();
```

For now, we are not using `requestAnimationFrame` because we just want to test our code without any animation. We will add the `requestAnimationFrame` method later.

## Setting Up the Canvas

Before we draw anything, we need to set some initial values for the canvas and where we will draw the clock. I want to draw in the center of the canvas. Let's add the following code:

```

function clock() {
  const now = new Date();
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');

  // Setup canvas
  ctx.save(); // Save the default state
  ctx.clearRect(0, 0, 500, 500); // Clear the entire canvas
  ctx.translate(250, 250); // Move the origin to the center of the canvas
  ctx.rotate(-Math.PI / 2); // Rotate the canvas -90 degrees
  // Set some default styles
  ctx.strokeStyle = '#000000';
  ctx.fillStyle = '#f4f4f4';
  ctx.lineWidth = 5;
  ctx.lineCap = 'round';
}

clock();

```

Remember, this will run before every frame, so we need to save the default state of the canvas and restore it after we are done drawing. We do this with the `save()` and `restore()` methods. We will also use these methods before and after we draw certain parts of the clock.

We also need to clear the canvas before we draw anything. We also need to move the origin to the center of the canvas and rotate the canvas -90 degrees. This will make it easier to draw the clock. We then set some default styles for the canvas.

## Draw clock face

Now, let's add the code to draw the clock face and border:

```

function clock() {
  const now = new Date();
  const canvas = document.getElementById('canvas');
  const ctx = canvas.getContext('2d');

  // Setup canvas
  ctx.save(); // Save the default state
  ctx.clearRect(0, 0, 500, 500); // Clear the entire canvas
  ctx.translate(250, 250); // Move the origin to the center of the canvas
  ctx.rotate(-Math.PI / 2); // Rotate the canvas -90 degrees

  // Set some default styles
  ctx.strokeStyle = '#000000';
  ctx.fillStyle = '#f4f4f4';
  ctx.lineWidth = 5;
  ctx.lineCap = 'round';

  // Draw clock face

```

```

    ctx.save();
    ctx.beginPath();
    ctx.lineWidth = 14;
    ctx.strokeStyle = '#800000';
    ctx.arc(0, 0, 142, 0, Math.PI * 2, true);
    ctx.stroke();
    ctx.fill();
    ctx.restore();

    ctx.restore(); // Restore the default state
}

clock();

```

We used the `save()` and `restore()` methods to save the default state of the canvas before we draw the clock border and restore it after we are done drawing the clock border. We also set the `lineWidth` to 14 and the `strokeStyle` to `#800000` before we draw the clock border. We created the circle using the `arc()` method. The `arc()` method takes 6 parameters: the x and y coordinates of the center of the circle, the radius of the circle, the starting angle, the ending angle, and a boolean value that determines if the circle is drawn clockwise or counter-clockwise. We set the starting angle to 0 and the ending angle to `Math.PI * 2` to draw a full circle.

## Draw hour lines

Now, we will draw the hour lines on the clock. I am just going to add this code rather than repeating all of the code again. Just be sure to add it right below the clock border `restore()` and above the last default state `restore()`:

```

// Draw hour lines
ctx.save();
for (let i = 0; i < 12; i++) {
  ctx.beginPath();
  ctx.rotate(Math.PI / 6);
  ctx.moveTo(100, 0);
  ctx.lineTo(120, 0);
  ctx.stroke();
}
ctx.restore();

```

Here, we used a for loop to draw 12 lines. We used the `rotate()` method to rotate the canvas 30 degrees before drawing each line. We then used the `moveTo()` and `lineTo()` methods to draw the lines.

## Draw minute lines

```

// Draw minute lines
ctx.save();
ctx.lineWidth = 4;

```

```

for (let i = 0; i < 60; i++) {
    // Do not draw on hour lines
    if (i % 5 !== 0) {
        ctx.beginPath();
        ctx.moveTo(117, 0);
        ctx.lineTo(120, 0);
        ctx.stroke();
    }
    ctx.rotate(Math.PI / 30);
}
ctx.restore();

```

We did the same thing here except we used a for loop to draw 60 lines. We used the `rotate()` method to rotate the canvas 6 degrees before drawing each line. We then used the `moveTo()` and `lineTo()` methods to draw the lines. We used the `stroke()` method to draw the lines. We also used an if statement to only draw the lines that are not on the hour marks. We used the `lineWidth` to 4 to make the minute marks thinner than the hour marks.

## Creating the hands

Now we have to create the clock hands. The placement on these are based on the current time. Let's first, get the hours, minutes, and seconds from the current time and log it.

```

const hr = now.getHours() % 12;
const min = now.getMinutes();
const sec = now.getSeconds();
console.log(` ${hr}:${min}:${sec}`);

```

## Set hand colors

Right now, the `strokeStyle` is set to `#000`. I want the hands to be `#800000` like the border. So we need to set that here before we draw the hands:

```

// Color for hands
ctx.strokeStyle = '#800000';

```

## Draw hour hand

Now, let's draw the hour hand:

```

// Write Hours
ctx.save();
ctx.rotate(
    (Math.PI / 6) * hr + (Math.PI / 360) * min + (Math.PI / 21600) * sec
);

```

```
ctx.lineWidth = 14;
ctx.beginPath();
ctx.moveTo(-20, 0);
ctx.lineTo(80, 0);
ctx.stroke();
ctx.restore();
```

The hour hand will not move yet, because we aren't using `requestAnimationFrame`, but it should be in the correct place based on the time.

We used the `rotate()` method to rotate the canvas based on the current time. We used the `lineWidth` to 14 to make the hour hand thicker than the minute and second hands. We then used the `moveTo()` and `lineTo()` methods to draw the hour hand.

## Draw minute hand

```
// Draw minute hand
ctx.save();
ctx.rotate((Math.PI / 30) * min + (Math.PI / 1800) * sec);
ctx.lineWidth = 10;
ctx.beginPath();
ctx.moveTo(-28, 0);
ctx.lineTo(112, 0);
ctx.stroke();
ctx.restore();
```

## Draw second hand

```
// Draw second hand
ctx.save();
ctx.rotate((sec * Math.PI) / 30);
ctx.strokeStyle = '#FF7F50';
ctx.fillStyle = '#FF7F50';
ctx.lineWidth = 6;
ctx.beginPath();
ctx.moveTo(-30, 0);
ctx.lineTo(100, 0);
ctx.stroke();
ctx.beginPath();
ctx.arc(0, 0, 10, 0, Math.PI * 2, true);
ctx.fill();
ctx.restore();
```

We also created the center circle of the clock here with the `arc()` method. We made it the same color as the second hand.

## Animating the clock

Now, we just need to add the `requestAnimationFrame` to animate the clock:

```
function clock() {  
    // ...  
    requestAnimationFrame(clock);  
}  
  
requestAnimationFrame(clock);
```

We now have a working clock!

# Animated Clock Project - Part 2

In the last lesson, we used `canvas` to draw an animated clock. In this video, we will add a form with some CSS to change the clock colors and save the clock as an image. This is a good opportunity for you to try the color changing on your own. It is all stuff that we have worked with. You can get the value of the color inputs in the JS and simply add them as the `strokeStyle` and `fillStyle` where needed.

## Creating the form

Here is the HTML for the form. This is actually where we want to set the default colors. The clock will directly correspond to the colors in the form.

```
<div class="card">
  <h3>Clock Style</h3>
  <form>
    <div class="form-input">
      <label for="face-color">Face Color</label>
      <input type="color" value="#f4f4f4" id="face-color" />
    </div>
    <div class="form-input">
      <label for="border-color">Border Color</label>
      <input type="color" value="#800000" id="border-color" />
    </div>
    <div class="form-input">
      <label for="line-color">Number Lines Color</label>
      <input type="color" value="#000000" id="line-color" />
    </div>
    <div class="form-input">
      <label for="large-hand-color">Large Hands Color</label>
      <input type="color" value="#800000" id="large-hand-color" />
    </div>
    <div class="form-input">
      <label for="second-hand-color">Second-hand Hand Color</label>
      <input type="color" value="#FF7F50" id="second-hand-color" />
    </div>
    <div class="form-input">
      <button id="save-btn" type="submit">Save As Image</button>
    </div>
  </form>
</div>
<canvas id="canvas" width="500" height="500"></canvas>
```

We have a little bit of CSS as well...

```
@import url('https://fonts.googleapis.com/css2?
family=Roboto:wght@300;400;700&display=swap');
```

```

body {
  font-family: 'Roboto', sans-serif;
  font-size: 18px;
}

button {
  background: #333;
  color: #fff;
  border: 0;
  padding: 10px 20px;
  border-radius: 5px;
  cursor: pointer;
}

.card {
  background: #f4f4f4;
  border-left: 1px solid #ccc;
  border-bottom: 1px solid #ccc;
  padding: 7px 20px;
  width: 300px;
  position: absolute;
  bottom: 0;
  left: 0;
}

.form-input {
  margin-bottom: 20px;
}

label {
  margin-right: 10px;
  font-weight: bold;
}

```

Now, in the JS, we need to get the values of the inputs. We can set variables to either the elements themselves or the values of the elements. I like to set them to the element and then just add `.value()` to get the value.

```

const faceColor = document.getElementById('face-color');
const borderColor = document.getElementById('border-color');
const lineColor = document.getElementById('line-color');
const largeHandColor = document.getElementById('large-hand-color');
const secondHandColor = document.getElementById('second-hand-color');

```

## Face & Border Color

For the face color, we are using the default `fillStyle`, So we need to add a `fillStyle` to the face code between the `save()` and `restore()`. We will set the `fillStyle` to `faceColor.value`. The `strokeStyle`

is the border, so we will set that to `borderColor.value`.

```
// Draw clock face/border
ctx.save();
ctx.beginPath();
ctx.lineWidth = 14;
ctx.fillStyle = faceColor.value; // Add this
ctx.strokeStyle = borderColor.value; // Add this
ctx.arc(0, 0, 142, 0, Math.PI * 2, true);
ctx.stroke();
ctx.fill();
ctx.restore();
```

Now if you change those color inputs, it should reflect on the clock.

## Number Lines Color

For the number lines, we need to add a `strokeStyle` right before the loop for the hour and minute lines. You could also put it inside the loop, but I like to keep it outside. We will set the `strokeStyle` to `lineColor.value`.

```
// Draw hour lines
ctx.save();
ctx.strokeStyle = lineColor.value; // Add this
for (let i = 0; i < 12; i++) {
  ctx.beginPath();
  ctx.rotate(Math.PI / 6);
  ctx.moveTo(100, 0);
  ctx.lineTo(120, 0);
  ctx.stroke();
}
ctx.restore();

// Draw minute lines
ctx.save();
ctx.lineWidth = 4;
ctx.strokeStyle = lineColor.value; // Add this
for (let i = 0; i < 60; i++) {
  if (i % 5 !== 0) {
    ctx.beginPath();
    ctx.moveTo(117, 0);
    ctx.lineTo(120, 0);
    ctx.stroke();
  }
  ctx.rotate(Math.PI / 30);
}
ctx.restore();
```

## Large Hand Color

```
// Draw hour hand
ctx.save();
ctx.rotate(
  (Math.PI / 6) * hr + (Math.PI / 360) * min + (Math.PI / 21600) * sec
);
ctx.strokeStyle = largeHandColor.value; // Add this
ctx.lineWidth = 14;
ctx.beginPath();
ctx.moveTo(-20, 0);
ctx.lineTo(80, 0);
ctx.stroke();
ctx.restore();

// Draw minute hand
ctx.save();
ctx.rotate((Math.PI / 30) * min + (Math.PI / 1800) * sec);
ctx.strokeStyle = largeHandColor.value; // Add this
ctx.lineWidth = 10;
ctx.beginPath();
ctx.moveTo(-28, 0);
ctx.lineTo(112, 0);
ctx.stroke();
ctx.restore();
```

## Second Hand Color

We will add the value to the `strokeStyle` and `fillStyle` for the second hand. The `fillStyle` pertains to the clock center dot.

```
// Draw second hand
ctx.save();
ctx.rotate((sec * Math.PI) / 30);
ctx.strokeStyle = secondHandColor.value; // Add this
ctx.fillStyle = secondHandColor.value; // Add this
ctx.lineWidth = 6;
ctx.beginPath();
ctx.moveTo(-30, 0);
ctx.lineTo(100, 0);
ctx.stroke();
ctx.beginPath();
ctx.arc(0, 0, 10, 0, Math.PI * 2, true);
ctx.fill();
ctx.restore();
```

## Save As Image

To save a canvas as an image, we can use the `toDataURL()` method. This will return a base64 encoded string. We can then essentially create a link and click it programmatically. We will do this in an event listener for the save button.

```
document.getElementById('save-btn').addEventListener('click', () => {
  const canvas = document.getElementById('canvas');
  const dataURL = canvas.toDataURL('image/png');
  const link = document.createElement('a');
  link.download = 'clock.png';
  link.href = dataURL;
  link.click();
});
```

Now, you can change the clock to any style that you want and download it as an image.

# Web Audio API

---

The Audio API is a powerful tool for creating audio in the browser. It is a low-level API that allows you to create audio nodes and connect them together to create a sound. The nodes can be used to create effects, filters, and more. The API is also used to play audio files.

## <audio> Element

The `<audio>` element is used to play audio files. It is a simple element, but you can interact with it using the JavaScript API. The element has a `src` attribute that is used to set the source of the audio file. Of course, you can change this within the JavaScript to play different audio files. I will show you how to do this in the next lesson where we build an audio player. The `<audio>` element also has a `controls` attribute that will show the default browser controls. Let's embed a simple audio file in our HTML.

`summer.mp3` is included in the resources for this lesson. You can also use a remote link to an audio file.

```
<audio src="summer.mp3" id="audio" controls></audio>
```

Since, I used the `controls` attribute, I can control the audio, however, usually we will create our own interface and control everything using the JavaScript API. Let's look at the JavaScript API.

## JavaScript API

Here are some of the common methods and properties of the audio element.

- `play()`
- `pause()`
- `currentTime`
- `duration`
- `volume`

Let's remove the `controls` attribute and create our own buttons as well as a `current-time` div and a volume slider. This will be really ugly for now, but in another lesson, we will create a really cool looking audio player.

```
<audio src="summer.mp3" id="audio"></audio>

<button id="play">Play</button>
<button id="pause">Pause</button>
<button id="stop">Stop</button>
<div id="current-time"></div>
<input id="volume" type="range" min="0" max="1" step="0.01" value="1" />
```

In the JavaScript, let's bring all of the elements in

```
const audio = document.getElementById('audio');
const play = document.getElementById('play');
const pause = document.getElementById('pause');
const stop = document.getElementById('stop');
const currentTime = document.getElementById('current-time');
const volume = document.getElementById('volume');
```

## play() & pause()

Then we can create events for the play and pause buttons and call the `play()` and `pause()` methods.

```
play.addEventListener('click', () => {
  audio.play();
});

pause.addEventListener('click', () => {
  audio.pause();
});
```

## Stopping Audio

We use `pause()` because there is no `stop()` method. We can use `pause()` and set the `currentTime` to `0` to stop the audio.

```
const stop = document.getElementById('stop');

stop.addEventListener('click', () => {
  audio.pause();
  audio.currentTime = 0;
});
```

Now, the track is reset to the beginning.

## timeUpdate Event

The `timeupdate` event is fired when the `currentTime` is updated. In our project we will create a progress bar, but for now, let's just view the `currentTime`

```
audio.addEventListener('timeupdate', () => {
  // console.log(audio.currentTime);
  currentTime.innerHTML = audio.currentTime;
});
```

## Volume

We can change the volume, by adding a listener on to the slider

```
volume.addEventListener('change', () => {  
    audio.volume = volume.value;  
});
```

## Adding filters & effects

We can create audio interfaces, etc just by bringing in the audio element like we did and then using the API methods, however if you wanted to add filters, effects, etc, you would need to create an audio context and audio nodes. We will cover this in the next lesson.

# Audio Context & Nodes For Filtering & Effects

---

In the last lesson, we saw how to control audio using the web audio API. However, the API is more powerful than just being used to control. We can also add all kinds of effects and filters. I am not going to pretend I know about this stuff, as far as adding "biquad filters" and all that. I'm far from an audio engineer. But I will show you how you can use them.

So we are going to use the same audio file as before. We will create an audio context and then create a source node. We will then create a **biquad filter** and connect the source node to the filter. Then we will connect the filter to the destination. The destination is the speakers. We will also create an oscillator node and connect it to the destination. This will allow us to hear the audio file and the oscillator at the same time.

In our HTML, we have a new **filter** select and filter frequency slider as well as an **oscillator** select and an oscillator frequency slider as well as a stop button for the oscillator.

```
<audio src="summer.mp3" id="audio"></audio>
<button id="play">Play</button>
<button id="pause">Pause</button>
<button id="stop">Stop</button>
<div id="current-time"></div>

<input id="volume" type="range" min="0" max="1" step="0.01" value="1" />

<h4>Filter</h4>
<!-- Filter Select -->
<select id="filter">
  <option value="none">None</option>
  <option value="lowpass">Lowpass</option>
  <option value="highpass">Highpass</option>
  <option value="bandpass">Bandpass</option>
  <option value="lowshelf">Lowshelf</option>
  <option value="highshelf">Highshelf</option>
  <option value="peaking">Peaking</option>
  <option value="notch">Notch</option>
  <option value="allpass">Allpass</option>
</select>

<label for="filter-frequency">Filter Frequency</label>
<input
  id="filter-frequency"
  type="range"
  min="0"
  max="1000"
  step="1"
  value="440"
/>
```

```

<h4>Oscillator</h4>
<select id="oscillator">
  <option value="none">None</option>

  <option value="sine">Sine</option>
  <option value="square">Square</option>
  <option value="sawtooth">Sawtooth</option>
  <option value="triangle">Triangle</option>
</select>

<label for="oscillator-frequency">Oscillator Frequency</label>
<input
  id="oscillator-frequency"
  type="range"
  min="0"
  max="1000"
  step="1"
  value="440"
/>

<button id="oscillator-stop">Stop Oscillator</button>

```

Let's bring in some additional elements:

```

const filterEl = document.getElementById('filter');
const filterFrequency = document.getElementById('filter-frequency');
const oscillatorEl = document.getElementById('oscillator');
const oscillatorFrequency = document.getElementById('oscillator-frequency');
const oscillatorStop = document.getElementById('oscillator-stop');

```

## Add Context & nodes

In order to do stuff like this, we need to create an `audio context` and then create a `source node`. We don't want to do this in the global scope because Chrome will complain about auto play policies. So we will just create the context variable and then initialize it in the `play` function. We will do the same for the source node.

```

let audioCtx, gainNode;

play.addEventListener('click', () => {
  // Create an AudioContext & connect the source to the gain node
  audioCtx = new AudioContext();
  const source = audioCtx.createMediaElementSource(audio);
  gainNode = audioCtx.createGain();
  source.connect(gainNode);
  gainNode.connect(audioCtx.destination);
  audio.play();
});

```

Our player still works the same. We can play, pause, etc. We just created the audio context, then the source and we connected the source to the gain node. Then we connected the gain node to the destination. The destination, which are the speakers.

## Filter

Now we are going to make it so that we can select a filter. This is called a **biquad filter**. We will create the filter variable in the global scope so that the frequency event handler can access it.

```
let filter;
filterEl.addEventListener('change', () => {
  filter = audioCtx.createBiquadFilter();
  filter.type = filterEl.value;
  gainNode.disconnect();
  gainNode.connect(filter);
  filter.connect(audioCtx.destination);
});
```

Let's also add the frequency event handler.

```
filterFrequency.addEventListener('change', () => {
  filter.frequency.value = filterFrequency.value;
  gainNode.disconnect();
  gainNode.connect(filter);
  filter.connect(audioCtx.destination);
});
```

We created a **biquad filter** and then set the type to the value of the select. We then disconnect the gain node from the destination and connect it to the filter. Then we connect the filter to the destination. This will allow us to hear the audio file with the filter applied. We also created the frequency handler.

Let's try it out and play the audio file. We can select a filter and hear the difference. We can also change the frequency of the filter. We can do this by adding an event listener to the filter frequency slider.

## Oscillator

We can also add an oscillator. We will create the oscillator variable in the global scope so that the frequency event handler can access it. We also need to create a stop button handler.

```
let oscillator;
oscillatorEl.addEventListener('change', () => {
  oscillator = audioCtx.createOscillator();
  oscillator.type = oscillatorEl.value;
  oscillator.frequency.value = 440; // 440Hz
```

```
oscillator.connect(audioCtx.destination);
oscillator.start();
});

oscillatorFrequency.addEventListener('change', () => {
  oscillator.frequency.value = oscillatorFrequency.value;
});

oscillatorStop.addEventListener('click', () => {
  oscillator.stop();
});
```

Now if we choose a type of oscillator, we can hear it. It's VERY annoying. We can also change the frequency of the oscillator. We can also stop the oscillator.

Again, I can't really tell you the point of an oscillator. Maybe some of you are really good with audio and can tell me. But this is how you can use it with the audio API.

# Music Player Project

---

Now, we are going to create a really nice looking music player. This is the same player that we used in my 20 vanilla projects course. I figured, why create a new project that will show you the same exact thing? This player also has some CSS transitions and animations.

There is a folder called `music` with 3 songs in it. There is also a folder called `images` with 3 cover images. We will be using these songs and images for the player. In this project, the music file and image file should be the same.

## The HTML

Let's go ahead and take a look at the HTML. It is really simple and minimal.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <link
      rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.10.2/css/all.min.css"
    />
    <link rel="stylesheet" href="style.css" />
    <script src="script.js" defer></script>
    <title>Music Player</title>
  </head>
  <body>
    <h1>Music Player</h1>

    <div class="music-container" id="music-container">
      <div class="music-info">
        <h4 id="title"></h4>
        <div class="progress-container" id="progress-container">
          <div class="progress" id="progress"></div>
        </div>
      </div>

      <audio src="music/ukulele.mp3" id="audio"></audio>

      <div class="img-container">
        
      </div>
      <div class="navigation">
        <button id="prev" class="action-btn">
          <i class="fas fa-backward"></i>
        </button>
```

```

        <button id="play" class="action-btn action-btn-big">
            <i class="fas fa-play"></i>
        </button>
        <button id="next" class="action-btn">
            <i class="fas fa-forward"></i>
        </button>
    </div>
</div>
</body>
</html>

```

Notice, we did include the Font Awesome 5 CDN. We will be using some icons for the play, prev, next button, etc. We also have an area for the progress bar, an `h4` for the title, and an image for the cover. We have the default audio file and image as the ukulele song, but this player will have 3 songs and cover images.

## The CSS

Here is the CSS. I am not going to go over it all. It is pretty simple and self explanatory. The one thing I want to point out is the `.play` class. This class is applied to the music container via javascript. This is what makes the image rotate as well as the title slide up.

```

@import url('https://fonts.googleapis.com/css?family=Lato&display=swap');

* {
    box-sizing: border-box;
}

body {
    background-image: linear-gradient(
        0deg,
        rgba(247, 247, 247, 1) 23.8%,
        rgba(252, 221, 221, 1) 92%
    );
    height: 100vh;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    font-family: 'Lato', sans-serif;
    margin: 0;
}

.music-container {
    background-color: #fff;
    border-radius: 15px;
    box-shadow: 0 20px 20px 0 rgba(252, 169, 169, 0.6);
    display: flex;
    padding: 20px 30px;
    position: relative;
}

```

```
margin: 100px 0;
z-index: 10;
}

.img-container {
  position: relative;
  width: 110px;
}

.img-container::after {
  content: '';
  background-color: #fff;
  border-radius: 50%;
  position: absolute;
  bottom: 100%;
  left: 50%;
  width: 20px;
  height: 20px;
  transform: translate(-50%, 50%);
}

.img-container img {
  border-radius: 50%;
  object-fit: cover;
  height: 110px;
  width: inherit;
  position: absolute;
  bottom: 0;
  left: 0;
  animation: rotate 3s linear infinite;

  animation-play-state: paused;
}

.music-container.play .img-container img {
  animation-play-state: running;
}

@keyframes rotate {
  from {
    transform: rotate(0deg);
  }

  to {
    transform: rotate(360deg);
  }
}

.navigation {
  display: flex;
  align-items: center;
  justify-content: center;
  z-index: 1;
```

```
}

.action-btn {
    background-color: #fff;
    border: 0;
    color: #dfdbdf;
    font-size: 20px;
    cursor: pointer;
    padding: 10px;
    margin: 0 20px;
}

.action-btn.action-btn-big {
    color: #cdc2d0;
    font-size: 30px;
}

.action-btn:focus {
    outline: 0;
}

.music-info {
    background-color: rgba(255, 255, 255, 0.5);
    border-radius: 15px 15px 0 0;
    position: absolute;
    top: 0;
    left: 20px;
    width: calc(100% - 40px);
    padding: 10px 10px 10px 150px;
    opacity: 0;
    transform: translateY(0%);
    transition: transform 0.3s ease-in, opacity 0.3s ease-in;
    z-index: 0;
}

.music-container.play .music-info {
    opacity: 1;
    transform: translateY(-100%);
}

.music-info h4 {
    margin: 0;
}

.progress-container {
    background: #fff;
    border-radius: 5px;
    cursor: pointer;
    margin: 10px 0;
    height: 4px;
    width: 100%;
}
```

```
.progress {  
    background-color: #fe8daa;  
    border-radius: 5px;  
    height: 100%;  
    width: 0%;  
    transition: width 0.1s linear;  
}
```

## The JavaScript

Now for the fun part. First we will bring in all of the elements that we need.

```
const musicContainer = document.getElementById('music-container');  
const playBtn = document.getElementById('play');  
const prevBtn = document.getElementById('prev');  
const nextBtn = document.getElementById('next');  
  
const audio = document.getElementById('audio');  
const progress = document.getElementById('progress');  
const progressContainer = document.getElementById('progress-container');  
const title = document.getElementById('title');  
const cover = document.getElementById('cover');
```

## Setup & Load Song

Next, we will create an array of songs and their cover images. Remember, they are named the same, so we just have an array with 3 titles/image names

```
const songs = ['hey', 'summer', 'ukulele'];
```

We need a way to keep track of the songs. So we will create a song index. We will start with the song with the index of 2.

```
let songIndex = 2;
```

Now we will load the song into the DOM. We will set the title, audio source, and cover image source.

```
loadSong(songs[songIndex]);  
  
function loadSong(song) {  
    title.innerText = song;  
    audio.src = `music/${song}.mp3`;
```

```
    cover.src = `images/${song}.jpg`;
}
```

## Play & Pause Song

The play and pause button are the same, so we will have an event listener and then choose to run either the `playSong` or `pauseSong` function based on if the song is playing or not.

This is also where we will either add or remove the `play` class from the music container. This will make the image spin and display the song info.

```
playBtn.addEventListener('click', () => {
  const isPlaying = musicContainer.classList.contains('play');

  if (isPlaying) {
    pauseSong();
  } else {
    playSong();
  }
});
```

Let's create both methods

```
function playSong() {
  musicContainer.classList.add('play'); // Makes image spin and display info
  playBtn.querySelector('i.fas').classList.remove('fa-play');
  playBtn.querySelector('i.fas').classList.add('fa-pause');

  audio.play();
}

function pauseSong() {
  musicContainer.classList.remove('play');
  playBtn.querySelector('i.fas').classList.add('fa-play');
  playBtn.querySelector('i.fas').classList.remove('fa-pause');

  audio.pause();
}
```

## Prev & Next Buttons

Now, let's add the prev and next functionality. We need to change the song index and then load the song.

```
prevBtn.addEventListener('click', prevSong);
nextBtn.addEventListener('click', nextSong);
```

```

function prevSong() {
    songIndex--;

    if (songIndex < 0) {
        songIndex = songs.length - 1;
    }

    loadSong(songs[songIndex]);

    playSong();
}

function nextSong() {
    songIndex++;

    if (songIndex > songs.length - 1) {
        songIndex = 0;
    }

    loadSong(songs[songIndex]);

    playSong();
}

```

## Show Progress Bar

We need to show the progress bar. It should be relative to where in the song is being played. We listen for the `timeupdate` event on the audio element. This event is fired when the time indicated by the `currentTime` attribute has been updated. We will then calculate the percentage of the song that has been played and set the width of the progress bar.

```

audio.addEventListener('timeupdate', updateProgress);

function setProgress(e) {
    const width = this.clientWidth;
    const clickX = e.offsetX;
    const duration = audio.duration;

    audio.currentTime = (clickX / width) * duration;
}

```

## Set Progress Bar

We also want to be able to click on the progress bar and have the song go to that point

```
progressContainer.addEventListener('click', setProgress);

function setProgress(e) {
  const width = this.clientWidth;
  const clickX = e.offsetX;
  const duration = audio.duration;

  audio.currentTime = (clickX / width) * duration;
}
```

That's it! We now have a cool little music player.

# Drum Machine Project

---

In this project, we will create a drum machine that plays sounds when the user presses specific keys. We will use the `Audio` object to play the sounds.

## The HTML

Let's start with the HTML.

```
<header>
  <h1>Drum Machine</h1>
</header>
<div class="keys">
  <div data-key="65" class="key">
    <kbd>A</kbd>
    <span class="sound">clap</span>
  </div>
  <div data-key="83" class="key">
    <kbd>S</kbd>
    <span class="sound">hihat</span>
  </div>
  <div data-key="68" class="key">
    <kbd>D</kbd>
    <span class="sound">kick</span>
  </div>
  <div data-key="70" class="key">
    <kbd>F</kbd>
    <span class="sound">openhat</span>
  </div>
  <div data-key="71" class="key">
    <kbd>G</kbd>
    <span class="sound">boom</span>
  </div>
  <div data-key="72" class="key">
    <kbd>H</kbd>
    <span class="sound">ride</span>
  </div>
  <div data-key="74" class="key">
    <kbd>J</kbd>
    <span class="sound">snare</span>
  </div>
  <div data-key="75" class="key">
    <kbd>K</kbd>
    <span class="sound">tom</span>
  </div>
</div>

<audio data-key="65" src="sounds/clap.wav"></audio>
<audio data-key="83" src="sounds/hihat.wav"></audio>
```

```
<audio data-key="68" src="sounds/kick.wav"></audio>
<audio data-key="70" src="sounds/openhat.wav"></audio>
<audio data-key="71" src="sounds/boom.wav"></audio>
<audio data-key="72" src="sounds/ride.wav"></audio>
<audio data-key="74" src="sounds/snare.wav"></audio>
<audio data-key="75" src="sounds/tom.wav"></audio>
```

We are creating a `div` for each key. We are using the `data-key` attribute to store the key code for each key. We went over key codes in past lessons. We are also adding a `span` element to display the name of the sound. At the bottom, we are adding an `audio` element for each sound. We are using the `data-key` attribute to store the key code for each sound.

## The CSS

Just some simple CSS to style the page.

```
@import url('https://fonts.googleapis.com/css?family=Poppins:300,400,700');

html,
body {
    font-family: 'Poppins', sans-serif;
    margin: 0;
    padding: 0;
}

header {
    background: #f4f4f4;
}

h1 {
    margin: 0;
    padding: 0;
    text-align: center;
}

.keys {
    font-size: 40px;
    width: 500px;
    margin: 100px auto;
}

.keys kbd {
    display: inline-block;
    padding: 10px 20px;
    margin: 0 10px;
    border: 1px solid #ccc;
    border-radius: 5px;
    background: #eee;
    box-shadow: 0 0 5px rgba(0, 0, 0, 0.1);
    font-size: 20px;
```

```
    font-weight: 300;
}

.keys .playing kbd {
  transform: scale(1.2);
  transition: all 0.07s;
}
```

## The JavaScript

Surprisingly, this will only take about 13 or 14 lines of code.

```
window.addEventListener('keydown', playSound);

function playSound(e) {
  const audio = document.querySelector(`audio[data-key="${e.keyCode}"]`);
  const key = document.querySelector(`.key[data-key="${e.keyCode}"]`);
  if (!audio) return; // stop the function from running all together
  audio.currentTime = 0; // rewind to the start
  audio.play();
  key.classList.add('playing');

  setTimeout(() => {
    key.classList.remove('playing');
  }, 100);
}
```

We are adding an event listener to the `window` object. We are listening for the `keydown` event. When the event fires, we are calling the `playSound` function. The `playSound` function takes an event object as a parameter. We are using the `keyCode` property of the event object to get the key code of the key that was pressed. We are using the `querySelector` method to get the `audio` element and the `key` element that have the same `data-key` attribute as the key code of the key that was pressed. We are using the `play` method to play the sound. We are using the `currentTime` property to rewind the sound to the start. We are using the `classList` property to add the `playing` class to the `key` element. We are using the `setTimeout` method to remove the `playing` class from the `key` element after 100 milliseconds.

Now, when you press a key, the sound will play and the key will animate. Have fun!

# Video API

---

Just like the audio API, the video API is a set of methods and properties that allow you to control the video element. In fact, you'll see that they both use many of the same methods.

First, let's use the HTML5 video element to play a video. I have included a `clouds.mov` video in the sandbox files, but you can use any video that you want.

```
<video controls width="700">
  <source src="media/clouds.mov" />
</video>
```

There are some attributes that you can use for the `video` element:

- `controls` - This will show the default controls for the video
- `autoplay` - This will automatically play the video when the page loads
- `muted` - This will mute the video (In Chrome, the video must be muted for autoplay to work)
- `loop` - This will loop the video
- `poster` - This will show an image before the video starts playing
- `width` - This will set the width of the video
- `height` - This will set the height of the video

Let's give the video a poster. We will use the `poster.png` image in the sandbox files.

```
<video controls width="700" poster="media/poster.png">
  <source src="media/clouds.mov" />
</video>
```

Let's also make it loop and autoplay. It must also be muted for autoplay.

```
<video controls autoplay loop width="700" poster="media/poster.png" muted>
  <source src="media/clouds.mov" />
</video>
```

## JavaScript API

The JavaScript API for the video element is very similar to the audio element. Let's add a play, pause, stop button and a current-time div to display the time. In the next video, we will create a nice looking video player, but for now, I just want you to learn the basic methods.

Let's add the buttons and also remove all of the attributes except for `width` and `poster`. We will use JavaScript to control the video.

```
<video width="700">
  <source src="media/clouds.mov" width="700" poster="media/poster.png" />
</video>

<div>
  <button id="play">Play</button>
  <button id="pause">Pause</button>
  <button id="stop">Stop</button>
  <div id="current-time"></div>
</div>
```

In the JavaScript, let's bring in what we need.

```
const video = document.querySelector('video');
const play = document.getElementById('play');
const pause = document.getElementById('pause');
const stop = document.getElementById('stop');
const currentTime = document.getElementById('current-time');
```

We will add an event listener on the play and pause button and then call the `play()` and `pause()` methods.

```
play.addEventListener('click', () => {
  video.play();
});

pause.addEventListener('click', () => {
  video.pause();
});
```

Just like with the audio API, there is no `stop()` method, so we will pause and reset the time.

```
stop.addEventListener('click', () => {
  video.pause();
  video.currentTime = 0;
});
```

To show the current time, we will listen for the `timeupdate` event and then display the current time.

```
video.addEventListener('timeupdate', () => {
  currentTime.textContent = video.currentTime;
});
```

---

In the next video, we will get a little more into this and create a custom video player.

# Video Player Project

---

Now, we are going to create a custom video player using the video API.

## The HTML

The HTML is pretty simple. We have our `video` element and then some custom controls and a time display. We are also including font awesome because we are using it for the icons.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Custom Video Player</title>
    <link href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css" rel="stylesheet" integrity="sha384-wvfXpqpZZVQGK6TAh5PVlG0fQNHSoD2xbE+QkPxCAF1NEvoEH3Sl0sibVc0QVnN" crossorigin="anonymous" />
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <video
      src="media/clouds.mov"
      poster="media/poster.png"
      id="video"
      class="screen"
    ></video>
    <div class="controls">
      <button class="btn" id="play">
        <i class="fa fa-play fa-2x"></i>
      </button>
      <button class="btn" id="stop">
        <i class="fa fa-stop fa-2x"></i>
      </button>
      <input
        type="range"
        id="progress"
        class="progress"
        min="0"
        max="100"
        step="0.1"
        value="0"
      />
    </div>
  </body>
</html>
```

```

        <span class="timestamp" id="timestamp">00:00</span>
    </div>

    <script src="script.js"></script>
</body>
</html>

```

## The CSS

The CSS is quite long, mostly because we used a custom slider. You don't really need to pay much attention to this part as this is not a CSS course.

```

@import url('https://fonts.googleapis.com/css?family=Questrial&display=swap');

* {
    box-sizing: border-box;
}

body {
    font-family: 'Questrial', sans-serif;
    background-color: steelblue;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    max-height: 100vh;
    margin: 30px 0;
}

h1 {
    color: #fff;
}

.screen {
    cursor: pointer;
    width: 60%;
    background-color: #000 !important;
    border-top-left-radius: 10px;
    border-top-right-radius: 10px;
}

.controls {
    background: #333;
    color: #fff;
    width: 60%;
    border-bottom-left-radius: 10px;
    border-bottom-right-radius: 10px;
    display: flex;
    justify-content: center;
    align-items: center;
}

```

```
padding: 10px;
}

.controls .btn {
  border: 0;
  background: transparent;
  cursor: pointer;
}

.controls .fa-play {
  color: #28a745;
}

.controls .fa-stop {
  color: #dc3545;
}

.controls .fa-pause {
  color: #ffff;
}

.controls .timestamp {
  color: #ffff;
  font-weight: bold;
  margin-left: 10px;
}

.btn:focus {
  outline: 0;
}

input[type='range'] {
  -webkit-appearance: none; /* Hides the slider so that custom slider can be made */
  width: 100%; /* Specific width is required for Firefox. */
  background: transparent; /* Otherwise white in Chrome */
}

input[type='range']::-webkit-slider-thumb {
  -webkit-appearance: none;
}

input[type='range']:focus {
  outline: none; /* Removes the blue border. You should probably do some kind of focus styling for accessibility reasons though. */
}

input[type='range']::-ms-track {
  width: 100%;
  cursor: pointer;

  /* Hides the slider so custom styles can be added */
  background: transparent;
```

```
border-color: transparent;
color: transparent;
}

/* Special styling for WebKit/Blink */
input[type='range']::-webkit-slider-thumb {
-webkit-appearance: none;
border: 1px solid #000000;
height: 36px;
width: 16px;
border-radius: 3px;
background: #ffffff;
cursor: pointer;
margin-top: -14px; /* You need to specify a margin in Chrome, but in Firefox
and IE it is automatic */
box-shadow: 1px 1px 1px #000000, 0px 0px 1px #0d0d0d; /* Add cool effects to
your sliders! */
}

/* All the same stuff for Firefox */
input[type='range']::-moz-range-thumb {
box-shadow: 1px 1px 1px #000000, 0px 0px 1px #0d0d0d;
border: 1px solid #000000;
height: 36px;
width: 16px;
border-radius: 3px;
background: #ffffff;
cursor: pointer;
}

/* All the same stuff for IE */
input[type='range']::-ms-thumb {
box-shadow: 1px 1px 1px #000000, 0px 0px 1px #0d0d0d;
border: 1px solid #000000;
height: 36px;
width: 16px;
border-radius: 3px;
background: #ffffff;
cursor: pointer;
}

input[type='range']::-webkit-slider-runnable-track {
width: 100%;
height: 8.4px;
cursor: pointer;
box-shadow: 1px 1px 1px #000000, 0px 0px 1px #0d0d0d;
background: #3071a9;
border-radius: 1.3px;
border: 0.2px solid #010101;
}

input[type='range']:focus::-webkit-slider-runnable-track {
background: #367ebd;
```

```

}

input[type='range']::-moz-range-track {
    width: 100%;
    height: 8.4px;
    cursor: pointer;
    box-shadow: 1px 1px 1px #000000, 0px 0px 1px #0d0d0d;
    background: #3071a9;
    border-radius: 1.3px;
    border: 0.2px solid #010101;
}

input[type='range']::-ms-track {
    width: 100%;
    height: 8.4px;
    cursor: pointer;
    background: transparent;
    border-color: transparent;
    border-width: 16px 0;
    color: transparent;
}

input[type='range']::-ms-fill-lower {
    background: #2a6495;
    border: 0.2px solid #010101;
    border-radius: 2.6px;
    box-shadow: 1px 1px 1px #000000, 0px 0px 1px #0d0d0d;
}

input[type='range']:focus::-ms-fill-lower {
    background: #3071a9;
}

input[type='range']::-ms-fill-upper {
    background: #3071a9;
    border: 0.2px solid #010101;
    border-radius: 2.6px;
    box-shadow: 1px 1px 1px #000000, 0px 0px 1px #0d0d0d;
}

input[type='range']:focus::-ms-fill-upper {
    background: #367ebd;
}

@media (max-width: 800px) {
    .screen,
    .controls {
        width: 90%;
    }
}

```

## The JavaScript

Let's start by bringing in what we need from the DOM:

```
const video = document.getElementById('video');
const play = document.getElementById('play');
const stop = document.getElementById('stop');
const progress = document.getElementById('progress');
const timestamp = document.getElementById('timestamp');
```

## Play/pause video

Let's create a function to toggle the video between playing and pausing and add the event listeners. I want it on the play button as well as the video itself.

```
function playPause() {
  if (video.paused) {
    video.play();
  } else {
    video.pause();
  }
}

video.addEventListener('click', playPause);
play.addEventListener('click', playPause);
```

## Update icon

Now let's update the icon. If it is playing, the icon should show a play icon, if it is paused, it should show a pause icon.

```
function updateIcon() {
  if (video.paused) {
    play.innerHTML = '<i class="fa fa-play fa-2x"></i>';
  } else {
    play.innerHTML = '<i class="fa fa-pause fa-2x"></i>';
  }
}

video.addEventListener('pause', updateIcon);
video.addEventListener('play', updateIcon);
```

## Stop video

Let's create the stop function and event.

```
function stopVideo() {
  video.currentTime = 0;
```

```
    video.pause();
}

stop.addEventListener('click', stopVideo);
```

## Progress bar

Let's make the progress bar function. We can use the `timeupdate` event to do so.

```
function updateProgress() {
  progress.value = (video.currentTime / video.duration) * 100;

  // Get minutes
  let mins = Math.floor(video.currentTime / 60);
  if (mins < 10) {
    mins = '0' + String(mins);
  }

  // Get seconds
  let secs = Math.floor(video.currentTime % 60);
  if (secs < 10) {
    secs = '0' + String(secs);
  }

  timestamp.innerHTML = `${mins}:${secs}`;
}

video.addEventListener('timeupdate', updateProgress);
```

## Set progress bar

We also want to be able to click anywhere on the bar and have it take us to that point in the video.

```
function setVideoProgress() {
  video.currentTime = (+progress.value * video.duration) / 100;
}

progress.addEventListener('change', setVideoProgress);
```

That's it! We now have a custom video player using the video API.

# Web Animations API

The Web Animations API lets us construct animations and control their playback with JavaScript. A lot of times we use a combination of CSS and JavaScript, but this API allows us to do all of the animation work in JavaScript.

We are going to build a little project for this section.

## The HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Web Animations API</title>
    <link
      rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.2.1/css/all.min.css"
      integrity="sha512-MV7K8+y+gLIBoVD59lQIYicR65iaqukzvf/nwasF0nqhPay5w/91JmVM2hMDcnK1OnMGCdVK+iQrJ71zPJQd1w=="
      crossorigin="anonymous"
      referrerpolicy="no-referrer"
    />
    <link rel="stylesheet" href="style.css" />
    <script src="script.js" defer></script>
  </head>
  <body>
    <div class="wrapper">
      <div class="controls">
        <button id="play"><i class="fa-solid fa-play"></i></button>
        <button id="pause"><i class="fa-solid fa-pause"></i></button>
        <button id="reverse">
          <i class="fa-solid fa-clock-rotate-left"></i>
        </button>
        <button id="speed-up"><i class="fa-solid fa-plus"></i></button>
        <button id="slow-down">Slow Down</button>
      </div>
      <div id="ball">
        <svg
          xmlns="http://www.w3.org/2000/svg"
          width="194"
          height="194"
          version="1.1"
        >
          <circle fill="#000000" cx="97" cy="97" r="97" />
        
```

```

<path
    fill="#ffffff"
    d="m 94,9.2 a 88,88 0 0 0 -55,21.8 l 27,0 28,-14.4 0,-7.4 z m 6,0
0,7.4 28,14.4 27,0 a 88,88 0 0 0 -55,-21.8 z m -67.2,27.8 a 88,88 0 0 0
-20,34.2 l 16,27.6 23,-3.6 21,-36.2 -8.4,-22 -31.6,0 z m 96.8,0 -8.4,22 21,36.2
23,3.6 15.8,-27.4 a 88,88 0 0 0 -19.8,-34.4 l -31.6,0 z m -50,26 -20.2,35.2
17.8,30.8 39.6,0 17.8,-30.8 -20.2,-35.2 -34.8,0 z m -68.8,16.6 a 88,88 0 0 0
-1.8,17.4 88,88 0 0 0 10.4,41.4 l 7.4,-4.4 -1.4,-29 -14.6,-25.4 z m 172.4,0.2
-14.6,25.2 -1.4,29 7.4,4.4 a 88,88 0 0 0 10.4,-41.4 88,88 0 0 0 -1.8,-17.2 z m
-106,57.2 -15.4,19 L 77.2,182.6 a 88,88 0 0 0 19.8,2.4 88,88 0 0 0 19.8,-2.4 l
15.4,-26.6 -15.4,-19 -39.6,0 z m -47.8,2.6 -7,4 A 88,88 0 0 0 68.8,180.4 l
-14,-24.6 -25.4,-16.2 z m 135.2,0 -25.4,16.2 -14,24.4 a 88,88 0 0 0 46.4,-36.6
l -7,-4 z"
/>
</svg>
</div>
</div>
</body>
</html>

```

Here we have some buttons for controlling the animation. I will get to those later. First, we want to focus on making the ball roll. We are using an **svg**, but you could use anything.

## The CSS

Before we do anything with JavaScript, I want to show you how we can do the same animation using just CSS.

```

body {
    background: #000;
    color: #fff;
}

html,
body {
    height: 100%;
}

.wrapper {
    position: relative;
    width: 100%;
    height: 100%;
    overflow: hidden;
}

#ball {
    /* animation: roll infinite 3s linear; */
    color: white;
    width: 25%;
}

```

```
position: absolute;
top: 50%;
left: 50%;
transform-origin: 0 0;
transform: rotate(0) translate3D(-50%, -50%, 0);
backface-visibility: hidden;
will-change: transform, color;
}

path {
  fill:currentColor;
}

.controls {
  display: flex;
  align-items: center;
  justify-content: center;
  margin-top: 20px;
}

button {
  padding: 10px 20px;
  margin: 5px;
  background: #333;
  color: #fff;
  font-size: 18px;
  cursor: pointer;
  border-radius: 5px;
}

button:hover {
  background: blue;
}

/* The Animation */
@keyframes roll {
  0% {
    transform: rotate(0) translate3D(-50%, -50%, 0);
    color: white;
  }
  30% {
    color: blue;
  }
  100% {
    transform: rotate(360deg) translate3D(-50%, -50%, 0);
    color: white;
  }
}
```

In the CSS keyframe we are moving the ball by starting at one position and then rotating it 360 degrees. We are also changing the color of the ball at 30% of the animation.

Now, comment out the CSS, because we are going to do the same thing using JavaScript.

## The JavaScript

Let's start out by bringing in everything that we need.

```
const ball = document.getElementById('ball');
const play = document.getElementById('play');
const pause = document.getElementById('pause');
const reverse = document.getElementById('reverse');
const speedUp = document.getElementById('speed-up');
const slowDown = document.getElementById('slow-down');
```

Now, we will create a keyframe object. We do this by creating an array of multiple objects. Each object represents a key from the original CSS.

```
const rollAnimation = [
  { transform: 'rotate(0) translate3D(-50%, -50%, 0)', color: 'white' },
  { color: 'blue', offset: 0.3 },
  { transform: 'rotate(360deg) translate3D(-50%, -50%, 0)', color: 'white' },
];
```

We can also set the timing in an options object.

```
const rollOptions = {
  duration: 3000,
  iterations: Infinity,
};
```

Now, let's make the element animate.

```
const roll = ball.animate(rollAnimation, rollOptions);
```

Now, we get the same animation as we did with CSS. We can also control the animation with JavaScript. Let's add the play and pause button functionality.

```
play.addEventListener('click', () => roll.play());
pause.addEventListener('click', () => roll.pause());
```

We can also have it go in reverse.

```
reverse.addEventListener('click', () => roll.reverse());
```

If we want to go from reverse to clicking play again and having it move forward, then we have to add the `playbackRate` to `play`.

```
play.addEventListener('click', () => {
  roll.playbackRate = 1;
  roll.play();
});
```

Now, it should work.

We can also speed up and slow down the animation using the `playbackRate`.

```
speedUp.addEventListener(
  'click',
  () => (roll.playbackRate = roll.playbackRate * 2)
);

slowDown.addEventListener(
  'click',
  () => (roll.playbackRate = roll.playbackRate * 0.5)
);
```

This is a very simple animation, but you can imagine how powerful it can be when you get into some really complex animations.

# Web Speech API - Speech Recognition

---

The Web Speech API is a set of JavaScript APIs that allow you to add speech recognition and speech synthesis to your web applications. In this lesson, we will look at speech recognition.

Speech recognition is the ability to convert spoken words into text. The Web Speech API provides a `SpeechRecognition` interface that lets you add speech recognition to your web applications.

## Creating a `SpeechRecognition` Object

To use the `SpeechRecognition` interface, you need to create a `SpeechRecognition` object. You can do this by calling the constructor of the `SpeechRecognition` interface. I am also going to set the language to English. Feel free to use something else.

```
const SpeechRecognition =
  window.SpeechRecognition || window.webkitSpeechRecognition;

const rec = new SpeechRecognition();

rec.lang = 'en-US';
```

## Setting the Speech Recognition Mode

The `SpeechRecognition` interface has a `continuous` property that lets you set the speech recognition mode. The value of the `continuous` property is a boolean. If the value is `true`, the speech recognition will continue until you stop it. If the value is `false`, the speech recognition will stop after a short pause. Let's set it to `true`.

```
rec.continuous = true;
```

To start the speech recognition, we use the `start()` method.

```
rec.start();
```

This will cause the browser to start to listen for speech. We can respond by using the `onresult` event.

```
rec.onresult = function (e) {
  console.log(e.results);
};
```

The event passed in will include a `results` array, which will contain an object with a `transcript` property. The `transcript` property will contain the text that was recognized.

I want to say a color and have the background of the page change to that color.

```
const script = e.results[i][0].transcript;
document.body.style.backgroundColor = transcript;
```

This will let us do it once, however, if we want this to be continuous, we have to loop through the array and set the background color for each result.

```
for (let i = e.resultIndex; i < e.results.length; i++) {
  const script = e.results[i][0].transcript;
  document.body.style.backgroundColor = script;
}
```

Now, we can keep saying colors and it should respond.

Let's make it so if we don't say a color, we get an alert message. We'll create an array of accepted colors.

```
const acceptedColors = [
  'red',
  'blue',
  'green',
  'yellow',
  'pink',
  'brown',
  'purple',
  'orange',
  'black',
  'white',
];
```

In the loop, let's make the transcript lowercase and trim any whitespace. Then we will check to see if the color is in the array. Here is the final code.

```
const SpeechRecognition =
  window.SpeechRecognition || window.webkitSpeechRecognition;

const rec = new SpeechRecognition();

rec.lang = 'en-US';
rec.continuous = true;

rec.onresult = function (e) {
```

```
console.log(e.results);

const acceptedColors = [
  'red',
  'blue',
  'green',
  'yellow',
  'pink',
  'brown',
  'purple',
  'orange',
  'black',
  'white',
];

for (let i = e.resultIndex; i < e.results.length; i++) {
  const script = e.results[i][0].transcript.toLowerCase().trim();

  if (acceptedColors.includes(script)) {
    document.body.style.backgroundColor = script;
  } else {
    alert('Say a color');
  }
}

rec.start();
```

# Speech Synthesis

---

[SpeechSynthesis](#) is a Web API that allows you to have your browser speak text. It is part of the [Web Speech API](#).

We are going to learn this by creating a small project that lets us type in a textbox and reads what we type out loud. We will also be able to change the voice. You can get the simple HTML and CSS from the sandbox files.

Let's start by creating a `synth` variable that will hold the [SpeechSynthesis](#) object.

```
const synth = window.speechSynthesis;
```

We will add an event listener for the form submit and get the textarea value.

```
function onSubmit(e) {
  e.preventDefault();

  const textInput = document.getElementById('text-input');

  document.getElementById('form').addEventListener('submit', onSubmit);
}
```

We need to create a [SpeechSynthesisUtterance](#) object that will hold the text we want to speak. Then we can use the `synth.speak()` method to speak the text.

```
function onSubmit(e) {
  e.preventDefault();

  const textInput = document.getElementById('text-input');

  const utterThis = new SpeechSynthesisUtterance(textInput.value);

  synth.speak(utterThis);
}
```

## Getting the voices

Now, let's create a function that will get the voices and populate the select element with them.

```
const voiceSelect = document.getElementById('voice-select');
let voices;
```

```

function addVoicesToSelect() {
  voices = synth.getVoices();

  for (let i = 0; i < voices.length; i++) {
    const option = document.createElement('option');
    option.textContent = `${voices[i].name} - ${voices[i].lang}`;

    if (voices[i].default) {
      option.textContent += ' - DEFAULT';
    }

    option.setAttribute('data-lang', voices[i].lang);
    option.setAttribute('data-name', voices[i].name);
    voiceSelect.appendChild(option);
  }
}

```

We got the select element outside of the function because we will need it in the submit function as well. We also create a `voices` variable to hold the array of voices that we get from the `synth.getVoices()` method.

Then we loop through the voices and create an option element for each one. We set the `textContent` to the name and language of the voice. If the voice is the default one, we add `- DEFAULT` to the text. We also set the `data-lang` and `data-name` attributes to the language and name of the voice. Finally, we append the option to the select element.

We need to call the `addVoicesToSelect()` function once to populate the select element with the voices. We also need to call it again when the voices change. We can do this by adding an event listener for the `voiceschanged` event. We will put this at the bottom above the form event listener.

```

addVoicesToSelect();
if (speechSynthesis.onvoiceschanged !== undefined) {
  speechSynthesis.onvoiceschanged = addVoicesToSelect;
}

```

## Changing the voice

Edit the `onSubmit()` function to change the voice.

```

function onSubmit(e) {
  e.preventDefault();

  const textInput = document.getElementById('text-input');

  const utterThis = new SpeechSynthesisUtterance(textInput.value);

```

```
const selectedOption =
  voiceSelect.selectedOptions[0].getAttribute('data-name');
for (let i = 0; i < voices.length; i++) {
  if (voices[i].name === selectedOption) {
    utterThis.voice = voices[i];
  }
}

synth.speak(utterThis);
}
```

Here, we get the selected option from the select element. Then we loop through the voices and check if the name of the voice is the same as the selected option. If it is, we set the `utterThis.voice` to the voice.

Now, you should be able to type in the text box, select a voice and have it read out loud.

# What Is OOP?

---

Now we're going to get into **Object Oriented Programming** or **OOP** using JavaScript. The way that we write object-oriented code with JavaScript is a little different than some other languages and even a bit controversial and I will get to that soon, but first, I want to talk about what OOP actually is outside of specific languages.

**OOP** is a **programming paradigm**. In short, a programming paradigm is a way of thinking about programming, and a way of structuring your code. You could dumb it down to the word "style". There are many different programming paradigms and certain languages are built for certain paradigms. The most common are **procedural**, **object-oriented** and **functional**. You can read more about paradigms [here](#)

What we have been doing so far is mostly **procedural** programming, although, we have definitely used some elements of **OOP** and even **functional** programming. JavaScript is an extremlet flexible language and you can do things in many different ways. Which can be a good thing, because you have a lot of freedom, but it can be bad because it can get really confusing and also cause you to sometimes write code that isn't that great. In **procedural** programming, we write functions that perform actions, and we call those functions to perform those actions. In **object-oriented** programming, we write objects that contain both data and functions, and we interact with those objects to perform actions. Some languages force you to write all of your code within objects.

## What Is An Object?

An object is a 'self-contained' piece of code and is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method. So, essentially, an object is a **collection of properties and methods**. Objects are used as building blocks and can interact with one another.

We have already had some experience working with objects such as **Math** and **Date**. In fact, just about everything in JavaScript is an object. Primitive data types like strings are not objects, however they are treated like objects when we use methods like **toUpperCase()**. As I stated earlier in the course, when we use a method like that on a string, JavaScript is actually creating a temporary object behind the scenes to perform the action. This is called a **wrapper object**.

JavaScript has many was to create objects inclidng object literals, constructor functions, which we'll be covering soon, classes, factories, etc.

Any entity in your application could be an object. For example, if you are building a blog, you might use objects like **users** and **posts**. Just to give you a better idea of what would be looked at as an object, here is a list of random objects:

- **user**
- **post**
- **comment**
- **UI component**
- **product**

- `order`
- `cart`
- `customer`
- `employee`
- `message`
- `book`

## Why Use OOP?

There are a lot of reasons to use OOP for certain projects, but the main reason is that it makes your code easier to understand and easier to maintain. It helps prevent what we call 'Spaghetti Code'. OOP is also very flexible and can be used to build complex applications. It is also very easy to reuse code in OOP, which is one of the main reasons why it is so popular.

## When To Use OOP

Where OOP really shines and makes sense is when you are building a complex application and will need multiple instances of an object. It's also helpful if you're working on a team. If you are building a relatively simple application, you probably should not use OOP. For the stuff that we have done up to this point, I would not use it. And when I say I would not use OOP, I mean I wouldn't create custom classes or constructor functions. In JavaScript, we're always using objects, so we're always using some aspect of OOP.

## Components of OOP

Technically we have created objects many times as we have been using `object literals`. However, I wouldn't call that `OOP`. We've just been using object literals to create a single object and structure data into key/value pairs. With OOP, we create blueprints to create multiple instances of an object.

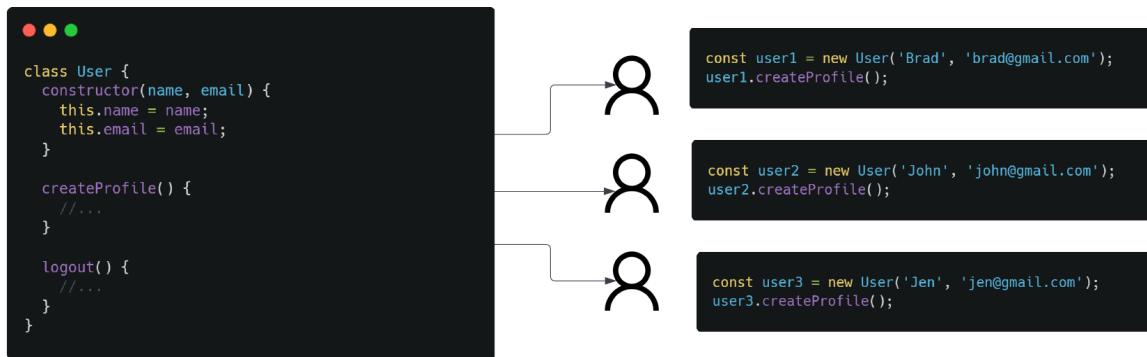
Now OOP within JavaScript is a bit different than most languages. A lot of that has to do with the inner mechanics and the browser environment. In JavaScript, we use something called `constructor functions` as well as `prototypes`. With these we can create blueprints to create objects. Most languages use `classes`. JavaScript doesn't use classes in the core language, however, the ES6 update gave us the class syntax. This is called `syntactic sugar`. So we have the easier to write class syntax, but it's really using constructors and prototypes under the hood. I'm going to show you both ways.

Some important components of OOP in JavaScript are:

- `Constructor Functions` Are the way that JavaScript creates a blueprint for an object.
- `Prototypes` A way for objects to inherit properties and methods from one another.
- `Classes` Another way to create a blueprint for an object. They are not a part of the core JavaScript language, but they are a part of the `ES6` specification. So we can use classes as a type of syntactic sugar to make our code easier to read and write. Classes are a core part of many other languages.
- `Instances` Objects are instantiated from a blueprint like a constructor or a class. This way we can create multiple objects that all have their own property values and can share the same methods.

Below is an example of a very simple class. I'm not showing you this to teach you the syntax yet, I'm just showing you this to give you some perspective of how OOP works. This class is essentially a blueprint for

an object and it includes properties and methods. We can use that blueprint to create what are called **instances** of that object. in this case, users.



The class alone is just the blueprint. It's useless in our code if we don't use it. You can think of it just like a blueprint for a house. The blueprint is just a piece of paper, it's not a house. We have to use the blueprint to build the house. In this case, the blueprint is for a user and each user is the actual object. We can create as many users as we want using the blueprint.

Now like I said, JavaScript doesn't actually have classes in the core language, it uses something called **constructor functions** and **prototypes**, but we can still write our code using classes and it gets transpiled into regular JavaScript. We will talk about both constructors, prototypes and classes in more detail later.

In the next lesson, I just want to go over some of the main principles of OOP.

# The 4 Basic Principles Of OOP

---

There are 4 basic principles of OOP that you should know about.

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

These principles are the foundation of OOP and are used in almost every OOP language. Right now, you are not expected to know how to implement these principles in your code. You will learn that later. For now, just know what they are and what they do.

## Abstraction

Abstraction is where we hide all but the relevant parts of an object in order to reduce complexity and increase efficiency.

The easiest way to explain and understand abstraction is to look at something like your car. As a user, do you really need to understand the details of what happens when you start your car? No, that's not your job. You don't need to know how it works. You just need to know how to use it. So you need to know about the ignition, gas, brakes, etc. You don't need to know how the fuel pump works to drive a car. That's abstraction. It's the process of hiding the details of how something works and only showing the important parts.

Since objects can get pretty large and complex, it's not usually possible to show the user all the details of how the object works. There also is no reason to. The user of the object should only see and access what they need.

We deal with abstraction everywhere in programming. There are 100 things that I can think of that we have already done in this course that have used abstraction. For example, when we create an event listener, we use the `fetch()` method. We don't need to know exactly how it works. We just need to know how to use it. Even the public APIs that we have used also have a lot of abstraction. We just hit an endpoint. We have no clue what goes on behind the scenes. Another example is when we're working with events and call `addEventListener()` method. We don't need to know how the event listener works. We just need to know how to use it. As a programmer, you'll both use and create abstractions.

One of the most useful things about abstraction is that it reduces the complexity of our code and the impact of change. If we have a class of `User` and we want to add a new property, we don't have to change the code everywhere that we use the `User` class. We can just add the new property to the class and it will be available everywhere.

## Encapsulation

Encapsulation is the process of wrapping up data and methods into a single unit such as a class or function. This concept is also often used to hide the properties or state of an object from the outside. This is called information hiding or data hiding. When we create a class for example, the properties and

methods are 'encapsulated' into that specific class. This is a way to keep our code organized and easy to understand. It also means that certain properties and methods are only accessible from within the class. Some languages have access modifiers like `private` and `public` keywords to indicate which properties and methods are accessible from the outside. JavaScript doesn't have those keywords, but the latest ES2022 update does include private fields using a `#` symbol. We'll look at that later.

I don't expect you to understand this example yet, but it uses encapsulation. The `radius` property is encapsulated into the `Circle` class. We can't access the `radius` property from outside the `Circle` class. So it's hidden from the outside world. We can access the `area` property from outside the class with what we call a `getter`.

```
class Circle {  
    #radius;  
    constructor(value) {  
        this.#radius = value;  
    }  
    get area() {  
        return Math.PI * Math.pow(this.#radius, 2);  
    }  
}
```

I know this is confusing now, but after the next two sections, you'll fully understand this. Another example of encapsulation that you should already understand is the scope chain. When we create a variable within a function, it's encapsulated in that function's scope. We can't access it from the global or any higher level scope.

## Inheritance

Inheritance is the process of inheriting the properties and methods from a parent class. This is a way to reuse code and reduce redundancy. We can create a parent class with some properties and methods, and then create child classes that inherit those properties and methods as well as have their own.

Even when we work with elements in the DOM, all HTML elements share some common properties and methods. For example, all HTML elements have a `style` property and a `addEventListener()` method. We can create a parent class of `HTMLElement` that has those properties and methods. Then we can create child classes of `HTMLButtonElement`, `HTMLDivElement`, etc. that inherit those properties and methods. Now, we can create instances of the `HTMLButtonElement` class and use the `style` property and the `addEventListener()` method.

To give you a very simple example, we have a class of `User` that has a property of `name` and a method of `sayHello()`. We can create a child class called `Admin` that inherits the `name` property and the `sayHello()` method. We can then add a new method to the `Admin` class called `deleteUser()`. Now, we can create an instance of the `Admin` class and use the `name` property, the `sayHello()` method, and the `deleteUser()` method.

```

class User {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

class Admin extends User {
  deleteUser(user) {
    users = users.filter((u) => u.name !== user.name);
  }
}

```

## Polymorphism

Polymorphism is a scary word, but it's not that intimidating when you break it down. The word **poly** means **many** and **morph** means **form**. SO it describes situations in which something occurs in several different forms. This is a way to reuse code. It allows us to do away with long **if/else** or **switch** statements and use a single method that will work for different types of objects. A very common example of polymorphism is when we use the same method name for different types of objects.

For instance, we have a **User** object with a **sayHello** method. Then we have an **Admin** class that extends the **User** class and also has a **sayHello** method that does something different. We could have 10 more classes with a **sayHello** and they all do something different. So we're using **many forms** of **sayHello** (poly morph)

```

class User {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

class Admin extends User {
  sayHello() {
    console.log(`Hello, my name is ${this.name}. I am an admin.`);
  }
}

```

So those are about the simplest way I can explain the 4 principles. As we move through the next dozen or so videos, these will come into perspective.

We have been working with **object literals** throughout this course. They are the simplest form of an object and great when you only need one instance of an object, which is usually the case when building simple applications. But what if you need to create multiple instances of an object? That's where **constructor functions** come in, which I will get to soon. But first, let's look at **object literals** a little closer. I also want to talk about the **this** keyword, which is used a lot in object oriented programming.

Let's say that we have a program that works with shapes, for whatever reason. I want to create a **Rectangle** object. So I'll use an object literal in this case. Let's give it some properties.

```
const rect = {  
    name: 'Rectangle 1',  
    width: 10,  
    height: 10,  
};
```

We can easily access properties with dot notation.

```
console.log(rect.name); // Rectangle 1
```

This is nothing new right? We've been doing stuff like this all along. One thing that we haven't done is added a method to an object literal. Let's add a method to our **Rectangle** object that calculates the area of the rectangle.

```
const rect = {  
    name: 'Rectangle 1',  
    width: 10,  
    height: 10,  
    area: function () {  
        console.log('Show area');  
    },  
};
```

We can call the method like this:

```
rect.area(); // Show area
```

## this keyword

The **this** keyword is used to refer to the current object. Let's have our **area** method return the area of the rectangle, which would be **width \* height**.

```
const rect = {
  name: 'Rectangle 1',
  width: 10,
  height: 10,
  area: function () {
    return this.width * this.height;
  },
};

console.log(rect.area()); // 100
```

As you can see, we can access the properties of the object using `this`. This is very useful when you have multiple objects that have the same properties and methods. You can use the `this` keyword to access the properties of the current object.

The `this` keyword seems to be confusing to a lot of people. It's actually pretty simple. It just refers to the current object. If you are in a method, it refers to the object that the method is part of. If you are in a function or the global scope, it refers to the global object, which is the `window` object in the browser.

Let's go outside of the object literal and try to access the `this` keyword.

```
console.log(this); // Window {parent: Window, opener: null, top: Window,
length: 0, frames: Window, ...}
```

It gives us the `window` object. Remember, when we run our script, the first thing that the `execution context` does is create the `global object`, which is the `window` object in the browser. So `this` refers to the `window` object.

## Object Literal Drawbacks

There are a few drawbacks to using object literals. The first is that you can't create multiple instances of the same object. Let's say that we want to create another `rect` object. We can't do this with an object literal. We need to basically just create a new one from scratch.

```
const rect2 = {
  name: 'Rectangle 2',
  width: 20,
  height: 20,
  area: function () {
    return this.width * this.height;
  },
};

console.log(rect2.area()); // 400
```

Imagine if we needed to create 100 rects. We would have to create 100 different object literals. This is not very efficient. We need a way to create multiple instances of the same object. This is where [constructor functions](#) come in, which we will talk about in the next lesson.

# Constructor Functions

---

Object literals are very useful in many ways, but what if you need a way to create multiple instances of the same object? This is where constructor functions come in.

Constructor functions are functions that are essentially blueprints to create new objects. They define properties and behaviors that will belong to the new object.

We have used constructor functions many times already. For instance, when we created a new `Date` object, we used the `Date` constructor function.

```
const now = new Date();
```

Anytime we use the `new` keyword, we are using a constructor function. Now, we want to create our own constructor function. Let's create a constructor function for a `Rectangle` object.

```
function Rectangle(name, width, height) {
  this.name = name;
  this.width = width;
  this.height = height;
  // We can also hardcode properties
  this.type = 'Rectangle';
}
```

We use the `function` keyword to create a function. We give it a name, which is `Rectangle`. The first letter is capitalized, which is a convention for constructor functions. We pass in the parameters that we want to use to create the object. We use the `this` keyword to set the properties of the object.

What is returned from the constructor function is the new object that is created. We could actually do this:

```
function Rectangle(name, width, height) {
  this.name = name;
  this.width = width;
  this.height = height;

  return this;
}
```

But there is no need to because the `this` keyword is implicitly returned by default.

We can now create a new `Rectangle` object like this:

```
const rect1 = new Rectangle('Rectangle 1', 10, 10);
console.log(rect1); // Rectangle {name: "Rectangle 1", width: 10, height: 10}
```

So as you can see, when we log the rectangle, it is an object and it is prefixed with `Rectangle`. This is because we used the `Rectangle` constructor function to create the object.

You'll also notice the `prototype` object. Inside of that is the constructor function. We can actually add methods to the `prototype` and we're going to get into that soon.

We can now access the properties of the object like this:

```
console.log(rect1.name); // Rectangle 1
```

When we use the `new` keyword, the following things happen:

1. A new empty object is created.
2. The constructor function is called with the arguments that we passed in.
3. The `this` keyword is set to the new empty object.
4. The new object is returned from the constructor function.

## Multiple Instances

The main advantage of using a constructor function is that we can create multiple instances of the same object. Let's create another `Rectangle` object.

```
const rect2 = new Rectangle('Rectangle 2', 20, 20);
console.log(rect2); // Rectangle {name: "Rectangle 2", width: 20, height: 20}
```

So you see, we did not need to create an entirely new constructor function to create a new `Rectangle` object. We just used the same constructor function and passed in different arguments. This is much more efficient than creating a new object literal every time we want to create a new object.

## Methods

We can also add methods to our constructor function. Let's add an `area` method to our `Rectangle` constructor function. Those of you that are a little more advanced may say we should add this to the `prototype` object. We will get into that later.

```
function Rectangle(name, width, height) {
  this.name = name;
  this.width = width;
  this.height = height;
  this.area = function () {
    return this.width * this.height;
```

```
};

console.log(rect1.area()); // 100
```

## constructor Property

Every object has a `constructor` property. This property is a reference to the constructor function that was used to create the object. Let's log the `constructor` property of the `rect1` object.

```
console.log(rect1.constructor); // f Rectangle(name, width, height) { this.name = name; this.width = width; this.height = height; this.area = function () { return this.width * this.height; }; }
```

## instanceof Operator

We can use the `instanceof` operator to check if an object was created by a constructor function. Let's check if the `rect1` object was created by the `Rectangle` constructor function.

```
console.log(square1 instanceof Rectangle); // true
```

## Encapsulation & Abstraction In Practice

This brings us to the first of the four principles of object-oriented programming, which is `encapsulation`. Encapsulation is the idea that we should group related variables and functions together. We should also hide the implementation details from the outside world. This is what we have done with the `Rectangle` constructor function. We have encapsulated the properties and methods that belong to the `Rectangle` object.

One of the benefits of encapsulation is that we can change the implementation details without affecting the outside world. For instance, we could change the name of the `area` method to `calculateArea` without affecting the outside world.

```
function Rectangle(name, width, height) {
    this.name = name;
    this.width = width;
    this.height = height;
    this.calculateArea = function () {
        return this.width * this.height;
    };
}
```

Another advantage is when we call `square1.calculateArea()` we don't have to pass in any arguments. This is because the `this` keyword is set to the `square1` object. We can access the properties of the `rect1` object inside of the `calculateArea` method.

## Abstraction

Abstraction is the idea that we should only expose the necessary details to the outside world. We should hide the implementation details. We have already seen this with the `Rectangle` constructor function. We have hidden the implementation details of the `Rectangle` object. We have only exposed the properties and methods that we want the outside world to use.

In the next lesson, we will look more at `literals` and `constructors` as well as `boxing` and `unboxing`.

# Literals vs Constructors & Boxing

We somewhat went over this already, but I want to talk a little bit about literals vs constructor functions when it comes to all types of data including `strings`, `numbers`, etc. We have constructors for strings, numbers, booleans, arrays, and objects. We can use either literals or constructor functions to create these. Literals are obviously shorter and more common and easier but let's take a look at both syntax and see what the differences and similarities are.

```
const strLit = 'Hello';
const strObject = new String('Hello');

console.log(strLit, typeof strLit); // Hello string
console.log(strObj, typeof strObj); // String {"Hello"} object
```

We see that `typeof` returns `string` for the literal and `object` for the object.

## Boxing

When we use a method such as `toUpperCase()` on a string literal, JavaScript creates a wrapper object around it. This is called `boxing`. So essentially, it is the same as calling `toUpperCase()` on a string object.

```
console.log(strLit.toUpperCase());
// Same as
console.log(strObj.toUpperCase());
```

## `valueOf` property & Unboxing

We can use the `valueOf` method to get the primitive value of the string object. This is called `unboxing` because we are unboxing the object and getting the primitive value.

```
console.log(strObj.valueOf()); // Hello
```

## `constructor` Property

Because of boxing, there is still a `constructor` property on the string literal. We can use the `constructor` property to see what the constructor function is for the string literal and object.

```
console.log(strLit.constructor); // [Function: String]
console.log(strObj.constructor); // [Function: String]
```

## instanceof Operator

We can also use the `instanceof` operator to check if an object is an instance of a constructor function. The literal will return `false` and the object will return `true`.

```
console.log(strLit instanceof String); // false
console.log(strObj instanceof String); // true
```

Let's look at some other type literals vs constructor functions.

```
const numLit = 20;
const numObj = new Number(20);

const boolLit = true;
const boolObj = new Boolean(true);

const arrLit = [1, 2, 3, 4, 5];
const arrObj = new Array(1, 2, 3, 4, 5);

const funcLit = function (x) {
    return x * x;
};

const funcObj = new Function('x', 'return x * x');

console.log(funcObj(10));

const objLit = { name: 'Jeff' };
const objObj = new Object({ name: 'Jeff' });
```

As you can see, there is even a constructor for objects. When we execute the following code:

```
const obj = {};
```

JavaScript actually creates an object behind the scenes and then assigns it to the variable `obj`. So the following code is actually the same as the code above.

```
const obj = new Object();
```

Hopefully this helps you understand the differences between literals and constructor functions. I know it can be confusing at first, but once you get the hang of it, it will be much easier to understand. Also, even if you do not fully understand, that doesn't mean you can't be a great developer. A lot of this stuff is behind the scenes and just stuff you have to learn and practice.



# Object Properties

Now, we're going to talk about properties and how we can add and remove them as well as look at some methods that we can use to get information about an object. Now, I just want to mention that a lot of this stuff, we have already talked about because they also pertain to `object literals`. So, we're going to talk about them again, but just in the context of `constructor functions`.

Let's use the same `Rectangle` constructor function that we used in a previous lesson. We're going to create a `rect1` object and log it to the console.

```
function Rectangle(name, width, height) {  
    this.name = name;  
    this.width = width;  
    this.height = height;  
    this.area = function () {  
        return this.width * this.height;  
    };  
}  
  
const rect1 = new Rectangle('Rectangle 1', 10, 10);  
console.log(rect1); // Rectangle {name: "Rectangle 1", width: 10, height: 10,  
area: f}
```

## Adding & Removing Properties

We add and remove properties the same way that we would with `object literals`. Let's add a `color` property to the `rect1` object.

```
rect1.color = 'red';  
console.log(rect1); // Rectangle {name: "Rectangle 1", width: 10, height: 10,  
area: f, color: "red"}
```

We can also remove properties from objects using `delete`. Let's delete the `color` property from the `rect1` object.

```
delete rect1.color;  
console.log(rect1); // Rectangle {name: "Rectangle 1", width: 10, height: 10,  
area: f}
```

## hasOwnProperty Method

We can use the `hasOwnProperty` method to check if an object has a specific property. Let's check if the `rect1` object has a `color` and a `name` property.

```
console.log(rect1.hasOwnProperty('color')); // false  
console.log(rect1.hasOwnProperty('name')); // true
```

## Object.keys Method

We can use the `Object.keys` method to get an array of all the properties of an object. Let's get the properties of the `rect1` object.

```
console.log(Object.keys(rect1)); // (3) ["name", "width", "height"]
```

## Object.values Method

We can use the `Object.values` method to get an array of all the values of an object. Let's get the values of the `rect1` object.

```
console.log(Object.values(rect1)); // (3) ["Rectangle 1", 10, 10]
```

## Object.entries Method

We can use the `Object.entries` method to get an array of all the properties and values of an object. Let's get the properties and values of the `rect1` object.

```
console.log(Object.entries(rect1)); // (3) [Array(2), Array(2), Array(2)]
```

## Looping Through Object Properties

We can use a `for...of` loop to loop through all the entries of an object. Let's loop through the `rect1` object.

```
for (let entry of Object.entries(rect1)) {  
  console.log(entry); // (2) ["name", "Rectangle 1"], (2) ["width", 10],  
  (2) ["height", 10], (2) ["area", f]  
}
```

This will give us all properties, even the method. We can use `typeof` to check if the value is a function. Let's loop through the `rect1` object and check if the value is a function.

```
for (let [key, value] of Object.entries(rect1)) {
  if (typeof value !== 'function') {
    console.log(` ${key}: ${value}`);
    // name: Rectangle 1, width: 10, height: 10
  }
}
```

We can also use a `for...in` loop to loop through all the properties of an object. Let's loop through the `rect1` object.

```
for (let key in rect1) {
  console.log(` ${key}: ${rect1[key]}`);
}
```

## Object.assign Method

We can use the `Object.assign` method to copy the properties and values of one object to another.

Let's create a new object with a `color` property

```
const obj = { color: 'green' };
```

Now let's create a new object that has the properties of `rect` and `obj`

```
const rect2 = Object.assign(obj, rect1);
console.log(rect2); // Rectangle {name: "Rectangle 1", width: 10, height: 10,
color: "green"}
```

## Spread Operator

We can do the same exact thing with the spread operator (...). Let's create a new object that has the properties of `rect` and `obj`

```
const rect3 = { ...obj, ...rect1 };
console.log(rect3);
```

# Prototypes & The Prototype Chain

---

Alright, so now we're going to look at everyone's favorite topic, **prototypes**. This is a topic that I think is made to be over-complicated in a lot of tutorials. I will try and give you the simplest definition that I can.

## What Are Prototypes?

Prototypes are a special type of enumerable object where additional methods and properties can be attached and shared across all of the instances of its constructor function. In JavaScript, every function and object has a prototype. The prototype itself is an object. When we try to access a property on an object, JavaScript will first look for that property on the object. If it doesn't find it, it will look at the object's **prototype**. If it still doesn't find it, it will look at the prototype's prototype. This continues until it finds the property or it reaches the end of what we call the **prototype chain**, which is null.

So the dumbed down version is that prototypes are an extra space to hold properties and methods that we can use on our objects. We can add properties and methods to the prototype and they will be available to all objects that inherit from that prototype. This is called **prototypical inheritance**.

We've used many prototype methods in the previous lessons. For instance, when we have an array and we call the **map()** method, JavaScript will look for the **map()** method on the array. If it doesn't find it, it will look at the array's prototype. It will find the **map()** method and execute it.

## Viewing Prototypes

Open your browser console and create an object. It can be empty. Then type the variable name and hit **enter**

```
> const x = {};
< undefined
> x
< - { } ⓘ
  ▼ [[Prototype]]: Object
    ► constructor: f Object()
    ► hasOwnProperty: f hasOwnProperty()
    ► isPrototypeOf: f isPrototypeOf()
    ► propertyIsEnumerable: f propertyIsEnumerable()
    ► toLocaleString: f toLocaleString()
    ► toString: f toString()
    ► valueOf: f valueOf()
    ► __defineGetter__: f __defineGetter__()
    ► __defineSetter__: f __defineSetter__()
    ► __lookupGetter__: f __lookupGetter__()
    ► __lookupSetter__: f __lookupSetter__()
    ► __proto__: (...)

    ► get __proto__: f __proto__()
    ► set __proto__: f __proto__()
```

you will see that it has a **Prototype** property. This is the prototype of the object. If you create an array or a function, it will also have a **Prototype** property. Everything in JS is an object and every object has a prototype (which is also an object).

Notice in the object's prototype, there is a `constructor`. The `constructor` property is a reference to the function that was used to create the object.

There are some other familiar properties in the prototype, like `toString()`, `valueOf()` and `hasOwnProperty()`. These are all methods that are inherited from the `Object.prototype` also called `objectBase`.

## Prototype Chain & Prototypical Inheritance

Let's create an array in the console. You'll see the `Array.prototype` or the `arrayBase` with some very familiar methods like `push()`, `pop()` as well as high order array methods that we have used such as `map()`, `filter()` and `forEach()`.

```
> const arr = [];
< undefined
> arr
< -> []
  ↴
    length: 0
  ▼ [[Prototype]]: Array(0)
    ► at: f at()
    ► concat: f concat()
    ► constructor: f Array()
    ► copyWithin: f copyWithin()
    ► entries: f entries()
    ► every: f every()
    ► fill: f fill()
    ► filter: f filter()
    ► find: f find()
    ► findIndex: f findIndex()
    ► findLast: f findLast()
    ► findLastIndex: f findLastIndex()
    ► flat: f flat()
    ► flatMap: f flatMap()
    ► forEach: f forEach()
    ► includes: f includes()
    ► indexOf: f indexOf()
    ► join: f join()
    ► keys: f keys()
    ► lastIndexOf: f lastIndexOf()
    length: 0
```

If we scroll down, you will see another `prototype` object.

```

▶ reverse: f reverse()
▶ shift: f shift()
▶ slice: f slice()
▶ some: f some()
▶ sort: f sort()
▶ splice: f splice()
▶ toLocaleString: f toLocaleString()
▶ toString: f tostring()
▶ unshift: f unshift()
▶ values: f values()
▶ Symbol(Symbol.iterator): f values()
▶ Symbol(Symbol.unscopables): {at: true, copyWithin: true, entries:
  ▶ [[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f tostring()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __definegetter__()
    ▶ __defineSetter__: f __definesetter__()
    ▶ __lookupGetter__: f __Lookupgetter__()
    ▶ __lookupSetter__: f __Lookupsetter__()
    ▶ __proto__: ...
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()

```

This is because the `Array.prototype` inherits from the `Object.prototype`. This is inheritance in what we call the prototype chain. The prototype is itself an object, so the prototype will have its own prototype, making what's called a prototype chain. The chain ends when we reach a prototype that has null for its own prototype.

Let's use our `Rectangle` object from the previous lesson and create an object and look at the prototype in the console.

```

function Rectangle(name, width, height) {
  this.name = name;
  this.width = width;
  this.height = height;
  this.area = function () {
    return this.width * this.height;
  };
}

const rect1 = new Rectangle('Rectangle 1', 10, 10);
console.log(rect1);

```

```

script.js:11
▼ Rectangle {name: 'Rectangle 1', width: 10, height: 10, area: f} ⓘ
  ▶ area: f ()
  ▶ height: 10
  ▶ name: "Rectangle 1"
  ▶ width: 10
  ▶ [[Prototype]]: Object
    ▶ constructor: f Rectangle(name, width, height)
    ▶ [[Prototype]]: Object

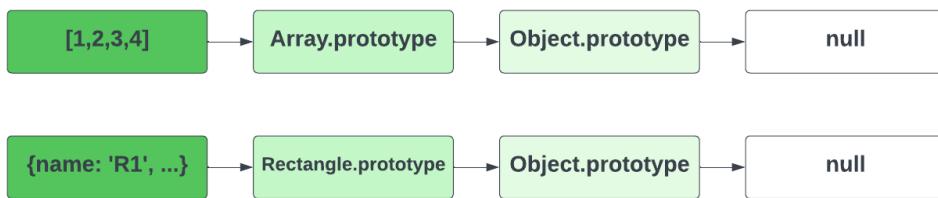
```

In addition to the values, we can see the prototype, which includes the `constructor` property. You can see the 3 arguments that the `Rectangle` constructor function takes.

Notice that this also includes the `Object.prototype`. That's because the `Rectangle` constructor function inherits from the `Object.prototype` or `objectBase`. This is why we can use the `toString()` method on our `Rectangle` object.

```
console.log(rect1.toString());
// [object Object]
```

So, the prototype chain looks something like this for the array and our `Rectangle` objects:



`Object.getPrototypeOf()` is a method that returns the prototype of an object. Let's use it to get the prototype of the `rect1` object.

```
console.log(Object.getPrototypeOf(rect1));
```

We see the same thing that we saw in the console.

In the next lesson, I'll show you how to add properties and methods to the prototype.

# Adding Methods & Properties to Prototypes

In the previous lesson, we learned about [prototypes](#). In this lesson, we will learn how to add methods and properties to [prototypes](#).

## Adding Methods to Prototypes

Let's look at the [Rectangle](#) object we created in the previous lesson.

```
function Rectangle(name, width, height) {  
    this.name = name;  
    this.width = width;  
    this.height = height;  
    this.area = function () {  
        return this.width * this.height;  
    };  
}
```

It's clear that every rectangle will take in a [name](#), [width](#), and [height](#). However, the [area](#) method is the absolute same for every rectangle. it will always be the [width](#) x [height](#). So instead of keeping the [area\(\)](#) method on the object itself, we can move it to the [prototype](#) of the [Rectangle](#) object.

First, remove the method from the [Rectangle](#) constructor function.

```
function Rectangle(name, width, height) {  
    this.name = name;  
    this.width = width;  
    this.height = height;  
}
```

Then, add the [area\(\)](#) method to the [Rectangle.prototype](#).

```
Rectangle.prototype.area = function () {  
    return this.width * this.height;  
};
```

We still have access to the [area\(\)](#) method on every rectangle object, but it's now on the [prototype](#).

```
const rect1 = new Rectangle('Rectangle 1', 10, 10);  
  
console.log(rect1.area()); // 100
```

Let's add another method for the `perimeter` of the rectangle.

```
Rectangle.prototype.perimeter = function () {
  return this.width * 2 + this.height * 2;
};
```

Now, we can access:

```
console.log(rect1.perimeter()); // 40
```

Let's log the `rect1` object to see what it looks like.

```
console.log(rect1);
```

As you can see, we now have 2 methods in addition to the `constructor`.

```
script.js:22
▼ Rectangle {name: 'Rectangle 1', width: 10, height: 10} ⓘ
  height: 10
  name: "Rectangle 1"
  width: 10
  ▼ [[Prototype]]: Object
    ► area: f ()
    ► perimeter: f ()
    ► constructor: f Rectangle(name, width, height)
    ▶ [[Prototype]]: Object
```

Also, notice that it inherits `Object.prototype`.

## Using arguments

We can also pass in arguments to the `prototype` methods. Let's add a method to change the name. It will take in a `newName` argument.

```
Rectangle.prototype.changeName = function (newName) {
  this.name = newName;
};

rect1.changeName('Rectangle One');
console.log(rect1); // Rectangle {name: "Rectangle One", width: 10, height: 10,
area: f, perimeter: f, ...}
```

I'm not going to leave this method, on the `prototype`. I just wanted to show that they can take in arguments.

# Using Object.create()

The `Object.create()` method creates a new object, using an existing object as the prototype of the newly created object. This is an alternate way to create objects and set the prototypes of an object.

Let's look at creating a rectangle object using a constructor function and adding a few prototype methods:

```
function Rectangle(height, width) {  
    this.height = height;  
    this.width = width;  
}  
  
Rectangle.prototype.area = function () {  
    return this.height * this.width;  
};  
  
Rectangle.prototype.perimeter = function () {  
    return 2 * (this.height + this.width);  
};  
  
Rectangle.prototype.isSquare = function () {  
    return this.height === this.width;  
};  
  
const rectangle1 = new Rectangle(4, 4);  
console.log(rectangle1.isSquare(), rectangle1.area());
```

Nothing new here. We have already learned this. I want to show you another way to do this using the `Object.create()` method.

First, we create an object that will be the prototype(s) of the `Rectangle` object:

```
const rectanglePrototypes = {  
    isSquare: function () {  
        return this.height === this.width;  
    },  
    area: function () {  
        return this.height * this.width;  
    },  
    perimeter: function () {  
        return 2 * (this.height + this.width);  
    },  
};
```

I'm going to put the `Object.create()` method inside of a function called `createRectangle()`. This function will take in the `height` and `width` and return a new `Rectangle` object:

```
function createRectangle(height, width) {
  return Object.create(rectanglePrototypes, {
    height: {
      value: height,
    },
    width: {
      value: width,
    },
  });
}

const rectangle2 = createRectangle(5, 8);
console.log(rectangle2, rectangle2.isSquare());
```

As you can see, we get the same result as before. We have created a new `Rectangle` object using the `Object.create()` method. We have also set the prototype of the `Rectangle` object to the `rectanglePrototypes` object.

Both examples do the same thing and give the same result. The `Object.create()` method is just another way to create objects and set the prototype of an object.

When we use the `new` keyword, the prototype of the object includes the constructor function's prototype property. This happens automatically. When we use the `Object.create()` method, we can manually set the prototype to any object we want.

You could use either method to create objects and set prototypes. Using the `new` keyword is more common, but you could use either method.

`Object.create()` is also very useful when we use inheritance. We will learn about inheritance in the next lesson.

# Prototypical Inheritance & the `call()` Method

I've already talked about prototypical inheritance. We saw that the `Rectangle` constructor function as well as any array as well as many other things inherits from the `Object.prototype` or `objectBase`. This is why we can use the `toString()` method on our `Rectangle` object or on an array.

What I want to do now is show you how we can use inheritance with our own objects. I want to create a `Shape` constructor function that will be the base for other shapes. So we can create a `Circle`, `Square`, etc and have them all inherit from the `Shape` constructor function as well as any prototype methods we add to the `Shape` constructor function.

Let's create a very simple constructor function for a `Shape`:

```
function Shape(name) {  
    this.name = name;  
}
```

The only property I am going to add to the `Shape` constructor function is the `name` property.

## `call()`

The `call()` method allows us to call a function with a given `this` value and arguments provided individually. The `call()` method is a predefined JavaScript method. It can be used for a variety of purposes. We will use it to call the `Shape` constructor function from the `Rectangle` constructor function.

```
function Rectangle(name, height, width) {  
    Shape.call(this, name);  
  
    this.height = height;  
    this.width = width;  
}
```

```
const rect1 = new Rectangle('Rectangle 1', 10, 10);  
console.log(rect1); // Rectangle { name: 'Rectangle 1', height: 10, width: 10 }
```

As you can see, we are calling the `Shape` constructor function from the `Rectangle` constructor function using `call()`. We are passing in the `this` value and the `name` property. This will set the `name` property on the `Rectangle` object. Then we can add the `height` and `width` properties to the `Rectangle` object.

## Why use `call()`?

If we did not use the `call()` method, we would get an error that says **Uncaught TypeError: Shape is not a constructor**. This is because the `Shape` constructor function would be called as a regular function and in a regular function, the `this` value is the `window` object. We need to use the `call()` method to set the `this` value to the `Rectangle` object.

Let's do the same with a `Circle` constructor function:

```
function Circle(name, radius) {  
  Shape.call(this, name);  
  
  this.radius = radius;  
}
```

```
const circle1 = new Circle('Circle 1', 10);  
console.log(circle1); // Circle { name: 'Circle 1', radius: 10 }
```

## Inheriting Prototype Methods

Let's add a `logName()` method to the `Shape` constructor function and try to use it on the `Rectangle` object

```
Shape.prototype.logName = function () {  
  console.log(`Shape name is: ${this.name}`);  
};  
  
rect1.logName();
```

This will throw an error: **Uncaught TypeError: rect1.logName is not a function**. The prototype methods are not inherited automatically from the `Shape` constructor function.

We can fix this by using the `Object.create()` method. We're going to set the `Rectangle.prototype` to the `Shape.prototype` using the `Object.create()` method.

```
Rectangle.prototype = Object.create(Shape.prototype);
```

We have to use the `Object.create()` method because if you just set the `Rectangle.prototype` to the `Shape.prototype`, the `constructor` property and the prototype chain will be broken.

Now, let's try to call the `logName()` method on the `Rectangle` object:

```
rect1.logName(); // Shape name is: Rectangle 1
```

---

It works! We can also call the `logName()` method on the `Circle` object:

```
circle1.logName(); // Shape name is: Circle 1
```

So again, JavaScript will look for `logName()` directly on the `Rectangle` or `Circle` object. It will not find it, so it will look at the object's prototype which is the `Shape.prototype`. It will find the `logName()` method and will use it.

## Set constructor property

Right now, if we log the constructor of the `Rectangle` object, we will get the `Shape` constructor function:

```
console.log(rect1.constructor); // [Function: Shape]
```

This may cause some unexpected results in the future, so you probably want to set the `constructor` property on the `Rectangle.prototype` to the `Rectangle` constructor function.

```
Rectangle.prototype.constructor = Rectangle;  
  
console.log(rect1.constructor); // [Function: Rectangle]
```

We can do the same for the `Circle` constructor function:

```
Circle.prototype.constructor = Circle;
```

## Polymorphism - Overwriting Prototype Methods

You should also be able to overwrite prototype methods. Let's add a `logName()` method to the `Rectangle` constructor function:

```
Rectangle.prototype.logName = function () {  
  console.log(`Rectangle name is: ${this.name}`);  
};  
  
rect1.logName(); // Rectangle name is: Rectangle 1
```

This is an example of polymorphism. We are able to have the same method name on different objects and have different results. We could have 100 different objects and all of them could have a `logName()` method and they would all do something different.

# OOP Game Challenge

---

## Instructions:

- Create a constructor to create a `Player` object with a name
- Player should have a `name` as well as a `lvl` set to `1` by default and `points` set to `0` by default
- Create a method on the prototype called `gainXp` that takes in a number from 1-10 and adds it to the players `points`. If the current `points` are  $\geq 10$  then add 1 to the `lvl` and decrement the points by 10.
- Create another prototype method called `describe` that displays the players stats (name, lvl, points)

You should be able to use the Player object like this:

```
let player1 = new Player('Bob');
let player2 = new Player('Alice');

player1.gainXp(5);
player2.gainXp(7);
player1.gainXp(3);
player2.gainXp(2);
player1.gainXp(8);
player2.gainXp(4);

console.log(player1.describe()); // Bob is level 2 with 6 experience points
console.log(player2.describe()); // Alice is level 2 with 3 experience points
```

► Click For Solution

```
function Player(name) {
  this.name = name;
  this.lvl = 1;
  this.points = 0;
}

Player.prototype.gainXp = function (xp) {
  this.points += xp;
  if (this.points >= 10) {
    this.lvl++;
    this.points -= 10;
  }
};

Player.prototype.describe = function () {
  return `${this.name} is level ${this.lvl} with ${this.points} experience
  points`;
};
```

```
let player1 = new Player('Bob');
let player2 = new Player('Alice');

player1.gainXp(5);
player2.gainXp(7);
player1.gainXp(3);
player2.gainXp(2);
player1.gainXp(8);
player2.gainXp(4);

console.log(player1.describe());
console.log(player2.describe());
```

# Classes

---

We have talked about OOP quite a bit and as I've said before, JavaScript does not have classes at its core. However, The ES6 spec does have classes and they are very similar to classes in other languages. I would say that most OOP languages use classes, including Java, C#, C++, PHP and Objective-C. If you have worked with a language that uses classes, then this should be pretty familiar to you.

What I mean when I say JavaScript does not have classes at it's core is that the code that is actually being run looks like what we have already learned. Constructor functions, prototypes, etc. However, ES6 gave us what is called a "syntactic sugar" for classes. This means that we can write code that looks like classes, but it is actually just constructor functions and prototypes under the hood.

It has become pretty popular to use classes in JavaScript. Many people think that it is a much easier and less confusing syntax. Also, a lot of people are coming from class-based languages. So, it is nice to have a syntax that is similar to what they are used to. What you use is completely up to you. We saw the basics of constructors/prototypes, now let's get into classes.

## Creating a Class

To create a class, we use the `class` keyword. We then give it a name. The name should be capitalized. This is just a convention. It is not required. We then use the `constructor` keyword to create a constructor function. This is the function that will be called when we create a new instance of the class. We can then add properties to `this` just like we would in a constructor function. We can also add methods to the prototype just like we would in a constructor function. Here is an example:

```
class Rectangle {  
    constructor(height, width) {  
        this.height = height;  
        this.width = width;  
    }  
  
    area() {  
        return this.height * this.width;  
    }  
}
```

We can then create a new instance of the class by using the `new` keyword. We then call the class like it was a function. We pass in the arguments that the constructor function expects. Here is an example:

```
const square = new Rectangle(10, 10);
```

We can then access the properties and methods on the instance just like we would with any other object. Here is an example:

```
console.log(square.area()); // 100
```

In the `area()` method, we used `this` to access the properties. We can also access methods with `this`. Let's add a few more methods, including a `logArea()` method and access `area()` within it.

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }

  area() {
    return this.height * this.width;
  }

  perimeter() {
    return 2 * (this.height + this.width);
  }

  isSquare() {
    return this.height === this.width;
  }

  logArea() {
    console.log('Rectangle Area: ' + this.area());
  }
}

console.log(square.perimeter()); // 40
console.log(square.isSquare()); // true
square.logArea(); // Rectangle Area: 100
```

## Under The Hood

Let's log the `square` variable that we created from the class to the console:

```
console.log(square);
```

Notice we see the same thing that we have been seeing when we use constructors/prototypes.



It is a `Rectangle` object with a `height` property of `10` and a `width` property of `10`. It also has an `area` method on the prototype. This is exactly what we would expect.

As I mentioned before, classes are just syntactic sugar for constructor functions and prototypes. So, what is actually happening when we create a class? Let's take a look:

```
function Rectangle(height, width) {
  this.height = height;
  this.width = width;
}

Rectangle.prototype.area = function () {
  return this.height * this.width;
};

Rectangle.prototype.perimeter = function () {
  return 2 * (this.height + this.width);
};

Rectangle.prototype.isSquare = function () {
  return this.height === this.width;
};

Rectangle.prototype.logArea = function () {
  console.log('Rectangle Area: ' + this.area());
};

const square = new Rectangle('Square', 20, 20);
console.log(square.perimeter());
console.log(square.isSquare());
square.logArea();
```

The same code that we have been writing throughout this section. The only difference is that we are using the `class` keyword and the `constructor` keyword and the code looks a bit nicer.

It's up to you on how you want to write your code. My job is just to show you the different ways that you can write it. Some people like the constructor/prototype method instead of using the abstraction of classes. Some people like neat structure of classes. I prefer classes most of the time, but I'll use either.

# Class Inheritance

---

We looked at the ES6 class syntax in the previous lesson. In this lesson, we will look at inheritance in classes.

Inheritance is the ability to create a new class from an existing class. The new class will inherit all the properties and methods of the existing class. This is called a **subclass** or **child class**. The existing class is called a **superclass** or **parent class**.

Let's create a parent class of **Shape**. This class will have a constructor that takes in a name. It will also have a **logName** method that will log the name to the console. Here is an example:

```
class Shape {  
    constructor(name) {  
        this.name = name;  
    }  
  
    logName() {  
        console.log(this.name);  
    }  
}
```

Now, let's create a subclass of **Shape** called **Rectangle**. This class will have a constructor that takes in a height and width. It will also have an **area** method that will return the area of the rectangle. Here is an example:

```
class Rectangle extends Shape {  
    constructor(name, height, width) {  
        super(name);  
        this.height = height;  
        this.width = width;  
    }  
  
    area() {  
        return this.height * this.width;  
    }  
}
```

We did a couple of new things here. First, we used the **extends** keyword to create a subclass of **Shape**. This is how we tell JavaScript that **Rectangle** is a subclass of **Shape**. Second, we used the **super()** method to call the constructor of the parent class. We need to do this so that the **name** property is set on the instance. If we did not call **super**, the **name** property would not be set on the instance. Third, we added the **height** and **width** properties to the instance. Finally, we added the **area** method to the instance.

We can create a **Circle** class and do the same thing.

```
class Circle extends Shape {  
    constructor(name, radius) {  
        super(name);  
        this.radius = radius;  
    }  
  
    area() {  
        return Math.PI * this.radius * this.radius;  
    }  
}
```

We can now create a new instance of `Rectangle` and/or `Circle` and call the `logName` method. Here is an example:

```
const square = new Rectangle('Square', 10, 10);  
square.logName(); // Square
```

Let's log the `square` variable to the console.

```
console.log(square);
```



You can see that in the prototype, the constructor is the `Rectangle` class. The `prototype` property is the `Shape` class. This is how inheritance works in JavaScript. The `Rectangle` class inherits all the properties and methods of the `Shape` class. You can see the `logName()` function in the `Shape` prototype. This is how the `logName()` method is available on the `Rectangle` instance.

We can use `instanceOf` to check if our `square` instance is an instance of the `Rectangle` AND the `Shape` class. Which it is.

```
console.log(square instanceof Rectangle); // true  
console.log(square instanceof Shape); // true
```

## Overriding Methods

We can override methods in a subclass. Let's override the `logName` method in the `Rectangle` class.

```
logName() {  
    console.log('Rectangle name is: ' + this.name);  
}
```

```
square.logName(); // Rectangle name is: Square
```

# Static Methods

---

Static methods are methods that are available on the class itself. They are not available on the instances of the class. Static methods are often used to create utility functions or to hold data that is shared across all instances of the class. Let's create our `Shape` and `Rectangle` class from the previous example and add a static method called `getClassName`.

A static method is created the same way as a regular method. The only difference is that we use the `static` keyword.

```
class Shape {  
    constructor(name) {  
        this.name = name;  
    }  
  
    logName() {  
        console.log(this.name);  
    }  
  
    static getClassName() {  
        return 'Shape';  
    }  
}
```

```
class Rectangle extends Shape {  
    constructor(name, height, width) {  
        super(name);  
        this.height = height;  
        this.width = width;  
    }  
  
    area() {  
        return this.height * this.width;  
    }  
  
    static getClassName() {  
        return 'Rectangle';  
    }  
}
```

Let's try to create an instance of a `Rectangle` and call the `getClassName` method.

```
const rect = new Rectangle('Rectangle', 10, 20);
```

```
rect.getClassName(); // TypeError: rect.getClassName is not a function
```

This does not work, because the `getClassName` method is not available on the instance of the `Rectangle` class. It is only available on the class itself.

```
console.log(Shape.getClassName());  
console.log(Rectangle.getClassName());
```

So if you have a method where you don't need to access the instance of the class, you can make it a static method. This way you can call it directly on the class.

# bind() & this

---

So we've talked about the `this` keyword quite a bit. You know that when we create either a constructor function or a class, `this` refers to the current instance. However, if we use it in a regular function or the global scope, it refers to the `window` object. If we use it on an event handler, it refers to the element that the event was triggered on. So the `this` keyword is very dynamic and it changes depending on how we use it. That's why we have certain methods that allow us to set the `this` value manually. Those methods are `call()`, `apply()`, and `bind()`. We already looked at `call()` back in the prototypical inheritance lesson. We'll look at `apply()` later. Right now I'm going to show you how to use `bind()`. We'll need to know this for our project that's coming up as well.

So `bind()` is used to set the `this` value manually. It returns a new function where the `this` value is bound to the value we pass in. One very common use case for `bind()` is when we want to use a callback function.

Let's create a class called `App` and add a property of `serverName` and add an event listener in the constructor to listen for a button click and when that happens, we want to call a class method and use the `this` value from the `App` class.

```
class App {  
    constructor() {  
        this.serverName = 'http://localhost:3000';  
  
        document  
            .querySelector('button')  
            .addEventListener('click', this.handleClick);  
    }  
  
    handleClick() {  
        console.log(this.serverName);  
    }  
}  
  
const app = new App();
```

If we run this code, we get `undefined`. This is because the `this` value is the `window` object due to the fact that we are using a callback function, which is a regular function.

If we just log `this` inside the `handleClick()` method, we get the `window` object.

```
handleClick() {  
    console.log(this); // Window {window: Window, self: Window, document:  
document, name: "", location: Location, ...}  
}
```

So we need to use `bind()` to set the `this` value to the `App` class.

```
class App {  
  constructor() {  
    this.serverName = 'localhost';  
  
    document  
      .querySelector('button')  
      .addEventListener('click', this.handleClick.bind(this));  
  }  
  handleClick() {  
    console.log(this.serverName);  
  }  
}  
  
const app = new App();
```

Now, we get the `serverName` property value. If you log `this` inside the `handleClick()` method, you get the `App` class instance.

```
handleClick() {  
  console.log(this); // App {serverName: "localhost"}  
}
```

Methods like `call` and `bind` are very overwhelming to a lot of people, but I gave you two real life examples, which I think helps people understand them better.

# Getters & Setters With Classes

---

Getters and setters are methods that are used to get or set property values for objects. We can use them with classes, constructors, and object literals. I'm going to show you all three, but I'm going to start with classes, because it's the most common way to use them and the easier syntax.

There are a few reasons to use getters and setters. They allow you to control how a property is accessed. This is useful when you want to perform an action before returning the value of a property. For example, you may want to ensure that a property is always capitalized before getting or setting it.

They also allow you to keep the same syntax whether it's a regular property or a method.

They are also used with private properties. For example, you may have properties that you don't want to be accessed directly. Instead you'll have getters and setters for them. I'll get more into that soon.

I'm going to start off with a different example than our shapes classes. We're going to create a person class.

```
class Person {  
  constructor(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
  }  
}
```

With this class, I can access the first name and last name of a person like this:

```
const person = new Person('John', 'Doe');  
console.log(person.firstName); // John  
console.log(person.lastName); // Doe
```

But what if I want to make sure that the first name and last name are always capitalized? I check that before I get it and/or set it.

```
class Person {  
  constructor(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
  }  
  
  get firstName() {  
    return this._firstName.charAt(0).toUpperCase() + this._firstName.slice(1);  
  }  
}
```

```
    set firstName(value) {
      this._firstName = value.charAt(0).toUpperCase() + value.slice(1);
    }
}
```

Now, when I get or set the first name, it will always be capitalized.

```
const person = new Person('john', 'doe');
console.log(person.firstName); // John
```

I would probably do the same for the last name and create a utility function for capitalizing a string.

```
class Person {
  // ...
  get firstName() {
    return this.capitalizeFirst(this._firstName);
  }

  set firstName(value) {
    this._firstName = this.capitalizeFirst(value);
  }

  get lastName() {
    return this.capitalizeFirst(this._lastName);
  }

  set lastName(value) {
    this._lastName = this.capitalizeFirst(value);
  }

  capitalizeFirst(value) {
    return value.charAt(0).toUpperCase() + value.slice(1);
  }
}
```

Let's say that we want to have a full name property that is the first name and last name combined. We could create a method called `getFullName` that returns the full name. But we can also create a getter for it, which would make it look like a property.

```
get fullName() {
  return `${this.firstName} ${this.lastName}`;
}

set fullName(name) {
  const names = name.split(' ');
  this.firstName = names[0];
```

```
this.lastName = names[1];  
}
```

Now we can just access the full name like a property.

```
const person = new Person('brad', 'traversy');  
console.log(person.fullName); // John Doe
```

# Getters & Setters Using `defineProperty`

---

In the last lesson, I showed you how to create getters and setters with the `get` and `set` keywords inside of a class. But there's another way that we can do this that is common with `constructor functions` and that is using the `Object.defineProperty()` method.

It takes three arguments. The first is the object that we want to add the property to (`this`). The second is the name of the property that we want to add. And the third is an object that contains the getter and setter functions. Let's add a getter and setter for the `firstName` and `lastName` property

```
function Person(firstName, lastName) {
  this._firstName = firstName;
  this._lastName = lastName;

  Object.defineProperty(this, 'firstName', {
    get: function () {
      return this._firstName;
    },
    set: function (value) {
      this._firstName = value;
    },
  });
}

Object.defineProperty(this, 'lastName', {
  get: function () {
    return this._lastName;
  },
  set: function (value) {
    this._lastName = value;
  },
});
```

As you can see, the original properties are prefixed with an underscore. Because we want the non-underscore version to be the getter and setter.

Now, we can use it in the same way we used the class version. I'm going to use lowercase letters because I want to make it part of the getter that it returns uppercase

```
const person1 = new Person('john', 'doe');
console.log(person1.firstName); //john
console.log(person1.lastName); // doe
```

Let's create a method to capitalize the first letter. I am actually going to put this on to the `prototype`. We don't have to, but why not?

```
Person.prototype.capitalizeFirst = function (value) {
  return value.charAt(0).toUpperCase() + value.slice(1);
};
```

Now, let's add to the getters. Let's also create a getter for the `fullName`. In that getter, we can actually use the `firstName` and `lastName` getters. It will automatically be capitalized.

```
function Person(firstName, lastName) {
  this._firstName = firstName;
  this._lastName = lastName;

  Object.defineProperty(this, 'firstName', {
    get: function () {
      return this.capitalizeFirst(this._firstName);
    },
    set: function (value) {
      this._firstName = value;
    },
  });

  Object.defineProperty(this, 'lastName', {
    get: function () {
      return this.capitalizeFirst(this._lastName);
    },
    set: function (value) {
      this._lastName = value;
    },
  });

  Object.defineProperty(this, 'fullName', {
    get: function () {
      // Using the getters
      return this.firstName + ' ' + this.lastName;
    },
    set: function (value) {
      this._firstName = value;
    },
  });
}
```

Let's try it out:

```
console.log(person1.firstName); // John
console.log(person1.lastName); // Doe
console.log(person1.fullName); // John Doe
```

## Object Literal Syntax

Let's do the same thing using object literal syntax. We could create a new `capitalizeFirst()` function or we can use the `Person` prototype method.

```
const PersonObj = {
  _firstName: 'jane',
  _lastName: 'doe',

  get firstName() {
    return Person.prototype.capitalizeFirst(this._firstName);
  },

  set firstName(value) {
    this._firstName = value;
  },

  get lastName() {
    return Person.prototype.capitalizeFirst(this._lastName);
  },

  set lastName(value) {
    this._lastName = value;
  },

  get fullName() {
    // Using the getters
    return this.firstName + ' ' + this.lastName;
  },
};
```

Now, we can set the `width` and `height` properties and get and set the `area` property. I'm going to use the `Object.create()` method to create a new object that inherits from the `RectangleObj` object, but you could just as well set it directly on the `RectangleObj` object.

```
const person2 = Object.create(PersonObj);
console.log(person2.firstName); // Jane
console.log(person2.lastName); // Doe
console.log(person2.fullName); // Jane Doe
```

# Convention For Private Properties

---

Now we are going to get a bit deeper into encapsulation, which often includes the process of hiding data or hiding specific properties and methods of a class.

In many OOP languages that use classes, you can use specific keywords to indicate which properties and methods are accessible from outside the class. For example, in Java, you can use the `private` keyword to indicate that a property or method is only accessible from within the class. In JavaScript, we don't have those keywords, but there is a convention that is commonly used to indicate that a property or method is private. We use an underscore `_` before the property or method name.

There is also a new feature in ES2022 that allows us to use the `#` symbol to create private fields. This is a new feature that isn't yet supported in all browsers. We'll look at that in the next video. Right now, in the beginning of 2023, you'll probably run into the underscore convention more often than the `#` symbol. There are a few other ways to implement this as well including using `Symbols` and the `WeakMap` object. The underscore convention is definitely the most common at this point in time.

Let's create a new class called `Wallet` and add a constructor that has a `balance` and a `transactions` property. The `balance` will be 0 and `transactions` will be an empty array.

```
class Wallet {  
    constructor() {  
        this.balance = 0;  
        this.transactions = [];  
    }  
}
```

Now let's add a `deposit` method that takes an amount and adds it to the `balance` and adds a new transaction to the `transactions` array. We'll also add a `withdraw` method that takes an amount and subtracts it from the `balance` and adds a new transaction to the `transactions` array.

```
class Wallet {  
    constructor() {  
        this.balance = 0;  
        this.transactions = [];  
    }  
  
    deposit(amount) {  
        this.balance += amount;  
    }  
  
    withdraw(amount) {  
        this.balance -= amount;  
    }  
}
```

---

Now let's create a new instance of the `Wallet` class and call the `deposit` method with an amount of 300. Then withdraw an amount of 50. Then we'll log the `balance` property to the console.

```
const wallet = new Wallet();
wallet.deposit(300);
wallet.withdraw(50);
console.log(wallet.balance); // 250
```

It works as expected. However, we don't want to expose the `balance` property to the outside world. We want to make it private. We want to make it so that the only way to access the `balance` property is through the `deposit` and `withdraw` methods. This is part of encapsulation.

Now, like I said, JavaScript does not have a `private` keyword, but we can use the underscore convention to indicate that a property or method is private. So let's add an underscore to the `balance` property.

```
class Wallet {
  constructor() {
    this._balance = 0;
    this.transactions = [];
  }

  deposit(amount) {
    this._balance += amount;
  }

  withdraw(amount) {
    if (amount > this._balance) {
      console.log(`No enough funds`);
      return;
    }
    this._balance -= amount;
  }
}
```

This convention tells the developer that the `balance` property is private and should not be accessed directly. We do want to be able to get the balance. We just don't want to be able to set it directly. So let's add a `getBalance` method that returns the `balance` property.

```
class Wallet {
  constructor() {
    this._balance = 0;
    this.transactions = [];
  }

  deposit(amount) {
```

```

    this._balance += amount;
}

withdraw(amount) {
  if (amount > this._balance) {
    console.log(`No enough funds`);
    return;
  }
  this._balance -= amount;
}

getBalance() {
  return this._balance;
}
}

```

Now we can call the `getBalance` method to get the balance. We can't access the `balance` property directly.

```

const wallet = new Wallet();
wallet.deposit(300);
wallet.withdraw(50);
console.log(wallet.getBalance()); // 250

```

## Using a Getter

We could use a function, but I would prefer to use a getter to get the balance. Let's remove the function and add a getter.

```

class Wallet {
  constructor() {
    this._balance = 0;
    this.transactions = [];
  }

  deposit(amount) {
    this._balance += amount;
  }

  withdraw(amount) {
    if (amount > this._balance) {
      console.log(`No enough funds`);
      return;
    }
    this._balance -= amount;
  }

  get balance() {

```

```
        return this._balance;
    }
}
```

Now, we can access the `balance` through the getter

```
const wallet = new Wallet();
wallet.deposit(300);
wallet.withdraw(50);
console.log(wallet.balance); // 250
```

The `transactions` property should also be private, so let's add an underscore to that property and create another getter for that. Let's also create 2 new private methods called `_processDeposit` and `_processWithdrawal`. These methods will add a new transaction to the `transactions` array.

```
class Wallet {
  constructor() {
    this._balance = 0;
    this._transactions = [];
  }

  get balance() {
    return this._balance;
  }

  get transactions() {
    return this._transactions;
  }

  deposit(amount) {
    this._processDeposit(amount);
    this._balance += amount;
  }

  withdraw(amount) {
    this._processWithdrawal(amount);
    if (amount > this._balance) {
      console.log(`No enough funds`);
      return;
    }
    this._balance -= amount;
  }

  _processDeposit(amount) {
    console.log(`Depositing ${amount}`);
    this._transactions.push({
      type: 'deposit',
    });
  }

  _processWithdrawal(amount) {
    console.log(`Withdrawing ${amount}`);
    this._balance -= amount;
    this._transactions.push({
      type: 'withdrawal',
    });
  }
}
```

```
        amount,
    });
}

_processWithdraw(amount) {
    console.log(`Withdrawing ${amount}`);

    this._transactions.push({
        type: 'withdraw',
        amount,
    });
}
}
```

We made the 2 new methods private because there is absolutely no reason for the outside world to call these methods. They are only used internally by the `deposit` and `withdraw` methods. When we create documentation for this interface, we would not include these methods. There is no reason to. Hopefully, encapsulation makes sense to you now.

Now, we can call the `deposit` and `withdraw` methods and we can access the `balance` and `transactions` properties through the getters.

```
const wallet = new Wallet();
wallet.deposit(300);
wallet.withdraw(50);
console.log(wallet.balance); // 250
console.log(wallet.transactions); // [{type: 'deposit', amount: 300}, {type: 'withdraw', amount: 50}]
```

# Private Fields In ES2022

---

Up until recently, JavaScript did not have a way to create private entities in classes. We used the underscore convention for a long time. However, in ES2022, we now have a way to create private class fields by using the `#` symbol. This is a new feature that isn't yet supported in all browsers. Right now, in the beginning of 2023, you'll probably run into the underscore convention more often than the `#` symbol. But this seems to be the future of private fields in JavaScript.

Let's use our `Wallet` example from the last video and go through and see what we need to change.

First, there is now a concept of `fields` and `properties`. A `field` is a variable that is declared inside of a class. A `property` is a variable that is declared inside of an object. So, in the last video, we had a `balance` property and a `transactions` property. Now, we're going to have a `#balance` field and a `#transactions` field defined directly in the class as opposed to in the constructor.

```
class Wallet {  
    #balance = 0;  
    #transactions = [];  
}
```

The getters will remain the same. We'll just need to change the `balance` property to `#balance` and the `transactions` property to `#transactions`.

```
class Wallet {  
    #balance = 0;  
    #transactions = [];  
  
    get balance() {  
        return this.#balance;  
    }  
  
    get transactions() {  
        return this.#transactions;  
    }  
}
```

The `deposit` and `withdraw` methods will also remain the same. We'll just need to change the `balance` property to `#balance` and the `transactions` property to `#transactions`. We will also make the process methods private by adding the `#` symbol to the beginning of the method name.

```
class Wallet {  
    #balance = 0;  
    #transactions = [];
```

```

get balance() {
    return this.#balance;
}

get transactions() {
    return this.#transactions;
}

deposit(amount) {
    this.#processDeposit(amount);
    this.#balance += amount;
}

withdraw(amount) {
    this.#processWithdraw(amount);
    if (amount > this.#balance) {
        console.log(`No enough funds`);
        return;
    }
    this.#balance -= amount;
}

#processDeposit(amount) {
    console.log(`Depositing ${amount}`);

    this.#transactions.push({
        type: 'deposit',
        amount,
    });
}

#processWithdraw(amount) {
    console.log(`Withdrawing ${amount}`);

    this.#transactions.push({
        type: 'withdraw',
        amount,
    });
}

```

Now, we can create a new instance of the `Wallet` class and call the `deposit` and `withdraw` methods and then show the balance and transactions using the getters.

```

const myWallet = new Wallet();
myWallet.deposit(300);
myWallet.withdraw(50);
console.log(myWallet.balance); // 250
console.log(myWallet.transactions); // [ { type: 'deposit', amount: 300 }, {
type: 'withdraw', amount: 50 } ]

```

Now, let's try and access the `#balance` and `#transactions` fields directly. We'll get an error because they are private fields.

```
console.log(myWallet.#balance); // Uncaught SyntaxError: Private field  
'#balance' must be declared in an enclosing class  
console.log(myWallet.#transactions); // Uncaught SyntaxError: Private field  
'#transactions' must be declared in an enclosing class
```

Let's try to be criminals and directly change our balance to \$1,000,0000.

```
myWallet.#balance = 1000000;  
console.log(myWallet.balance); // Uncaught SyntaxError: Private field  
'#balance' must be declared in an enclosing class
```

What if we try and just do `myWallet.balance = 1000000`? This will work because we're not directly changing the `#balance` field.

```
myWallet.balance = 1000000;  
console.log(myWallet.balance); // 250
```

Let's try and call the `#processDeposit` method directly. We'll get an error because it is a private method. Same with `#processWithdraw`

```
myWallet.#processDeposit(100); // Uncaught SyntaxError: Private field  
'#processDeposit' must be declared in an enclosing class
```

So, you see, we now have full encapsulation and real private fields and methods in JavaScript. This is a huge step forward for JavaScript.

# Property Flags & Descriptors

---

Property flags are internal attributes of a property. They are not accessible directly, but we can use `Object.getOwnPropertyDescriptor`. The following is a list of available property flags:

- `[[Configurable]]` - if `true`, the property can be deleted and these attributes can be modified, otherwise not
- `[[Enumerable]]` - if `true`, the property will be returned in a `for...in` loop, otherwise not
- `[[Writable]]` - if `true`, the value of the property can be changed, otherwise not
- `[[Value]]` - the value of the property

## `getOwnPropertyDescriptor` Method

When we create a new object, whether it is an object literal or an instance from a constructor or a class, the flags for all properties are set to `true` by default. The `getOwnPropertyDescriptor()` method returns an object with property flags. This object is called a property descriptor.

Before, we create our own object and experiment, let's check the flags for the `Math.PI` property.

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');
console.log(descriptor); // {value: 3.141592653589793, writable: false,
enumerable: false, configurable: false}
```

As you can see, the property flags of the `Math.PI` property are `false`. If we try and change the value of the `Math.PI` property, it will not work.

```
Math.PI = 3;
console.log(Math.PI); // 3.141592653589793
```

Let's create an object literal called `rectObj` object to work with. It will have a `name`, `height` and `width` property. You could also use a constructor or a class to create the object, but an object literal is easier to work with.

```
const rectObj = {
  name: 'Rectangle 1',
  width: 10,
  height: 10,
};

console.log(rectObj); // {name: "Rectangle 1", width: 10, height: 10}
```

## Get Property Flags

Let's get the property flags of the `name` property of the `rectObj` object.

```
console.log(Object.getOwnPropertyDescriptor(rectObj, 'name'));
// {value: "Rectangle 1", writable: true, enumerable: true, configurable: true}
```

They are all set to `true` by default.

## Change Property Flags

We can use the `Object.defineProperty` method to change the property flags of an existing property. Let's change the property flags of the `name` property of the `rectObj` object.

```
Object.defineProperty(rectObj, 'name', {
  writable: false,
  enumerable: false,
  configurable: false,
});
```

if we check again the property flags of the `name` property, we can see that they have changed.

```
console.log(Object.getOwnPropertyDescriptor(rectObj, 'name'));
// {value: "Rectangle 1", writable: false, enumerable: false, configurable:
false}
```

Now let's try and change the name value to something else:

```
rectObj.name = 'Rectangle 1 Updated';
console.log(rectObj.name); // Rectangle 1
```

The name was not updated because we set the `writable` flag to `false`. Let's try and delete the `name` property:

```
delete rectObj.name;
console.log(rectObj.name); // Rectangle 1
```

The `name` property was not deleted because we set the `configurable` flag to `false`.

Let's try and enumerate through the properties of the `rectObj` object. I'm going to use a `for...of` loop on the `Object.entries` method to get the key and value of each property.

```
for (let [key, value] of Object.entries(rectObj)) {
  if (typeof value !== 'function') {
    console.log(` ${key}: ${value}`);
  }
}
```

Notice the `name` property was not returned because we set the `enumerable` flag to `false`.

So we have really limited the access to the `name` property of the `rectObj` object. We can't change the value, delete the property, or enumerate through it. We can't even change the property flags of the `name` property because we set the `configurable` flag to `false`.

## Object.getOwnPropertyDescriptors Method

The `Object.getOwnPropertyDescriptors` method returns an object with all the property flags of the object.

```
console.log(Object.getOwnPropertyDescriptors(rectObj));
// {name: {...}, width: {...}, height: {...}, area: {...}}
```

They are all true except for the `name` property, because we changed them.

# Sealing & Freezing Objects

Let's use our `rectObj` object from the previous lesson to demonstrate **sealing** and **freezing**.

```
const rectObj = {  
    name: 'Rectangle 1',  
    width: 10,  
    height: 10,  
};
```

## Sealing

Sealing an object prevents new properties from being added to it. Existing properties can still be modified or deleted. It sets the `configurable` flag to `false` for all existing properties. `writable` and `enumerable` flags are not affected.

```
Object.seal(rectObj);  
  
// Check flags  
let descriptor = Object.getOwnPropertyDescriptors(rectObj);  
console.log(descriptor); // {name: {...}, width: {...}, height: {...}, area: {...}}
```

Let's try and add a new property to the `rectObj` object.

```
rectObj.color = 'red';  
console.log(rectObj); // NOT added
```

Try removing an existing property from the `rectObj` object.

```
delete rectObj.width;  
console.log(rectObj); // NOT removed
```

I can not add a new property, however I can still modify existing properties.

```
rectObj.height = 20;  
console.log(rectObj); // Rectangle {name: "Rectangle 1", width: 10, height: 20}
```

## Freezing

Freezing an object prevents new properties from being added to it and prevents existing properties from being modified or deleted. It sets the `configurable` and `writable` flags to `false` for all existing properties.

Let's create a new object called `circleObj` and freeze it.

```
const circleObj = {  
  name: 'Circle 1',  
  radius: 10,  
};
```

```
Object.freeze(circleObj);  
  
// Check flags  
let descriptor = Object.getOwnPropertyDescriptors(circleObj);  
console.log(descriptor); // {name: {...}, width: {...}, height: {...}}
```

Let's try adding, removing and changing a frozen object:

```
// Try adding a new property  
circleObj.color = 'red';  
console.log(circleObj); // Not added  
  
// Try deleting a property  
delete circleObj.width;  
console.log(circleObj); // Not deleted  
  
// Try changing a property  
circleObj.name = 'Rectangle 2 Updated';  
console.log(circleObj); // Not changed
```

## Checking if an Object is Sealed or Frozen

We can use the `Object.isSealed` and `Object.isFrozen` methods to check if an object is sealed or frozen.

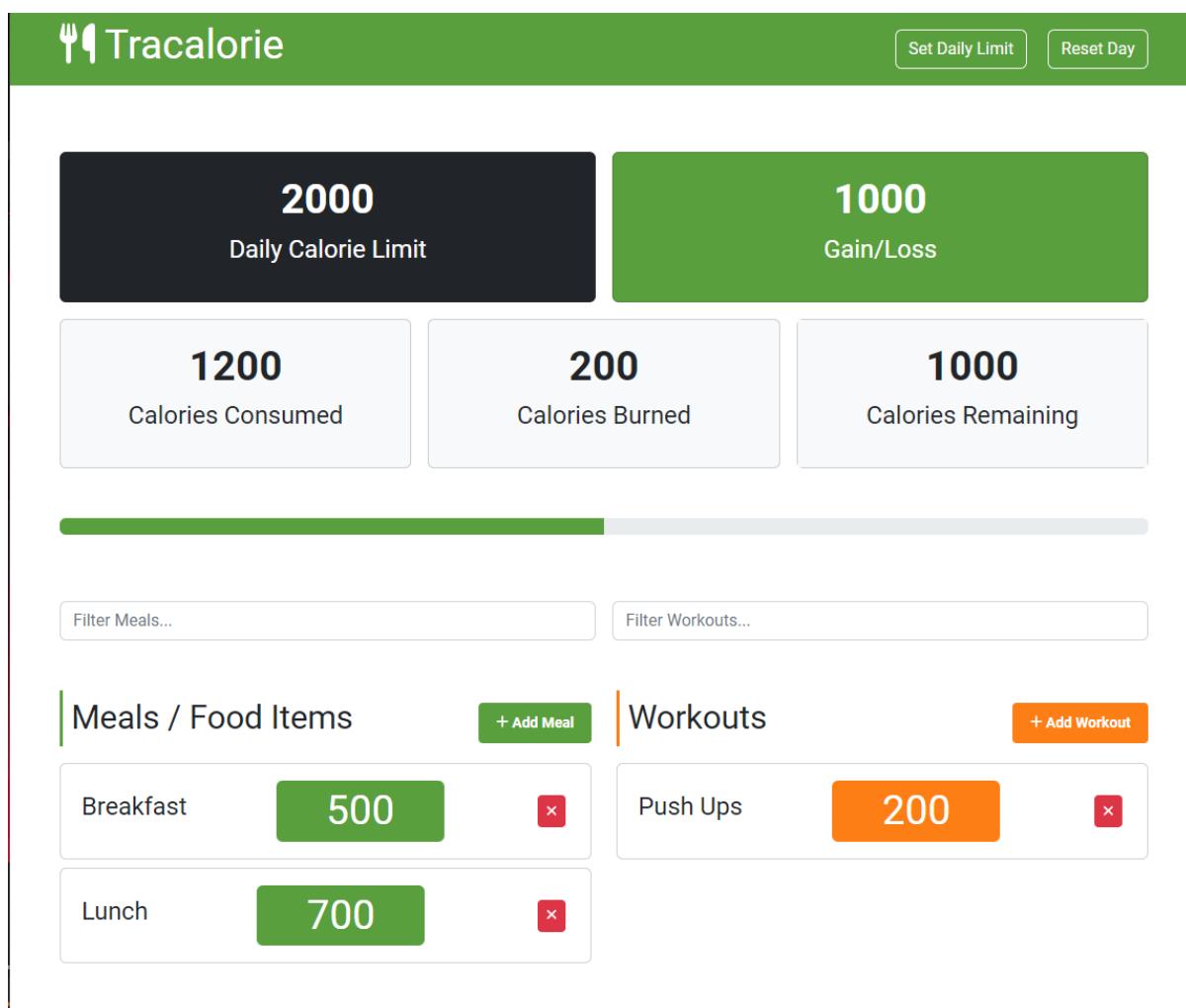
```
console.log('rectObj sealed? ', Object.isSealed(rectObj)); // true  
console.log('rectObj frozen? ', Object.isFrozen(rectObj)); // false  
console.log('circleObj frozen? ', Object.isFrozen(circleObj)); // true  
console.log('circleObj sealed? ', Object.isSealed(circleObj)); // true
```

Notice that `rectObj` is only frozen and not sealed. `circleObj` is both sealed and frozen. This is because if we freeze an object, it is automatically sealed.



# Tracalorie Project Intro

Now that you have an understanding of how OOP works, we are going to build a project that will help you understand how to use OOP in practice. We are going to build a calorie tracker app called Tracalorie. The app will allow users to add meals and workouts that will either add or remove (burn) calories from their total and track their calories. It is meant to be used as a daily tracker, but you could add to it and implement a schedule. Personally, I would not do that using client side local storage. It's just too much data. I would create a backend API and store the data in a database.



Let's look at the requirements for the app:

- Set a calorie limit
- Add meals and workouts
- Delete meals and workouts
- Filter meals and workouts
- Display total calories gain/loss
- A progress bar to visualize total calories gain/loss
- Display calories consumed and burned
- Reset calories and clear meals and workouts
- Use local storage to persist total calories, calorie limit, meals and workouts
- Use Bootstrap for styling and UI components (Modal & Collapse)

We will be using ES6 classes for this project. If you wanted to convert it to constructor functions and prototypes, that would not be difficult at all. I just prefer classes for something like this.

# Theme Setup

---

For this project, we will be using a custom Bootstrap 5 theme that I created. We are not going to be typing out the HTML/CSS. I have included a folder called `tracalorie_theme` in the lesson files. It is also in the main GitHub repo in the `_theme` folder. You can find the final code at <https://github.com/bradtraversy/tracalorie>.

Take the theme folder and rename it to `tracalorie`. Open it up with `VS Code` and open `index.html` with the `Live Server` extension.

You should see the UI, although nothing will work, because there is no JavaScript. the only things that will work that are dynamic is the modal when you click 'Set Daily Limit' and the collapse form when you click 'Add Meal' or 'Add Workout'. That is because we are including the Bootstrap JavaScript bundle. The rest of the JavaScript will be written from scratch.

## Sass Files

If you do not plan on changing any styles of the theme, you can delete the `SCSS` folder as well as all of the Bootstrap css files except for `bootstrap.css`. If you want to customize the styling, then you can use a Sass compiler like `Live Sass Compiler` to compile the `SCSS` files to `CSS`.

Now, that we have our UI and theme setup, we can start working on the JavaScript.

# Project Planning & Diagram

It's always a good idea to take time before starting a project and plan out the steps. This will give you sort of a map and will help you stay on track and make sure you don't miss anything.

The following diagram shows every class, property, method and relationship that will be needed for this project.



## App Class

The `App` class is basically the initializer. This is the only class that we instantiate in the global scope. Everything kicks off from there.

In the constructor, we will instantiate the `CalorieTracker` class, which is responsible for anything to do with tracking and displaying calorie related data. We will also have all of the event listeners in the constructor of the `App` class.

We will be using the tracker within all of the `App` class methods, so we will store it in a property called `_tracker`.

We are abiding by the underscore convention for private properties and methods that should only be accessed within that class.

## CalorieTracker Class

This is the main class for all of our calorie tracking and displaying. It will hold properties for the total calories, daily limit, meals and workouts. It will also have methods for adding and deleting items as well as displaying calorie data in the DOM.

## Meal & Workout Classes

The `Meal` and `Workout` classes will be very simple. They will only have a constructor with an `id`, `name` and `calories` property. When we use the `addMeal()` or `addWorkout()` methods in the tracker, we will be passing in an object instance of the `Meal` or `Workout` class.

## Storage Class

The `Storage` class will be responsible for storing and retrieving data from `localStorage`. We will store the total calories, calorie limit, meals and workouts. This will let us persist the data even if the user refreshes the page. All of the methods in this class will be static. There is no need to instantiate this class because there is only one `localStorage`.

Hopefully, this gives you an idea of how the project will be structured.

# Base Tracker, Meal & Workout Classes

---

We are going to start off by creating the base `CalorieTracker` class, as well as the `Meal` and `Workout` classes. We will be using the `Meal` and `Workout` classes to instantiate meal and workout objects that we will be passing into the `CalorieTracker` class.

Let's create the class with the constructor and properties.

```
class CalorieTracker {  
    constructor() {  
        this._calorieLimit = 2000;  
        this._totalCalories = 0;  
        this._meals = [];  
        this._workouts = [];  
    }  
}
```

We set the calorieLimit to a hardcoded **2000**, but later on, that will be controlled via the app. We also set the totalCalories to **0** and the meals and workouts to empty arrays.

We are also going to create a class to add a meal and a workout, which will just add the meal or workout to the respective array and add or remove the calories to the total calories. A workout burns calories, so we will subtract the calories from the total calories, and a meal adds calories, so we will add the calories to the total calories.

```
class CalorieTracker {  
    // ...  
  
    addMeal(meal) {  
        this._meals.push(meal);  
        this._totalCalories += meal.calories;  
    }  
  
    addWorkout(workout) {  
        this._workouts.push(workout);  
        this._totalCalories -= workout.calories;  
    }  
}
```

Now let's create the `Meal` and `Workout` classes. They will be very simple. They will only have a constructor with an `id`, `name` and `calories` property. When we use the `addMeal()` or `addWorkout()` methods in the tracker, we will be passing in an object instance of the `Meal` or `Workout` class.

The ID has to be different for each meal or workout, so I am going to use `Math.Random().toString(16)` which will give us a random hexadecimal string. We will then slice off the first two characters. There are many other ways to create random IDs, this is just one of them.

---

```

class Meal {
  constructor(name, calories) {
    this.id = Math.random().toString(16).slice(2);
    this.name = name;
    this.calories = calories;
  }
}

class Workout {
  constructor(name, calories) {
    this.id = Math.random().toString(16).slice(2);
    this.name = name;
    this.calories = calories;
  }
}

```

Now, we should be able to instantiate the `CalorieTracker` class and add meals and workouts to it.

```

const tracker = new CalorieTracker();

const breakfast = new Meal('Breakfast', 400);
tracker.addMeal(breakfast);

const run = new Workout('Morning Run', 300);
tracker.addWorkout(run);

```

We can see the results, by logging the properties. Now these are private properties, so we should not be accessing them directly, but for now, we will just log them to the console to see the results.

```

console.log(tracker._meals); // [Meal { id: '1a2b3c', name: 'Breakfast',
calories: 400 }]
console.log(tracker._workouts); // [Workout { id: '4d5e6f', name: 'Morning
Run', calories: 300 }]
console.log(tracker._totalCalories); // 100

```

# Display Stats

---

In the last lesson, we created a class for the calorie tracker. In this lesson, we will create methods to display the stats on the UI.

We are going to have methods to display the total calories, the calorie limit, the calories consumed, the calories burned, and the calories remaining.

```
class CalorieTracker {  
    // ...  
  
    _displayCaloriesTotal() {  
        const totalCaloriesEl = document.getElementById('calories-total');  
        totalCaloriesEl.innerHTML = this._totalCalories;  
    }  
  
    _displayCaloriesLimit() {  
        const calorieLimitEl = document.getElementById('calories-limit');  
        calorieLimitEl.innerHTML = this._calorieLimit;  
    }  
  
    _displayCaloriesConsumed() {  
        const caloriesConsumedEl = document.getElementById('calories-consumed');  
  
        const consumed = this._meals.reduce(  
            (total, meal) => total + meal.calories,  
            0  
        );  
  
        caloriesConsumedEl.innerHTML = consumed;  
    }  
  
    _displayCaloriesBurned() {  
        const caloriesBurnedEl = document.getElementById('calories-burned');  
  
        const burned = this._workouts.reduce(  
            (total, workout) => total + workout.calories,  
            0  
        );  
  
        caloriesBurnedEl.innerHTML = burned;  
    }  
  
    _displayCaloriesRemaining() {  
        const caloriesRemainingEl = document.getElementById('calories-remaining');  
  
        const remaining = this._calorieLimit - this._totalCalories;  
  
        caloriesRemainingEl.innerHTML = remaining;  
    }  
}
```

```
}
```

So this is pretty simple. We are grabbing elements from the DOM and inserting data into them based on the data we have in the class. We are also using the `reduce()` method to get the total calories consumed and burned.

Now we need to call these methods in the constructor so that they run as soon as the class is instantiated.

```
class CalorieTracker {  
  constructor() {  
    this._calorieLimit = 2000;  
    this._totalCalories = 0;  
    this._meals = [];  
    this._workouts = [];  
  
    this._displayCaloriesLimit();  
    this._displayCaloriesTotal();  
    this._displayCaloriesConsumed();  
    this._displayCaloriesBurned();  
    this._displayCaloriesRemaining();  
  }  
}
```

Now we need to call these methods in the `addMeal()` and `addWorkout()` methods so that they run whenever a new meal or workout is added. Instead of calling them all in both methods, let's create a private method called `_render()` that will call all of these methods.

```
class CalorieTracker {  
  //...  
  
  addMeal(meal) {  
    this._meals.push(meal);  
    this._totalCalories += meal.calories;  
    this._render();  
  }  
  
  addWorkout(workout) {  
    this._workouts.push(workout);  
    this._totalCalories -= workout.calories;  
    this._render();  
  }  
  
  _render() {  
    this._displayCaloriesTotal();  
    this._displayCaloriesConsumed();  
    this._displayCaloriesBurned();  
    this._displayCaloriesRemaining();  
  }  
}
```

```
    }  
}
```

Now let's test it out. We can add a meal and a workout and see if the stats are updated.

```
const tracker = new CalorieTracker();  
  
const breakfast = new Meal('Breakfast', 400);  
tracker.addMeal(breakfast);  
  
const run = new Workout('Morning Run', 320);  
tracker.addWorkout(run);  
  
console.log(tracker._meals);  
console.log(tracker._workouts);  
console.log(tracker._totalCalories);
```

As you add meals and workouts, the stats should update. If you refresh the page, the stats should reset to their default values.

# Progress Bar & Max Calorie Alert

---

Before we move on to the `App` class, let's add a progress bar and a max calorie alert. The alert is going to turn the "Calories Remaining" box and the progress bar red by adding the `bg-danger` class.

Bootstrap progress bars work by setting the `width` to a percentage. We can calculate the percentage by dividing the calories consumed by the max calories and multiplying by 100.

Let's create a method in the `CalorieTracker` class called `_displayCaloriesProgress` and add the following code:

```
_displayCaloriesProgress() {  
  const progressEl = document.getElementById('calorie-progress');  
  const percentage = (this._totalCalories / this._calorieLimit) * 100;  
  const width = Math.min(percentage, 100);  
  progressEl.style.width = `${width}%`;  
}
```

We then want to add call this in the constructor and in the `_render()` method.

```
class CalorieTracker {  
  constructor() {  
    //...  
    this._displayCalorieProgress();  
  }  
}
```

```
class CalorieTracker {  
  _render() {  
    //...  
    this._displayCalorieProgress();  
  }  
}
```

Now let's add the alert. We want to add the `bg-danger` class to the "Calories Remaining" box and the progress bar if the calories remaining is 0 or less. We can put this in the `_displayCaloriesRemaining` method.

```
class CalorieTracker {  
  _displayCaloriesRemaining() {  
    const caloriesRemainingEl = document.getElementById('calories-remaining');  
    const progressEl = document.getElementById('calorie-progress');
```

```
const remaining = this._calorieLimit - this._totalCalories;
caloriesRemainingEl.innerHTML = remaining;
if (remaining <= 0) {
    caloriesRemainingEl.parentElement.classList.remove('bg-light');
    caloriesRemainingEl.parentElement.classList.add('bg-danger');
    progressEl.classList.add('bg-danger');
    progressEl.classList.remove('bg-success');
} else {
    caloriesRemainingEl.parentElement.classList.remove('bg-danger');
    caloriesRemainingEl.parentElement.classList.add('bg-light');
    progressEl.classList.remove('bg-danger');
    progressEl.classList.add('bg-success');
}
}
```

# App Class, New Meal & Workout

---

Now that we have our base `CalorieTracker` class, let's create the `App` class. The `App` class is going to be the main class that will control the entire application. It will be responsible for initializing the `CalorieTracker` class and it is where all of the event listeners will be added.

We are going to set the tracker instance as a property on the `App` class.

```
class App {  
  constructor() {  
    this._tracker = new CalorieTracker();  
  }  
}
```

In the final version of the app, we will have an event handler called `_newItem` that will be responsible for adding a new meal or workout to the tracker. For now, we will have two separate event handlers, `_newMeal` and `_newWorkout`. We will add the event listeners in the constructor.

We also need access to the current instance by using the `this` keyword. Since `this._newMeal` and `this._newWorkout` are being passed as callbacks, the `this` keyword will refer to the element that the event was called on. To get around this, we can use the `bind` method to bind the `this` keyword to the `App` class instance.

```
document  
  .getElementById('meal-form')  
  .addEventListener('submit', this._newMeal.bind(this, 'meal'));  
  
document  
  .getElementById('workout-form')  
  .addEventListener('submit', this._newWorkout.bind(this, 'workout'));
```

Let's start with the `_newMeal` event handler. We are going to get the values from the form and create a new `Meal` object. We will then add the meal to the tracker and render the tracker.

```
_newMeal(e) {  
  e.preventDefault();  
  const name = document.getElementById('meal-name');  
  const calories = document.getElementById('meal-calories');  
  
  if (name.value === '' || calories.value === '') {  
    alert('Please fill in all fields');  
    return;  
  }
```

```

    // Create a new meal
    const meal = new Meal(name.value, +calories.value);

    // Add the meal to the tracker
    this._tracker.addMeal(meal);
}

```

This will create a new meal and we should see the reflection in the calorie stats. We won't see the meal in the list yet because we haven't rendered that, but it is being added to the tracker.

I also want to clear the form inputs and close the Bootstrap collapse component when we submit, so let's do that.

```

_newMeal(e) {
e.preventDefault();
const name = document.getElementById('meal-name');
const calories = document.getElementById('meal-calories');

if (name.value === '' || calories.value === '') {
  alert('Please fill in all fields');
  return;
}

// Create a new meal
const meal = new Meal(name.value, +calories.value);

// Add the meal to the tracker
this._tracker.addMeal(meal);

// Clear the form
name.value = '';
calories.value = '';

// Collapse the form
const collapseMeal = document.getElementById('collapse-meal');
const bsCollapse = new bootstrap.Collapse(collapseMeal, {
  toggle: true,
});
}

```

Now to do this for the workout, is very similar. Let's create the `_newWorkout` event handler.

```

_newWorkout(e) {
e.preventDefault();
const name = document.getElementById('workout-name');
const calories = document.getElementById('workout-calories');

if (name.value === '' || calories.value === '') {

```

```
    alert('Please fill in all fields');
    return;
}

// Create a new workout
const workout = new Workout(name.value, +calories.value);

// Add the workout to the tracker
this._tracker.addWorkout(workout);

// Clear the form
name.value = '';
calories.value = '';

// Collapse the form
const collapseWorkout = document.getElementById('collapse-workout');
const bsCollapse = new bootstrap.Collapse(collapseWorkout, {
  toggle: true,
});
}
```

We have to instanciate the [App](#) class

```
const app = new App();
```

These two functions are very similar, so in the next lesson, we will refactor a bit and abide by the DRY principle.

# newItem() Refactor

---

So, the `_newMeal()` and `newWorkout()` methods in the last lesson were very similar. We are going to combine them into a single method called `_newItem()`.

First we need to change the event handlers. We are going to change the method name to `_newItem()` and pass in the type of item as a parameter.

```
document
  .getElementById('meal-form')
  .addEventListener('submit', this._newItem.bind(this, 'meal'));

document
  .getElementById('workout-form')
  .addEventListener('submit', this._newItem.bind(this, 'workout'));
```

Now, let's create the method, which will take in a type parameter.

```
_ newItem(type, e) {
  e.preventDefault();
  const name = document.getElementById(`#${type}-name`);
  const calories = document.getElementById(`#${type}-calories`);

  if (name.value === '' || calories.value === '') {
    alert('Please fill in all fields');
    return;
  }

  if (type === 'meal') {
    const meal = new Meal(name.value, +calories.value);
    this._tracker.addMeal(meal);
  }
  if (type === 'workout') {
    const workout = new Workout(name.value, +calories.value);
    this._tracker.addWorkout(workout);
  }

  name.value = '';
  calories.value = '';

  const collapseItem = document.getElementById(`collapse-${type}`);
  const bsCollapse = new bootstrap.Collapse(collapseItem, {
    toggle: true,
  });
}
```

As you can see, we just changed a lot of the hardcoded 'meal' and 'workout' strings to the type parameter. We also added a conditional to see which tracker method would be called.

# Display New Meal & Workout

---

So we are able to add a meal or workout via the forms and the calories are factored into the stats, but we still don't see the meal or workout. So we are going to create two methods, `_displayNewMeal()` and `_displayNewWorkout()`.

I would suggest removing any dummy items in the list from the HTML if you still have any there.

Let's start by calling them where they need to be called, which is in the public `addMeal()` and `addWorkout()` methods in the `CalorieTracker`.

```
addMeal(meal) {
  this._meals.push(meal);
  this._totalCalories += meal.calories;
  this._displayNewMeal(meal);
  this._render();
}

addWorkout(workout) {
  this._workouts.push(workout);
  this._totalCalories -= workout.calories;
  this._displayNewWorkout(workout);
  this._render();
}
```

Now, let's create the `_displayNewMeal()` method in the `CalorieTracker` class.

```
_displayNewMeal(meal) {
  const mealsEl = document.getElementById('meal-items');
  const mealEl = document.createElement('div');
  mealEl.classList.add('card', 'my-2');
  mealEl.setAttribute('data-id', meal.id);
  mealEl.innerHTML =
    `

#### ${meal.name}



${meal.calories}

`;
  mealsEl.appendChild(mealEl);
}
```

```
    mealsEl.appendChild(mealEl);
}
```

We just selected the element where we want to append the meal, created the element, added some classes and a `data-id` attribute, and then set the inner HTML. We are using the `data-id` attribute to get the ID of the meal when we want to delete it. We are also using the `delete` class to select the delete button when we want to delete the meal.

We will create the `_displayNewWorkout()` method in the same way.

```
_displayNewWorkout(workout) {
  const workoutsEl = document.getElementById('workout-items');
  const workoutEl = document.createElement('div');
  workoutEl.classList.add('card', 'my-2');
  workoutEl.setAttribute('data-id', workout.id);
  workoutEl.innerHTML = `
    <div class="card-body">
      <div class="d-flex align-items-center justify-content-between">
        <h4 class="mx-1">${workout.name}</h4>
        <div class="fs-1 bg-secondary text-white text-center rounded-2 px-2 px-sm-5">
          ${workout.calories}
        </div>
        <button class="delete btn btn-danger btn-sm mx-2">
          <i class="fa-solid fa-xmark"></i>
        </button>
      </div>
    </div>
  `;
  workoutsEl.appendChild(workoutEl);
}
```

Now when you add a meal or workout, you should see it in the list.

# Remove Meal & Workout

---

Now we want to be able to remove meals and workouts. We want to remove from the list, but we also want to account for the calories in the tracker. So we are going to start by adding event listeners in the [App](#) class.

```
document
  .getElementById('meal-items')
  .addEventListener('click', this._removeItem.bind(this, 'meal'));

document
  .getElementById('workout-items')
  .addEventListener('click', this._removeItem.bind(this, 'workout'));
````
```

As you can see, we are using event delegation and putting the event listener on the parent element. We are also passing in the type as a parameter. Now let's create the method.

```
```js
_removeItem(type, e) {
  if (
    e.target.classList.contains('delete') ||
    e.target.classList.contains('fa-xmark')
  ) {
    if (confirm('Are you sure?')) {
      const id = e.target.closest('.card').getAttribute('data-id');
      type === 'meal'
        ? this._tracker.removeMeal(id)
        : this._tracker.removeWorkout(id);
      const item = e.target.closest('.card');
      item.remove();
    }
  }
}
```

We are targeting the `delete` and `fa-xmark` classes. We are also using the `closest` method to get the parent element. We are then getting the `data-id` attribute and passing it to the `removeMeal` or `removeWorkout` method in the `CalorieTracker`. We are also removing the item from the DOM.

## Remove From Tracker

Now, let's create our public `removeMeal()` and `removeWorkout()` methods in the tracker.

```
removeMeal(id) {
  const index = this._meals.findIndex((meal) => meal.id === id);
  if (index !== -1) {
    const meal = this._meals[index];
    this._meals.splice(index, 1);
    this._totalCalories -= meal.calories;
    this._render();
  }
}

removeWorkout(id) {
  const index = this._workouts.findIndex((workout) => workout.id === id);
  if (index !== -1) {
    const workout = this._workouts[index];
    this._workouts.splice(index, 1);
    this._totalCalories += workout.calories;
    this._render();
  }
}
```

Now, when we click on the delete button, it will remove the item. We are also updating the total calories.

# Filter & Reset

---

In this lesson, we are going to add the functionality to filter items and reset the total calories as well as remove all items.

Let's start by adding an event listener for a `keyup` event on both the meal and workout filter inputs.

```
document
  .getElementById('filter-meals')
  .addEventListener('keyup', this._filterItems.bind(this, 'meal'));

document
  .getElementById('filter-workouts')
  .addEventListener('keyup', this._filterItems.bind(this, 'workout'));
````
```

We will create the `_filterItems` method in the `App` class.

```
```js
_filterItems(type, e) {
  const text = e.target.value.toLowerCase();
  document.querySelectorAll(`#${type}-items .card`).forEach((item) => {
    const name = item.firstChild.firstChild.textContent;
    if (name.toLowerCase().indexOf(text) != -1) {
      item.style.display = 'block';
    } else {
      item.style.display = 'none';
    }
  });
}
```

We are getting the text of the input. Then we are looping through all the items and checking if the text is in the name of the item. If it is, we display the item, otherwise we hide it. We don't need to use the tracker at all because it has nothing to do with manipulating calories or items.

## Adding Reset Functionality

Let's add an event listener for the reset button.

```
document
  .getElementById('reset')
  .addEventListener('click', this._reset.bind(this));
```

Create the `\\_reset` method in the `App` class.

```
_reset() {  
  if (confirm('Are you sure you want to reset everything?')) {  
    this._tracker.reset();  
    document.getElementById('meal-items').innerHTML = '';  
    document.getElementById('workout-items').innerHTML = '';  
    document.getElementById('filter-meals').value = '';  
    document.getElementById('filter-workouts').value = '';  
  }  
}
```

Anything to do with resetting calories and items will happen in the tracker, so we have a `_tracker.reset()` method. We are also clearing the items from the UI and resetting the filter inputs.

Let's create the tracker reset. Remember, this is a public method, so no underscore.

```
reset() {  
  this._totalCalories = 0;  
  this._meals = [];  
  this._workouts = [];  
  this._render();  
}
```

We are clearing the total calories, the meals and workouts arrays, and then we are rendering the tracker/stats.

# Set Calorie Limit

---

Before we start to get into localStorage, I want to make it so that when we submit the limit modal form, it changes the limit, which right now is hard-coded to 2000.

Let's add an event listener for the submit event on the limit form.

```
document
  .getElementById('limit-form')
  .addEventListener('submit', this._setLimit.bind(this));
```

Now, create the `_setLimit` method in the `App` class.

```
_setLimit(e) {
  e.preventDefault();
  const limit = document.getElementById('limit');

  if (limit.value === '') {
    alert('Please add a limit');
    return;
  }

  this._tracker.setLimit(+limit.value);
  limit.value = '';

  const modalEl = document.getElementById('limit-modal');
  const modal = bootstrap.Modal.getInstance(modalEl);
  modal.hide();
}
```

So, the actual setting of the calorie limit is done within the tracker. We are also clearing the input and hiding the modal.

let's create the tracker `setLimit` method. Remember, this is a public method, so no underscore.

```
setLimit(calorieLimit) {
  this._calorieLimit = calorieLimit;
  this._displayCalorieLimit();
  this._render();
}
```

We are setting the calorie limit and then calling the `displayCalorieLimit` and `render` methods.

Now whatever you set it to, the rest of the app will use that value. However, it will not stick if you refresh the page. Nothing will right now. So in the next few lessons, we will implement localStorage to save the data.

# Storage Class & Calorie Limit Persist

In the last lesson, we added the ability to set the calorie limit. However, if you refresh the page, it will go back to the default of 2000. In this lesson, we will implement localStorage to save the data.

Let's create a **Storage** class:

```
class Storage {}
```

This class will have all **static** methods. Meaning we do not have to instantiate the class. There is no need to because we never need more than one instance. So we can call the methods directly from the class (`Storage.methodName()`).

Let's add a method to get the calorie limit from localStorage:

```
class Storage {
    static getCalorieLimit(defaultLimit = 2000) {
        let calorieLimit;
        if (localStorage.getItem('calorieLimit') === null) {
            calorieLimit = defaultLimit;
        } else {
            calorieLimit = +localStorage.getItem('calorieLimit');
        }
        return calorieLimit;
    }
}
```

We are first checking **if** there is a `calorieLimit` in `localStorage`. If not, we are setting it to the `defaultLimit`. If there is, we are parsing it to an integer and returning it.

Now, we need a method to set the calorie limit in `localStorage`:

```
class Storage {
    static setCalorieLimit(calorieLimit) {
        localStorage.setItem('calorieLimit', calorieLimit);
    }
}
```

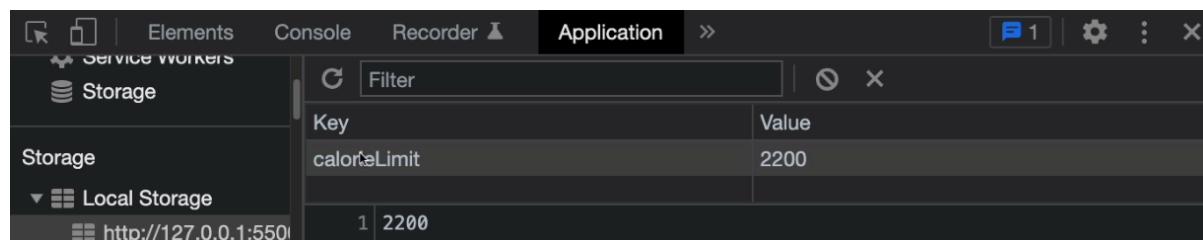
Let's change the default limit in the **CalorieTracker** class to use the **Storage** class:

```
class CalorieTracker {  
  constructor() {  
    this.calorieLimit = Storage.getCalorieLimit();  
  }  
}
```

When we set the limit in the `CalorieTracker` class, we need to update the `localStorage`:

```
setLimit(calorieLimit) {  
  this._calorieLimit = calorieLimit;  
  Storage.setCalorieLimit(calorieLimit); // Add this line  
  this._displayCalorieLimit();  
  this._render();  
}
```

Now, when you submit the limit form in the modal, it will update local storage and persist in the app. If you look in the `Application` tab in the devtools, and click on `Local Storage`, you should see the key/value pair.



The screenshot shows the Chrome DevTools Application tab with the Local Storage panel open. The sidebar on the left lists 'Storage' and 'Local Storage'. Under 'Local Storage', there is an entry for the URL 'http://127.0.0.1:5501'. The main area displays a table with one row:

Key	Value
calorieLimit	2200

# Persist Total Calories to Storage

---

We want the total calories to be stored in localStorage and we want that number to update when we add a meal or a workout.

Let's create a method to get the total calories from localStorage:

```
static getTotalCalories(defaultCalories = 0) {  
  let totalCalories;  
  if (localStorage.getItem('totalCalories') === null) {  
    totalCalories = defaultCalories;  
  } else {  
    totalCalories = +localStorage.getItem('totalCalories');  
  }  
  return totalCalories;  
}
```

As well as a method to update the calories

```
static updateCalories(calories) {  
  localStorage.setItem('totalCalories', calories);  
}
```

Now, let's update the `_totalCalories` property in the tracker to use the `getTotalCalories()` method:

```
class CalorieTracker {  
  constructor() {  
    this._totalCalories = Storage.getTotalCalories(0);  
    // ...  
  }  
}
```

When we add a meal or a workout, we need to update the total calories:

```
addMeal(meal) {  
  this._meals.push(meal);  
  this._totalCalories += meal.calories;  
  Storage.updateCalories(this._totalCalories); // Add this line  
  this._render();  
}
```

```
addWorkout(workout) {
  this._workouts.push(workout);
  this._totalCalories -= workout.calories;
  Storage.updateCalories(this._totalCalories); // Add this line
  this._render();
}
```

We also want to do it when we remove a meal or workout:

```
removeMeal(id) {
  const index = this._meals.findIndex((meal) => meal.id === id);
  if (index !== -1) {
    const meal = this._meals[index];
    this._meals.splice(index, 1);
    this._totalCalories -= meal.calories;
    Storage.updateCalories(this._totalCalories);
    this._render();
  }
}

removeWorkout(id) {
  const index = this._workouts.findIndex((workout) => workout.id === id);
  if (index !== -1) {
    const workout = this._workouts[index];
    this._workouts.splice(index, 1);
    this._totalCalories += workout.calories;
    Storage.updateCalories(this._totalCalories);
    this._render();
  }
}
```

Now, when we refresh the page, the total calories will be the same as before. However, we won't see the meals and workouts because we haven't implemented that in local storage yet.

So the calories are now being tracked and saved in local storage, but the calories consumed and burned are not being displayed as well as the meals and workouts themselves. This is because we have yet to save them in local storage, so that is what we are going to do now.

Let's create a `getMeals()` method in the `Storage` class:

```
class Storage {  
    // ...  
    static getMeals() {  
        let meals;  
        if (localStorage.getItem('meals') === null) {  
            meals = [];  
        } else {  
            meals = JSON.parse(localStorage.getItem('meals'));  
        }  
        return meals;  
    }  
}
```

We are just checking to see if it exists. If it does not, we return an empty array. If it does, we get it from local storage, parse the array and return the meals.

Let's create a method to store the meals in local storage:

```
static saveMeal(meal) {  
    const meals = Storage.getMeals();  
    meals.push(meal);  
    localStorage.setItem('meals', JSON.stringify(meals));  
}
```

We are getting the meals from local storage, pushing the new meal to the array and then setting the item in local storage.

We are going to change the `_meals()` property to store the array returned from the `getMeals()` method:

```
class CalorieTracker {  
    constructor() {  
        this._meals = Storage.getMeals();  
        // ...  
    }  
}
```

We will use the `saveMeal()` method in the `addMeal()` method:

```

addMeal(meal) {
  this._meals.push(meal);
  this._totalCalories += meal.calories;
  Storage.updateCalories(this._totalCalories);
  Storage.saveMeal(meal);
  this._displayNewMeal(meal);
  this._render();
}

```

So now we are storing meals in local storage, but we are not displaying them. Let's do that now by creating a `loadItems()` method in the `CalorieTracker` class:

```

class CalorieTracker {
  loadItems() {
    this._meals.forEach((meal) => this._displayNewMeal(meal));
  }
}

```

We are just looping through the meals and displaying them.

We will call the `loadItems()` method in the `App` constructor:

```

class App {
  constructor() {
    //...
    this._tracker.loadItems();
  }
}

```

Let's also clean up the `App` constructor by putting all of the event listeners into a single method:

```

class App {
  constructor() {
    // ...
    this._loadEventListeners();
  }

  _loadEventListeners() {
    document
      .getElementById('meal-form')
      .addEventListener('submit', this._newItem.bind(this, 'meal'));

    document
      .getElementById('workout-form')
      .addEventListener('submit', this._newItem.bind(this, 'workout'));
  }
}

```

```
document
  .getElementById('meal-items')
  .addEventListener('click', this._removeItem.bind(this, 'meal'));

document
  .getElementById('workout-items')
  .addEventListener('click', this._removeItem.bind(this, 'workout'));

document
  .getElementById('filter-meals')
  .addEventListener('keyup', this._filterItems.bind(this, 'meal'));

document
  .getElementById('filter-workouts')
  .addEventListener('keyup', this._filterItems.bind(this, 'workout'));

document
  .getElementById('reset')
  .addEventListener('click', this._reset.bind(this));

document
  .getElementById('limit-form')
  .addEventListener('submit', this._setLimit.bind(this));
}

}
```

# Save Workouts To Local Storage

---

Now we want to save our workouts. I want you to try this on your own. It is the same exact process we used with meals. The only difference is that we are going to use the `workouts` array instead of the `meals` array.

Let's continue...

We will add a `getWorkouts()` method to the `Storage` class:

```
static getWorkouts() {
    let workouts;
    if (localStorage.getItem('workouts') === null) {
        workouts = [];
    } else {
        workouts = JSON.parse(localStorage.getItem('workouts'));
    }
    return workouts;
}
```

As well as a `saveWorkout()` method:

```
static saveWorkout(workout) {
    const workouts = Storage.getWorkouts();
    workouts.push(workout);
    localStorage.setItem('workouts', JSON.stringify(workouts));
}
```

We will set the default for `_workouts` to `Storage.getWorkouts()`:

```
class CalorieTracker {
    constructor() {
        // ...
        this._workouts = Storage.getWorkouts(); // Add this line
    }
}
```

Now call the `saveWorkout()` method in the `addWorkout()` method:

```
addWorkout(workout) {
    this._workouts.push(workout);
    this._totalCalories -= workout.calories;
    Storage.updateCalories(this._totalCalories);
```

```
    Storage.saveWorkout(workout); // Add this line
    this._displayNewWorkout(workout);
    this._render();
}
```

Finally, to display them on the page, we will add to the `loadItems()` method in the `CalorieTracker` class:

```
loadItems() {
  this._meals.forEach(meal => this._displayNewMeal(meal));
  this._workouts.forEach(workout => this._displayNewWorkout(workout));
}
```

# Remove Meals & Workouts From Local Storage

---

Let's go ahead and add a method called `removeMeal()` to the `CalorieTracker` class:

```
static removeMeal(id) {
  const meals = Storage.getMeals();
  meals.forEach((meal, index) => {
    if (meal.id === id) {
      meals.splice(index, 1);
    }
  });
  localStorage.setItem('meals', JSON.stringify(meals));
}
```

We simply loop through the meals and if the meal id matches the id we pass in, we remove it from the array. Then we save the array to local storage.

Let's also create `removeWorkout()`

```
static removeWorkout(id) {
  const workouts = Storage.getWorkouts();
  workouts.forEach((workout, index) => {
    if (workout.id === id) {
      workouts.splice(index, 1);
    }
  });
  localStorage.setItem('workouts', JSON.stringify(workouts));
}
```

Now we will call these methods from the tracker. In the `removeMeal()` method of the `CalorieTracker` class, we will call the `removeMeal()` method of the `Storage` class:

```
removeMeal(id) {
  this._meals.forEach((meal, index) => {
    if (meal.id === id) {
      this._totalCalories -= meal.calories;
      this._meals.splice(index, 1);
    }
  });
  Storage.updateCalories(this._totalCalories);
  Storage.removeMeal(id); // Add this line
  this._render();
}
```

We will do the same with the `removeWorkout()` method:

```
removeWorkout(id) {
  this._workouts.forEach((workout, index) => {
    if (workout.id === id) {
      this._totalCalories -= workout.calories;
      this._workouts.splice(index, 1);
    }
  });
  Storage.updateCalories(this._totalCalories);
  Storage.removeWorkout(id); // Add this line
  this._render();
}
```

Now, when you remove a meal or workout, it will stay gone when you reload the page because we removed them from localstorage.

# Clear Storage Items

---

Now that we are saving our meals and workouts to local storage, we need a way to clear them. We already have the functionality of clearing them from the UI, so we just need to clear them from local storage.

It is up to you if you want to also clear the **calorie limit**. You may want to keep that but reset the total calories and the item arrays.

Let's create a `clearAll()` method in the `Storage` class:

If you wanted to clear everything including the limit, you could simply use the `.clear()` method on the `localStorage` object:

```
static clearAll() {  
  localStorage.clear();  
}
```

If you want to keep the limit, you can just remove the other 3 individually:

```
static clearAll() {  
  localStorage.removeItem('meals');  
  localStorage.removeItem('workouts');  
  localStorage.removeItem('totalCalories');  
}
```

Now in the `CalorieTracker()` class, in the `reset()` method, call the `clearAll()` method in the storage class:

```
reset() {  
  this._totalCalories = 0;  
  this._meals = [];  
  this._workouts = [];  
  Storage.clearAll();  
  this._render();  
}
```

# What Are Modules

---

So what are modules? Modules are just JavaScript files that we can import into other JavaScript files. We can then use the code that is in the module in the file that we import it into. Now an import is not like a PHP include for instance, where we simply just have everything accessible to us. We need to export what we want from the file. This could be a function, a class or even just a variable value or an object. Modules can be our own files and code or they can be part of a package that we install using [NPM](#) or the [Node Package Manager](#). If you want to use NPM modules in the front-end, you need to use a module bundler like Webpack, which we're going to get into later. For now, let's just focus on how to create our own modules using our own files.

The reason that modules are so important is because they allow us to break our code up into different files. This makes our code more organized and easier to maintain. Using our [Tracalorie](#) app as an example, we had a single `app.js` file with 5 different classes. That's not really ideal. Imagine if we had 100 classes. It would be more organized to put each class into its own module. We can then import the modules into the `app.js` file. This way, we can keep our code organized and maintainable. Like I said, later on, we'll refactor our Tracalorie app to use modules.

Modules are obviously **modular**, so we can reuse them where we want. You may have a module with some utility classes to add commas to a number or something. Where ever you want to use that, you can import it and use it.

Using a module bundler also means that we can use NPM packages. So you have access to over a million 3rd-party modules to enhance your application. You can't use NPM with the way that we've been doing things.

Also, when you use something like Webpack or Parcel, you have access to tools to optimize your project. Whether it's minifying your code or using Sass or a custom dev server, you can create a very customized environment.

## Types Of Modules

There are a few different types of modules when it comes to JavaScript. The two main types are [CommonJS Modules](#) and [ES Modules](#) or [ES6 Modules](#). [CommonJS Modules](#) are the modules that are usually used in Node.js. We're going to talk about Node more in the next video. When you use a front-end framework like React, Angular or Vue, you'll be using [ES Modules](#). I'm going to show you both types.

ES Modules and CommonJS modules have a different syntax, but the idea is the same. We export what we want from a specific module/file and import it into another. We can export variables, functions, classes, etc.

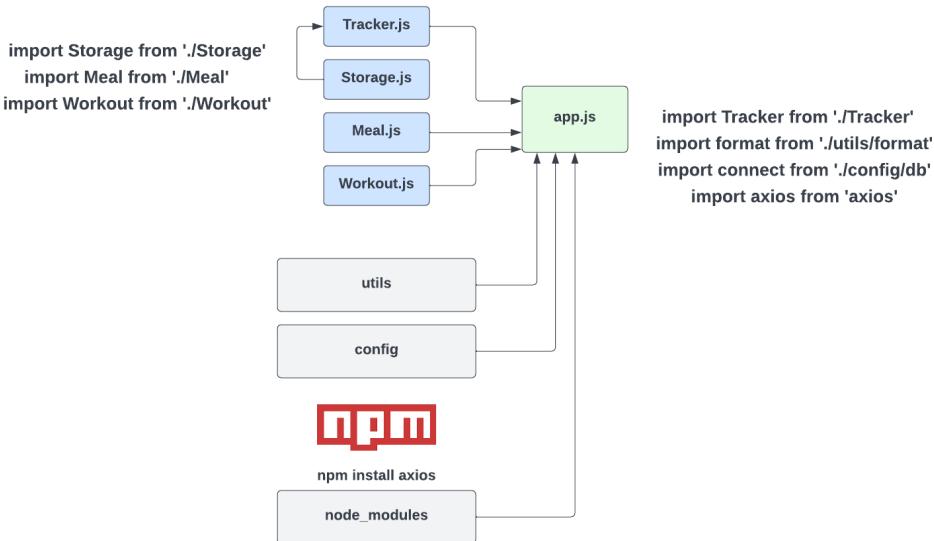
## Modules & The Browser

When it comes to using modules in the browser, there is support for [ESM](#) or [ES Modules](#) in newer browsers, however they're not supported in older browsers. So in order to use them, you will usually use a module bundler like [Webpack](#) or [Parcel](#) to bundle our modules into a single file that can be used in the

browser. I will show you how to use ES Modules directly in the browser, but it is recommended that you use some kind of module bundler.

## Visual Example

Let's look at an example of what we could do for the Tracalorie project if we broke it up into modules:



We could separate out the tracker, storage, meal and workout classes into their own files. We would export the whole class, that way we could use whatever methods we needed from that class. Since we use the tracker, the meal and the workout class in the main App class, we would import it into app.js. The Storage class is used in the tracker, so we would import that into `Tracker.js`. The syntax we're looking at here is the [ES6 Module](#) syntax. CommonJS is a bit different. I'm going to show you both.

I added some other files and folders just as an example. We may have a config folder and a utile folder with more code we want to import. We could also install 3rd-party modules/packages using [NPM](#). For instance, if we installed the `axios` package, which is an HTTP client, we could import it where we needed it.

In order to teach you about the [CommonJS](#) modules as well as [NPM](#) modules, we need to talk about and install Node.js. So we'll do that in the next lesson.

# What Is Node.js?

---

Alright, so this is not a Node.js course, so we won't be doing a deep dive into Node. I do want you to be familiar with it though, because it's a very popular environment for JavaScript and even as a front-end developer, you'll be working with **NPM** and to use NPM, you need to install Node.js.

So Node.js is a runtime environment. Up to this point, we've been executing our JavaScript code in the browser. Node.js is simply another environment to execute JavaScript code. It allows us to write server-side code with JavaScript. Just like other languages, such as Python, Ruby and Java. Server-side code is code that can interact with databases and the filesystem, etc.

Node is built on Chrome's V8 JavaScript engine. It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. The way Node works is very similar to how the browser works. It uses an event loop, callbacks, promises, `async/await`, etc. It's essentially the browser environment minus the `window` object. Node does have a global object, it's just not called `window`, it's called `global`. In Node, we don't have a document object model. So it's not used for creating interfaces like we do in the browser. It's used for creating back-end applications and APIs. It's also used for creating command line tools among other things.

## Installing Node.js

Installing Node is extremely easy. Just go to the [Node.js website](#) and download the latest version. Once it's installed, you can check the version by opening up your terminal/command line and typing `node -v`. This should return the version number. You can also check the version of `npm` by typing `npm -v`. This should also return the version number.

Let's create a folder to work in and an `app.js` file

```
mkdir node-playground
cd node-playground
touch app.js
code app.js
```

## Running a .js file

Add `console.log('Hello World');` to the `app.js` file.

There is no browser in this environment, so `console.log()` will output to the terminal.

Now let's run this file in the terminal. We can do this by typing `node app.js`. We can also leave off the `.js` extension and just do `node app`. You should see `Hello World` logged to the terminal.

Instead of just doing a hello world, let's fetch some data.

The fetch API actually was not part of Node.js up until recently. We had to use an NPM package like `node-fetch` or `axios` if we wanted to make requests.

```
async function getUser() {  
  const response = await fetch('https://api.github.com/users/bradtraversy');  
  const data = await response.json();  
  console.log(data);  
}  
getUser();
```

Now let's run this file in the terminal with `node app.js`.

You should see the data logged to the console. Now let's run this file in the terminal.

# CommonJS Modules

---

Let's look at the first type of module syntax called **CommonJS**. This is an older syntax that is used in Node.js. You won't use this much on the front-end, but if you plan on being a full-stack or back-end developer, you'll need to know it.

Let's create an **app.js** file and a **utils.js** file. Let's say that the app file is the main entry point and utils is just some extra utility functions that we want to be able to use in multiple files.

In **utils.js**, I am going to create a function that will capitalize the first letter of every word.

```
function capitalizeWords(str) {
  return str
    .toLowerCase()
    .split(' ')
    .map((word) => word[0].toUpperCase() + word.substr(1))
    .join(' ');
}
```

I want to be able to import this into the **app.js** file and use it. In order to do that, we need to export it. We do that by using the **module.exports** object. We can then set it equal to the function that we want to export.

```
module.exports = capitalizeWords;
```

Now, I can import it into the **app.js** file. In order to do that, we use the **require()** function. We pass in the path to the file that we want to import. We can then store it in a variable and use it.

```
const capitalizeWords = require('./utils');

console.log(capitalizeFirst('hello world')); // Hello World
```

As you can see, we can import the function and use it in the **app.js** file.

## Exporting Multiple functions

In many cases, you will want to export more than one thing from a file. Let's create another function in **utils.js**.

```
function makeMoney(amount) {
  return ` $$ {amount} `;
```

To export both functions, we can use the `module.exports` object again. We can set it equal to an object with the functions as properties.

```
module.exports = {  
    capitalizeWords,  
    makeMoney,  
};
```

Now when we import it in the `app.js` file, we can destructure it and get both functions.

```
const { capitalizeWords, makeMoney } = require('./utils');  
  
console.log(capitalizeWords('hello world')) // Hello World  
console.log(makeMoney(100)) // $100
```

## Exporting Classes

Functions are not the only thing that we can import/export. Let's create a file named `Person.js` and create a class in it.

```
class Person {  
    constructor(name, age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    greet() {  
        console.log(`Hello, my name is ${this.name} and I am ${this.age}`);  
    }  
}
```

Now, let's export it

```
module.exports = Person;
```

Now we can use it in the `app.js` file.

```
const Person = require('./Person');  
  
const person1 = new Person('John', 30);  
person1.greet(); // Hello, my name is John and I am 30
```

Remember, this is only for Node.js. You won't be able to use this on the front-end, at least without tooling.

# NPM Modules

---

NPM stands for **Node Package Manager**. It's a package manager for JavaScript. It's used to install 3rd-party packages and modules that we can use in our projects. There are over a million packages that you can install in a single command. Packages are hosted at [npmjs.com](https://npmjs.com).

Let's create another **app.js** file. I'm going to install and use **Axios**, which is a 3rd-party HTTP client that is similar to the Fetch API, but even more powerful.

When you create a Node.js app, the first thing that you usually do is run **npm init**. This will create a **package.json** file. This file is used to store information about your project. It's also used to store information about the packages that you install. It's kind of like a manifest file. It's also used to store scripts that you can run from the command line. We'll talk more about that later. For now, let's just run **npm init** and accept all of the defaults.

```
npm init
```

You can also run **npm init -y** to accept all of the defaults without having to answer any questions.

In this file, you will see some standard information about the project. You will also see a **dependencies** object. This is where the information about the packages that we install will be stored. Let's install **axios**.

```
npm install axios
```

Now, if you look at your **package.json** file, you will see **axios** listed in your projects dependencies. You will also see a new folder called **node\_modules**. This is where the packages that we install will be stored. If you look in it you will see a bunch of folders because our dependencies/packages use other dependencies. So this file gets VERY large. It is important that you do not commit this folder to version control. You should add it to your **.gitignore** file. Let's do that now.

```
echo "node_modules" >> .gitignore
```

Now, when we commit our code, the **node\_modules** folder will not be included.

Let's go to our **app.js** and import axios

```
import axios from 'axios';
```

Now we can use it to fetch some data from an API. Let's use the **JSON Placeholder** API to fetch a single post.

```
async function getPost() {
  const res = await axios.get('https://jsonplaceholder.typicode.com/posts/1');
  console.log(res.data);
}
getPost();
```

We are able to use `axios` because we installed and imported it.

Let's install one more package called `Lodash`. This is a utility library that has a lot of useful methods. Let's install it. We can use `i` as a shorthand for `install`.

## Global Packages

You can also use the `-g` option to install packages globally. This means it's not installed in your local project in the `node_modules` folder. It's installed globally on your system and you can use it wherever you want.

An example would be `create-react-app`. This is a utility that you can run to generate a new React boilerplate. You could install this globally like this:

```
npm install -g create-react-app
```

You could then run it from anywhere.

## Dev Dependencies

Sometimes, you may use packages that are strictly for your development environment. An example would be `nodemon`, which monitors your application so you don't have to keep running it. Nodemon would not be needed in your production environment, so you could install it as a dev dependency with the `-D` option

```
npm install -D nodemon
```

Then, in your `package.json` file, it would be put in a `devDependencies` object.

## NPM Scripts

Let's say that we want to run our `app.js` file. We could run `node app.js`, but that's kind of long. We can create a script in our `package.json` file to make this easier. Let's add a `start` script.

```
{
  "name": "npm-modules",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
```

```
"scripts": {  
  "start": "node app.js"  
},  
"keywords": [],  
"author": "",  
"license": "ISC",  
"dependencies": {  
  "axios": "^0.21.1",  
  "lodash": "^4.17.21",  
  "uuid": "^8.3.2"  
}  
}
```

Now we can run `npm start` to run our `app.js` file.

Note that if you use something other than `start`, you will have to run `npm run <script name>`. Let's change `start` to `dev`. Let's add a `dev` script that will run our app with Nodemon

```
{  
  "name": "npm-modules",  
  "version": "1.0.0",  
  "description": "",  
  "main": "app.js",  
  "scripts": {  
    "start": "node app.js",  
    "dev": "nodemon app.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "axios": "^0.21.1",  
    "lodash": "^4.17.21",  
    "uuid": "^8.3.2"  
  }  
}
```

To run Nodemon, we would do the following

```
npm run dev
```

So those are the basics of NPM. You'll be using it quite a bit as a JavaScript developer.

# ES Modules

---

When it comes to the front-end and the browser, things can be a bit more complicated. You're not going to be using CommonJS modules in the browser. You technically can if you use a module bundler, but ES Modules are much more popular and more modern.

## Browser Compatibility

Modern browsers support ES Modules, however older browsers do not. This is why it is suggested that you use a module bundler like Webpack to bundle our modules into a single file that can be used in the browser. We'll talk more about module bundlers in the next video. Using something like Webpack or Parcel has other benefits as well. You can install and use [NPM](#) modules. You can install plugins to do all kinds of things to optimize your code. They typically have some kind of local web server as well. Although, that's not really a big deal, because we have [Live Server](#) installed in VS Code.

In this lesson, we are not going to use any module bundlers. We're just going to use ES Modules in Chrome. The import/export syntax is the same whether you use a module bundler or not.

## The `type="module"` Module Attribute

Create an `index.html` file and an `app.js` file to start with. Since we are going to be using modules in the browser, when you add the `<script>` tag to the HTML file, you need to add the `type="module"` attribute. This tells the browser that this is an ES Module. You can also add the `defer` attribute, which will make sure that the script is loaded after the HTML is loaded. This is just a good practice to get into.

```
<!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" />
    <script src="app.js" type="module" defer></script>
    <title>ES Modules</title>
  </head>
  <body>
    <h1>ES Modules</h1>
  </body>
</html>
```

## Creating Modules

Let's do the same thing that we did in the [CommonJS](#) lesson. Although, this time I'm going to create a folder called `modules` and put my files in there. Let's start with our `modules/utils.js` file.

```

function capitalizeWords(str) {
  return str
    .toLowerCase()
    .split(' ')
    .map((word) => word[0].toUpperCase() + word.substr(1))
    .join(' ');
}

function makeMoney(amount) {
  return ` $$ ${amount} `;
}

export { capitalizeWords, makeMoney };

```

As you can see, to export, we use the `export` keyword. We can export multiple things by separating them with a comma. So remember, with [CommonJS](#), we use `module.exports = {}`, with ES Modules, we use `export {}`.

Now, let's use it in the `app.js` file.

```

import { capitalizeWords, makeMoney } from './modules/utils.js';

console.log(capitalizeWords('hello world'));
console.log(makeMoney(100));

```

So instead of using `require()` like we did with [CommonJS](#), we use the `import` keyword. Since we are exporting and importing multiple things, we use curly braces and commas. When we do not use a bundler, we need to include the file extension on the import.

Now, let's create a `Person.js` file and export a single class.

```

class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age}`);
  }
}

export default Person;

```

Notice we used `export default` instead of `export`. This is because we are only exporting one thing. We can only use `export default` once per file. If we try to use it more than once, we will get an error. You

can use as many `export` statements as you want.

Let's import `Person`

```
import Person from './modules/Person.js';

const person = new Person('Mark', 29);
person.greet();
```

Notice we didn't use curly braces. That's because we exported as default.

So that is the [ES Module syntax](#). It's a bit nicer than [CommonJS](#). You can also use [Node.js](#) with ES Modules, there is just a little bit of setup that goes into it. In the next video, I will show you how to setup the webpack module bundler.

# Webpack Module Bundler

So now that you understand the concept of modules, let's talk about module bundlers. A module bundler is a tool that takes all of your modules and bundles them into one or more files that can be used in the browser. There are a few popular module bundlers out there, but in this video, we'll be focusing on [Webpack](#). There are some others such as [Vite](#), [Parcel](#) and [Snowpack](#). I'll be honest and say that Webpack is probably the most difficult of the bunch, but it's the one that's been around forever, so you'll probably run into it at some point. Also, if you learn Webpack now, learning the others will be simple.

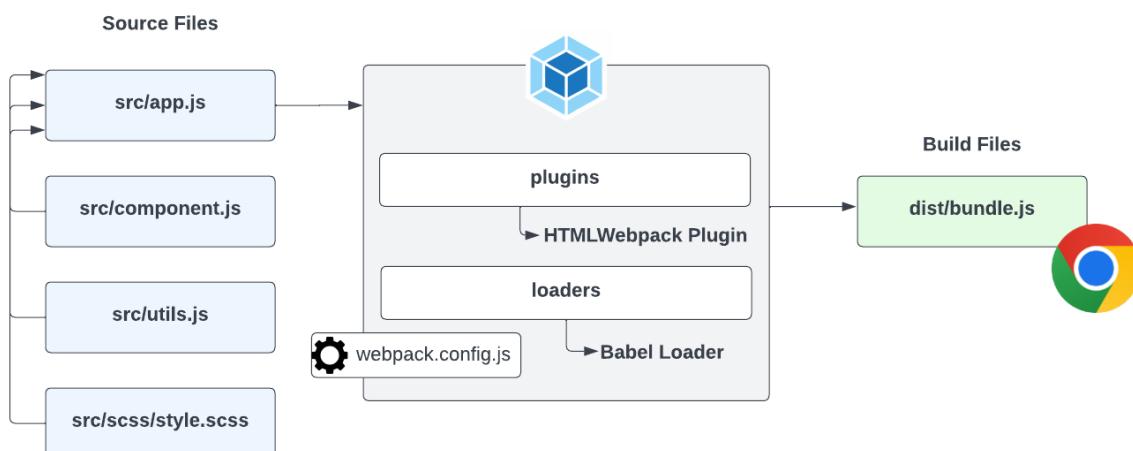
Webpack is used by a lot of companies. It's very popular and has a lot of features. It's also very flexible and can be configured to do a lot of different things. The next few videos will all have to do with creating a Webpack project from scratch. We'll start with the basics and then add more and more features as we go.

I can almost guarantee that you will be a bit overwhelmed if you've never used a module bundler. It takes some time to get used to them, but the good news is, once you create this boilerplate, you can use it for pretty much any project that you create. So, it's worth it to spend some time learning about it. The code that we write that pertains to the module bundler will only be a small part of your application and is reusable across applications.

## How It Works

The way that we've been doing things is pretty simple. We create our JavaScript code and run it in the browser. However, when you get into more advanced JavaScript and start building bigger applications, you're probably going to use some kind of build tool such as Webpack. You may get into a framework like React, but this is the same process that goes on under the hood with front-end frameworks.

The way that Webpack and other bundlers work is that they take all of your JavaScript files and bundle them into one or more files that can be used in the browser. This allows you to create all kinds of files and modules, use NPM packages, import CSS files and other assets and much more. As you can see from this diagram. We write our code in the source or [src](#) folder, then we run it through Webpack which bundles it into one or more files that can be used in the browser.



The files in your `dist` or `build` folder are the files that you would use in production. You would upload those files to your server. The files in your `src` folder are the files that you would use during development. So you need to get used of the idea that you have `development` or `source` files and `production` or `build` files.

There's a few parts of Webpack that I want to mention here and we'll see these in action very soon.

## Webpack Config

First, Webpack is configured using a file called `webpack.config.js`. This file is a JavaScript file that exports an object. The object contains all of the configuration for Webpack. You can specify your input and output files, project mode, as well as any loaders and plugins, which I'll talk about now.

## Loaders

Webpack has something called `loaders`. Loaders are used to process different types of files and convert them into modules that can be used in your application. For instance, if you want to import a CSS file into your JavaScript, you would use a CSS loader. If you want to import an image, you would use an image loader. There are loaders for pretty much anything you can think of. You can even create your own loaders if you want. The Babel loader, which I have as an example here, is commonly used. Babel is a transpiler that will take your modern JavaScript code and transpile it to older JavaScript that older browsers can understand. We'll look at Babel later.

## Plugins

Webpack also has something called plugins. Plugins are used to extend the functionality of Webpack. For instance, if you want to minify your JavaScript, you would use a minification plugin. If you want to extract your CSS into a separate file, you would use a CSS extraction plugin. We'll be using the `HTMLWebpackPlugin` plugin to automatically generate our HTML production files and the `WebpackDevServer` plugin to give us a nice auto reload dev server to work with.

Alright, in the next lesson, we're going to create a basic Webpack setup.

# Webpack Setup

---

Alright, now we are going to setup Webpack. Like I said, your configuration and what your tooling does can be as simple or advanced as you want. I'm going to try and keep it pretty simple. What I want to do is create a `webpack-starter` boilerplate that you can re-use in future applications. We're also going to use it to refactor the `Tracalorie` project. This is the boilerplate that I use. I will include the finished project in this lesson. I have a very similar project at <https://github.com/bradtraversy/webpack-starter>. It does have a few extra bells and whistles, but it is very similar and used in the same way.

Let's start off by creating a folder called `webpack-starter`. Open it with your text editor and run `npm init` to create a `package.json` file.

## Create folders

We are going to create a few folders. Create a `src` folder and a `dist` folder. The `src` folder is where we are going to put our source code. The `dist` folder is where we are going to put our bundled code. We are going to use the `dist` folder for production.

Create an `index.html` file in the `dist` folder. We'll put some boilerplate HTML in there for now. We are going to create a `script` tag and point it to a file called `main.js`. This file does not exist yet. This is the output file that Webpack is going to create for us.

Now, create a file called `index.js` in the `src` folder. This will be the entry point for our application. Let's also create a file called `message.js` in the `src` folder. This is going to be a module that we are going to import into the `index.js` file.

In the `message.js` file, let's export an object with an id and a text field. You could also export functions, classes, etc.

```
export default {
  id: 1,
  text: 'Hello World',
};
```

We could also give it a variable name and export the variable

```
const message = {
  id: 1,
  text: 'Hello World',
};

export default message;
```

Bring it into the `index.js` file and log the text message.

```
import message from './message.js';
console.log(message);
```

Now, this is not code that would work on its own in the browser. We need to bundle it with Webpack. We are going to use Webpack to bundle our code into a single file.

## Install Webpack

Now we are going to install Webpack. Run `npm install -D webpack webpack-cli`. We use the `-D` to save as a `development dependency`. Meaning this is a dev tool and it won't be used in production. This will get put in the `devDependencies` section in the package.json file.

## .gitignore

Let's create a `.gitignore` file. We don't want to commit the `node_modules` folder to GitHub. We also don't want to commit the `dist` folder. We are going to create that folder when we build our application. We are going to add the `dist` folder to the `.gitignore` file.

```
node_modules
dist
```

## Create Build Script

Let's open up the `package.json` file and create a build script. What I mean is a command that we can run to take our source code and bundle it into our production files. We don't have a config file yet, so I am just going to add the mode flag and set it to production.

```
"scripts": {
  "build": "webpack --mode production"
}
```

Now, let's run `npm run build`. This will run the build script. We should see a `main.js` file show up in the `dist` folder. If we open up the `index.html` file in the browser, we should see `Hello World` in the console. I'll just use `live-server` for now, but I will show you a webpack dev server that we can use in a bit.

## Create webpack.config.js

Now we are going to create a `webpack.config.js` file. This is where we are going to put our Webpack configuration. We use the `CommonJS` module syntax in this particular file. We are going to export `index`, an object with our configuration. For now, let's add an entry and an output. I'm going to change the output

file from `main.js` to `bundle.js`. We will also put the `mode` in here and take it out of the `package.json` file. We will also set it to `development` for now.

```
const path = require('path');

module.exports = {
  mode: 'development',
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
};
```

We used the `path` module to get us the absolute path. It's a built-in module in Node. We are going to use the `__dirname` variable to get us the current directory. We are going to use the `resolve` method to resolve the path.

Now let's change the script name in our HTML from `main.js` to `bundle.js`. Now, let's run `npm run build`. We should see a `bundle.js` file show up in the `dist` folder. If we open up the `index.html` file in the browser, we should see `Hello World` in the console. You can delete the `main.js` file now.

Notice we see a lot more stuff in the output file. This is because we are in `development` mode. It added some extra code to help us debug. You can change it to production when you are ready to deploy. The result is the same though.

## NPM Modules

We are now able to build much more advanced applications just being able to import files. We can also use **NPM** modules for our frontend apps.

Let's install a package called `UUID`, which will generate unique IDs for us. Run `npm install uuid`. Now, let's import it into the `message.js` file and use it for the id.

```
import { v4 as uuidv4 } from 'uuid';
```

```
export default {
  id: uuidv4(),
  text: 'Hello World',
};
```

Now, build again with `npm run build`. You should see a `bundle.js` file in the `dist` folder. If you open up the `index.html` file in the browser, you should see a different ID in the console everytime you refresh the page. We are now using the `UUID` package in our frontend app.

I'm going to remove `UUID`, because I don't want it in the boilerplate. I just wanted to show you we can use NPM modules.

```
npm remove uuid
```

```
// message.js
export default {
  id: 1,
  text: 'Hello World',
};
```

Make sure that you run `npm run build` again after you remove the package. Right now, you have to do this for every single change, but I will show you how to set up a dev server that will watch your files soon.

So, now that you know how to import files and use NPM modules, then build your production files with Webpack, let's move on to the next lesson and talk about `loaders`, specifically the `css` and `style` loaders.

# CSS & Style Loaders

In webpack, **loaders** are modules that transform the source code of an application as it's being bundled. They allow you to pre-process files as you include them in your application, and provide a means to transform code written in one language (such as TypeScript or CSS) into JavaScript that the browser can understand. You can also use loaders to use frameworks like React or Vue in your application.

## CSS & Style Loaders

We are going to use a loader to transform our CSS files and let us import them directly into our JavaScript. Let's install the **style-loader** and the **css-loader**. Run `npm install -D style-loader css-loader`.

After installing loaders, you need to configure them in the config file. Loaders are specified in the **module.rules** field of the config file. Each rule consists of a regular expression that determines which files the loader should be applied to, and an array of loaders to use.

```
module.exports = {
  mode: 'development',
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
    ],
  },
};
```

Here, we are saying that any file that ends with `.css` should be processed by the **style-loader** and the **css-loader**. The **style-loader** will add the CSS to the DOM by injecting a `<style>` tag, and the **css-loader** will interpret `@import` and `url()` like `import/require()` and will resolve them.

Now let's create a file at `src/css/style.css` and make the background of the page purple.

```
body {
  background: purple;
  color: white;
}
```

Now, we should be able to simply import the CSS file into our JavaScript file. Let's import it into the `index.js` file.

```
import './css/style.css';
```

Now, let's build again with `npm run build`. You should see a `bundle.js` file in the `dist` folder. If you open up the `index.html` file in the browser, you should see a purple background. We are now using the CSS loader to import our CSS file into our JavaScript file.

If you look at the `bundle.js` file, and search for `purple`, you will see that the CSS is now in the JavaScript file. This is because the `style-loader` injected the CSS into the DOM by injecting a `<style>` tag.

# HTMLWebpackPlugin

---

So, the way that we have things right now, it's ok, but it's a very basic setup. We created the `index.html` file and we added a script tag to it. The issue with this is that we have this static HTML file, so if I were to share this code, I have to include the `dist` folder. So, what we're going to do is we're going to use the `HTMLWebpackPlugin` to generate the `index.html` file for us. This means that we don't have to worry about creating the `dist` folder. So, let's go ahead and install it.

```
npm i -D html-webpack-plugin
```

Then, we're going to import it into our `webpack.config.js` file.

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
```

Then, we're going to add it to the plugins array.

```
plugins: [
  new HtmlWebpackPlugin({
    title: 'Webpack App'
    filename: 'index.html'
  })
]
```

Now, we should be able to delete the whole `dist` folder. It will be regenerated on build.

```
rm -rf dist
```

```
npm run build
```

Now the `dist` folder will be created with both the `bundle.js` and `index.html` files. If we open up the `index.html` file in the browser, we should see `Hello World` in the console. We are now using the `HTMLWebpackPlugin` to generate the `index.html` file for us.

The issue that we have now is if I add anything to the html file, such as an `<h1>`, it will go away when we build again. We fix this by adding a template. We add the template file to the `src` folder and then we add the template property to the `HTMLWebpackPlugin` and set it to the template file.

```
plugins: [
  new HtmlWebpackPlugin({
    title: 'Webpack App'
    filename: 'index.html'
    template: './src/index.html'
  })
]
```

I'm using `index.html` as the name, but you could call it anything that you want (eg. `template.html`). Whatever we put in `src/index.html` will be in the `dist/index.html` file. So, let's go ahead and add some basic HTML and add an `<h1>` tag to the `src/index.html` file. We can also use values from the `webpack.config.js` file in the template. So, let's go ahead and add the title to the `<title>` tag and the `<h1>` tag.

```
<!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><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <h1><%= htmlWebpackPlugin.options.title %></h1>
  </body>
</html>
```

```
npm run build
```

Now, take a look at `dist/index.html`. You should see the title and the `<h1>` tag that we added to the `src/index.html` file. So now, we do not have to create a `dist` folder to share our code. We simply run `npm run build` and it creates the `dist` folder.

In the next lesson, we're going to install the `webpack-dev-server` to make development easier.

# Webpack DevServer Plugin

---

It gets quite annoying to have to run `npm run build` after every little change. There is a plugin that we can use that will watch our files and rebuild them for us. Let's install it.

```
npm install -D webpack-dev-server
```

We now want to create a new NPM script to run the server. Let's open up the `package.json` file and add a new script.

```
"scripts": {  
  "build": "webpack",  
  "start": "webpack serve"  
}
```

## Add config

Before we run it, I just want to add some values to the config file. Right above the `modules` object, add the following:

```
devServer: {  
  static: {  
    directory: path.resolve(__dirname, 'dist'),  
  },  
  port: 3000,  
  open: true,  
  hot: true,  
  compress: true,  
  historyApiFallback: true,  
},  
module: {  
  //..  
}
```

So here, we are telling the dev server to serve the files from the `dist` folder. We are also telling it to open the browser on start and to use hot module replacement. We are also telling it to compress the files and to fallback to the `index.html` file if it can't find the file.

Now, let's run `npm run dev` and it should open up on `http://localhost:3000`. If we make a change to the `index.js` file, it should rebuild the application and refresh the browser for us.

If I make a change, like change the message to 'Hello World from Webpack', it should rebuild the application and refresh the browser for us.



# Babel Loader

---

The last thing that I want to do is setup [Babel](#), which is a JavaScript transpiler that will take any modern JavaScript that we write and transpile it into older code that older browsers can understand. These days, I guess it is not completely mandatory, but it's still a good idea to make our code more backwards compatible.

We can install Babel and the Babel loader with the following command:

```
npm install -D babel-loader @babel/core @babel/preset-env
```

Then in the Webpack config, we need to add a rule to the module object. We also need to add a new property to the config object called [resolve](#). This will tell Webpack what file extensions we want it to resolve. We are going to add [.js](#). We also want to exclude anything in the [node\\_modules](#) folder. We also want to add the preset-env to the options object.

```
module: {
  rules: [
    //..
    {
      test: /\.js$/,
      exclude: /node_modules/,
      use: [
        {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env'],
          },
        },
      ],
    },
  ],
},
```

Now, we can run `npm run dev` or `npm run build` and it should work as expected.

If you want to test it out, you can use an arrow function in your source code, and you will see it will be a regular function in your build file.

So we have a modern front-end dev environment setup. We have Webpack, Babel, and DevServer all setup and working. We can now start to refactor the Tracalorie app using this [webpack-starter](#) project.

# Tracalorie Webpack Refactor

---

Now we are going to take the Tracalorie project, which is just a bunch of classes in a single file, and refactor it to use Webpack. We're going to use the `webpack-starter` boilerplate. I will include it in this lesson, if you don't have it already. You will also need the `tracalorie` project so we can copy the files and code over to the webpack project. I will include that as well.

## Set Up Files

Copy the `webpack-starter` boilerplate and rename it to `tracalorie-webpack`. Open the `package.json` and change the name to `tracalorie-webpack`. You can also change the description and change `main` from `index.js` to `app.js`. This is optional, but it is a convention that I like. Especially, where this is where the `App` class is going to be.

Change the `entry` field in the `webpack.config.js` from `./src/index.js` to `./src/app.js`.

Delete the `message.js` file if it's there. That was just created to show you how modules work. You can remove everything in the `app.js` file as well.

## Bring Over Assets & HTML

In the `src` folder, create a `css` folder. Bring over the following stylesheets from the Tracalorie project:

- `style.css`
- `bootstrap.css`

We will be installing `bootstrap` and `fontawesome` using NPM, so we don't need to bring over the `fontawesome.css` file. We do want the `bootstrap.css` because it is a customized theme.

Bring over the `webfonts` folder into the `src` folder as well.

Copy everything from the `index.html` file and paste it into the `index.html` file in the `src` folder. We need to make some changes here though.

Delete all of these lines from the `<head>`:

```
<link rel="stylesheet" href="css/fontawesome.css" />
<link rel="stylesheet" href="css/bootstrap.css" />
<link rel="stylesheet" href="css/style.css" />
<script src="js/bootstrap.bundle.min.js" defer></script>
<script src="js/app.js" defer></script>
```

We don't need to include the `app.js` because now that we are using Webpack along with the `HTMLWebpackPlugin`, it will automatically include the `app.js` file for us. We don't need the stylesheet links because we will be importing our styles. Since we are going to install Bootstrap, we do not need the `bootstrap.bundle.min.js` file either.

Paste the everything in from the `<body>`. Make sure there are no hardcoded dummy meals or workouts in the HTML.

## Install Bootstrap & Fontawesome

Now we need to install Bootstrap and Fontawesome. We will be using the `bootstrap` and `@fortawesome/fontawesome-free` packages. Install them using NPM:

```
npm install bootstrap @fortawesome/fontawesome-free
```

Now open your `app.js` file and add the following imports:

```
import '@fortawesome/fontawesome-free/js/all.js';
import { Modal, Collapse } from 'bootstrap';
import './css/bootstrap.css';
import './css/style.css';
```

We need to bring in the `Modal` and `Collapse` classes from Bootstrap in order to use them in our JavaScript.

## Create The CalorieTracker class

Create a file named `CalorieTracker.js` in the `src` folder. Copy the `CalorieTracker` class from the Tracalorie project and paste it into the `CalorieTracker.js` file.

At the end of the file, export the class with the following line:

```
export default CalorieTracker;
```

## Create The Meal & Workout Classes

Since the Meal and Workout classes are so simple, let's put the in a single file. Create a file called `Items.js` in the `src` folder. Copy the `Meal` and `Workout` classes from the Tracalorie project and paste them into the `Items.js` file and export them like this:

```
export { Meal, Workout };
```

## Create The Storage Class

Create a file called `Storage.js` in the `src` folder. Copy the `Storage` class from the Tracalorie project and paste it into the `Storage.js` file. Export the class like this:

```
export default Storage;
```

## Import Storage Into CalorieTracker

The Storage class is only used in the tracker, so import it at the top of `CalorieTracker.js` like this:

```
import Storage from './Storage';
```

## Import CalorieTracker & Items Into App

In the `src/app.js` folder, import the `CalorieTracker` and `Items` classes like this:

```
import CalorieTracker from './CalorieTracker';
import { Meal, Workout } from './Items';
```

## Create The App Class

Now copy the entire App class to the `src/app.js` file. Also, instantiate the App class:

```
const app = new App();
```

## Run The Server

Now we should be all set to run the Webpack dev server. Run the following command:

```
npm run dev
```

## Use The UUID Package

Instead of using hex values for the item IDs, let's use the `uuid` package. Install it using NPM:

```
npm install uuid
```

Now, in the `Items.js` file, import the `uuid` package like this:

```
import { v4 as uuidv4 } from 'uuid';
```

Then, in the `Meal` and `Workout` classes, replace the `id` property with the following:

```
this.id = uuidv4();
```

This is a great example of how you can use packages to add functionality to your projects.

That's it, now you have a modern, modularized project using Webpack. This makes our app more scalable and easier to maintain.

# Symbols

---

It was a long time ago that we talked about the primitive types of JavaScript, such as strings, numbers and booleans, however, we did not really address Symbols, which are also a primitive data type. They are a bit harder to understand than the others, so I wanted to wait a while before we talked about them.

Symbols are used to create unique identifiers for objects. They are created using the `Symbol()` function, which can be called with an optional string as its parameter. This string is used to describe the symbol, and is useful when debugging code. The value that you pass in is strictly for identification. Let's create a couple symbols.

```
const sym1 = Symbol('foo');
const sym2 = Symbol('bar');

console.log(sym1); // Symbol(foo)
console.log(sym2); // Symbol(bar)
console.log(typeof sym1); // symbol
```

It is important to remember that when we create a symbol, we do not use the `new` keyword. It is not a constructor function. This prevents developers from creating an explicit Symbol wrapper object instead of a new Symbol value. We can do that with strings, numbers and booleans, but not with symbols.

We can get the read-only `description` property of a Symbol, but keep in mind, this is just an identifier, it is not a value.

```
console.log(Symbol('sym1').description); // sym1
console.log(Symbol().description); // undefined
```

## Symbols are unique

Symbols are unique. So if we do the following, we get false.

```
console.log(Symbol('sym1') === Symbol('sym1'));
```

This is because the `Symbol()` function creates a new symbol each time it is called. This is why we can pass in a string to the function. It is used to identify the symbol, but it is not used to create the symbol.

One use case for Symbols is when we want to create objects that have private properties. We can use Symbols to create private properties that are not accessible outside of the object. Let's create an object that has a private property.

```
const user = {
  [Symbol('id')]: 1,
  name: 'John',
  email: 'john@gmail.com',
};

user.id = '123';

console.log(user); // { name: 'John', email: 'john@gmail.com', id: '123',
[Symbol(id)]: 1 }
```

Changing the id does not change the symbol value, it just adds a new property to the object.

This will not work either

```
console.log(user[Symbol('id')]); // undefined
```

So symbols are a good way to hide properties from the outside world. If you're creating a library or something like that.

## Non-Enumerable

We can also see that the symbol is not enumerable. It is not returned when we use `Object.keys()` or `Object.values()`.

```
console.log(Object.keys(user)); // [ 'name', 'email', 'id' ]
console.log(Object.values(user)); // [ 'John', 'john@gmail.com', '123' ]
```

Since Symbols are not enumerable, we cannot use `for...in` to iterate over them.

```
for (let key in user) {
  console.log(key);
}
```

## getOwnPropertySymbols

We can use `Object.getOwnPropertySymbols()` to get an array of all the symbols in an object.

```
console.log(Object.getOwnPropertySymbols(user)); // [ Symbol(id) ]
```

## Well-known Symbols

There are a few well-known symbols that are used by JavaScript. These are used to create special objects that have special properties. For example, the `Symbol.iterator` symbol is used to create objects that are iterable. We will talk about this in a later chapter.

## Symbol.for()

There is another way to create symbols. We can use the `Symbol.for()` function. This function takes a string as its parameter. It will search for an existing symbol that has the same string as its description. If it finds one, it will return that symbol. If it does not find one, it will create a new symbol with the string as its description.

The Symbols are stored in a global symbol registry. So it first checks the registry before creating it. Having it in a global registry means that if we create a symbol using `Symbol.for()`, we can access it from anywhere in our code.

```
const sym3 = Symbol.for('foo');
const sym4 = Symbol.for('foo');

console.log(sym3 === sym4); // true
```

`sym3` and `sym4` are the same symbol. They are both referencing the same symbol in the global symbol registry.

## Symbol.keyFor()

We can use the `Symbol.keyFor()` function to get the description of a symbol. It takes a symbol as its parameter. It will return the description of the symbol if it is in the global symbol registry. If it is not in the global symbol registry, it will return `undefined`.

```
console.log(Symbol.keyFor(sym3)); // foo
```

If we use `Symbol()` to create a symbol, it will not be in the global symbol registry, so it will return `undefined`.

```
console.log(Symbol.keyFor(sym1)); // undefined
```

## toString() & valueOf()

These do exactly what you think. They get the Symbol in string form and the primitive value of the Symbol.

```
// toString() - returns a string representation of a symbol
console.log(sym1.toString()); // Symbol(sym1)
```

```
console.log(sym3.toString()); // Symbol(foo)

// valueOf - returns a primitive value of a symbol
console.log(sym1.valueOf()); // Symbol(sym1)
console.log(sym3.valueOf()); // Symbol(foo)
```

## Well-Known Symbols

There are some well-known symbols, also called 'built-in symbols' that are used by JavaScript engines to provide certain language features and are shared by all built-in objects. For example, `Symbol.iterator` is utilized to iterate over items in arrays, strings, or even to define your own iterator function. We're going to go over iterators very soon and we'll look more into this, but in all honesty, you probably won't use these all too much, but I did want to mention them.

You can actually see all of the built-in symbols by running the following code:

```
console.log(Object.getOwnPropertyNames(Symbol));

// ['length', 'name', 'prototype', 'for', 'keyFor', 'asyncIterator',
'hasInstance', 'isConcatSpreadable', 'iterator', 'match', 'matchAll',
'replace', 'search', 'species', 'split', 'toPrimitive', 'toStringTag',
'unscopables']
```

If you want to look more into these, you can, but I would say they're beyond the scope of this course because I only want to focus on things that you'll use on a frequent basis.

# Iterators

---

An **iterator** is an object that defines a sequence and potentially a return value upon completion. It's an object that implements the **iterator protocol** by having a `next()` method. The `next()` method returns an object with two properties: `value` and `done`. The `value` property is the value of the next item in the sequence. The `done` property is a boolean that is `true` if the iterator is done with its sequence.

## Iterator Example 1

We're going to create an `app` object that has an array of baseball teams.

```
const app = {
  teams: ['Red Sox', 'Yankees', 'Astros', 'Dodgers'],
};
```

We want to create an iterator that will iterate through this teams array and return the next team in the sequence. We need to create a function in the object called `next` that returns the next team in the sequence. We also need to create a variable called `nextIndex` that will keep track of the next team in the sequence.

The `next()` function should return an object with a `value` and a `done` property. The `value` property is the next team in the sequence. The `done` property is `true` when the sequence is complete.

```
const app = {
  nextIndex: 0,
  teams: ['Red Sox', 'Yankees', 'Astros', 'Dodgers'],
  next() {
    if (this.nextIndex >= this.teams.length) {
      return { done: true };
    }

    const returnValue = { value: this.teams[this.nextIndex], done: false };
    this.nextIndex++;
    return returnValue;
  },
};
```

We can call the `next()` function to get the next team in the sequence.

```
console.log(app.next()); // { value: 'Red Sox', done: false }
console.log(app.next()); // { value: 'Yankees', done: false }
console.log(app.next()); // { value: 'Astros', done: false }
console.log(app.next()); // { value: 'Dodgers', done: false }
console.log(app.next()); // { done: true }
```

We can use a while loop with our iterator.

```
let next1 = app.next();

while (!next1.done) {
  console.log(next1.value);
  next1 = app.next();
}
```

Now, even though this is an iterator, is not iterable such as a built int Array or Map in JavaScript. We can not use a for...of loop with our iterator.

```
for (const team of app) {
  console.log(team); // TypeError: app is not iterable
}
```

In order for this to be iterable and work with a for...of loop, we need to add a `Symbol.iterator` to our object.

## Iterator Example 2

In this example, we will do the same thing as the previous example, but we will use the built-in `Symbol.iterator`.

```
const app = {
  teams: ['Red Sox', 'Yankees', 'Astros', 'Dodgers'],
  [Symbol.iterator]: function () {
    let nextIndex = 0;
    return {
      next: () => {
        return nextIndex < this.teams.length
          ? { value: this.teams[nextIndex++], done: false }
          : { done: true };
      },
    };
  },
};
```

So here, we are using `Symbol.iterator` to create a function where we are returning an object that has a `next()` function. The `next()` function returns the next team in the sequence. If the sequence is complete, it returns `done: true`.

We can use the iterator like this:

```
const iterator = app[Symbol.iterator]();
console.log(iterator.next().value); // Red Sox
console.log(iterator.next().value); // Yankees
console.log(iterator.next().value); // Astros
console.log(iterator.next().value); // Dodgers
console.log(iterator.next().done); // true
```

We could also use this iterator in a `while` loop.

```
let next = iterator2.next();

while (!next.done) {
  console.log(next.value);
  next = iterator2.next();
}
```

Or we could use it in a `for...of` loop.

```
// For of loop
for (const team of app2) {
  console.log(team); // Res Sox, Yankees, Astros, Dodgers
}
```

So we're creating our own iterators. You may not use this stuff right now as a beginner, but it's good to know that this is how iterators work.

# Generators

---

Generators are similar to iterators. In fact, they are basically used to "generate" iterators. You can use them for the same thing, but they have a much easier syntax and are easier to maintain. Generators are functions that can be paused and resumed. They are created using the `function*` syntax. We can use the `yield` keyword to pause the function. We can also use the `yield*` keyword to delegate to another generator function.

Let's look at an example.

```
function* createTeamIterator(teams) {
  for (let i = 0; i < teams.length; i++) {
    yield teams[i];
  }
}

const teams = ['Red Sox', 'Yankees', 'Astros', 'Dodgers'];

const iterator = createTeamIterator(teams);

console.log(iterator.next().value); // Red Sox
console.log(iterator.next().value); // Yankees
console.log(iterator.next().value); // Astros
console.log(iterator.next().value); // Dodgers
console.log(iterator.next().done); // true
```

As you can see, we get the same results as we did with the iterator. The only difference is that the syntax is much easier to read and maintain.

We can also use a `for...of` loop with a generator.

```
for (const team of createTeamIterator(teams)) {
  console.log(team);
}
```

Or we can use a spread operator.

```
console.log([...createTeamIterator(teams)]);
```

We can also use `destructuring` to get the values.

```
const [first, second, third] = createTeamIterator(teams);
console.log(first, second, third);
```

# Profile Scroller Project

---

In this project, we will use a simple generator function to create a dating app profile scroller. We are using people, but you could use any type of data.

## The HTML

Here is the HTML. There is a dummy person in the HTML, but we will be using JavaScript to add more people.

```
<div class="container">
  
  <div class="profile-info">
    <h3>Jamie Williams</h3>
    <p>26 Years Old</p>
    <p>From London, UK</p>
    <p>Female looking for male</p>
    <button id="next">Show Next</button>
  </div>
</div>
```

## The CSS

We have some simple CSS

```
url('https://fonts.googleapis.com/css2?
family=Poppins:wght@300;400;500;600;700&display=swap');
body { font-family: 'Poppins', sans-serif; background: #f5f5f5; font-size: 18px;
} .container { width: 400px; margin: 100px auto; padding: 40px 20px;
background: #fff; border: 1px solid #ccc; border-radius: 5px; box-shadow: 0 0 10px #ccc;
text-align: center; } img { border-radius: 50%; } button { background: purple;
color: #fff; border: 1px solid #ccc; border-radius: 10px; padding: 14px 25px;
margin: 10px; cursor: pointer; font-size: inherit; } button:hover { background: #fff; color: purple; }
```

## The JavaScript

Let's create an array of people with their info.

```
const people = [
  {
    name: 'John Smith',
```

```
age: 35,
gender: 'male',
location: 'New York, NY',
 imageURL: 'https://randomuser.me/api/portraits/men/1.jpg',
looking: 'Male looking for female',
},
{
name: 'Jamie Williams',
age: 26,
gender: 'female',
location: 'Los Angeles, CA',
 imageURL: 'https://randomuser.me/api/portraits/women/1.jpg',
looking: 'Female looking for male',
},
{
name: 'Bob Johnson',
age: 42,
gender: 'male',
location: 'Chicago, IL',
 imageURL: 'https://randomuser.me/api/portraits/men/2.jpg',
looking: 'Male looking for male',
},
{
name: 'Shannon Jackson',
age: 29,
gender: 'female',
location: 'Los Angeles, CA',
 imageURL: 'https://randomuser.me/api/portraits/women/2.jpg',
looking: 'Female looking for female',
},
];

```

In a real-world app, this would be coming from an API.

Let's bring in everything we need from the DOM.

```
const container = document.querySelector('.container');
const img = container.querySelector('img');
const profileInfo = container.querySelector('.profile-info');
const nextBtn = container.querySelector('#next');
```

Now we can create a generator function that will yield a person from the array. We will then set the generator to a variable called `generator`.

```
function* peopleGenerator() {
let index = 0;
while (true) {
yield people[index++ % people.length];
```

```
    }

const generator = peopleGenerator();
```

Now we will create the event listener for the button. When the button is clicked, we will get the next person from the generator and set the DOM elements to the new person's info. We will also make the initial 'click' to get the first person.

```
nextBtn.addEventListener('click', () => {
  const person = generator.next().value;
  img.src = person.imageURL;
  profileInfo.querySelector('h3').textContent = person.name;
  profileInfo.querySelectorAll('p')[0].textContent = `${person.age} Years Old`;
  profileInfo.querySelectorAll('p')[1].textContent = `From ${person.location}`;
  profileInfo.querySelectorAll('p')[2].textContent = person.looking;
});

nextBtn.click();
```

Now we have a profile scroller. You can click the **next** button and it will show the next person. Feel free to add more people to the array.

# Sets

---

Sets are a data structure that allows you to store a collection of unique values. They are unordered and they do not allow duplicates. Sets are useful when you want to store a collection of values that you want to check for membership, but you don't care about the order of the values.

## Creating a Set

To create a set, you use the `new Set()` constructor. You can pass in an iterable, such as an array, to initialize the set with values.

```
const set = new Set([1, 2, 3, 4]);
```

Sets contain only unique values. If you try to add a value that already exists in the set, it will be ignored.

```
const set = new Set([1, 2, 2, 3, 3, 4]);
// set = {1, 2, 3, 4}
```

## Adding to a Set

To add a value to a set, you use the `add()` method.

```
set.add(5);
```

## Checking for Membership

To check if a value is in a set, you use the `has()` method.

```
set.has(5); // true
set.has(6); // false
```

## Removing from a Set

To remove a value from a set, you use the `delete()` method.

```
set.delete(5);
set.has(5); // false
```

## Converting a Set to an Array

To convert a set to an array, you use the `Array.from()` method.

```
const setArray = Array.from(set);
```

## Converting an Array to a Set

To convert an array to a set, you use the `new Set()` constructor.

```
const arraySet = new Set([1, 2, 3, 4, 5]);
```

## Iterating using a for...of loop

To iterate through a set, you can use a `for...of` loop.

```
for (let item of set) {
  console.log(item);
}
```

## Create an iterator

To create an iterator, you use the `values()` method.

```
const iterator = set.values();

console.log(iterator.next()); // {value: 1, done: false}
console.log(iterator.next()); // {value: 2, done: false}
console.log(iterator.next()); // {value: 3, done: false}
console.log(iterator.next()); // {value: 4, done: false}
console.log(iterator.next()); // {value: undefined, done: true}
```

This should make sense after the last few lessons.

## Clear the set

```
set.clear();
```

In the next lesson, we'll look at maps.

# Maps

---

Maps are another data structure that we can use in JavaScript. Maps were introduced in ES6. They are similar to objects, but the keys can be ANY type, not just strings. You can even have an object or an array for a key. Maps are iterable, so we can loop through them as well.

## The Map Object

The Map object is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.

```
const map = new Map();
```

We can add data to a map using the `set` method.

```
map.set('name', 'John');
map.set(1, 'Number One');
```

We can get data from a map using the `get` method.

```
map.get('name'); // John
map.get(1); // Number One
```

We can also use the `size` property to get the number of items in the map.

```
map.size; // 2
```

We can also use the `has` method to check if a key exists in the map.

```
map.has('name'); // true
map.has('age'); // false
```

We can also use the `delete` method to remove a key-value pair from the map.

```
map.delete('name');
map.size; // 1
```

We can also use the `clear` method to remove all key-value pairs from the map.

```
map.clear();
map.size; // 0
```

We can also use the `forEach` method to loop through the map. Let's create a new map and add some data to it.

```
const peopleMap = new Map();
map.set('Brad', { phone: '555-555-5555', email: 'brad@gmail.com' });
map.set('John', { phone: '555-555-5555', email: 'john@gmail.com' });
map.set('Jill', { phone: '555-555-5555', email: 'jill@gmail.com' });
```

Now we can loop through the map using the `forEach` method.

```
peopleMap.forEach((person) => {
  console.log(person.email);
});
```

We can also get the `keys`, `values` and `entries` from the map. These 3 methods return an iterator, so this should make sense if you watched the videos on iterators.

```
peopleMap.keys(); // MapIterator {"Brad", "John", "Jill"}
peopleMap.values(); // MapIterator {...}, {...}, {...}
peopleMap.entries(); // MapIterator {"Brad" => {...}, "John" => {...}, "Jill" =>
...}
```

Let's use the `next()` method to get the first value from the `keys` iterator.

```
const iterator = peopleMap.values();

console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

To convert a map to an array, we can use the `Array.from` method.

```
const peopleArray = Array.from(peopleMap);
```

To convert to an array of values, we can use the `Array.from` method and pass in the `values` method.

```
const peopleArray = Array.from(peopleMap.values());
```

To convert to an array of keys, we can use the `Array.from` method and pass in the `keys` method.

```
const peopleArray = Array.from(peopleMap.keys());
```

# Poll Project

---

In this project, we will create a very simple poll application that uses a `Map` to keep track of the votes. Like with anything else, using a `Map` in an actual project, helps you understand it better.

We will create a poll form to vote for your favorite JavaScript framework. The poll will have 5 options, and the user can only vote for one option. The results will be displayed after the user votes.

## The HTML

I used Bootstrap 5 for this project. Here is the HTML with the Bootstrap classes:

```
<!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" />
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css"
      rel="stylesheet"
      integrity="sha384-GLh1TQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA6j6gD"
      crossorigin="anonymous"
    />
    <script src="script.js" defer></script>

    <title>Poll</title>
  </head>
  <body class="bg-light">
    <div class="card w-50 m-auto border-round mt-5 shadow-lg">
      <div class="card-header bg-primary text-white text-center">
        <h1>Favorite JS Framework</h1>
      </div>
      <div class="card-body p-5 fs-4">
        <form id="poll-form">
          <div class="form-check m-2 p-3 border-bottom">
            <input
              class="form-check-input"
              type="radio"
              name="poll-option"
              id="poll-option"
              value="React"
            />
            <label class="form-check-label" for="poll-option"> React </label>
          </div>
          <div class="form-check m-2 p-3 border-bottom">
            <input
```

```

        class="form-check-input"
        type="radio"
        name="poll-option"
        id="poll-option"
        value="Vue"
      />
      <label class="form-check-label" for="poll-option"> Vue </label>
    </div>
    <div class="form-check m-2 p-3 border-bottom">
      <input
        class="form-check-input"
        type="radio"
        name="poll-option"
        id="poll-option"
        value="Angular"
      />
      <label class="form-check-label" for="poll-option"> Angular </label>
    </div>

    <div class="form-check m-2 p-3 border-bottom">
      <input
        class="form-check-input"
        type="radio"
        name="poll-option"
        id="poll-option"
        value="Svelte"
      />
      <label class="form-check-label" for="poll-option"> Svelte </label>
    </div>

    <div class="form-check m-2 p-3">
      <input
        class="form-check-input"
        type="radio"
        name="poll-option"
        id="poll-option"
        value="Other"
      />
      <label class="form-check-label" for="poll-option"> Other </label>
    </div>

    <input
      type="submit"
      value="Submit"
      class="btn btn-dark btn-lg btn-block w-100 mt-5"
    />
  </form>

  <div id="results" class="m-4"></div>
</div>
</div>
</body>
</html>

```

## The JavaScript`

Let's create a map to correspond to the options in the poll. The keys will be the options, and the values will be the number of votes for each option. The values will be set to 0 initially.

```
const poll = new Map();
poll.set('React', 0);
poll.set('Vue', 0);
poll.set('Angular', 0);
poll.set('Svelte', 0);
poll.set('Other', 0);
```

Let's setup an event listener for the form. When the user submits the form, we will get the value of the selected option, and increment the corresponding value in the map.

```
document.getElementById('poll-form').addEventListener('submit', submitForm);

function submitForm(e) {
  e.preventDefault();
  const selectedOption = document.querySelector(
    "input[name='poll-option']:checked"
  );

  if (!selectedOption) {
    alert('Please select an option');
    return;
  }

  let voteCount = poll.get(selectedOption.value);
  poll.set(selectedOption.value, voteCount + 1);
  displayResults();
}
```

We are using the `get()` method to get the value of the selected option. We are also using the `set()` method to increment the value of the selected option.

Let's create a function to display the results. We will loop through the map and display the results in a list.

```
function displayResults() {
  const results = document.getElementById('results');
  results.innerHTML = '';
  for (let [option, votes] of poll) {
    const optionElement = document.createElement('div');
    optionElement.classList.add(
      'border-bottom',
```

```
'p-2',
'd-flex',
'justify-content-between'
);
optionElement.innerHTML = `<strong>${option}:</strong> ${votes} votes`;
results.appendChild(optionElement);
}
}
```

When you vote, you will see each option and the number of votes for that option. To make it so the user can only vote once, we can disable the form after the user votes.

Add this to the `submitForm()` function:

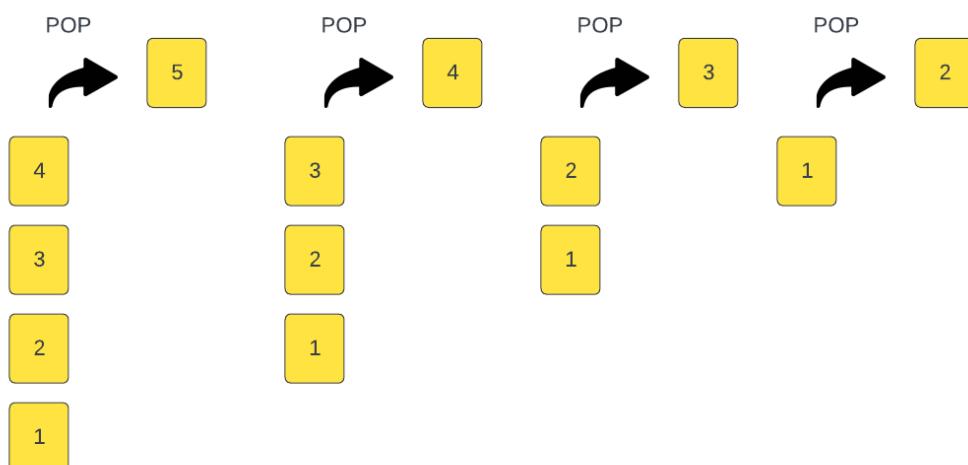
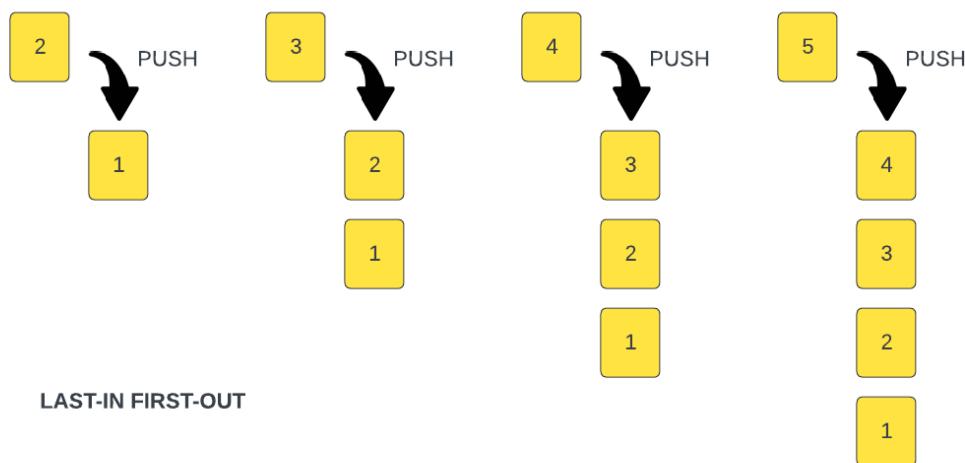
```
// disable form fields and submit button
document
.getElementById('poll-form')
.querySelectorAll('input, button')
.forEach((el) => {
  el.setAttribute('disabled', true);
});
``
```

# Stacks

---

So we have looked at built-in data structures like arrays, objects, sets and maps, but now I want to cover some custom data structures. We'll start with stacks, which if you've been following along, you should already be somewhat familiar with because we talked about the `call stack` earlier in the course.

Stacks are **LIFO** (Last In First Out) data structures. The last item added to the stack will be the first item removed from the stack. We talked about stacks way back when we were looking at the call stack. The call stack is a stack of functions that are called. The last function called is the first function to be removed from the stack.



Since stacks are not built in to JavaScript, we will have to create our own.

We will create a class called `Stack` that will have the following methods:

- `push()`
- `pop()`
- `peek()`
- `length()`
- `isEmpty()`

We will have a constructor where we initialize our data. We will use an array to store our data. We will also have a `count` property that will keep track how many items are in the stack. We will use underscores, because they are all private properties, meaning the should not be accessed outside of the class.

```
class Stack {  
    constructor() {  
        this._items = [];  
        this._count = 0;  
    }  
}
```

## push()

Now we can add the `push()` method. This method will add an item to the top of the stack. Some people just use the built in `push()` method for arrays, but we are going to do this without any help from pre-defined methods. We will use the count as the index and set the item. Then increment the count by 1,

```
push(item) {  
    this._items[this._count] = item;  
    this._count++;  
}
```

## pop()

Now we can add the `pop()` method. This method will remove an item from the top of the stack. Again, we could take the easy way out and use the `Array.prototype.pop()` method, but we want this to be barebones.

First, we check to see if the stack is empty and return "Underflow" if it is. That's what it's called when you try and pop off an item that is not there. It's the opposite of a stack overflow.

Then we store the last item of the stack into a variable 'item'. This item is going to be returned at the end of the method.

It decrements the count property by 1, to reflect that the stack now has one less item.

It starts a for loop, where the variable `i` is initialized to the value of the count, and the loop continues as long as `i` is less than the length of the `items` array.

Inside the for loop, the method assigns the value of `this.items[i+1]` to `this.items[i]`. This has the effect of shifting all items in the array to the left by one index. This way it removes the last item of the array by shifting all the items to the left by one index.

The variable `i` is incremented by one on each iteration, so the loop can continue processing the next item of the array.

The loop continues until all the items have been shifted to the left.

Finally, the method updates the length property of the items array to the value of the count property. This reduces the length of the array, effectively removing the last item.

The last step is returning the variable 'item' that was stored in the first step. This variable holds the value of the last item of the stack before popping it out.

```
pop() {
    if (this._count === 0) {
        return 'Underflow';
    }
    const item = this._items[this._count - 1];
    this.count--;
    for (let i = this._count; i < this._items.length; i++) {
        this._items[i] = this._items[i + 1];
    }
    this._items.length = this._count;
    return item;
}
```

## isEmpty()

We will need to check if the count is === 0 a couple times, so let's create an `isEmpty()` method

```
isEmpty() {
    return this._count === 0;
}
```

Add the `isEmpty()` to the `pop()` method

```
if (this.isEmpty()) {
    return 'Underflow';
}
```

Now we can add the `peek()` method. This method will return the item at the top of the stack. We will return the item from the `data` array with an index that is 1 less than the count.

## peek()

```
peek() {
    if (this.isEmpty()) {
        return 'No items in Stack';
    }
}
```

```
    return this._items[this._count - 1];
}
```

## length()

Now we can add the `length()` method. This method will return the length of the stack. We will return the `count` property.

```
length() {
    return this._count;
}
```

## clear()

This will clear all items and set the count to 0

```
clear() {
    this._items = [];
    this._count = 0;
}
```

Now we can create a new instance of the `Stack` class and add some items to it.

```
const stack = new Stack();
```

At the bottom of the script, let's check the length, top item and if empty

```
console.log('Top item: ', stack.peek());
console.log('Stack Length: ', stack.length());
console.log(stack.isEmpty() ? 'Stack is empty' : 'Stack is not empty');
```

Let's add some items

```
stack.push('First Item');
stack.push('Second Item');
```

We should see a length of 2 and a top item of Second Item

Let's add a couple more items. We will add some strings

```
stack.push('Third Item');
stack.push('Fourth Item');
```

Now we should see a length of **4** and a top item of **Fourth Item**

Let's remove an item

```
stack.pop();
```

We should see a length of **3** and a top item of **Third Item**

Let's remove a couple more items

```
stack.pop();
stack.pop();
```

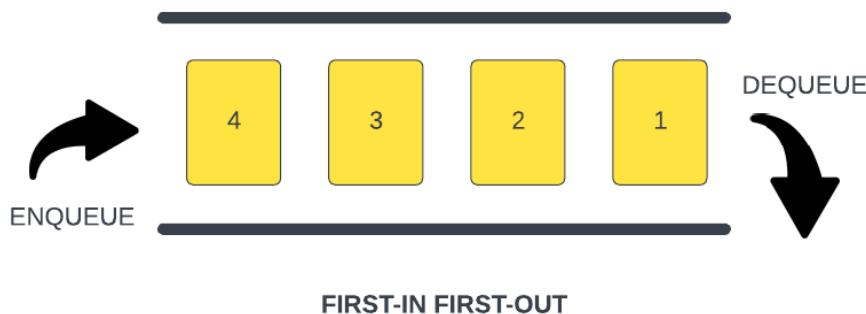
You should be able to call **stack.clear()** at any time and it will clear all items.

We should see a length of **1** and a top item of **First Item**

So we have created a stack class that we can use in our applications. We can use this class to create a stack of any type of data.

# Queues

Queues are another type of data structure specifically designed to operate in a FIFO context (first-in first-out), where items are inserted into one end of the container and extracted from the other.



We saw an example of this when we looked at the JavaScript [Task Queue](#). The [Task Queue](#) is a queue of functions that are waiting to be executed. The first function in the queue is the first function to be executed.

Let's create a class called [Queue](#) that will have the following methods:

- `enqueue()`
- `dequeue()`
- `peek()`
- `length()`
- `isEmpty()`

We will have a constructor where we initialize our data. We will use an array to store our data. We will also have a `count` and a `front` property that will keep track of the front of the queue. We will use underscores to signify that these are private properties.

```
class Queue {  
  constructor() {  
    this._items = [];  
    this._count = 0;  
    this._front = 0;  
  }  
}
```

## [enqueue\(\)](#)

Now we can add the `enqueue()` method. It takes in a single parameter, `item`, which is the item that is being added to the queue.

It assigns the value of the item parameter to the next empty index of the `items` array. The next empty index is determined by the current value of the `count` property. Then it increments the `count` property by 1 to reflect that the queue now has one more item.

```
enqueue(item) {  
    this._items[this._count] = item;  
    this._count++;  
}
```

## dequeue()

Now we can add the `dequeue()` method. This method will remove an item from the front of the queue. We first check to see if there is anything in the queue.

It stores the first element of the queue into a variable 'item'. This element is going to be returned at the end of the method.

It starts a for loop, where `i` is initialized to the value of the `first` property, and the loop continues as long as `i` is less than the `count` property - 1.

Inside the for loop, the method assigns the value of `this._items[i+1]` to `this._items[i]`. This way it removes the first element of the array by shifting all the elements to the left by one index.

Then `i` is incremented by one on each iteration, so the loop can continue processing the next element of the array.

Finally, the method decrements the `count` property by 1 to reflect that the queue now has one less item, and updates the length property of the `items` array to the value of the `count` property. This reduces the length of the array, effectively removing the first element.

```
dequeue() {  
    if (this._count === 0) {  
        return 'Underflow';  
    }  
    const item = this._items[this._front];  
    for (let i = this._front; i < this._count - 1; i++) {  
        this._items[i] = this._items[i + 1];  
    }  
    this._count--;  
    this._items.length = this._count;  
    return item;  
}
```

## isEmpty()

Let's add a method to check to see if the queue is empty

```
isEmpty() {
    return this.length() === 0;
}
```

You can then add that to the `dequeue()` method

```
if (this.isEmpty()) {
    return 'Underflow';
}
```

## peek()

Now we can add the `peek()` method. This method will return the item at the front of the queue. We will return the item from the `data` array at the `front` property.

```
peek() {
    if (this.isEmpty()) {
        return 'No items in Queue';
    }
    return this._items[this._front];
}
```

## length()

Now we can add the `length()` method. This method will return the length of the queue. We will return the `back` property minus the `front` property.

```
length() {
    return this.back - this.front;
}
```

Now we can create a new instance of the `Queue`

```
const queue = new Queue();
```

Lets show the front item, lenth and if empty

```
console.log('Front Item: ', queue.peek());
console.log('Queue Length: ', queue.length());
console.log(queue.isEmpty() ? 'Queue is empty' : 'Queue is not empty');
```

Above that, go ahead and add a couple items

```
queue.enqueue('First item');
queue.enqueue('Second item');
queue.enqueue('Third item');
```

You should see the first item as the front

Let's remove an item

```
queue.dequeue();
```

Now you should see the second item as the front

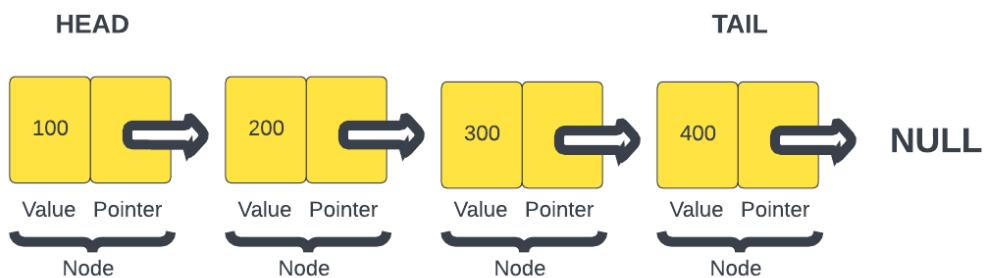
So you can see that the **Queue** class is working as expected, which is first in first out.

# Linked Lists

A linked list is a data structure that contains a sequence of nodes. They are similar to arrays, but the difference is that elements are not stored in a particular memory location or index. Rather, each element is a separate object that contains a pointer or a link to the next object in that list.

Linked Lists are a common interview question. You probably won't use them in your day to day work, but learning this stuff helps your overall understanding of data structures and programming in general.

Let's look at a diagram to help you visualize a linked list.



As you can see a linked list is made up of **nodes**. Each node contains a **value** and a **pointer** to the next node. The first node is called the **head** and the last node is called the **tail**. The **tail** node will point to **null** since there is nothing after it.

Before, we crate our class, I'll just use object literals to represent our nodes.

```
const node1 = {  
    value: 100,  
};  
  
const node2 = {  
    value: 200,  
};
```

Here we just have two nodes. We can connect them by setting the **next** property on **node1** to be **node2**.

```
node1.next = node2;
```

Now we have a linked list with two nodes. We can add more nodes by setting the **next** property on **node2** to be another node.

```
const node3 = {
  value: 300,
};

node2.next = node3;
```

That is a very simplistic example of a linked list.

Now, we will create 2 classes, a `Node` class and a `LinkedList` class. The `Node` class will represent a node in the linked list. It will have a `value` property and a `next` property. The `next` property will point to the next node in the list.

```
class Node {
  constructor(value) {
    this.value = value;
    this.next = null;
  }
}
```

We can initialize a node with:

```
const node1 = new Node(100);
console.log(node1); // Node { value: 100, next: null }
```

Don't leave this in your code, it is just to show you how we will be initializing nodes within the `LinkedList` class.

Let's create a class called `LinkedList` that will have the following methods:

- `insertFirst()`
- `insertLast()`
- `insertAt()`
- `getAt()`
- `removeAt()`
- `printListData()`
- `clearListData()`

We will have a constructor where we initialize our data. We will use a `head` property to keep track of the first node in the list and a `tail` property to keep track of the last node in the list. I am also going to add some comments for the methods that we will be creating.

```
class LinkedList {
  constructor(value) {
    this.head = null;
```

```
    this.length = 0;
}

// Insert first node (head)

// Insert last node (tail)

// Insert at index

// Get at index

// Remove at index

// Print the list data

// Clear list data
}
```

## insertFirst()

We pass in the data for this node. Then we initialize a new node using the `Node` class. We set the `next` property on the new node to be the `head` property. Then we set the `head` property to be the new node. Finally, we increment the `length` property by `1`.

```
// Insert first node (head)
insertFirst(data) {
  const newNode = new Node(data);
  newNode.next = this.head;
  this.head = newNode;
  this.length++;
}
```

We can test this method by creating a new instance of the `LinkedList` class and calling the `insertFirst()` method.

```
const list = new LinkedList();
list.insertFirst(100);
console.log(list); // LinkedList { head: Node { value: 100, next: null },
length: 1 }
```

The `head` property is now pointing to the new node, which has a `value` of `100` and a `next` value pointing to `null` because it is the only node in the list. The `length` property is now `1`.

If we add another node with...

```
list.insertFirst(200);
console.log(list); // LinkedList { head: Node { value: 200, next: Node { value:
100, next: null } }, length: 2 }
```

That node with the value of `200` is now the `head` of the list. The `next` property on that node is pointing to the node with the value of `100`. The `length` property is now `2`.

## insertLast()

Now we want to be able to add a node to the end of the list. We will create a method called `insertLast()`.

```
// Insert last node (tail)
insertLast(data) {
  const newNode = new Node(data);
  let current = this.head;
  while (current.next) {
    current = current.next;
  }
  current.next = newNode;
  this.length++;
}
```

We create a new node with the data that is passed in. Then we create a variable called `current` and set it to the `head` property. We use a `while` loop to loop through the list until we get to the `tail` node. The `tail` node will have a `next` property of `null`. We set the `next` property on the `tail` node to be the new node. Finally, we increment the `length` property by `1`.

We can test this method by creating a new instance of the `LinkedList` class and calling the `insertLast()` method.

```
const list = new LinkedList();
list.insertFirst(100);
list.insertFirst(200);
list.insertFirst(300);
list.insertLast(400);
console.log(list);
```

Now the `tail` node has a `value` of `400` and a `next` property of `null`. The `length` property is now `4`.

## printListData()

Let's jump down to the `printListData()` method. This will help us see the data in the list.

```
// Print the list data
printListData() {
    let current = this.head;
    let list = '';
    while (current) {
        list += current.value + ' ';
        current = current.next;
    }
    console.log(list);
}
```

We create a variable called `current` and set it to the `head` property. We create a variable called `list` and set it to an empty string. We use a `while` loop to loop through the list. We add the `value` property of the current node to the `list` variable. We then set the `current` variable to be the `next` property on the current node. We log the `list` variable to the console.

We can test this method by creating a new instance of the `LinkedList` class and calling the `printListData()` method.

```
const list = new LinkedList();
list.insertFirst(100);
list.insertFirst(200);
list.insertFirst(300);
list.insertLast(400);
list.printListData(); // 300 200 100 400
```

## insertAt()

Now we want to be able to insert a node at a specific index. We will create a method called `insertAt()`.

```
// Insert at index
insertAt(data, index) {
    if (index > 0 && index > this.length) {
        return;
    }

    if (index === 0) {
        this.insertFirst(data);
        return;
    }

    const newNode = new Node(data);
    let current, previous;
    current = this.head;
    let count = 0;

    while (count < index) {
```

```

        previous = current;
        count++;
        current = current.next;
    }

    newNode.next = current;
    previous.next = newNode;
    this.length++;
}

```

We check if the index is greater than 0 and greater than the length of the list. If it is, we return. If the index is `0`, we call the `insertFirst()` method. We create a new node with the data that is passed in. Then we create a variable called `current` and set it to the `head` property. We create a variable called `previous` and set it to `null`. We create a variable called `count` and set it to `0`. We use a `while` loop to loop through the list until we get to the index that we want to insert the node at. We set the `previous` variable to be the `current` variable. We increment the `count` variable by `1`. We set the `current` variable to be the `next` property on the `current` variable. We set the `next` property on the `newNode` to be the `current` variable. We set the `next` property on the `previous` variable to be the `newNode` variable. Finally, we increment the `length` property by `1`.

We can test this method by creating a new instance of the `LinkedList` class and calling the `insertAt()` method.

```

const list = new LinkedList();
list.insertFirst(100);
list.insertFirst(200);
list.insertFirst(300);
list.insertLast(400);
list.insertAt(500, 2);
list.printListData(); // 300 200 500 100 400

```

## getAt()

Now we want to be able to get a node at a specific index. We will create a method called `getAt()`.

```

// Get at index
getAt(index) {
    let current = this.head;
    let count = 0;
    while (current) {
        if (count == index) {
            console.log(current.value);
        }
        count++;
        current = current.next;
    }
}

```

```
    return null;
}
```

We create a variable called `current` and set it to the `head` property. We create a variable called `count` and set it to `0`. We use a `while` loop to loop through the list. We check if the `count` variable is equal to the index that we want to get. If it is, we log the `value` property of the current node to the console. We increment the `count` variable by `1`. We set the `current` variable to be the `next` property on the `current` variable. We return `null` if the index is not found.

We can test this method by creating a new instance of the `LinkedList` class and calling the `getAt()` method.

```
const list = new LinkedList();
list.insertFirst(100);
list.insertFirst(200);
list.insertFirst(300);
list.insertLast(400);
list.insertAt(500, 2);
list.getAt(2); // 500
list.getAt(3); // 100
```

## removeAt()

Now we want to be able to remove a node at a specific index. We will create a method called `removeAt()`.

```
// Remove at index
removeAt(index) {
  if (index > 0 && index > this.length) {
    return;
  }

  let current = this.head;
  let previous;
  let count = 0;

  if (index === 0) {
    this.head = current.next;
  } else {
    while (count < index) {
      count++;
      previous = current;
      current = current.next;
    }
    previous.next = current.next;
  }
  this.length--;
}
```

We check if the index is greater than 0 and greater than the length of the list. If it is, we return. We create a variable called `current` and set it to the `head` property. We create a variable called `previous` and set it to `null`. We create a variable called `count` and set it to `0`. If the index is `0`, we set the `head` property to be the `next` property on the `current` variable. Otherwise, we use a `while` loop to loop through the list until we get to the index that we want to remove the node at. We increment the `count` variable by `1`. We set the `previous` variable to be the `current` variable. We set the `current` variable to be the `next` property on the `current` variable. We set the `next` property on the `previous` variable to be the `next` property on the `current` variable. Finally, we decrement the `length` property by `1`.

We can test this method by creating a new instance of the `LinkedList` class and calling the `removeAt()` method.

```
const list = new LinkedList();
list.insertFirst(100);
list.insertFirst(200);
list.insertFirst(300);
list.insertLast(400);
list.insertAt(500, 2);
list.printListData(); // 300 200 500 100 400
list.removeAt(2);
list.printListData(); // 300 200 100 400
```

As you can see, the value of `500` was removed from the list.

## clearList()

Now we want to be able to clear the list. We will create a method called `clearList()`.

```
// Clear list
clearList() {
    this.head = null;
    this.length = 0;
}
```

We set the `head` property to `null` and set the `length` property to `0`.

We can test this method by creating a new instance of the `LinkedList` class and calling the `clearList()` method.

```
const list = new LinkedList();
list.insertFirst(100);
list.insertFirst(200);
list.insertFirst(300);
list.insertLast(400);
list.insertAt(500, 2);
```

```
list.printListData(); // 300 200 500 100 400  
list.clearList();  
list.printListData(); // ""
```

That's it, we have now created a linked list data structure in JavaScript.

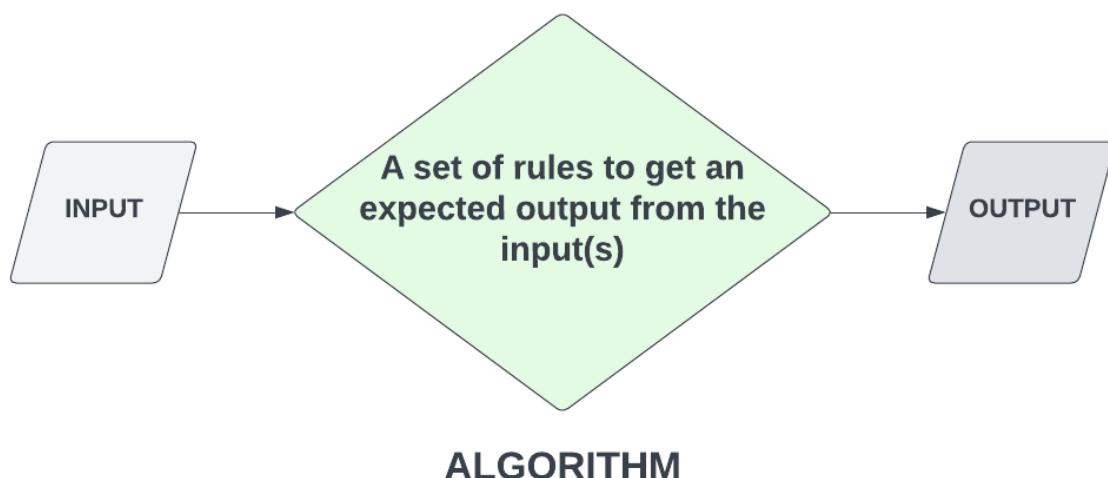
# What Are Algorithms? / Section Intro

---

We looked at some data-structures, which really help you become a better programmer and are also very popular in software development job interviews. Algorithms are another side of the same coin. In fact, you hear the term "data structures and algorithms" a lot. In this section, we'll be not only solving some algorithms, but we'll be writing unit-tests to run to see if we did them correctly. I'll talk more about testing in the next lesson. Let's look at what an algorithm actually is.

Algorithms are used in computer programming to solve problems. They are also used in mathematics, science, engineering, and many other fields.

I like to keep things as simple as possible, so a simple explanation is that an algorithm is a set of instructions for accomplishing a task. Usually, you have an input or a set of inputs that you run through an algorithm to get an expected output. People like to think algorithms have to be this crazy braniac type thing that only tech geniuses can solve, but the truth is, an algorithm can be as simple as a `getSum(n1, n2)` function. You're passing in input, in the form of function arguments, which are the 2 numbers to be added and you have an expected output. You can also write tests to validate that algorithm. In fact, that will be the first test we write.



We did the FizzBuzz challenge a while ago when we were learning about loops. That's actually a very popular algorithm for job interviews.

This section is going to have challenges that you can either stop and try yourself or just follow along to see the solution. I would encourage you to try them yourself. I'm almost hesitant to say that because solving algorithms is a skill that takes time to develop. It's not something that you can just pick up and do. There are many of you that won't be able to do any of these on your own. Just know that that is absolutely fine.

For those of you that do want to try them on your own, keep in mind that you can search for the methods and functions that you need to solve the problem.

These problems will range from easy to intermediate. We're not really doing anything advanced because that's just not what this course is.

IN the next lesson, we'll talk a little bit more about testing.

# What is Unit Testing

---

In software development, we often write tests to test our code. This is especially true for really large projects that teams work on. Now, testing can be a little controversial because it's quite divisive. Some developers say that you always need to test your code, some say it's a waste of precious time. Of course, there is no right or wrong opinion, like most things, it comes down to the developer and the project. Either way, it's good to at least know the basics.

There's many different types of testing. There's unit testing, end-to-end testing, there's integration testing, etc. For the purposes of this course, we're going to be focusing on unit testing, which is one of, if not the most common type. It's also usually the first type of testing done on a project.

Unit testing is a way of testing individual pieces of code, called units. Usually the unit you're testing is a function. It is a method of testing where you write test scripts that check if the code is doing what it's supposed to do. For example, if you have a function that adds two numbers together, you would write a test that calls the function with the numbers 1 and 2 and checks that the output is 3. If the function passes the test, it is working correctly. If it fails, it means there is a problem that needs to be fixed.

## Jest

There are all kinds of testing libraries and frameworks, not just for JavaScript, but for any language. Many of them are similar, they just have a different syntax. We will be using a very popular framework called **Jest**. Jest is pretty easy to use, there isn't even any needed configuration. We can install it with NPM and then just start writing tests. Jest is also very popular, so there's a lot of documentation and tutorials out there, including my own.

This is a very simple example that tests a function called `sum()`. We use the `test()` function and add a description of what should happen. Then we pass in a callback with an `expect` function that gets the function that we're testing passed in and then we're using the `toEqual()` method to see if it equals 3. There's all kinds of methods you can use.

## Testing Algorithms

What we'll be doing is writing a test for each algorithm to make sure that we do it correctly and get the right output. I'm trying to kill two birds with one stone here and teach you basic unit testing as well as problem solving with algorithms. I think it's a good combination.

In the next video, we will get Jest installed and setup our files.

# Unit Testing With Jest

---

Now, we're going to get setup with **Jest**, so that we can write unit tests.

Create a new folder to work in. You can call it whatever you want, but I'm going to call it **unit-testing-algorithms**.

We will be installing Jest using NPM, so we first need to run `npm init` to create a `package.json` file.

## Install Jest

Then, we can install Jest with

```
npm install -D jest
```

We are installing it as a dev-dependency, because we only need it for development.

## Create a `.gitignore` file

Next, create a `.gitignore` file and add `node_modules` to it. This will prevent the `node_modules` folder from being added to the repository.

## Create the test script

In your `package.json` file, add a `test` script:

```
"scripts": {  
  "test": "jest"  
}
```

Now, when we need to run our tests, we just run `npm test`.

The way it works, is any file where we want to test, we make a file with the same name, but with `.test.js` at the end. For example, if we want to test a file called `sum.js`, we would create a file called `sum.test.js`.

Let's create a file called `sum.test.js` and add the following code:

```
const sum = require('./sum');  
  
test('adds 1 + 2 to equal 3', () => {  
  expect(sum(1, 2)).toEqual(3);  
});
```

We are importing the `sum` function from the `sum.js` file. Then, we are using the `test` function to create a test. The first argument is the name of the test, and the second argument is a function that contains the actual test, which is we expect to get 3 as an output, when we have an input of 1 and 2.

Let's run the test with `npm test`. You should see a fail message like this

```
FAIL  ./sum.test.js
  × adds 1 + 2 to equal 3

  ● adds 1 + 2 to equal 3

    TypeError: sum is not a function

    2 |
    3 |   test('adds 1 + 2 to equal 3', () => {
  > 4 |     expect(sum(1, 2)).toBe(3);
    |     ^
    5 |   });
    6 |

      at Object.sum (sum.test.js:4:10)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:  0 total
Time:        0.351 s, estimated 1 s
```

Obviously, this will fail because we did not even create a `sum()` function yet.

It will show how many tests ran, passed and failed and will also give us an error message.

In your `sum.js` file, add the following code:

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

module.exports = sum;
```

Now, when you run `npm test`, you should see a success message like this:

```
PASS  ./sum.test.js
  ✓ adds 1 + 2 to equal 3 (1 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
```

```
Snapshots: 0 total  
Time: 0.413 s, estimated 1 s
```

Congrats! You just wrote your first unit test!

# Grouping Tests With `describe()`

---

In the last lesson, we used the `test()` function to create a test. However, we can also use the `describe()` function to group tests together. This is useful when we have a lot of tests for a single function.

Let's take it a step further and write some tests for the FizzBuzz algorithm. We did this challenge a while back. Basically, it needs to return `Fizz` if the number is divisible by 3, `Buzz` if the number is divisible by 5, `FizzBuzz` if the number is divisible by both 3 and 5, and the number itself if it is not divisible by 3 or 5.

Create a file called `fizzbuzz.js` and add the following code:

```
function fizzbuzz(num) {
  if (num % 3 === 0) {
    return 'Fizz';
  } else if (num % 5 === 0) {
    return 'Buzz';
  } else {
    return num;
  }
}
```

Notice, I left out the case where the number is divisible by both 3 and 5. We will add that later.

Let's create `fizzbuzz.test.js` and add the following code:

```
const fizzBuzz = require('./fizzbuzz');

describe('fizzBuzz', () => {
  it('should be a function', () => {
    expect(typeof fizzBuzz).toEqual('function');
  });
  it('should return a number', () => {
    expect(typeof fizzBuzz(1)).toEqual('number');
  });
  it('should return 1 with the input of 1', () => {
    expect(fizzBuzz(1)).toEqual(1);
  });
});
```

Here, we used the `describe()` function to group the tests together. The first argument is the name of the group, and the second argument is a function that contains the tests.

When we write a test, we use the `it()` function. The first argument is the name of the test, and the second argument is a function that contains the actual test. It is common convention to start the name of the test with `should`.

Then we use the `expect()` function to test the actual value. The first argument is the actual value, and the second argument is the expected value.

We are first testing if it is a function. Then if it returns a number if passed a number. Then if it returns 1 if passed 1.

Run `npm test` and you should get something like this

```
PASS  fizzbuzz/fizzbuzz.test.js
PASS  sum/sum.test.js

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.463 s, estimated 1 s
```

So we have 2 test suites, and 4 tests. 1 of them are from the `sum.test.js` file, and 3 of them are from the `fizzbuzz.test.js` file.

Instead of only testing for the number 1, let's add a few more cases. Replace the last test with

```
it('should return the number if not divisible by 3 or 5', () => {
  expect(fizzBuzz(1)).toEqual(1);
  expect(fizzBuzz(13)).toEqual(13);
  expect(fizzBuzz(17)).toEqual(17);
});
```

So we can add as many cases as we want to test.

Now, let's test if we get `Fizz` if the number is divisible by 3. Add the following code after the last test:

```
it('should return Fizz if divisible by 3', () => {
  expect(fizzBuzz(3)).toEqual('Fizz');
  expect(fizzBuzz(6)).toEqual('Fizz');
  expect(fizzBuzz(9)).toEqual('Fizz');
});
```

Also, if we get `Buzz` if the number is divisible by 5. Add the following code after the last test:

```
it('should return Buzz if divisible by 5', () => {
  expect(fizzBuzz(5)).toEqual('Buzz');
  expect(fizzBuzz(10)).toEqual('Buzz');
  expect(fizzBuzz(20)).toEqual('Buzz');
});
```

Run `npm test` and you should get something like this

```
PASS fizzbuzz/fizzbuzz.test.js
PASS sum/sum.test.js

Test Suites: 2 passed, 2 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        0.523 s, estimated 1 s
```

Finally, if we get **FizzBuzz** if the number is divisible by both 3 and 5. Add the following code after the last test:

```
it('should return FizzBuzz if divisible by 3 and 5', () => {
  expect(fizzBuzz(15)).toEqual('FizzBuzz');
  expect(fizzBuzz(30)).toEqual('FizzBuzz');
  expect(fizzBuzz(45)).toEqual('FizzBuzz');
});
```

```
npm test
```

That test fails because we did not add the case where the number is divisible by both 3 and 5. Let's add that. Replace the `fizzbuzz.js` file with the following code:

```
function fizzBuzz(number) {
  if (number % 3 == 0 && number % 5 == 0) {
    return 'FizzBuzz';
  } else if (number % 3 == 0) {
    return 'Fizz';
  } else if (number % 5 == 0) {
    return 'Buzz';
  } else {
    return number;
}
```

Now run the test again and it should pass.

## Clean up files

Let's create a folder for each algorithm and put the main file and test file into those. So you will have a folder called `sum` and a folder called `fizzbuzz`. Then, you will have a `sum.js` file and a `sum.test.js` file inside the `sum` folder, and a `fizzbuzz.js` file and a `fizzbuzz.test.js` file inside the `fizzbuzz` folder.

Let's move on to another algorithm in the next lesson

# Reverse A String

---

## Instructions

Create a function called `reverseString` that takes a string as an argument and returns the string reversed.

## Tests

Let's make sure that `reverseString` works as expected. it should be a function and return a string. It should also return the string reversed.

```
const reverseString = require('./reverseString');

describe('Reverse String', () => {
  it('should be a function', () => {
    expect(typeof reverseString).toEqual('function');
  });
  it('should return a string', () => {
    expect(typeof reverseString('hello')).toEqual('string');
  });
  it('should return the string reversed', () => {
    expect(reverseString('hello')).toEqual('olleh');
    expect(reverseString('Howdy')).toEqual('ydwH');
    expect(reverseString('Greetings from Earth')).toEqual(
      'htraE morf sgniteerG'
    );
  });
});
```

## Hint

There is no method to reverse a string, but there is a method to reverse an array and strings can be converted into arrays and vice versa.

If you want to try and create the function on your own, you can do so in the `reverseString.js` file. Then just run `npm test` to see if your function passes the tests.

► [Click For Solutions](#)

## Solution 1

```
function reverseString(str) {
  return str.split(' ').reverse().join(' ');
}
```

We can use the `split` method to split the string into an array of characters. We can then use the `reverse` method to reverse the array. Finally, we can use the `join` method to join the array back into a string.

This is probably the most straightforward way to solve this problem. Let's look at another solution.

## Solution 2

```
function reverseString(str) {  
  let reversed = '';  
  for (let character of str) {  
    reversed = character + reversed;  
  }  
  return reversed;  
}
```

We can use a `for` loop to iterate through the string. We can then add each character to the beginning of the `reversed` variable. We can then return the `reversed` variable.

## Solution 3

```
function reverseString(str) {  
  return str.split(' ').reduce((reversed, character) => {  
    return character + reversed;  
  }, '');  
}
```

We can use the `reduce` method to iterate through the string. We can then add each character to the beginning of the `reversed` variable. We can then return the `reversed` variable.

# Palindrome

---

A palindrome is a word that is spelled the same forwards and backwards. For example, **kayak** and **rotator** are both palindromes. **hello** and **world** are not palindromes.

## Instructions

Create a function called **palindrome()** that takes a string as an argument and returns **true** if the string is a palindrome and **false** if it is not.

## Tests

```
const palindrome = require('./palindrome');

describe('palindrome', () => {
  it('should be a function', () => {
    expect(typeof palindrome).toEqual('function');
  });
  it('should return a boolean', () => {
    expect(typeof palindrome('hello')).toEqual('boolean');
  });
  it('should return true if is a palindrome', () => {
    expect(palindrome('kayak')).toBeTruthy();
    expect(palindrome('rotator')).toBeTruthy();
    expect(palindrome('wow')).toBeTruthy();
  });
  it('should return false if is not a palindrome', () => {
    expect(palindrome('hello')).toBeFalsy();
    expect(palindrome('world')).toBeFalsy();
    expect(palindrome('bye')).toBeFalsy();
  });
  it('should return false if includes spaces', () => {
    expect(palindrome(' wow')).toBeFalsy();
    expect(palindrome('wow ')).toBeFalsy();
  });
});
```

We are testing to make sure that **palindrome** is a function and that it returns a boolean. We are also testing to make sure that it returns **true** if the string is a palindrome and **false** if it is not. It should also return **false** if the string includes spaces.

► Click For Solutions

## Solution 1

```
function palindrome(str) {  
    const reversed = str.split(' ').reverse().join(' ');\n    return str === reversed;\n}
```

This is very similar to one of the solutions for the `reverseString` problem. We can use the `split` method to split the string into an array of characters. We can then use the `reverse` method to reverse the array. Finally, we can use the `join` method to join the array back into a string. Then we simply need to check if the `str` is equal to the `reversed` string.

## Solution 2

```
function palindrome(str) {  
    return str.split(' ').every((char, i) => {  
        return char === str[str.length - i - 1];  
    });  
}
```

We can use the `every` method to check if every character in the string is equal to the character at the same index in the reversed string. We can use the `str.length - i - 1` to get the index of the character in the reversed string.

# Array Chunking

---

## Instructions

Create a function called `chunk()` that takes in an array and a size. The function should return a new array where each element is a sub-array of the given size.

## Tests

I am going to change it up a bit here and use the built-in Node.js `assert` library along with Jest. We don't have to do this and it will do the same thing, but it's just to show you there are multiple ways to test and you may run into this. You can read more about `assert` [here](#).

An `assertion` is a statement that is either true or false. If it is false, then the program will throw an error. We can use `assert` to test our code. We can use `assert.equal` to test if two values are equal. We can also use `assert.deepEqual` to test if two objects are equal. We can also use `assert.ok` to test if a value is truthy.

```
const chunk = require('./arraychunk');
const assert = require('assert');

describe('Array Chunking', () => {
  it('Should implement array chunking', () => {
    assert.deepEqual(chunk([1, 2, 3, 4], 2), [
      [1, 2],
      [3, 4],
    ]);
    assert.deepEqual(chunk([1, 2, 3, 4], 3), [[1, 2, 3], [4]]);
    assert.deepEqual(chunk([1, 2, 3, 4], 5), [[1, 2, 3, 4]]);
  });
});
```

We are going to use `assert.deepEqual` to test if the returned array is equal to the expected array. We can use `assert.deepEqual` to test if two arrays are equal. We can also use `assert.deepEqual` to test if two objects are equal.

► Click For Solutions

Let's also change our functions up a bit and use arrow functions.

## Solution 1

```
const chunk = (array, size) => {
  const chunked = [];
  for (let element of array) {
    const last = chunked[chunked.length - 1];
```

```
if (!last || last.length === size) {
    chunked.push([element]);
} else {
    last.push(element);
}
return chunked;
};
```

This is a very common solution. We create an empty array called `chunked`. We then loop through the given array. We then check if the last element in `chunked` is equal to the given size. If it is, we push a new chunk into `chunked` with the current element. If not, we add the current element into the chunk.

## Solution 2

```
const chunk = (array, size) => {
    const chunked = [];
    let index = 0;
    while (index < array.length) {
        chunked.push(array.slice(index, index + size));
        index += size;
    }
    return chunked;
};
```

This is a very similar solution to the first one. We create an empty array called `chunked`. We then loop through the given array. We then use the `slice` method to get a slice of the given array from the current index to the current index plus the given size. We then push the slice into `chunked`. We then increment the index by the given size.

# Anagrams

---

An anagram is a word, phrase, or name formed by rearranging the letters of another, such as cinema, formed from iceman.

## Instructions

Create a function called `anagram()` that takes two strings as arguments and returns `true` if the strings are anagrams and `false` if they are not.

## Tests

```
const anagram = require('./anagram');

describe('Anagram', () => {
  it('should be a function', () => {
    expect(typeof anagram).toEqual('function');
  });
  it('should return a boolean', () => {
    expect(typeof anagram('ram', 'arm')).toEqual('boolean');
  });
  it('should return true if is an anagram', () => {
    expect(anagram('cinema', 'iceman')).toBeTruthy();
    expect(anagram('hello world', 'world hello')).toBeTruthy();
    expect(anagram('god', 'dog')).toBeTruthy();
  });
  it('should return false if is not an anagram', () => {
    expect(anagram('hello', 'fellow')).toBeFalsy();
    expect(anagram('world', 'twirl')).toBeFalsy();
    expect(anagram('lose', 'choose')).toBeFalsy();
  });
});
```

We are testing to make sure that `anagram` is a function and that it returns a boolean. We are also testing to make sure that it returns `true` if the string is a anagram and `false` if it is not.

► Click For Solutions

## Solution 1

```
function anagram(str1, str2) {
  const aCharMap = buildCharMap(str1);
  const bCharMap = buildCharMap(str2);

  if (Object.keys(aCharMap).length !== Object.keys(bCharMap).length) {
    return false;
```

```
}

for (let char in aCharMap) {
    if (aCharMap[char] !== bCharMap[char]) {
        return false;
    }
}

return true;
}

function buildCharMap(str) {
    const charMap = {};

    for (let char of str.replace(/\w/g, '').toLowerCase()) {
        charMap[char] = charMap[char] + 1 || 1;
    }

    return charMap;
}
```

We can use a helper function called `buildCharMap` to build a character map for each string. We can then loop through the character map of the first string and check if the character exists in the second string's character map. If it does, we can check if the character count is the same. If it is not, we can return `false`. If it is, we can continue looping. If we make it through the entire loop, we can return `true`.

# Get Elements By Tag

---

We are going to get a little bit more advanced and realistic with our unit testing. I don't expect anyone to get this without watching, unless you're a very experienced JS developer. We are going to test a function that takes a DOM tree and a tag name and returns an array of all the elements in the tree with that tag name.

This shows you how you can write code that has to do with the DOM and still test it without having to actually create a DOM tree. We won't have any actual HTML or DOM elements. We will just create some mock elements within our test code.

Remember, we have no DOM. We are working within Node.js. In order to create mock elements, we need to use the `jsdom` test environment. First off, we need to create a Jest config file and add the following to it:

```
const config = {
  testEnvironment: 'jsdom',
};

module.exports = config;
```

With recent versions of jest, we need to install an additional package for this to work. So open your terminal and run the following command:

```
npm install -D jest-environment-jsdom
```

Now, we should be able to create mock elements in our tests.

## Tests

```
const getElementsByTag = require('./getelementsbytag');

describe('Get Elements By Tag', () => {
  it('should be a function', () => {
    expect(typeof getElementsByTag).toEqual('function');
  });
  it('should return an array', () => {
    expect(Array.isArray(getElementsByTag())).toEqual(true);
  });
  it('should return an empty array if no root element is passed in', () => {
    expect(getElementsByTag()).toEqual([]);
  });
  it('should return only the root element in the array if no tagName is passed in', () => {
```

```

    const root = document.createElement('div');
    expect(getElementsByTag(root)).toEqual([root]);
  });
  it('should return the correct elements', () => {
    const root = document.createElement('div');

    // Add some child elements to the root
    const p1 = document.createElement('p');
    const p2 = document.createElement('p');
    const span = document.createElement('span');
    root.appendChild(p1);
    root.appendChild(span);
    span.appendChild(p2);

    // Call the function and save the result
    const result = getElementsByTag(root, 'p');

    // Assert that the result is an array containing the two p elements
    expect(result).toEqual([p1, p2]);
  });
}

```

## The Code

We want to pass in a `root` element and a `tagName`. We want to return an array of all the elements in the tree that have that tag name. We will use the `tagName` property of the element to check if it matches the tag name we are looking for.

```

function getElementsByTag(root, tagName) {
  if (!root) return [];
  if (!tagName) return [root];

  let result = [];

  // Check if the root is the tag we are looking for, if so, add it to the
  result
  if (root.tagName.toLowerCase() === tagName.toLowerCase()) {
    result.push(root);
  }

  // Check if the root has any children, if so, recursively call
  getElementsByTagName on each child
  // This will merge the results of each child into the result array
  if (root.hasChildNodes()) {
    for (let child of root.children) {
      result = result.concat(getElementsByTagName(child, tagName));
    }
  }

  return result;
}

```

```
}

module.exports = getElementsByTag;
```

We start off by checking if the `root` is `null` or `undefined`. If it is, we return an empty array. This is because we don't want to return `null` or `undefined` from our function. We want to return an array.

if we pass in only the `root` and no `tagName`, we return an array containing only the `root` element.

Next, we create an empty array called `result`. This is where we will store all the elements that match the tag name we are looking for.

We then check if the `root` element has the tag name we are looking for. If it does, we add it to the `result` array. We use the `tagName` property of the element to check if it matches the tag name we are looking for. We use `toLowerCase` to make sure we are comparing the tag names in a case-insensitive way.

Next, we check if the `root` element has any children. If it does, we loop through each child and recursively call `getElementsByTagName` on each child. `Recursive` means that we are calling the same function within the function. We then merge the results of each child into the `result` array.

Finally, we return the `result` array.

# Has Duplicate IDs

---

We are going to continue to work with the DOM test environment and test a function that takes a DOM tree and returns `true` if there are any duplicate IDs in the tree and `false` if there are not.

## Tests

```
const hasDuplicateIds = require('./hasduplicateids');

describe('DOM Tree Has Duplicate IDs', () => {
  it('should be a function', () => {
    expect(typeof hasDuplicateIds).toEqual('function');
  });
  it('should return an boolean', () => {
    expect(typeof hasDuplicateIds()).toEqual('boolean');
  });
  it('should return false if no root element is passed in', () => {
    expect(hasDuplicateIds()).toEqual(false);
  });
  it('should return true if there are duplicate ids', () => {
    // Create a mock element tree
    const root = document.createElement('div');
    const child1 = document.createElement('div');
    const child2 = document.createElement('div');
    root.appendChild(child1);
    root.appendChild(child2);

    // Add duplicate ids to the tree
    root.id = 'root';
    child1.id = 'child';
    child2.id = 'child';

    // Call the function and save the result
    const result = hasDuplicateIds(root);

    // Assert that the result is true
    expect(result).toEqual(true);
  });
  it('should return false if there are no duplicate ids', () => {
    // Create a mock element tree
    const root = document.createElement('div');
    const child1 = document.createElement('div');
    const child2 = document.createElement('div');
    root.appendChild(child1);
    root.appendChild(child2);

    // Add duplicate ids to the tree
    root.id = 'root';
    child1.id = 'child1';
  });
}
```

```

child2.id = 'child2';

// Call the function and save the result
const result = hasDuplicateIds(root);

// Assert that the result is true
expect(result).toEqual(false);
});
});

```

## Keeping Our Tests DRY By Using `beforeEach` and `afterEach`

As you can see, we have some repeating code where we are creating our root element, adding children, ids, etc. We can use a `beforeEach` block to set up the mock element tree and then use `afterEach` to clean up the mock element tree.

What these blocks do is run before and after each test. This allows us to set up the mock element tree before each test and then clean it up after each test.

```

const hasDuplicateIds = require('./hasduplicateids');

describe('DOM Tree Has Duplicate IDs', () => {
  let root;
  beforeEach(() => {
    root = document.createElement('div');
    const child1 = document.createElement('div');
    const child2 = document.createElement('div');
    root.appendChild(child1);
    root.appendChild(child2);
  });
  afterEach(() => {
    root = null;
  });
  it('should be a function', () => {
    expect(typeof hasDuplicateIds).toEqual('function');
  });
  it('should return an boolean', () => {
    expect(typeof hasDuplicateIds()).toEqual('boolean');
    root.id = 'root';
    root.children[0].id = 'child';
    root.children[1].id = 'child';
    const result = hasDuplicateIds(root);
    expect(typeof result).toEqual('boolean');
  });
  it('should return false if no root element is passed in', () => {
    expect(hasDuplicateIds()).toEqual(false);
  });
  it('should return true if there are duplicate ids', () => {
    root.id = 'root';
    root.children[0].id = 'child';
  });
}

```

```

root.children[1].id = 'child';
const result = hasDuplicateIds(root);
expect(result).toEqual(true);
});
it('should return false if there are no duplicate ids', () => {
  root.id = 'root';
  root.children[0].id = 'child1';
  root.children[1].id = 'child2';
  const result = hasDuplicateIds(root);
  expect(result).toEqual(false);
});
});

```

## The Code

```

function hasDuplicateIds(root, idSet = new Set()) {
  if (!root) return false;

  // If the root has an id and the idSet already has that id, return true
  if (idSet.has(root.id)) return true;

  // If the root has an id, add it to the idSet
  root.id && idSet.add(root.id);

  // If the root has children, recursively call the function on each child
  if (root.hasChildNodes()) {
    for (let child of root.children) {
      const result = hasDuplicateIds(child, idSet);
      if (result) return true;
    }
  }

  return false;
}

module.exports = hasDuplicateIds;

```

We are passing in a root element to the tree and a set of ids. We are using a set because it is a data structure that only allows unique values. If we add a value to the set that already exists, it will not be added.

Then we are checking if the root has an id and if the idSet already has that id. If it does, we return true. If it does not, we add the id to the idSet.

Then we are checking if the root has children. If it does, we are recursively calling the function on each child. If any of the children return true, we return true. If none of the children return true, we return false.

# fs Module

---

The `fs` module is a core module that allows us to interact with the file system. It is used to create, read, update, and delete files and folders. Obviously, from the browser environment, we cannot access the file system. However, we can use the `fs` module in Node.js.

There are methods for doing just about anything with files and folders. There are also asynchronous and synchronous versions of each method. The asynchronous versions are preferred because they do not block the execution of the program. The synchronous versions are useful when we need to wait for the operation to complete before continuing.

There is also a callback and promise version. For the callback, it is just `require('fs')` and for the promise version, it is `require('fs/promises')`.

Let's look at some of the most common methods.

First, we need to bring the module in.

```
const fs = require('fs');
```

## Creating a file

We use the `writeFile` method to create a file. The first argument is the path to the file. The second argument is the content of the file. The third argument is a callback function that will be called when the operation is complete.

```
// Callback version
fs.writeFile('file.txt', 'Hello World', (err) => {
  if (err) throw err;
  console.log('File created!');
});
```

`writeFile` is asynchronous as it takes in a callback function to run when the operation is complete. If we want to use a promise instead, we can use the `writeFile` method from the `fs/promises` module.

```
// Promise version
const fs = require('fs/promises');

fs.writeFile('file.txt', 'Hello World')
  .then(() => console.log('File created!'))
  .catch((err) => console.log(err));
```

We can also use `async/await`. Of course, it has to be wrapped in a function that is marked as `async`. Let's do that and let's pass in the name of the file and the content as arguments.

```
// Async/Await version
async function createFile(filename, content) {
  try {
    await fs.writeFile(filename, content);
    console.log('File created!');
  } catch (err) {
    console.error(err);
  }
}

createFile('file.txt', 'Hello World');
```

Like I said, there is also a synchronous version of `writeFile`. It is called `writeFileSync`. It does not take in a callback function. Instead, it returns `undefined` if the operation is successful. If there is an error, it will throw an exception. Let's put this in a function as well

```
function createFileSync(filename, content) {
  try {
    fs.writeFileSync('file.txt', 'Hello World');
    console.log('File created!');
  } catch (err) {
    console.error(err);
  }
}
```

For the rest of these examples, we will use the `async/await` version, because that is what I would use in a real application. It's completely up to you which version you use.

## Reading a file

To read from a file, we can use the `readFile` method. The first argument is the path to the file. The second argument is the encoding of the file. If you are using the standard callback version, the third argument is a callback function that will be called when the operation is complete. I will be using promises with `Async/Await` in the examples below.

```
async function readFile(filename) {
  try {
    const data = await fs.readFile(filename, 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

```
readFile('file.txt');
```

## Delete a file

We can use the `unlink` method to delete a file.

```
async function deleteFile(filename) {
  try {
    await fs.unlink(filename);
    console.log(`File ${filename} deleted!`);
  } catch (err) {
    console.error(err);
  }
}

deleteFile('file.txt');
```

## Rename a file

We can use the `rename` method to rename a file.

```
async function renameFile(oldName, newName) {
  try {
    await fs.rename(oldName, newName);
    console.log(`File ${oldName} renamed to ${newName}!`);
  } catch (err) {
    console.error(err);
  }
}

renameFile('file.txt', 'new-file.txt');
```

## Create a folder

We can use the `mkdir` method to create a folder.

```
async function createFolder(folderName) {
  try {
    await fs.mkdir(folderName);
    console.log(`Folder ${folderName} created!`);
  } catch (err) {
    console.error(err);
  }
}
```

## Move a file to a folder

We can use the `rename` method to move a file to a folder. I am using a couple methods from the `path` module to get the file name and to join the folder path and the file name. We will look at the path module in the next lesson.

```
async function moveFileToFolder(filePath, folderPath) {
  try {
    const newFilePath = path.join(folderPath, path.basename(filePath));
    await fs.rename(filePath, newFilePath);
    console.log(`File ${filePath} moved to ${newFilePath}`);
  } catch (err) {
    console.error(err);
  }
}

moveFileToFolder('file.txt', 'folder');
```

There are many more methods to work with files and folders. To see the full list, check out the [Node.js documentation](#).

# path Module

---

The `path` module is a core module that allows us to work with file and directory paths. It is used to get the base name of a file, get the extension of a file, create absolute paths, and much more. One big reason that this is useful is because the path separator, among other things can vary between operating systems. For example, on Windows, the path separator is `\` and on Linux and macOS, it is `/`. The `path` module will take care of this for us.

Let's look at some of the most common methods.

First, we need to bring the module in.

```
const path = require('path');
```

Let's create a variable that holds the path to a file.

```
const myFilePath = 'subfolder/anotherfolder/index.js';
```

## basename() - Getting the base name of a file

We use the `basename` method to get the base name of a file. The first argument is the path to the file. The second argument is the extension of the file. If we do not pass in the second argument, it will return the base name with the extension. If we pass in the second argument, it will remove the extension from the base name.

```
const base1 = path.basename(myFilePath);
const base2 = path.basename(myFilePath, '.js');
console.log(base1, base2); // index.js index
```

## extname() - Getting the extension of a file

We use the `extname` method to get the extension of a file.

```
const ext = path.extname(myFilePath);
console.log(ext); // .js
```

## dirname() - Getting the directory name of a file

We use the `dirname` method to get the directory name of a file.

```
const dir = path.dirname(myFilePath);
console.log(dir);
```

## join() - Creating a path

We use the `join` method to create a path. It takes in any number of arguments and joins them together to create a path. It will also normalize the path. For example, if we pass in `subfolder`, `anotherfolder`, and `index.js`, it will join them together with the correct path separator and it will remove any extra path separators.

```
const myPath = path.join('subfolder', 'anotherfolder', 'index.js');
console.log(myPath); // subfolder/anotherfolder/index.js
```

If you are on Windows, you will see that the path separator is `\`. If you are on Linux or macOS, you will see that the path separator is `/`.

## resolve() - Creating an absolute path

We use the `resolve` method to create an absolute path. It takes in any number of arguments and joins them together to create a path. It will also normalize the path, just like the `join` method. The difference is that the `resolve` method will return an absolute path.

```
const resolved = path.resolve('subfolder', 'anotherfolder', 'index.js');
console.log(resolved);
```

You may see `resolve` and `join` used interchangeably. The difference is that `resolve` will always return an absolute path, while `join` will return a relative path if the first argument is a relative path.

The next two things we are going to look at are `__filename` and `__dirname`. These are environment variables that Node provides for us. They are not part of the `path` module, but they are often used with `path` module methods.

## `__dirname` - Getting the directory name of the current file

The `__dirname` is an environment variable that tells you the absolute path of the directory containing the currently executing file. Whenever you are pointing to the file that you are in, you should use `__dirname` instead of `./`. This will ensure that your code will work on any operating system.

```
console.log(__dirname);
```

## `__filename` - Getting the file name of the current file

The `__filename` is an environment variable that is similar to the `__dirname` environment variable. It tells you the absolute path of the currently executing file as well as the file name.

```
console.log(__filename);
```

# os Module

---

The `os` module is a core module that allows us to work with the operating system. It is used to get information about the operating system, get information about the user, and much more. One big reason that this is useful is because the operating system can vary between operating systems. For example, on Windows, the path separator is `\` and on Linux and macOS, it is `/`. The `os` module will take care of this for us. You also may want to have different functionality or content based on the operating system.

## os.arch()

This method returns a string identifying the operating system CPU architecture for which the Node.js binary was compiled. Possible values are `'arm'`, `'arm64'`, `'ia32'`, `'mips'`, `'mipsel'`, `'ppc'`, `'ppc64'`, `'s390'`, `'s390x'`, `'x32'`, and `'x64'`.

```
const os = require('os');
```

```
console.log(os.arch()); // x64
```

## os.platform()

This method returns a string identifying the operating system platform on which Node.js is running. Possible values are `'aix'`, `'darwin'`, `'freebsd'`, `'linux'`, `'openbsd'`, `'sunos'`, and `'win32'`.

```
console.log(os.platform()); // darwin
```

## os.cpus()

This method returns an array of objects containing information about each logical CPU core.

```
console.log(os.cpus()); // [ { model: 'Intel(R) Core(TM) i7-1068NG7 CPU @ 2.30GHz',
```

## os.freemem()

This method returns the amount of free system memory in bytes as an integer.

```
console.log(os.freemem()); // 17179869184
```

To display in gigabytes, we can use the following code:

```
console.log(`Free memory: ${os.freemem() / 1024 / 1024 / 1024} GB`);
```

## os.totalmem()

This method returns the total amount of system memory in bytes as an integer.

```
console.log(os.totalmem()); // 17179869184
```

To display in gigabytes, we can use the following code:

```
console.log(`Total memory: ${os.totalmem() / 1024 / 1024 / 1024} GB`);
```

## os.homedir()

This method returns the home directory of the current user as a string.

```
console.log(os.homedir()); // /Users/username
```

## os.uptime()

This method returns the system uptime in seconds as an integer.

```
console.log(os.uptime()); // 123456
```

Convert to days, hours, minutes, and seconds:

```
const uptime = os.uptime();
const days = Math.floor(uptime / 60 / 60 / 24);
const hours = Math.floor(uptime / 60 / 60) % 24;
const minutes = Math.floor(uptime / 60) % 60;
const seconds = Math.floor(uptime) % 60;

console.log(
  'Uptime: ',
  `${days} days, ${hours} hours, ${minutes} minutes, ${seconds} seconds`
);
```

## os.hostname()

This method returns the hostname of the operating system as a string.

```
console.log(os.hostname()); // hostname
```

## os.networkInterfaces()

This method returns an object containing network interfaces that have been assigned a network address.

```
console.log(os.networkInterfaces());
```

# url and querystring Modules

---

These are two different modules, but they are related. The `url` module is used to parse and manipulate URLs. The `querystring` module is used to parse and manipulate query strings.

## url Module

Let's start with the `url` module.

### `url.parse()`

This method takes a URL string and returns an object. The object has properties for each part of the URL. The following code shows how to use this method:

```
const url = require('url');
```

```
const myURL = url.parse(  
  'https://www.example.com/listing?id=1000&premium=true'  
);  
console.log(myUrl);
```

```
{  
  host: 'example.com',  
  port: null,  
  hostname: 'example.com',  
  hash: null,  
  search: '?id=1000&premium=true',  
  query: 'id=1000&premium=true',  
  pathname: '/listing',  
  path: '/listing?id=1000&premium=true',  
  href: 'https://example.com/listing?id=1000&premium=true'  
}
```

As you can see, we get a lot of information here. We can use this information to manipulate the URL.

### `url.format()`

This method takes an object and returns a URL string. It is basically the opposite of `url.parse()`. The following code shows how to use this method:

```
const myURL2 = url.format({  
  protocol: 'https',
```

```
host: 'www.example.com',
pathname: 'listing',
query: {
  id: 1000,
  premium: true,
},
});
console.log(myURL2);
```

## querystring Module

The `querystring` module is used to parse and manipulate query strings. Query strings are the options that you see in a URL after the `?` character.

Let's create a variable with a query string with the year, month and day:

```
const myQueryString = 'year=2023&month=january&day=20';
```

### querystring.parse()

Now, we can use the `querystring.parse()` method to parse the query string into an object:

```
const q = querystring.parse(myQueryString);
console.log(q.month, q.day, q.year); // january 20 2017
```

We can get the query string from the google url in the previous example:

```
const googleQuery = querystring.parse(myURL.search.slice(1));
console.log(googleQuery.q); // how to parse url nodejs
```

### querystring.stringify()

We can use the `querystring.stringify()` method to convert an object into a query string:

```
const myQueryString2 = querystring.stringify({
  year: 2023,
  month: 'january',
  day: 20,
});

console.log(myQueryString2); // year=2023&month=january&day=20
```

So, both of these modules can be useful for certain tasks in certain applications.

# http module - Creating a Server

The `http` module is a core module that allows us to create a server and listen for requests. You can create a complete web server with just this module, however, it is not recommended. The `http` module is low-level and does not provide many features that are needed in a production environment. For example, it does not handle routing, sessions, or any static files. To create a production-ready web server, you would use a framework, such as Express. Express as well as other frameworks are built on top of the `http` module. We will be using Express in the next lesson.

## Creating a Server

To create a server, we use the `createServer` method. This method takes a callback function that is called every time a request is made to the server. The callback function takes two parameters, `request` and `response`. The `request` object contains information about the request, such as the URL, headers, and body. The `response` object is used to send a response back to the client. The `response` object has a method called `end` that takes a string as a parameter. This string is what is sent back to the client.

```
const http = require('http');

const server = http.createServer((request, response) => {
  response.end('Hello World');
});

server.listen(5000, () => {
  console.log('Server is listening on port 5000. Ready to accept requests!');
});
```

We now actually have a server running on our localhost on port 5000. So you can make requests to that server. You can use the browser, since it is a `GET` request by default. Or you can use a tool like Postman. Or you can use the `curl` command in the terminal.

```
curl localhost:5000
```

I would suggest using `Postman`. That's what I'll be using. You can get it from <https://www.postman.com/> for free.

If you are using your browser, open up the `network` tab in the developer tools. Then refresh the page. You should see a request to `localhost:5000`. If you click on it, you can see the response. You can also see the request and response headers.

Congratulations! You have created your first server! It doesn't do anything except say hello, but it is a server.

## Routing

The `http` module does not have any built-in routing. So we have to implement it ourselves. We can do this by checking the `request.url` property. If the URL matches a certain pattern, we can send a response back to the client. If the URL does not match, we can send a 404 response.

```
const server = http.createServer((request, response) => {
  const url = request.url;

  if (url === '/') {
    response.writeHead(200, { 'content-type': 'text/html' });
    response.end('<h1>Welcome</h1>');
  } else if (url === '/about') {
    response.writeHead(200, { 'content-type': 'text/html' });
    response.end('<h1>About Us</h1>');
  } else {
    response.writeHead(404, { 'content-type': 'text/html' });
    response.end('<h1>Page not found</h1>');
  }
});
```

It is important to note that when you change anything, you have to restart your server. You can do this by pressing `ctrl + c` in the terminal. Then run `node index.js` again.

Creating routes and doing other things without a framework can get very tedious. We would have to write a lot of `if` statements to handle all the different routes. What I have done is created a route for the index page, about and then a 404 page. If you go to `localhost:5000/about`, you should see the about page. If you go to `localhost:5000/anythingelse`, you should see the 404 page.

Remember we talked about status codes? The `200` response means `ok` so I used that on the index and about pages. The `404` response means `not found` so I used that on the 404 page.

## Returning an HTML File

We can also return an HTML file instead of a string. We can do this by using the `fs` module. The `fs` module allows us to read and write files. We can use the `readFile` method to read an HTML file and send it back to the client.

```
const fs = require('fs');

const server = http.createServer((request, response) => {
  const url = request.url;

  if (url === '/') {
    fs.readFile('index.html', (error, file) => {
      if (error) {
        console.log(error);
        response.writeHead(500, { 'content-type': 'text/html' });
        response.end('<h1>Sorry, we have a problem on our end</h1>');
      } else {
```

```

        response.writeHead(200, { 'content-type': 'text/html' });
        response.end(file);
    }
});

} else if (url === '/about') {
    response.writeHead(200, { 'content-type': 'text/html' });
    response.end('<h1>About Us</h1>');
} else {
    response.writeHead(404, { 'content-type': 'text/html' });
    response.end('<h1>Page not found</h1>');
}
});

```

You can go ahead and create an about and a 404 page if you want.

## Returning a JSON File

Node.js can of course server HTML files, but usually we want to create a backend API with Node.js. So we want to return JSON data. You saw earlier that we used public APIs like the GitHub API, the movie database, the chuck norris joke API and many others. They all served JSON data. We can put some JSON data in a file and return it to the client.

let's change the `about` route to `posts`. We will also get rid of the `fs` module. We will put some JSON with some blog posts at the top of the file. Then when we hit the endpoint of `http://localhost:5000/posts`, we will return the JSON data.

```

const posts = [
    { id: 1, title: 'Post One', body: 'This is post one' },
    { id: 2, title: 'Post Two', body: 'This is post two' },
];

const server = http.createServer((request, response) => {
    const url = request.url;

    if (url === '/') {
        response.writeHead(200, { 'content-type': 'application/json' });
        response.end(JSON.stringify({ success: true, message: 'Welcome' }));
    } else if (url === '/posts') {
        response.writeHead(200, { 'content-type': 'application/json' });
        response.end(JSON.stringify({ success: true, data: posts }));
    } else {
        response.writeHead(404, { 'content-type': 'application/json' });
        response.end(JSON.stringify({ success: false, error: 'Not found' }));
    }
});

```

When creating a response with JSON data, you have to set the `content-type` header to `application/json`. When formatting your JSON, it's common to have a `success` property that is a

boolean. If the request was successful, it will be `true`. If it was not successful, it will be `false`. You can also have an `error` property if the request was not successful. You can also have a `data` property if the request was successful. You can put whatever data you want in there. This is the convention that I like to use.

Now if you visit <http://localhost:5000/posts>, you should see the JSON data.

## Getting a single post

We can also get a single post. We can do this by using the `request.url` property. We can use the `split` method to split the URL into an array. Then we can get the last item in the array. That will be the id of the post. We can then use that id to get the post from the array.

```
const posts = [
  { id: 1, title: 'Post One', body: 'This is post one' },
  { id: 2, title: 'Post Two', body: 'This is post two' },
];

const server = http.createServer((request, response) => {
  const url = request.url;
  const id = url.split('/')[2];

  if (url === '/') {
    response.writeHead(200, { 'content-type': 'application/json' });
    response.end(JSON.stringify({ success: true, message: 'Welcome' }));
  } else if (url === '/posts') {
    response.writeHead(200, { 'content-type': 'application/json' });
    response.end(JSON.stringify({ success: true, data: posts }));
  } else if (url === `/posts/${id}`) {
    const post = posts.find((post) => post.id === Number(id));
    response.writeHead(200, { 'content-type': 'application/json' });
    response.end(JSON.stringify({ success: true, data: post }));
  } else {
    response.writeHead(404, { 'content-type': 'application/json' });
    response.end(JSON.stringify({ success: false, error: 'Not found' }));
  }
});
```

Now, if you go to <http://localhost:5000/posts/1>, you should see the first post. If you go to <http://localhost:5000/posts/2>, you should see the second post.

So this is the VERY beginning of creating an API. Like I said, usually you are not going to do it this way with 'vanilla Node.js'. It would be too tedious, you would need to write a ton of code and it wouldn't be as secure as if you used a framework. But this is just to give you an idea of how it works.

# Express API Setup

---

Now we are going to get into some back-end web development. We are going to build a simple REST API using Express. This will be a random ideas API, so the data will be ideas that people post. We'll keep it simple and just have the idea text, a tag, the date and a username. We won't be adding authentication or anything, even what we're doing now is way beyond what I planned for this course. We'll just add the username to the request.

We will implement **CRUD** functionality, which is create, read, update and delete. Express handles a lot of the heavy lifting for us, so we can focus on the important stuff.

Create a folder called `randomideas-api` and `cd` into it.

First, we need to install Express. Let's initialize our project with a `package.json` file:

```
npm init -y
```

Then we can install Express:

```
npm install express
```

Now we can create a file called `server.js`. You can call it absolutely anything you want, but `server.js` is a common name for the main file in a Node.js API.

Let's start by

- Importing Express
- Creating an instance of the `express` function
- Creating a route that responds to a `GET` request to the `/` path
- Listening on port 5000

```
const express = require('express');

const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(5000, () => {
  console.log('Server is listening on port 5000');
});
```

As you can see, this is much simpler and cleaner than the vanilla Node.js server we built in the previous lesson. We have this `app` object that we can use to create routes. We can also use it to listen for requests on a specific port, etc.

Now, let's add an NPM script to our `package.json` file so we can start the server with `npm start`:

```
"scripts": {  
  "start": "node server.js"  
},
```

Now you can run `npm start` to start the server. You should see the message `Server is listening on port 5000` in your terminal.

As far as data, I'm just going to keep the data in memory for now. Create an array with a few ideas right in the `server.js` file under where we initialized the `app` object:

```
const ideas = [  
  {  
    id: 1,  
    text: 'Positive NewsLetter, a newsletter that only shares positive,  
    uplifting news',  
    tag: 'Technology',  
    username: 'TonyStark',  
    date: '2022-01-02',  
  },  
  {  
    id: 2,  
    text: 'Milk cartons that turn a different color the older that your milk is  
    getting',  
    tag: 'Inventions',  
    username: 'SteveRogers',  
    date: '2022-01-02',  
  },  
  {  
    id: 3,  
    text: 'ATM location app which lets you know where the closest ATM is and if  
    it is in service',  
    tag: 'Software',  
    username: 'BruceBanner',  
    date: '2022-01-02',  
  },  
];
```

Let's create a route that responds to a `GET` request to the `/api/ideas` path. It will return an array of ideas. Add this to the bottom of your `server.js` file above the `app.listen()` call:

```
// Get all ideas
app.get('/api/ideas', (req, res) => {
  res.json(ideas);
});
```

The reason that I used `/api/ideas` instead of just `/ideas` is because I want to keep the API separate from the front-end. I don't want to have to worry about the front-end routes conflicting with the API routes.

All of our routes take in a callback with a request and response object. The request object contains information about the request, such as the path, query string, headers, etc. The response object contains methods for responding to the request, such as `send()`, `json()`, `status()`, etc. We are sending back a JSON response with the `res.json()` method. This will automatically set the `Content-Type` header to `application/json`.

Now we can test it out in Postman. Create a new GET request to `http://localhost:5000/api/ideas`. You should see the array of ideas in the response body.

Let's also create a route for individual ideas. We can use a route parameter to get the ID of the idea we want to get. Add this to the bottom of your `server.js` file:

```
// Get single idea
app.get('/api/ideas/:id', (req, res) => {
  const idea = ideas.find((idea) => idea.id === parseInt(req.params.id));

  if (!idea) {
    res.status(404).json({ success: false, error: 'Resource not found' });
  } else {
    res.json({ success: true, data: idea });
  }
});
```

We specified '`:id`' in the route. We can then access that route parameter using `req.params.id`.

Then we are using the `find()` method to find the idea with the ID that matches the route parameter. If we don't find a idea, we send a 404 status code and an error. If we do find an idea, we send it back as a successful JSON response.

You have to restart the server to see the changes. I will show you a tool to get around that soon.

Test it out in Postman. Create a new GET request to `http://localhost:5000/api/ideas/1`. You should see the idea with the ID of 1 in the response body.

Try going to `http://localhost:5000/api/ideas/4`. You should see the error message in the response body.

In the next lesson, I will show you how to use a package called `Nodemon` to automatically restart the server when you make changes. We will also clean up our code a bit and create a separate file for our routes.

# Nodemon & Routes Folder

---

So right now, we have to keep restarting our app everytime we make a change. This is not ideal. We can use a package called `nodemon` to automatically restart our app when we make changes.

Let's install as a dev dependency:

```
npm install -D nodemon
```

Now, in your `package.json` file, you can add a script to run your app with `nodemon`:

```
"scripts": {  
  "start": "node server.js",  
  "dev": "nodemon server.js"  
}
```

From now on, in development, you can run `npm run dev` to start your app with `nodemon` and you will not have to restart your app everytime you make a change.

## Cleaning up the routes

Right now we have our idea routes in the main `server.js` file. This is not ideal. We should move them to a separate file. Let's create a new folder called `routes` and inside that folder, create a new file called `ideas.js`.

In order to use the Express router, we need to import it:

```
const express = require('express');  
const router = express.Router();
```

Now, we can move our ideas and idea routes to this file. We do have to make a couple changes to the routes. We are using the router now, so we need to change the `app.get()` to `router.get()`. We also need to change the paths from `/api/ideas` to `/` because we are already in the `/api/ideas` route. The single idea route will be `/:id` because we are already in the `/api/ideas` route.

Lastly, we need to export the router (I always forget this 😊):

```
module.exports = router;
```

## Hook up routes folder

Now we need to hook up the routes folder. In your `server.js`, you can remove the 2 idea routes and add the following:

```
const ideasRouter = require('./routes/ideas');
app.use('/api/ideas', ideasRouter);
```

`app.use` is a way to add middleware to your app. In this case, we are adding the `ideasRouter` middleware to the `/api/ideas` route. This means that all the routes in the `ideasRouter` will be prefixed with `/api/ideas` automatically.

So now the api should work the same as before, but we have a cleaner `server.js` file. You can go ahead and test out the `/api/ideas` and `/api/ideas/:id` routes in postman.

In the next lesson, we will make it so that we can add ideas with a POST request.

# Handling POST Requests & Data

---

Now that we have a basic API set up, we can start adding more functionality. We will start by adding the ability to create ideas with a POST request. This means that we need to send some data with the request.

In the `routes/ideas.js` file, we will add a new route to handle the POST request:

```
// Create idea
router.post('/ideas', (req, res) => {
  const idea = {
    id: ideas.length + 1,
    text: req.body.text,
    tag: req.body.tag,
    username: req.body.username,
    date: new Date().toISOString().slice(0, 10),
  };

  ideas.push(idea);

  res.json({ success: true, data: idea });
});
```

We are not using a database, so we are just going to add the idea to an array. We will also send back the idea that was created. Usually, when you integrate a database, it will create the ID automatically, but we are just going to increment the ID by 1.

The data that we send with our HTTP request will have the data of the idea. We can access this data with `req.body`. In order to do this, we do have to add a piece of middleware to our app. In the `server.js` file, add the following:

```
// Body parser middleware
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
```

This will allow us to access the data that we send with the request. Now, we can test out the POST request in Postman. In the body, we can add the text, tag and username of the idea. The endpoint stays the same - `http://localhost:5000/api/ideas`. However, we will also need to change the request type to POST.

## Sending data to the server

We need to send our idea data in the request body. In Postman, we can do this by clicking on the body tab and selecting `x-www-form-urlencoded`. We can then add the text, tag and username of the idea.

When you send your request, you should get a response with the idea that was created.

Remember, your data is just being saved in memory. If you restart the server, the data will be lost.

If you make a GET request right after, you will see that the idea was added to the array.

If we were using a database, most of this would be the same. We make the requests the same way, etc. It would actually be easier once the database is all configured, because we don't have to worry about the IDs. Also, there are methods we could use like `findByIdAndUpdate` to update the idea. Here, we are doing it manually.

In the next lesson, we will add the update and delete functionality

# Handling PUT & DELETE Requests

---

We can read and create ideas via our API, now let's add the functionality to update and delete ideas.

## Updating Ideas

In the `routes/ideas.js` file, we will add a new route to handle the PUT request:

```
// Update idea
router.put('/:id', (req, res) => {
  const idea = ideas.find((idea) => idea.id === parseInt(req.params.id));

  if (!idea) {
    res.status(404).json({ success: false, error: 'Resource not found' });
  } else {
    idea.text = req.body.text || idea.text;
    idea.tag = req.body.tag || idea.tag;

    res.json({ success: true, data: idea });
  }
});
```

We are going to find the idea that we want to update by the ID. If the idea is not found, we will send back an error. If the idea is found, we will update the text and/or tag of the idea. We are not going to allow username updates. We will then send back the updated idea.

Try it out in Postman. Make a `PUT` request to `http://localhost:5000/api/ideas/1`. Remember to choose `PUT` as the request type and to send the `text` and `tag` in the request body.

## Deleting Ideas

In the `routes/ideas.js` file, we will add a new route to handle the `DELETE` request:

```
// Delete idea
router.delete('/:id', (req, res) => {
  const idea = ideas.find((idea) => idea.id === parseInt(req.params.id));

  if (!idea) {
    res.status(404).json({ success: false, error: 'Resource not found' });
  } else {
    const index = ideas.indexOf(idea);
    ideas.splice(index, 1);

    res.json({ success: true, data: {} });
  }
});
```

---

We are going to find the idea that we want to delete by the ID. If the idea is not found, we will send back an error. If the idea is found, we will find the index of the idea in the array and remove it. We will then send back an empty object.

Now, make a **DELETE** request to <http://localhost:5000/api/ideas/1>. Remember to choose DELETE as the request type.



We now have a complete **CRUD** REST API. We can create, read, update and delete ideas.

# What Is MongoDB?

---

Alright, so we have really reached a point in the course where this is totally optional and well beyond JavaScript fundamentals. However, if you're interested in becoming a full-stack developer, you'll need to know about databases. A database is technically and organized collection of data, but a lot of times, we use the term "database", when talking about [database management systems](#). An example of a database management system is [MongoDB](#) or [MySQL](#) or [PostgreSQL](#). These work in different ways, but the goal is the same and that is to store and manage data. In our case, we need a place to store the ideas for our application/api. We're going to use [MongoDB](#).

Database systems can run and operate on a single file system or across multiple nodes or clusters. There are also different types of databases that store and retrieve data in different ways. For instance a MySQL database uses tables and columns while MongoDB uses collections and documents.

MongoDB is a document database. It stores data in JSON-like documents. A Document Database is a type of NoSQL database, which means it does not use SQL to query data, so it's different than a relational database such as MySQL. There are pros and cons to every database including MongoDB, MySQL, Postgres and so on. NoSQL databases are typically faster and easier to scale than SQL databases. However, they are not as mature and do not have as many features as SQL databases. So it's really up to you as well as the project that you're working on to decide which database to use.

MongoDB is very popular in the JavaScript world. I think one of the reasons for that is because it is structured similarly to JavaScript objects. It's also very easy to get started with. You have probably heard of the [MERN](#) stack, which is a popular stack for building full-stack applications. [MERN](#) stands for MongoDB, Express, React, and Node. There is also the [MEAN](#) and [MEVN](#) stacks, where they use Angular or Vue instead of React.

## Collections & Documents

MongoDB stores data in collections and documents. A collection is a group of documents. A document is a set of key-value pairs. It's essentially JSON. It's actually something called BSON, which is a superset of JSON. If you're familiar with relational databases like MySQL, you can think of a collection as a table and a document as a row or record. This could be an example of a document in a users collection.

```
{  
  "name": "John Doe",  
  "email": "john@gmail.com",  
  "password": "SOME_HASHED_PASSWORD"  
}
```

## MongoDB Atlas

Another reason I like to use MongoDB is because they have an incredible cloud-hosted version called [Atlas](#). So you don't even have to install MongoDB on your own server. It's simple to get started with. We are going to go through this process in the next video. It is absolutely free to get started. you get a small

amount of storage and a small amount of bandwidth. However, if you want to scale up, you can do that as well.

## MongoDB Compass

Another tool that I like to use is called [MongoDB Compass](#). It's a GUI for MongoDB. It's a free tool that you can use to view your data. It's very easy to use. You can also use it to import and export data. So, it's a very powerful tool. I'll show you how to get that setup as well.

## Mongoose

When it comes to integrating Mongo into your app or API that uses Node.js, there is a tool called [Mongoose](#) that is extremely powerful and very popular. It's a library that makes it easy to work with MongoDB. You create a [model](#) of your data and then you can use that model to create, read, update, and delete data from your database. You can also do things like validation and data sanitization. So, it's a very powerful tool. We'll be using Mongoose, but we'll just be scratching the surface of what it can do. Mongoose is installed using NPM.

# MongoDB Atlas Setup

---

Like I said in the last lesson, Atlas is the cloud version of MongoDB and it is incredibly popular and I would say much more popular than the on-premise version of MongoDB. So, we're going to go through the process of setting up an account and creating a cluster.

## Create an Account

Go to <https://www.mongodb.com/cloud/atlas> and click the **Try Free** button.

Fill out the form and click **Create Account**. You can also sign up with Google.

## Create an Organization

Once you log in, create an organization. You can name it whatever you want. I'm going to name mine **Traversy Media**.

## Create a Project

Now, create a project. I'm going to name mine **Node API**.

## Create a Database

Click on 'Build a Database' and select the "Shared" plan, which is 100% free. Keep the default selections, which includes AWS for the provider. You can name your cluster whatever you want. I'm going to name mine **node-api-cluster**. Click **Create Cluster**.

## Create a Database User

Add a username and password and click "create user"

## Whitelist Your IP Address

Choose "My Local Environment" and then click **Add IP Address**. This will add your current IP address to the whitelist. Click **Confirm**.

It may take a few minutes for your cluster to be created. Once it's created, you can click on 'Browse Collections' and then 'Add My Own Data'. Enter a database name. I just used 'ideas\_db'. Also, and a collection name of 'ideas'.

Just a note, when you create new collections, you do not have to do it here, you can do it in your application.

You will be able to see all of your data from here. You can also edit data, however if you want to manage data directly from your application, I would suggest a tool called **MongoDB Compass**. We'll go over that a little later.

For now, let's go back to the Atlas dashboard/overview and click on [Connect](#). Click on [Connect Your Application](#) and select [Node.js](#) from the dropdown. Copy the connection string and paste it anywhere for now. Mine looks like this

```
mongodb+srv://brad:@node-api-cluster.4nsqqlb.mongodb.net/ideas_db?retryWrites=true&w=majority
```

Change [`<password>`](#) to your password and add whatever you named your database after the last slash. I called mine 'ideas\_db'.

In the next lesson, we will connect to the database via our application.

# Connecting To MongoDB

---

Now that we have our cloud database hooked up to our application, we need to connect to it. We'll use the [Mongoose](#) library to do this.

## Installing Mongoose & dotenv

We are actually going to install two packages right now. One is Mongoose, which is called an "ODM" or "Object Document Mapper". This is a library that allows us to connect to our MongoDB database and interact with it. The other is called [dotenv](#), which is a library that allows us to store environment variables in a file called `.env`. We'll use this to store our MongoDB connection string.

```
npm install mongoose dotenv
```

## .env Setup

Let's add our MongoDB connection string as an environment variable. Create a file called `.env` in your root folder. Add the connection string that you copied from MongoDB Atlas in the last lesson. It should look something like this:

```
MONGODB_URI=mongodb+srv://<username>:<password>@cluster0-0z0z0.mongodb.net/ideas_db?retryWrites=true
```

I also add my port here. I'm going to use port 5000. You can use whatever port you want.

```
PORT=5000
```

Now, in the `server.js` file, we need to require the dotenv package and call the config method. This will load the environment variables from the `.env` file.

```
require('dotenv').config();
```

Now, I will create a variable called `PORT` and set it to the value of the `PORT` environment variable. If it doesn't exist, I will set it to 5000. Then, just use that in the `app.listen` method.

```
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server is listening on port ${PORT}`);
});
```

## Connecting To MongoDB

Now, let's connect to MongoDB. We'll use the Mongoose library to do this. Instead of putting this code directly in the `server.js` file, I'm going to create a folder called `config` and in it, create a new file called `db.js`. This will be a separate file that will handle all of our database connection logic.

```
const mongoose = require('mongoose');

const connectDB = async () => {
  const conn = await mongoose.connect(process.env.MONGODB_URI);
  console.log(`MongoDB Connected: ${conn.connection.host}`);
};

mongoose.set('strictQuery', true);

module.exports = connectDB;
```

Now, in the `server.js` file, we need to require the `connectDB` function and call it.

```
const connectDB = require('./config/db');

connectDB();
```

Now when you start your server, you should see a message that says "MongoDB Connected".

Congrats, you've connected to your MongoDB database!

# Mongoose Data Model

---

With Mongoose, we can create a data model that will define the structure of our documents. This is similar to how we defined the structure of our tables in SQL. We can also define the data types of each field, as well as add validation to each field.

Create a folder called `models` and a new file called `Idea.js` inside of it. In this file, we will define our data model and schema.

```
const mongoose = require('mongoose');

const IdeaSchema = new mongoose.Schema({
  text: {
    type: String,
    required: [true, 'Please add a text field'],
  },
  tag: {
    type: String,
  },
  username: {
    type: String,
  },
  date: {
    type: Date,
    default: Date.now,
  },
});

module.exports = mongoose.model('Idea', IdeaSchema);
```

In this file, we are importing the `mongoose` module and creating a new schema. We are then exporting the model using the `mongoose.model()` method. The first argument is the name of the model, and the second argument is the schema. The schema is just a collection of fields that we want to have in our documents.

We can now use this model to query our database, which we will do in the next lesson.

# Database Queries

---

Now that we have Mongoose setup and a data model for our ideas, we can use really easy methods to query our database. We will be using the `Idea` model that we created in the last lesson.

Let's go into our `routes/ideas.js`

We will start by importing the `Idea` model:

```
const Idea = require('../models/Idea');
```

## Async/await

Mongoose methods return promises, so we can use `async/await` to handle the promises. We will be using `async/await` in all of our routes. So before the callback function, we will add `async` and then we can use `await` before the Mongoose method.

## Get All Ideas

Now in the GET / route, let's get the ideas from the database and send them instead of the hardcoded ones. Yes, I know there are not any in there yet, but that's ok, we will still just get an empty array.

We can do this easily by using the `find()` method.

```
// Get all posts
router.get('/', async (req, res) => {
  try {
    const ideas = await Idea.find();
    res.json({ success: true, data: ideas });
  } catch (err) {
    res.json({ success: false, message: err });
  }
});
```

Now, go to Postman and go to the route `http://localhost:5000/api/ideas`. You should see an empty array.

## Create an Idea

Now let's add a new idea to the database. We can use the `save()` method to save the idea to the database.

```
// Create idea
router.post('/', async (req, res) => {
```

```

const idea = new Idea({
  text: req.body.text,
  tag: req.body.tag,
  username: req.body.username,
});

try {
  const savedIdea = await idea.save();
  res.json({ success: true, data: savedIdea });
} catch (err) {
  res.json({ success: false, message: err });
}
);

```

Now, go to Postman and create a new POST request to <http://localhost:5000/api/ideas>. Be sure to add the text, tag and username in the body of the request. You should get a response with the idea that was created.

Now, make a get request to <http://localhost:5000/api/ideas> and you should see the idea that you just created.

It has the text, tag, username, date and an `_id` field. This is the unique identifier for the idea. We will use this to get a single idea and to delete an idea. You will also see `_v`, this is the version number of the document. This is used by Mongoose to keep track of changes to the document.

## Single Idea

Let's update the route that gets a single idea by it's ID. We can use the `findById()` method to get a single idea by it's ID.

```

// Get single idea
router.get('/:id', async (req, res) => {
  try {
    const idea = await Idea.findById(req.params.id);
    res.json({ success: true, data: idea });
  } catch (err) {
    res.json({ success: false, message: err });
  }
});

```

Go to Postman and make a GET request to [http://localhost:5000/api/ideas/ID\\_OF\\_IDEA](http://localhost:5000/api/ideas/ID_OF_IDEA). You should get the idea that you created.

## Update Idea

Now for the update route, we will use `findByIdAndUpdate()` to update the text and tag

```
// Update idea
router.put('/:id', async (req, res) => {
  try {
    const updatedIdea = await Idea.findByIdAndUpdate(
      req.params.id,
      {
        $set: {
          text: req.body.text,
          tag: req.body.tag,
          username: req.body.username,
        },
      },
      { new: true }
    );
    res.json({ success: true, data: updatedIdea });
  } catch (err) {
    res.json({ success: false, message: err });
  }
});
```

Now, make a PUT request to [http://localhost:5000/api/ideas/ID\\_OF\\_IDEA](http://localhost:5000/api/ideas/ID_OF_IDEA) and update the title and body. You should get the updated idea back.

## Delete Idea

Finally, we will add the code to delete the idea. We will use the `findByIdAndDelete()` method to delete the idea.

```
// Delete idea
router.delete('/:id', async (req, res) => {
  try {
    await Idea.findByIdAndDelete(req.params.id);
    res.json({ success: true, data: {} });
  } catch (err) {
    res.json({ success: false, message: err });
  }
});
```

We now have a fully functional API that can create, read, update and delete ideas from the database.

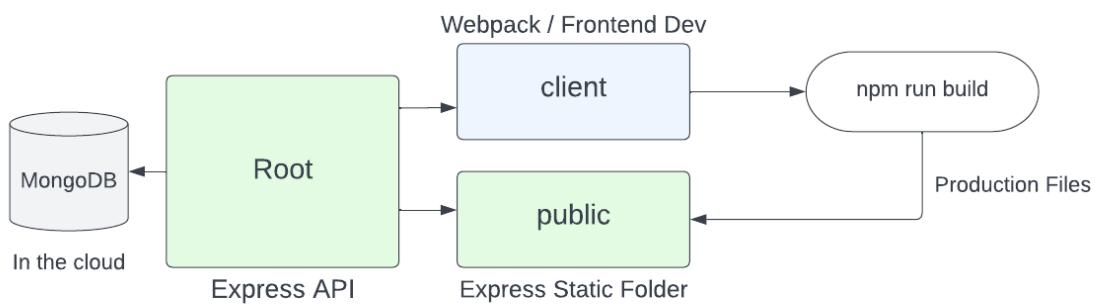
In the next section, we will create a simple front-end to interact with our API.

# Fullstack Workflow

---

Now that we have our API done, we are going to start working on the front end. Before we jump into the code, I just want to go over the structure and the workflow of our project.

Here is a diagram of the structure of our project:



We already created the backend, which consists of the root folder with our Express API as well as the MongoDB database.

Now, we will create a `client` folder. This will be our frontend development folder. We will use the [webpack-starter](#) as a base for our frontend.

We will configure Webpack to build our production files into a `public` folder in the root. This will be the folder that we will serve our static files from. With Express, we can serve static files from any folder. We will use this folder to serve our production `index.html` file as well as our bundled JavaScript and CSS files.

Let's start setting up the `client` folder in the next lesson.

# Client Folder Setup

---

In the last lesson, I explained the structure of our app, now we will implement it. I renamed my api folder from `randomideas-api` to `randomideas-app`, because now it will be a fullstack app. I also created a `client` folder in the root of the project.

## Setting the Static Folder

Express has a built in middleware function called `express.static` that allows us to serve static files from any folder. We will use this to serve our production files from the `public` folder.

Create a `public` folder in the root and add the following code to your `server.js` file:

```
const path = require('path');
// ...

// Static Folder
app.use(express.static(path.join(__dirname, 'public')));
```

Now, if you add an `index.html` file to the `public` folder, you can access it at <http://localhost:5000/index.html>.

## Client Folder & Webpack Starter

I am going to use the Webpack starter that we created a few lessons ago. Create a folder called `client` and copy the contents of the `webpack-starter` folder, including the `src` folder into your `client` folder. You should not have a `dist` folder in your `client` folder. Delete it if you have one. `public` will be the new `dist` folder. Also, do not copy any `node_modules` folder. We will install our dependencies after.

From the client, install your frontend dependencies:

```
cd client
npm install
```

## Setting the Build Folder

In your `webpack.config.js` file, change the `output` path to

```
output: {
  path: path.resolve(__dirname, '../public'),
  filename: 'bundle.js',
},
```

and for the devServer path...

```
devServer: {  
  static: {  
    directory: path.resolve(__dirname, '../public'),  
  }  
},
```

Now, when you run `npm run build`, your files will be built into the `public` folder. You can try it now. Open a new terminal and cd into your `client` folder. Run `npm run build` and you should see your files in the `public` folder.

Now, run your devserver from the client folder:

```
npm run dev
```

So at this point, you should have one terminal running your API (localhost:5000) and another terminal running your devserver (localhost:3000).

## RandomIdeas Theme

In this lesson, you will have a zip file called `randomideas-theme.zip`. It is just an index.html and a style.css. Add the index.html to the client, overwriting the one from `webpack-starter` and add the style.css to the `src/css` folder.

In the index.html, remove the link to the style.css. We can import that into our JS. Also, remove the script tag for the bundle.js. That is automatically added at build time.

Now, in your `index.js` file, import the style.css file:

```
import '../css/style.css';
```

Delete the import of `message.js` and delete that file. You should now just see the hardcoded ideas in the html file.

# Modal Component

---

Now, we want to create a Modal that includes a form to add a new idea. We are not using Bootstrap or anything, so we need to do this from scratch. The CSS already includes the transition and classes to show and hide the modal.

We are breaking everything into components as if we were using a framework. Let's create a folder called `components` and a file called `Modal.js` inside that folder.

Add the following:

```
class Modal {  
  constructor() {  
    this._modal = document.querySelector('#modal');  
    this._modalBtn = document.querySelector('#modal-btn');  
    this.addEventListeners();  
  }  
  
  addEventListeners() {  
    this._modalBtn.addEventListener('click', this.open.bind(this));  
    window.addEventListener('click', this.outsideClick.bind(this));  
  }  
  
  open() {  
    this._modal.style.display = 'block';  
  }  
  
  close() {  
    this._modal.style.display = 'none';  
  }  
  
  outsideClick(e) {  
    if (e.target === this._modal) {  
      this.close();  
    }  
  }  
}  
  
export default Modal;
```

We are using the `querySelector` method to get the modal and the button in the constructor. We are also adding an event listener to the button to open the modal and to the window to close the modal if the user clicks outside of it. We are using the `bind` method to make sure that the `this` keyword is referring to the class. We have a method to open the modal and a method to close the modal. We also have a method to check if the user clicked outside of the modal and close it if they did. Finally, we are exporting the class so we can use it in other files.

Now bring it into your 'index.js' file and create an instance of the class.

```
import Modal from './components/Modal';
new Modal();
```

Now you should be able to click on the modal button and open the modal.

# IdeaForm Component

---

Now that we have a Modal component, I also want to have a component for the form. Let's create a file called `IdeaForm.js` inside the `components` folder.

Add the following code:

```
import IdeaList from './IdeaList';

class IdeaForm {
  constructor() {
    this._formModal = document.querySelector('#form-modal');
  }

  addEventListeners() {
    this._form.addEventListener('submit', this.handleSubmit.bind(this));
  }

  handleSubmit(e) {
    e.preventDefault();

    if (
      !this._form.elements.text.value ||
      !this._form.elements.tag.value ||
      !this._form.elements.username.value
    ) {
      alert('Please enter all fields');
      return;
    }

    const idea = {
      text: this._form.elements.text.value,
      tag: this._form.elements.tag.value,
      username: this._form.elements.username.value,
    };

    console.log(idea);

    // Clear fields
    this._form.elements.text.value = '';
    this._form.elements.tag.value = '';
    this._form.elements.username.value = '';

    this.render();

    document.dispatchEvent(new Event('closemodal'));
  }

  render() {
```

```

this._formModal.innerHTML = `
<form id="idea-form">
<div class="form-control">
  <label for="idea-text">Enter a Username</label>
  <input type="text" name="username" id="username" value="${localStorage.getItem('username') ? localStorage.getItem('username') : ''}" />
</div>
<div class="form-control">
  <label for="idea-text">What's Your Idea?</label>
  <textarea name="text" id="idea-text"></textarea>
</div>
<div class="form-control">
  <label for="tag">Tag</label>
  <input type="text" name="tag" id="tag" />
</div>
<button class="btn" type="submit" id="submit">Submit</button>
</form>
`;
this._form = document.querySelector('#idea-form');
this.addEventListener();
}
}

export default IdeaForm;

```

So we created a class for the form component and we are rendering the HTML from our JavaScript. We are also adding event listeners to the form. We are also storing the form in a property so we can access it later.

We need the modal to close when the form is submitted. We can do this by dispatching an event from the form component. We can listen for this event in the modal component.

```
document.dispatchEvent(new Event('closemodal'));
```

Now, in the `components/Modal.js` file, we can listen for this event and close the modal. Add this to the `addEventListener` method:

```
document.addEventListener('closemodal', () => this.close());
```

Go to your HTML and remove the form from the modal. We are going to render it with JavaScript.

```
<div id="modal" class="modal">
  <div id="form-modal" class="modal-box"></div>
</div>
```

Bring in your IdeaForm component in the `index.js` file and render it.

```
import IdeaForm from './components/IdeaForm';
const ideaForm = new IdeaForm();
ideaForm.render();
```

Now your form should display and submit.

# IdeaList Component

---

Before we fetch our ideas from the API, let's create a component for the list of ideas. Create a file called `IdeaList.js` inside the `components` folder. For now, we will just use dummy data in the file.

Add the following code:

```
class IdeaList {
  constructor() {
    this._idealListEl = document.querySelector('#idea-list');
    this._ideas = [
      {
        id: 1,
        text: 'Idea 1',
        tag: 'Business',
        username: 'John',
        date: '02/01/2023',
      },
      {
        id: 2,
        text: 'Idea 2',
        tag: 'Technology',
        username: 'Jill',
        date: '02/01/2023',
      },
    ];
  }

  render() {
    this._idealListEl.innerHTML = this._ideas
      .map((idea) => {
        const tagClass = this.getTagClass(idea.tag);
        return `
          <div class="card">
            <button class="delete"><i class="fas fa-times"></i></button>
            <h3>
              ${idea.text}
            </h3>
            <p class="tag tag-technology">Technology</p>
            <p>
              Posted on <span class="date">${idea.date}</span> by
              <span class="author">${idea.username}</span>
            </p>
          </div>
        `;
      })
      .join('');
  }
}
```

```
export default IdeaList;
```

Now, delete the `div` elements with the class `card` from the `index.html` file. We will render the ideas using the `IdeaList` component.

In your `index.js` file, import the `IdeaList` component and create an instance of it. Call the `render` method on the instance.

```
import IdeaList from './components/IdeaList';

const idealist = new IdeaList();
idealist.render();
```

## Colored Tags

Now, let's add some color to the tags. We will use the `tag` property of the idea to determine the color of the tag. We will create a method called `getTagClass` that will return the class name based on the tag. I already have classes in the css for tag-technology, tag-inventions and a few others. We will put the allowed tags in a `set`.

```
class IdeaList {
  constructor() {
    this._idealiste1 = document.querySelector('#idea-list');
    this._ideas = [
      {
        id: 1,
        text: 'Idea 1',
        tag: 'Business',
        username: 'John',
        date: '02/01/2023',
      },
      {
        id: 2,
        text: 'Idea 2',
        tag: 'Technology',
        username: 'Jill',
        date: '02/01/2023',
      },
    ];
    this._validTags = new Set();
    this._validTags.add('technology');
    this._validTags.add('software');
    this._validTags.add('business');
    this._validTags.add('education');
    this._validTags.add('health');
    this._validTags.add('inventions');
  }
}
```

```

getTagClass(tag) {
  tag = tag.toLowerCase();
  let tagClass = '';
  if (this._validTags.has(tag)) {
    tagClass = `tag-${tag}`;
  } else {
    tagClass = '';
  }
  return tagClass;
}

render() {
  this._idealListEl.innerHTML = this._ideas
    .map((idea) => {
      const tagClass = this.getTagClass(idea.tag);
      return `
        <div class="card">
          <button class="delete"><i class="fas fa-times"></i></button>
          <h3>
            ${idea.text}
          </h3>
          <p class="tag ${tagClass}">${idea.tag.toUpperCase()}</p>
          <p>
            Posted on <span class="date">${idea.date}</span> by
            <span class="author">${idea.username}</span>
          </p>
        </div>
      `;
    })
    .join('');
}

export default IdealList;

```

Now, the tags that are in the set, will have a color. The rest will be black.

# API Service

---

Now, we will fetch our data from the backend. In the `ideaList` component, delete the hardcoded objects and make `ideas` an empty array.

```
this._ideas = [];
```

Create the file `services/ideasApi.js` and add the following code:

```
import axios from 'axios';

class IdeasApi {
  constructor() {
    this._apiUrl = 'http://localhost:5000/api/ideas';
  }

  getIdeas() {
    return axios.get(this._apiUrl);
  }
}

export default new IdeasApi();
```

Now, since we are using `axios` to make the API call, we need to install it. Run the following command in the terminal:

```
npm install axios
```

In the `index.js` file, import the `ideasApi` service and call the `getIdeas` method and add the following code.

```
import IdeasApi from '../services/ideasApi';

class IdeaList {
  constructor() {
    this._ideaListEl = document.querySelector('#idea-list');
    this._ideas = [];
    this.getIdeas();

    this._validTags = new Set();
    this._validTags.add('technology');
    this._validTags.add('software');
```

```

        this._validTags.add('business');
        this._validTags.add('education');
        this._validTags.add('health');
        this._validTags.add('inventions');
    }

    async getIdeas() {
        try {
            const res = await IdeasApi.getIdeas();
            this._ideas = res.data.data;
            this.render();
        } catch (error) {
            console.log(error);
        }
    }

    getTagClass(tag) {
        tag = tag.toLowerCase();
        let tagClass = '';
        if (this._validTags.has(tag)) {
            tagClass = `tag-${tag}`;
        } else {
            tagClass = '';
        }
        return tagClass;
    }

    render() {
        this._idealListEl.innerHTML = this._ideas
            .map((idea) => {
                const tagClass = this.getTagClass(idea.tag);
                return `
                    <div class="card">
                        <button class="delete"><i class="fas fa-times"></i></button>
                        <h3>
                            ${idea.text}
                        </h3>
                        <p class="tag ${tagClass}">${idea.tag.toUpperCase()}</p>
                        <p>
                            Posted on <span class="date">${idea.date}</span> by
                            <span class="author">${idea.username}</span>
                        </p>
                    </div>
                `;
            })
            .join('');
    }

    export default IdeaList;

```

We are calling the `getIdeas` method in the constructor. This will fetch the ideas from the API and render them on the page.

## Enabling CORS

You will probably get an error that says your request is being blocked by CORS. This is because the API is running on a different port than the frontend. To fix this, we need to enable CORS in the backend. In the `server.js` file, add the following code:

```
const cors = require('cors');

// cors middleware
app.use(
  cors({
    origin: ['http://localhost:5000', 'http://localhost:3000'],
    credentials: true,
  })
);
```

You need to install cors in the backend, so make sure you are in the root folder in the terminal and type

```
npm install cors
```

Also, when you start your backend server, do not use Nodemon. Start or restart it with `npm start`

Now, when you make your request from your frontend, you should see the ideas. If you don't, check the console for any errors.

You can also open up postman and make a POST request to add an idea and you should see it reflect in the browser.

# Create Idea via Form & POST Request

---

Now, we want to be able to submit our form and create a new idea. We will use the `POST` method to do this. We will also use the `axios` library to make the API call.

In the `services/ideasApi.js` file, add the following method to the class:

```
createIdea(data) {
  return axios.post(this._ apiUrl, data);
}
```

Now go to `components/ideaForm.js` and add the following code:

```
import IdeasApi from '../services/ideasApi';
```

Make the `handleSubmit` method `async` and add the following code:

```
async handleSubmit(e) {
  e.preventDefault();

  const idea = {
    text: this._form.elements.text.value,
    tag: this._form.elements.tag.value,
    username: this._form.elements.username.value,
  };

  // Add idea to server
  const newIdea = await IdeasApi.createIdea(idea);

  // Clear fields
  this._form.elements.text.value = '';
  this._form.elements.tag.value = '';
  this._form.elements.username.value = '';

  document.dispatchEvent(new Event('closemodal'));
}
```

Update the DOM

We also want the DOM to get updated, so let's create a method in the `ideaList` component

```
addIdeaToList(idea) {
  this._ideas.push(idea);
  this.render();
}
```

Now, bring in the `ideaList` component in the `components/ideaForm.js` file and add the following code right under where we called the `IdeasApi.createIdea` method:

```
import IdeaList from './IdeaList';

constructor() {
  this._formModal = document.querySelector('#form-modal');
  this._ideaList = new IdeaList(); // <--- Add this line
}

// Add idea to list
this._ideaList.addIdeaToList(newIdea.data.data);
```

Now, try and add a new idea via the form.

# Save Username to Local Storage

---

We want the username that is put into the first idea submit into local storage and then pre-fill the username field with that value.

Go into `components/ideaForm.js` and add the following code:

```
async handleSubmit(e) {
  e.preventDefault();

  if (
    !_form.elements.text.value ||
    !_form.elements.tag.value ||
    !_form.elements.username.value
  ) {
    alert('Please enter all fields');
    return;
  }

  // Save user to local storage
  localStorage.setItem('username', _form.elements.username.value); // <-
  -- Add this line

  // ...

  // Render the form again
  this.render();

  document.dispatchEvent(new Event('closemodal'));
}
```

## Pre-fill The Username Field

In the `render` method, add the following code:

```
render() {
  this._formModal.innerHTML = `
<form id="idea-form">
<div class="form-control">
  <label for="idea-text">Enter a Username</label>
  <input type="text" name="username" id="username" value="${localStorage.getItem('username') ? localStorage.getItem('username') : ''}" />
</div>
<div class="form-control">
  <label for="idea-text">What's Your Idea?</label>
```

```
    <textarea name="text" id="idea-text"></textarea>
  </div>
  <div class="form-control">
    <label for="tag">Tag</label>
    <input type="text" name="tag" id="tag" />
  </div>
  <button class="btn" type="submit" id="submit">Submit</button>
</form>
`;
this._form = document.querySelector('#idea-form');
this.addEventListeners();
}
```

Now, it will fetch the username from local storage and pre-fill the username field.

# Add Username to Server

---

We are going to jump back into the server and check for the username in the http request and match it to the username of the idea. If it matches, we will update/delete. If not, we will send a 403 error.

Go to the backend `routes/idea.js` file and add update the delete request function

```
// Delete idea
router.delete('/:_id', async (req, res) => {
  try {
    const idea = await Idea.findById(req.params.id);

    // Match the usernames
    if (idea.username === req.body.username) {
      await Idea.findByIdAndUpdateDelete(req.params.id);
      return res.json({ success: true, data: {} });
    }

    // Usernames do not match
    res.status(403).json({
      success: false,
      error: 'You are not authorized to delete this resource',
    });
  } catch (error) {
    console.log(error);
    res.status(500).json({ success: false, error: 'Something went wrong' });
  }
});
```

We want to do the same thing with the update:

```
// Update idea
router.put('/:_id', async (req, res) => {
  try {
    const idea = await Idea.findById(req.params.id);

    // Match the usernames
    if (idea.username === req.body.username) {
      const updatedIdea = await Idea.findByIdAndUpdate(
        req.params.id,
        {
          $set: {
            text: req.body.text,
            tag: req.body.tag,
          },
        },
        { new: true }
      );
      res.json(updatedIdea);
    } else {
      res.status(403).json({
        success: false,
        error: 'You are not authorized to update this resource',
      });
    }
  } catch (error) {
    console.log(error);
    res.status(500).json({ success: false, error: 'Something went wrong' });
  }
});
```

```
);

    return res.json({ success: true, data: updatedIdea });
}

// Usernames do not match
res.status(403).json({
    success: false,
    error: 'You are not authorized to update this resource',
});

} catch (error) {
    console.log(error);
    res.status(500).json({ success: false, error: 'Something went wrong' });
}
});
```

Now, you can try to make an update or delete from Postman and see if it works. you need to send the username in the request body and it has to match the idea username.

# Delete Functionality

Now we want to be able to delete ideas from the client. Let's start by adding the delete to the API service. I am also going to add an update, even though we are not going to use it in this project.

```
updateIdea(id, data) {
  return axios.put(`.${this._ apiUrl}/${id}`, data);
}

deleteIdea(id) {
  const username = localStorage.getItem('username')
  ? localStorage.getItem('username')
  : '';
  return axios.delete(`.${this._ apiUrl}/${id}`, {
    data: {
      username,
    },
  });
}
```

Now in the `components/ideaList.js` file, let's add an `addEventListeners` function that will add the event listeners to the delete buttons.

```
addEventListeners() {
  this._idealstEl.addEventListener('click', (e) => {
    if (e.target.classList.contains('fa-times')) {
      e.stopImmediatePropagation();
      const ideaId = e.target.parentElement.parentElement.dataset.id;
      this.deleteIdea(ideaId);
    }
  });
}
```

We used `stopImmediatePropagation` to stop the event from bubbling up to the parent element. We also get the idea id from the dataset on the parent element. We don't have an `data-id` attribute on the element yet, so let's add it.

In the `render` function, add the `data-id` attribute to the `div` with the class of `card`:

```
<div class='card' data-id='${idea._id}'>
  ${deleteBtn}
  <h3>${idea.text}</h3>
  <p class='tag ${tagClass}'>${idea.tag.toUpperCase()}</p>
  <p>
```

```
    Posted on <span class='date'>${idea.date}</span> by
    <span class='author'>${idea.username}</span>
  </p>
</div>
```

Now, create the `deleteIdea` function:

```
async deleteIdea(ideaId) {
  try {
    // Delete from server
    const res = await IdeasApi.deleteIdea(ideaId);
    this._ideas.filter((idea) => idea._id !== ideaId);
    this.getIdeas();
  } catch (error) {
    alert('You can not delete this resource');
  }
}
```

We just filtered out the idea from the array and then called the `getIdeas` function to re-render the list.

## Hide Delete Button

Let's only show the delete button to people that own that idea. We can do that by checking for the username in the idea and comparing it to the username in local storage.

Here is the entire `render` function:

```
render() {
  this._ideaListEl.innerHTML = this._ideas
    .map((idea) => {
      const tagClass = this.getTagClass(idea.tag);
      const deleteBtn =
        idea.username === localStorage.getItem('username')
          ? `<button class="delete"><i class="fas fa-times"></i></button>`
          : '';
      return `
        <div class="card" data-id="${idea._id}">
          ${deleteBtn}
          <h3>
            ${idea.text}
          </h3>
          <p class="tag ${tagClass}">${idea.tag.toUpperCase()}</p>
          <p>
            Posted on <span class="date">${idea.date}</span> by
            <span class="author">${idea.username}</span>
          </p>
        </div>
      `;
    });
}
```

```
    })
    .join(' ');
  this.addEventListener();
}
```

Now only ideas that you create should show the delete button

# Deploy To Render

---

There are many ways to deploy a fullstack app. You could use cloud hosting with something like Linode or Digital Ocean. You could use a service like Heroku. We are going to use Render.com. Render is a service that allows you to deploy your app to the cloud and manage it. It is a great service for deploying fullstack apps. It is also free for small apps.

## Create a Render Account

To get started, just sign in with GitHub. You can also sign up with an email address, but I recommend using GitHub.

## Checklist

In order to deploy successfully, there are a few things you need to do.

1. Change the mode in `webpack.config.js` to production
2. In your API service, change the `api_url` from `http://localhost:5000/api/ideas` to `/api/ideas`
3. Add a proxy. In your `webpack.config.js` file, add a proxy to the `devServer` object:

```
devServer: {  
  static: {  
    directory: path.resolve(__dirname, '../public'),  
  },  
  port: 3000,  
  open: true,  
  hot: true,  
  compress: true,  
  historyApiFallback: true,  
  proxy: {  
    '/api': 'http://localhost:5000',  
  },  
  
},
```

This tells our `devServer` to use the proxy when it sees a request to `/api`. This is the same as if we were using a proxy like Nginx.

4. Build your app. Run `npm run build` to build your app.
5. Push your app to GitHub.
6. Add your environment variables to Render. In your Render dashboard, click on the `Environment` tab. Add the following environment variables:

MONGODB\_URI - Your MongoDB connection string

NODE\_ENV - production

7. Add your app to Render. Click on the [Add App](#) button. Select the GitHub repo that you pushed your app to. Select the branch you want to deploy. Click [Create App](#).

Now you should be able to see your app running on Render. You can click on the [Open](#) button to see your app.