**Intro to JavaScript and Node.js**

**Lesson duration: 30 minutes**

**Learning Objectives**

* Understand what JavaScript is and where it came from
* Know what a JavaScript runtime is
* Understand why Node is useful

**Intro**

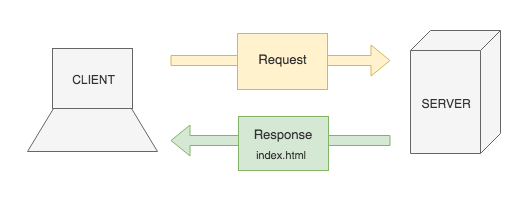
In this module we will learn JavaScript as a programming language. JavaScript was originally designed as a front-end tool to make web pages more interesting, but that isn’t all that it can do. JavaScript can be used to build entire web apps, native apps, mobile apps, games as well as a plethora of other things. Almost anything that can be done using other programming languages can be done using JavaScript.

Before we start learning the language, let’s take some time to learn what JavaScript is, where it came from and why it’s important.

**History of JavaScript**

The year is 1995. The internet has been around for a while and is becoming more widely used, but it’s still in its infancy. Netscape Navigator is the dominant web browser with over 75% market share. Web pages are constructed using pure HTML. CSS hasn’t been created yet.

When a user types a URL into their browser’s address bar, the client makes a request to a server. The server then responds by sending back an .html file, which is rendered by the browser.



*Request Response Cycle*

In order to update the information that is displayed in the browser, the client had to make another request to the server and wait for the server to respond with more HTML. Internet connections in the mid 90s were very slow, so this wasn’t ideal.

Netscape, realising that the web had to become more dynamic and interactive, recruited Brendan Eich to create a scripting language that would compliment Java, as that was the popular new programming language at the time. This new language was to be embedded in the HTML that was sent to the client, allowing logic to be performed client side without having to make additional requests to the server. It would be revolutionary, allowing web pages to change in real-time.

The language that Brendan Eich created was originally called LiveScript when it was released as part of the Netscape Navigator web browser in 1995, but was renamed JavaScript a few months later. This caused some confusion as people believed that it was somehow related to Java, but the name was purely a marketing ploy as Java was very popular at the time.

**ECMAscript**

A year later, in 1996, Netscape submitted JavaScript to ECMA International so that they could create a standard specification which could then be implemented in other web browsers. This led to the release of the official language specification known as ECMAScript or ES1 in 1997.

You’ll sometimes hear people refer to JavaScript as ECMAscript (or vice versa) although they’re slightly different things. JavaScript is a programming language, whereas ECMAscript is the specification that JavaScript adheres to.

ECMAscript has gone through various iterations over the years since it’s initial release in 1997, adding and improving language features with each new release.

**Official ECMAScript releases:**

* ES1: June 1997
* ES2: June 1998
* ES3: Dec 1999
* ES4: Targeted for 2008 (abandoned)
* ES5: Dec 2009
* ES2015 (ES6): June 2015
* ES2016 (ES7): June 2016
* ES2017 (ES8): June 2017

ECMAScript releases will be yearly from 2015 onwards so it will become more and more common to see them named by year.

You may also see the term ES.Next. This refers to whatever the next release of JS will be.

**Node.js**

As we discussed earlier, JavaScript was originally designed to run client-side, in the browser, to make web pages more dynamic and interactive. The back-end was typically written in another language, like Java, and JavaScript was sent to the client with the HTML.

All major internet browsers have a JavaScript runtime, an environment in which they can execute JavaScript code. Google Chrome has its V8 engine and Safari has WebKit, for example.

Node.js is a JavaScript runtime which was built on Chrome’s V8 JavaScript engine. Node provides a JavaScript environment outside of the browser, where we can run server-side code. With the help of Node, JavaScript has gained traction as a full-stack programming language. This means that we can write entire applications using only JavaScript, HTML and CSS.

**Writing Our First JS App**

We’re going to create our first JavaScript app and run it in the command-line using Node.

Before we get started we’ll have to check that we have Node installed on our machines. Typing node -v into the command-line should yield a version number, not a ‘command not found’ error.

node -v

*# -> vx.x.x*

The first thing that we’ll have to do is create a file to work in. We’re going to write a simple ‘hello world’ program, so we’ll name our file hello\_world.js.

touch hello\_world.js

.js is the file extension used for JavaScript files.

There is no hard and fast rule when it comes to naming JS files. The most common naming conventions are snake\_case, kebab-case or camelCase. None of these are considered to be better practice than the others, but it is bad practice to mix and match. Pick the one that you like best and be consistant throughout your project.

*// hello\_world.js*

console.log('Hello, World!');

The correct naming convention for variables and functions in JavaScript is camelCase.

Now that we’ve written some JS, we can use Node to run it.

node hello\_world.js

…And just like that, we’re JavaScript developers! Well, almost… We’re at the beginning of our journey towards becoming JavaScript developers.

**Recap**

What effect did JavaScript have on web development?

Answer

What is Node.js?

Answer

**Conclusion**

Now that we know a little bit about what JavaScript is and where it came from we’re ready to start learning the language. We’ll spend the next few weeks learning JavaScript as a programming language, writing JavaScript that runs in the browser, allowing us to create more interesting, dynamic and interactive web pages, and writing server-side JavaScript to handle routing and database queries.

**JavaScript Fundamentals**

**Lesson Duration: 60 minutes**

**Learning Objectives**

* Be able to declare variables
* Understand the differences between data types
* Be able to write conditionals
* Understand the concept of truthy and falsy values and it’s uses
* Know the truthy and falsy values
* Understand the equality operators

**Introduction**

When learning a new programming language there are a number of key things to be aware of. These include:

* whether it is dynamically or statically typed
* what the data types are
* whether it is weakly or strongly typed
* the truthy and falsy values

In this lesson we are going to look at what we mean by these terms and how they apply to JavaScript so that we are able to write programs knowing what the expected behaviour of the language is.

**Running a JavaScript file in Node**

Create a file called js\_fundamentals.js. We can use the console’s log method to output a value.

Note: Run the file with the command node js\_fundamentals.js.

*// js\_fundamentals.js*

console.log("Hello World!");

We can run the file using Node.

node js\_fundamentals.js

**Variables**

JavaScript, like other programming languages, allows us to declare variables and assign values to them, so that we can reference them later.

We declare variables using the keyword var. Let’s create a variable name with the value “Mickey”:

**var** name **=** "Mickey";

console.log("name:", name);

*// -> name: Mickey*

It’s good practice to pass a string label to console.log() as the first argument to more easily identify the output.

**Semicolons**

Notice that we end each line with a semicolon to tell the JavaScript runtime that it has reached the end of our statement.

JavaScript has a feature called automatic semicolon insertion which means that semicolons are optional for the most part. We will be writing our semicolons manually for the duration of this course however, as it’s important to know where they do and don’t belong.

**Dynamic Typing**

Dynamically typed languages type check at run time, in contrast to statically typed languages which type check at compile time. Type checking is the verification of a value’s type, so that the environment can determine which operations are safe or unsafe. For example, whether or not it is safe to call a particular method on it.

Knowing that JavaScript is dynamically typed, what would we expect to happen if we reassign the value of the following variable, greeting, to be a number?

**var** greeting **=** "hello";

 Answer

**Data Types**

Can you think of any reasons why it’s important to know which data type we are dealing with?

Answer

JavaScript has a number of primitive data types:

* number
* string
* null
* undefined
* boolean
* symbol

In JavaScript we can always check the data type of a value using the typeof operator, which returns the type of a value as a string.

**typeof** "hello";

*// -> string*

JavaScript wraps up primitive values in objects so we are able to call methods on them.

"no way!".toUpperCase();

*// -> NO WAY!*

**Number**

Number is the only numerical data type in JavaScript. It does not differentiate between whole and decimal numbers.

**typeof** 1;

*// -> 'number'*

**typeof** 1.5;

*// -> 'number'*

The standard arithmetic operators are available to us:

* Addition +

1 **+** 2;

*// -> 3*

* Subtraction -

1 **-** 2;

*// -> -1*

* Multiplication \*

2 **\*** 2;

*// -> 4*

* Division /

4 **/** 2;

*// -> 2*

* Exponentiation \*\*

2 **\*\*** 3;

*// -> 8*

* Remainder %

3 **%** 2;

*// -> 1*

**String**

Strings can be declared with:

* double quotation marks (")
* single quotation marks (')
* backticks (`)

We can also use the String() function to convert non-string values to strings.

String(5);

*// -> '5'*

We can use the + operator to concatenate strings.

**var** name **=** "Minnie";

"Hello, " **+** name;

*// -> 'Hello, Minnie'*

Using backticks to declare a string allows us to use string interpolation.

**var** name **=** "Donald";

`Hello, ${name}`;

*// -> 'Hello, Donald'*

A full list of string methods can be found here: [MDN docs: String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

**Null**

The null value is used when a value is deliberately absent. A database might return null if you are trying to fetch a record that doesn’t exist or you might return null from a search function when it finds no matches.

**Undefined**

Undefined is the default value and type of declared variables where no value has been assigned.

**var** myVariable;

myVariable;

*// -> undefined*

**typeof** myVariable;

*// -> 'undefined'*

undefined is different from ReferenceError, which is the error we get when trying to use a variable that hasn’t been declared.

bananas;

*// -> ReferenceError: bananas is not defined*

NaN is Not-A-Number. It is of type of number but has no numerical value. We get NaN when we perform some illegal mathematical operations. A numerical value + undefined is probably the most common operation that results in NaN. This might happen if you try to access a value that you believe is a number but is actually undefined.

**undefined** **+** 1;

*// -> NaN*

**Boolean**

A Boolean has one of two values, either true or false. Like in other languages the key role of the Boolean values is for control flow.

**Symbol**

In newer versions of JavaScript, we have a new type of primitive: Symbol. They are not widely used yet, and you will see them far less frequently than the other types of primitive we have looked at.

They function quite differently from symbols that you might have seen in other languages. In JavaScript, they are primarily used to create unique keys within key-value pairs.

**Control Flow**

**Conditionals**

As with other programming languages, JavaScript allows us to write conditionals. This is the syntax for the if statement:

*// if (first expression) {*

*// statement to run if first expression evaluated as true*

*// }*

*// else if (second expression) {*

*// statement to run if second expression evaluated as true*

*// }*

*// else {*

*// statement to run if all other expressions evaluated as false*

*// }*

**if** (1 **>** 0) {

**var** message **=** "1 is greater than 0";

} **else** **if** (1 **==** 0) {

**var** message **=** "1 is equal to 0";

} **else** {

**var** message **=** "1 is not greater than 0";

}

**Short-circuiting**

JavaScript employs short-circuiting. This means that in the above if statement, as the first condition is satisfied, the else if and else conditions and associated code blocks are never executed.

**Truthy and Falsy Values**

While we know that the boolean values are true or false, we often want to evaluate a non-boolean value as true or false when working with control flow. For example, as an if statement expects a boolean value as its condition, whatever you pass to it will be coerced to either true or false. null is a falsy value so will be coerced to false.

**if** (**null**) {

**var** result **=** 'The expression evaluates to true.';

} **else** {

**var** result **=** 'The expression evaluates to false.';

}

When we are learning a language we need to know which expressions and non-boolean values evaluate as true and which evaluate as false. The result of these evaluations are called the ‘truthy’ and ‘falsy’ values and are different for different languages.

**Task: (5 minutes)**

We can convert from any value to a boolean with the Boolean function, Boolean(). For example:

Boolean(**null**);

*// -> false*

Run the following in the Node REPL, use the Boolean function check out the truthy and falsy values:

* a string
* '' (an empty string)
* 0
* a number other than zero
* undefined
* NaN

Note: To enter the Node REPL run the command ‘node’ in the terminal.

What are the falsy values in JavaScript?

Answer

As these values are falsy, we can easily identify them using control flow and prevent them being displayed to the user where appropriate.

**Weak (or Loose) Typing**

JavaScript is a weakly typed language, which means it makes implicit conversions (or coercions) between data types at run time.

**Task: (5 minutes)**

Run the following in the Node REPL to see the results of JavaScript’s type coercion:

1. 3 + “hello”
2. "route" + 6 + 6
3. 6 + 6 + “route”

When we attempt to perform operations that act on values of different types JavaScript will coerce one or more of them to a different type in order to make it work.

**Equality**

**Types of Equality**

Let’s look at how some other expressions evaluate in JavaScript:

2 **==** 2; *// -> true*

2 **==** "2"; *// -> true*

2 **===** "2"; *// -> false*

The equivalent ‘not equal’ operators are != and !== respectively.

2 **!=** 2; *// -> false*

2 **!=** "2"; *// -> false*

2 **!==** "2"; *// -> true*

**Strict Equality (===)**

The triple equals (strict equality) operator compares for equality by checking if both the type and value are the same.

**Loose Equality (==)**

The double equals (abstract or loose equality) operator compares for equality *after* having coerced the values to a common type. This is a product of JavaScript being a weakly typed language, as previously discussed.

Because loose equality can cause unexpected behaviour, unless you have a good reason, it is good practice to use the triple equals (===), which only evaluates to true if both the value and the type match.

**Logical Operators**

We can use the logical operators ‘and’ (&&) and ‘or’ (||) to make logic expressions.

(1**+**1 **===** 2) **&&** (1**+**1 **===** 4); *// -> false*

(1**+**1 **===** 2) **||** (1**+**1 **===** 4); *// -> true*

We can also use ! for “not”.

**!true**;

*// -> false*

Short-circuiting also applied to the logical operators. This means that if the first expression in an && statement evaluates to false, JavaScript does not bother evaluating the second expression, because the statement will evaluate to false regardless.

Similarly, if the first expression in an || statement is true, JavaScript does not bother evaluating the second expression, because the statement will evaluate to true regardless.

**Recap**

1. What syntax has to be used to declare a string when using string interpolation?
2. What are the falsy values in JavaScript?
3. What does !0 return?
4. What does !0 && !1 return?
5. Why is it better practice to use the strict equality operator (===) over the loose equality operator (==)?

 Answers

**Conclusion**

In this lesson we saw how to write the basic syntax of JavaScript. We looked at JavaScript’s data types and now know it’s common truthy and falsy values.

We have learnt that Javascript is a dynamically and weakly typed language, meaning that type checking happens at run time and it performs implicit data type conversions. This prevents us from having to declare the types of our variables up front and means that we can reassign them to values of different types.

**Further Resources**

[JavaScript Equality Table](https://dorey.github.io/JavaScript-Equality-Table/)

[Mozilla JavaScript Docs](https://developer.mozilla.org/bm/docs/Web/JavaScript)

**Objects, Arrays & Loops**

**Lesson Duration: 75 minutes**

**Learning Objectives**

* Be able to create, access and modify arrays
* Be able to use common array methods: pop, push, shift, unshift and splice
* Be able to create, access and modify objects
* Know that object keys are strings, and values can be any type, including other objects
* Be able to write for loops
* Know when to use for...of and when to use for...in loops

**Intro**

We often want to store data in collections, or data structures, in order to make the data easier to work with or understand. In this lesson we are going to look at two data structures in Javascript:

1. Arrays - unordered collections where each element has an index position.
2. Objects (known in other languages as hashes, hashmaps and dictionaries) - unordered collections where data is stored in key-value pairs and values can be accessed by their key.

Storing data in collections enables us to organise, manipulate and iterate over it. Just like other programming languages, JavaScript has iteration statements, which allow us to create [loops](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration) so that we can carry out operations on a collection’s elements programatically.

In this lesson we are going to look at how we declare collections and manipulate and access their elements.

**Arrays**

Arrays are ordered collections and their elements are accessed with an integer index, the first element in an array being at index 0. Let’s look at how we declare arrays in JavaScript.

touch loops\_arrays\_objects.js

**Declaring an Array**

We can create an array with the array literal square bracket notation. ([])

**var** sports **=** [];

To put data in our array as we create it, we list the values separated by commas. (,)

**var** sports **=** ['football', 'tennis', 'rugby']; *// MODIFIED*

**Length Property**

We can get the current number of elements in the array with the Array’s length property.

**var** numberOfElements **=** sports.length;

console.log('number of elements:', numberOfElements);

*// -> number of elements: 3*

**Accessing Elements**

To get the first element we pass the index 0 to the array using square brackets again. We can access the other elements with their respective index numbers.

**var** firstSport **=** sports[0];

console.log('first sport:', firstSport);

*// -> first sport: football*

**var** secondSport **=** sports[1];

console.log('second sport:', secondSport);

*// -> second sport: tennis*

**Adding Elements to the End**

We can add a new element to the end of our array with the push method.

sports.push('curling');

sports.push('snooker');

sports.push('darts');

console.log('sports:', sports);

*// -> sports: [ 'football', 'tennis', 'rugby', 'curling', 'snooker', 'darts' ]*

**Removing Elements from the End**

Let’s get rid of that last item, with the pop method.

sports.pop();

console.log('sports:', sports);

*// -> sports: [ 'football', 'tennis', 'rugby', 'curling', 'snooker' ]*

pop also returns the removed item.

**var** removedSport **=** sports.pop(); *// MODIFIED*

console.log('removed sport:', removedSport); *// MODIFIED*

*// -> removed sport: darts*

console.log('sports:', sports);

*// -> sports: [ 'football', 'tennis', 'rugby', 'curling', 'snooker' ]*

pop and push operate on the end of the array. We can do the same to the start of the array using shift and unshift respectively.

**Adding Elements to the Start**

Add an item to the start with unshift.

sports.unshift('basketball');

console.log('sports:', sports);

*// -> sports: [ 'basketball', 'football', 'tennis', 'rugby', 'curling', 'snooker' ]*

**Removing Elements from the Start**

Remove the first item with shift. (also returns the removed item to us)

**var** removedSport **=** sports.shift();

console.log('removed sport:', removedSport);

*// -> removed sport: basketball*

console.log('sports:', sports);

*// -> sports: [ 'football', 'tennis', 'rugby', 'curling', 'snooker' ]*

**Manipulating Arrays with splice**

These methods are useful, but they only let us modify the beginning or end of the array. What about if we want to add or remove something from the middle of the array? To do this, we can use the Array method, [splice](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice).

Let’s remove 'curling' from the sports array using splice. We can see from the docs that splice has parameters:

1. start is the index position that we want to start removing from. We are removing 'curling' so we will pass it the index position, which is 3.
2. deleteCount is the number of items to remove. We only want to remove one element so we will pass it 1.
3. We would use the additional optional arguments if we wanted to use splice to add items into the array, which in this case we don’t.

splice returns an array of all of the items it removed, and modifies the original array we called it on.

**var** removedSport **=** sports.splice(3, 1);

console.log('removed sports:', removedSport);

*// -> removed sports: [ 'curling' ]*

console.log('sports:', sports);

*// -> sports: [ 'football', 'tennis', 'rugby', 'snooker' ]*

**Iterating Over Arrays**

When we have an array, and want to do the same thing with each value in the array, we can use a for loop. To loop through an array we can use the for...of statement. This probably looks similar to for loops you might have seen in other languages.

**for** (**var** currentSport **of** sports) {

**var** uppercasedSport **=** currentSport.toUpperCase();

console.log(uppercasedSport);

*// -> FOOTBALL*

*// -> TENNIS*

*// -> RUGBY*

*// -> SNOOKER*

}

JavaScript also supports the “long-form” for loop syntax. It allows us to run a counter for the index ourselves, and access the array’s elements manually using that index. This gives us complete control. We can achieve the same behaviour as the for...of loop above like so.

**for** (**var** i **=** 0; i **<** sports.length; i**++**) {

**var** currentSport **=** sports[i];

**var** uppercasedSport **=** currentSport.toUpperCase();

console.log(uppercasedSport);

*// -> FOOTBALL*

*// -> TENNIS*

*// -> RUGBY*

*// -> SNOOKER*

}

It can be quite a confusing syntax when you first see it. After the for we have three separate statements, separated by semi-colons. (;)

**for** (initialiseCounter; condition; incrementCounter) {

}

The initialiseCounter section runs once, before we begin looping. Here we tend to declare and set and initial value of some kind of counter variable.

The condition is checked before every iteration of the loop. If it’s true, we loop again, if it’s false, we stop looping and continue the program after the closing brace. (})

The incrementCounter section is run after each iteration of the loop. Here we usually want to change our counter variable in some way so that after the appropriate number of iterations, the condition is false and the loop can end.

You might not have seen the i++ syntax in the incrementCounter section of this for statement. ++ is essentially a shorthand for i += 1, as incrementing a number by 1 is so common, (for example, in most for loops) there’s a special operator to do just that. i += 1 is just a shorthand for i = i + 1.

So why bother with this more complicated syntax if we’re just doing the same thing as for...of does? This syntax is much more flexible, we can do almost anything with it. We could increment the index by 2 each iteration, skipping out every second element. Or we start our counter at the last index, decrement our index counter rather than incrementing, and iterate over the array backwards.

**Objects**

**Declaring Objects**

Javascript’s objects store data in key-value pairs. We can create one with the object literal notation, {}.

**var** movie **=** {};

console.log('movie:', movie);

*// -> movie: {}*

Our object is currently empty, it has no key-value pairs in it. We can add key-value pairs to our object literal as we define it. A key separated from its associated value by a colon (:), and each key-value pair is separated from the next by a comma (,). As *all* keys are strings, we don’t need to wrap them in quotes. (' / ")

**var** movie **=** { *// MODIFIED*

title: 'It\'s a Wonderful Life',

year: 1946,

language: 'Spanish'

};

console.log('movie:', movie);

*// -> movie: {*

*// -> title: 'It\'s a Wonderful Life',*

*// -> year: 1946,*

*// -> language: 'Spanish'*

*// -> }*

**Accessing a Property**

We can access an object’s value by using the dot notation (.).

**var** title **=** movie.title;

console.log('title:', title);

*// -> title: It's a Wonderful Life*

**Adding a Property**

Let’s add an array of cast to our movie object. We can add new properties using the same dot notation, this time assigning *to* the property rather than accessing the value *from* it.

movie.cast **=** ['James Stewart', 'Donna Reed'];

console.log('movie:', movie);

*// -> movie: {*

*// -> title: 'It\'s a Wonderful Life',*

*// -> year: 1946,*

*// -> language: 'Spanish',*

*// -> cast: ['James Stewart', 'Donna Reed']*

*// -> }*

**Modifying a Property**

Modifying an existing property looks exactly the same as adding a property. If the property already exists, its value is modified. Let’s change the language to 'English'.

movie.language **=** 'English';

console.log('movie:', movie);

*// -> movie: {*

*// -> title: 'It\'s a Wonderful Life',*

*// -> year: 1946,*

*// -> language: 'English',*

*// -> cast: ['James Stewart', 'Donna Reed']*

*// -> }*

As an alternative to dot notation (.) we can use square bracket notation ([]). We do this by inserting the property name as a string between square brackets.

movie['language'] **=** 'French';

console.log('movie:', movie);

*// -> movie: {*

*// -> title: 'It\'s a Wonderful Life',*

*// -> year: 1946,*

*// -> language: 'French',*

*// -> cast: ['James Stewart', 'Donna Reed']*

*// -> }*

This is a bit more typing than the dot notation, but is necessary in some cases. Dot notation won’t work when using a property name that contains special characters such a hyphens (-) or colons (:) etc.

movie.subtitle**-**language **=** 'German';

*// -> ReferenceError: Invalid left-hand side in assignment*

movie['subtitle-language'] **=** 'German'; *// MODIFIED*

console.log('movie:', movie);

*// -> movie: {*

*// -> title: 'It\'s a Wonderful Life',*

*// -> year: 1946,*

*// -> language: 'French',*

*// -> cast: ['James Stewart', 'Donna Reed'],*

*// -> subtitle-language: 'German'*

*// -> }*

Most of the time we’ll use camelCase instead of kebab-case, however, you may need to use the square bracket notation when dealing with JavaScript objects from an outside data source that uses special characters in their property names.

Another use case for square bracket notation is when you need to dynamically access properties using a variable. This allows us to write DRY, reusable code.

**var** propertyToAccess **=** 'subtitle-language'; *// NEW*

movie[propertyToAccess] **=** 'German'; *// MODIFIED*

console.log('subtitle language:', movie[propertyToAccess]);

*// -> subtitle language: German*

console.log('subtitle language:', movie.propertyToAccess);

*// -> subtitle language: undefined*

*// -> {*

*// -> title: 'It\'s a Wonderful Life',*

*// -> year: 1946,*

*// -> language: 'French',*

*// -> cast: ['James Stewart', 'Donna Reed'],*

*// -> subtitle-language: 'German'*

*// -> }*

Note: A common use-case for this is when a function accepts a key as an argument. Inside the function you will have access to the key via the parameter variable, and will need to use the square bracket notation if you want to access the value from the object.

**Deleting a Property**

We can also use the delete operator to delete an object’s property. Let’s delete the movie’s year property.

**delete** movie.year;

*// -> {*

*// -> title: 'It\'s a Wonderful Life',*

*// -> language: 'French',*

*// -> cast: ['James Stewart', 'Donna Reed'],*

*// -> subtitle-language: 'German'*

*// -> }*

**Nested Objects**

We’ve seen that the values of our object’s properties can be strings, numbers, even arrays, and they can be regular JavaScript objects as well. Like we did with the movie’s cast property that has an array or strings, we often want to nest data structures inside others. The hierarchical structure this creates helps us more easily access and manage subsets of data.

For example, if we want to store both critics ratings and audience ratings on our movie object, we may decide to wrap both properties in an ratings object, because ratings is a subset of the overall movie properties.

Let’s give our movie a new rating property, with the value of another object. The rating object will have two properties: critic and audience.

movie.ratings **=** {

critic: 94,

audience: 95

};

console.log('movie:', movie);

*// -> movie: {*

*// -> title: 'It\'s a Wonderful Life',*

*// -> language: 'French',*

*// -> cast: ['James Stewart', 'Donna Reed'],*

*// -> subtitle-language: 'German',*

*// -> ratings: { critic: 94, audience: 95 }*

*// -> }*

We can access properties on the ratings object by chaining the keys using the dot notation, to allow us to ‘drill-down’ into the data structure.

**var** audienceRating **=** movie.ratings.audience;

console.log('audience rating:', audienceRating);

**Iterating Over Objects**

As with arrays, a common task with a collection of data if iterating through all the pieces of data within it, one at a time. We can do this with the for...in statement. for...in gives us each key in the object in turn. We can then use the key to access the value.

**for** (**var** key **in** movie) {

**var** value **=** movie[key];

console.log(`The ${key} is ${value}`);

*// -> The title is It's a Wonderful life*

*// -> The language is French*

*// -> The cast is James Stewart,Donna Reed*

*// -> The subtitle-language is German*

*// -> The ratings is [object Object]*

}

**Accessing an Object’s keys**

We may not always know the properties of an object. We can use Object.keys() to get an array of all of an object’s keys. We pass the method the object that we want to get the keys from.

**var** keys **=** Object.keys(movie)

console.log('keys:', keys);

*// => keys: [ 'title', 'language', 'cast', 'subtitle\_language', 'ratings' ]*

**Task: 10 Minutes**

Note: Hand out [Task Start](https://codeclan.github.io/canvas_notes/course_javascript/week_1/day_1/objects_arrays_loops/arrays_objects_movies_task_start.js).

Complete the tasks in arrays\_objects\_movies\_task\_start.js

Example solution

**Recap**

What do the array methods pop, push, shift, unshift and splice do?

Answer

</br>

What types can objects’ keys and values be?

Answer

</br>

When might we want to use the “long form”, basic for loop? (for (var i = 0; i < array.length; i++){})

Answer

</br>

Which data structure do we iterate over with for...of loops?

Answer

</br>

Which data structure do we iterate over with for...in loops?

Answer

</br>

**Conclusion**

Objects and Arrays help us to organise and work with the data in our applications more efficiently. We can iterate over objects with the for...in loop and arrays with the for...of loop. If we want total control and extra flexibility, we can do all of the looping manually with a for loop. Loops help us avoid repeated code.

**Functions**

**Lesson Duration: 45 minutes**

**Learning Objectives**

* Be able to declare functions
* Understand that functions can be anonymous
* Know the differences between function declarations and anonymous function expressions

**Intro**

We’ve now covered most of the core building blocks of a programming language, but we’re still missing a big one: functions. Functions are incredibly useful, they let us split our code up into small, reuseable chunks. Breaking our code up like this lets us build DRY, modular applications. We can give that section of code a meaningful name, making our code more readable and more easily understood.

Let’s make a new file to have a look at functions in JavaScript.

touch functions.js

atom .

**Named Function Declarations**

There are a number of ways of declaring a function in JavaScript. First, we’ll look at named functions.

To declare the function we use the function keyword. This keyword is followed by a name for the function and brackets (()). Then braces ({}) are used to define the function body. To return a value from our function we use the return keyword.

**function** sayHello() {

**return** 'Hello World!';

}

We execute the code contained in a function by referring to it by it’s name and using brackets () to invoke (or call) it: sayHello().

**function** sayHello() {

**return** 'Hello World!';

}

console.log('sayHello message:', sayHello());

*// -> sayHello message: Hello World!*

sayHello() gives us back the string 'Hello World!' when our code runs.

**Arguments & Parameters**

We don’t always want our functions to output a hard-coded value; often we want the function to take some input, and return a different output depending on what was input. This makes our functions more flexible and let’s us reuse our code. Instead of saying ‘Hello world!’, we might want to take input from the outside world to decide who we say hello to.

What happens when we try to pass an argument to the function as we invoke it?

**function** sayHello() {

**return** 'Hello World!';

}

console.log('sayHello message:', sayHello('Danielle'));

*// -> sayHello message: Hello World!*

We still just get 'Hello World!', but that’s not too surprising, we haven’t named a parameter or done anything with the argument that we passed in. What is a bit surprising is that JavaScript doesn’t seem to have a problem with us passing in this unused argument. It’s simply ignored, the code runs fine with no errors.

To use the argument inside our function we need to declare a parameter by adding a name for the variable between the brackets (()). We can then reference it by name within the function.

**function** sayHello(name) { *// MODIFIED*

**return** `Hello ${name}!`; *// MODIFIED*

}

console.log('sayHello message:', sayHello('Danielle'));

*// -> sayHello message: Hello Danielle!*

Now that we’re expecting a parameter to be passed, and using it within our function, what happens if we don’t pass one in?

console.log('sayHello with no arguments:', sayHello()); *// NEW*

*// -> sayHello with no arguments: Hello undefined!*

Again, JavaScript doesn’t seem to care all that much. No errors, it still runs fine. It just declares an empty parameter that never gets a value passed into it. Our name’s value is undefined.

**Default Parameters**

JavaScript supports default parameter syntax similar to that found in many other languages, meaning we can set a default value to be used if no argument is passed in. Let’s set 'World' as the default value which will be taken by name if no value is passed to the function.

**function** sayHello(name **=** 'World') { *// MODIFIED*

**return** `Hello ${name}!`;

}

console.log('sayHello with no arguments:', sayHello());

*// -> sayHello with no arguments: Hello World!*

We can add more parameters by listing them between the brackets (), as a comma separated list ,.

**function** sayHello(greeting, name **=** 'World') { *// MODIFIED*

**return** `${greeting} ${name}!`;

}

console.log('sayHello message:', sayHello('Hi', 'Danielle')); *// MODIFIED*

*// -> sayHello message: Hi Danielle!*

Note: The parameter with a default must be last in the list, so that if only one argument is passed, the argument will be treated as greeting, and World! with be used as the value for name.

**Hoisting**

A slightly strange feature of the named function declaration is that the function declaration is “hoisted”. This means that when the JavaScript interpreter parses the code, and just before it actually runs, it essentially moves the function declaration to the top of the file. This means that we can use our function before it is declared.

console.log('sayHello message:', sayHello('Danielle', 'Hi')); *// MODIFIED*

*// -> sayHello message: Hi Danielle!*

**function** sayHello(name **=** 'World', greeting) { *// MODIFIED*

**return** `${greeting} ${name}!`;

}

This can be kind of confusing as it can make the code harder to follow. There is an argument that you should not depend on this hoisting behaviour at all.

[MDN documentation on hoisting](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting)

**Anonymous Function Expressions**

We’re about to see an incredibly powerful feature of JavaScript, which is a fairly uncommon feature in object oriented programming languages. Functions are first-class objects. This means that, like numbers, strings, arrays or objects, we can store functions in variables, put them in arrays or objects. We can also call methods on functions, pass functions into functions as arguments and even return a function from another function.

There’s a lot to this aspect of JavaScript, so we’ll see more of this in practice later in the module. For now, let’s try writing a new function and storing it in a variable.

**var** add **=** **function** (firstNumber, secondNumber) {

**return** firstNumber **+** secondNumber;

}

console.log('1 + 3 with add:', add(1, 3));

*// -> 1 + 3 with add: 4*

This function doesn’t look like our named function, sayHello, from before. We omitted the name. Since we can store function objects in variables, we can refer to them later by the name of the variable that we store them in, so in this case, the function name is optional. This is called an anonymous function expression.

It’s worth noting that anonymous function expressions are not “hoisted”, like named function declarations.

console.log('1 + 3 with add:', add(1, 3));

*// -> TypeError: add is not a function*

**var** add **=** **function** (firstNumber, secondNumber) {

**return** firstNumber **+** secondNumber;

}

**Task: 20 minutes**

1. Declare a **named** function that takes an array of numbers, and returns the sum, or total, of all of the numbers in the array.
2. Define an **anonymous** function expression that takes two arguments:
   * an object, for example, { name: 'Wojtek', age: 30 }
   * a string, for exmaple, 'name'

Your function should return true if the given string is a key on the given object and false if the object does not have a key that matches the string. Store this function in an appropriately named variable to invoke it.

Example solution

**Arrow Functions**

There’s actually one more way to define functions in JavaScript. Just like with anonymous function expressions, we’ll see more and more uses for this as the course goes on, but for now, let’s just take a look at the syntax of arrow function expressions.

**var** multiply **=** (firstNumber, secondNumber) **=>** {

**return** firstNumber **\*** secondNumber;

}

console.log('multiply 2 by 5:', multiply(2, 5));

*// -> multiply 2 by 5: 10*

We omit yet another part of the first named function declaration, this time it’s the function keyword that we’re dropping. Instead, an arrow function has a “fat arrow” (=>) pointing from the list of parameters in brackets (()) to the function body in braces ({}). This forms an anonymous function expression: () => {}. Arrow functions are *always* anonymous, they cannot be named. If we want to refer to them later they must be assigned into a variable.

**Implicit Return**

When our arrow function’s body only contains a single expression, we can write it on one line and omit the return keyword and the braces. The function will implicitly return the expression the arrow is pointing to.

**var** multiply **=** (firstNumber, secondNumber) **=>** firstNumber **\*** secondNumber; *// MODIFIED*

console.log('multiply 2 by 5:', multiply(2, 5));

multiply’s body only has one expression, firstNumber \* secondNumber, which is evaluated and returned implicitly. This makes writing a simple function very concise.

**Recap**

What are the three ways that we can declare functions in JavaScript?

Answer

What can named functions do that anonymous function expressions can’t?

Answer

What can arrow functions do that other functions can’t?

Answer

**Conclusion**

We use functions to organise our code, breaking it up into reuseable blocks, that can take arguments to remain flexible and useful in as many situations as possible. This stop us from writing the same, or very similar code multiple times.

We’ve also seen three different ways to create functions in JavaScript, and a couple of the differences. We’ll see more of these as they become relevant as the course continues.

**Variable Scope**

**Lesson Duration: 45 minutes**

**Learning Objectives**

* Understand global, function and block scope
* Know the difference between the var, let and const keywords
* Be able to use var, let and const variable declarations appropriately

**Intro**

Variables can help us organise and keep track of data in our code, we give a piece of data a name and can then pass it around and keep track of its value. We’ve seen that we can declare variables with the var keyword, but we haven’t talked about the details of how var affects the scope of a variable. We’re going to take a closer look at variable scoping, how var works, and then we’ll take a look at two more keywords JavaScript gives us for declaring variables: let and const.

**Function Scope with var**

When declaring variables so far we’ve used the var keyword. The var keyword declares a variable in the current function’s scope. It is only defined within that function, and can’t be seen outside it.

Let’s make a new file and take a look.

touch scope.js

**var** secretsFunction **=** **function** () {

**var** pinCode **=** [0, 0, 0, 0];

console.log('pinCode inside secretsFunction:', pinCode);

*// -> pinCode inside secretsFunction: [ 0, 0, 0, 0 ]*

}

secretsFunction();

console.log('pinCode outside secretsFunction:', pinCode);

*// -> ReferenceError: pinCode is not defined*

This means that our variables aren’t all in the same global scope. When working with globally scoped variables, it can be hard to be sure which variables are relevant to the current task, and we can end up with variable name clashes. Things would get really messy as our applications get larger. var saves us from this.

**Lexical Scope**

Another cool thing about variable scope in JavaScript is that our functions capture the variables from the parent scope in which they are defined. This is called lexical scope. This means we can access variables from inside the function that are declared outside of the function.

**var** name **=** 'Jill'; *// NEW*

**var** secretsFunction **=** **function** () {

**var** pinCode **=** [0, 0, 0, 0];

console.log('name inside secretsFunction:', name); *// MODIFIED*

*// -> name inside secretsFunction: Jill*

}

secretsFunction();

console.log('name outside secretsFunction:', name); *// MODIFIED*

*// -> name outside secretsFunction: Jill*

**Block Scope with let**

Function scope with var isn’t always ideal though as it can leave things in our scope that we don’t expect. Let’s write a function to find names that begin with a certain letter.

**var** filterNamesByFirstLetter **=** **function** (names, letter) {

**var** filteredNames **=** [];

**for** (**var** name **of** names) {

**if** (name[0] **===** letter) {

filteredNames.push(name);

}

}

**return** filteredNames

}

**var** students **=** ['Alice', 'Bob', 'Alyssia', 'Artem', 'Babs'];

**var** filteredStudents **=** filterNamesByFirstLetter(students, 'A');

console.log('filteredStudents:', filteredStudents);

Great, no problems here right? Everything’s working as we expect. However, var is actually doing something that it probably shouldn’t. What should the value of name be after the loop, just before we return?

**var** filterNames **=** **function** (names, letter) {

*// ...*

console.log('name after loop:', name); *// NEW*

*// -> name after loop: Babs*

**return** filteredNames

}

name was just a temporary variable for storing each name for one iteration of the loop. After the loop has finished it still exists within the scope of our function even though it’s not relevant anymore, and is just polluting our scope. This happens because we declared the variable using var. Function scope is not what we want here. We want a variable scoped only to the for loop’s block {}. We want block scope, and to get it, we can use the let keyword.

**var** filterNames **=** **function** (names, letter) {

**var** filteredNames **=** [];

**for** (**let** name **of** names) { *// MODIFIED*

**if** (name[0] **===** letter) {

filteredNames.push(name);

}

}

console.log('name after loop:', name);

*// -> ReferenceError: name is not defined*

**return** filteredNames

}

Variables declared with let are block scoped. This means they don’t exist outside the block ({}) that they are declared in. Like function scopes, any further, nested blocks within a certain block scope will be able to access the let declared variable. Anything outside the block (similarly to how function scopes work) won’t be able to access variable within the block where the let variable is declared.

Many people argue that you should never use var. Using let, you can still create variables that are scoped to a whole function, by declaring them at the “top level” of the function, outside any other blocks. This can often signal your intentions for the variable better than using var and its unusual function scoping behaviour.

Instead of using var like this:

**let** isItFive **=** **function** (number) {

**if** (number **===** 5) {

**var** result **=** **true**;

} **else** {

**var** result **=** **false**;

}

**return** result;

}

We would use let like this:

**let** isItFive **=** **function** (number) {

**let** result;

**if** (number **===** 5) {

result **=** **true**;

} **else** {

result **=** **false**;

}

**return** result;

}

**Paired discussion: 5 minutes**

Read the following code. Discuss it in pairs or small groups.

Without running the code, try to figure out what the value of the console.loged variables will be.

**let** temperature **=** 30;

**if** (temperature **>** 15) {

**let** jacket **=** **false**;

**var** happy **=** **true**;

} **else** {

**let** jacket **=** **true**;

**var** happy **=** **false**;

}

console.log('jacket after if-else blocks:', jacket);

console.log('happy after if-else blocks:', happy);

Answer

**Constants with const**

Sometimes we don’t ever intend for a variable to change. These types of value are referred to as constants. JavaScript provides yet another way to declare variables, for constants we can use const. The scope of const variables is block scoped just like let.

**let** calculateEnergy **=** **function** (mass) {

**const** speedOfLight **=** 299792458;

**return** mass **\*** speedOfLight **\*\*** 2;

}

**let** energyOfMe **=** calculateEnergy(75);

console.log('energyOfMe (if I had a mass of 75kg)', energyOfMe);

*// -> energyOfMe (if I had a mass of 75kg) 6740663840526132000*

The speed of light is a constant, we don’t want to be changing it during our function. The const keyword will help us out by throwing an error if we try to reassign to that variable. If we set it to 0, set it to itself multiplied by 2, or even if we increment it by 1 with the ++ operator it will give the error, TypeError: Assignment to constant variable..

**let** calculateEnergy **=** **function** (mass) {

**const** speedOfLight **=** 299792458;

speedOfLight**++** *// NEW*

*// -> TypeError: Assignment to constant variable.*

**return** mass **\*** speedOfLight **\*\*** 2;

}

**let** energyOfMe **=** calculateEnergy(75);

console.log('energyOfMe (if I had a mass of 75kg)', energyOfMe);

We have to remove any code which reassigns the value of speedOfLight in order to make our code run at all.

*// speedOfLight++ // MODIFIED*

There are a couple more variables in this code that we never want to reassign. What happens if we allow our function assignment by reassigned?

**let** calculateEnergy **=** **function** (mass) {

**const** speedOfLight **=** 299792458;

*// speedOfLight++*

**return** mass **\*** speedOfLight **\*\*** 2;

}

calculateEnergy **=** () **=>** 0

**let** energyOfMe **=** calculateEnergy(75);

console.log('energyOfMe (if I had a mass of 75kg)', energyOfMe);

*// -> energyOfMe (if I had a mass of 75kg) 0*

We probably shouldn’t allow our function to be reassigned now that we can prevent it. The same is true of the energyOfMe variable, why would we want to let it change? We get the result of the function, and all we do with it is log it out, we don’t want it to change.

**const** calculateEnergy **=** **function** (mass) { *// MODIFIED*

**const** speedOfLight **=** 299792458;

*// speedOfLight++*

**return** mass **\*** speedOfLight **\*\*** 2;

}

*// calculateEnergy = () => 0 // MODIFIED*

**const** energyOfMe **=** calculateEnergy(75); *// MODIFIED*

console.log('energyOfMe (if I had a mass of 75kg)', energyOfMe);

const isn’t going to let us make this mistake. Generally, using const as much as possible makes our code more robust and less prone to another developer (or ourselves) accidentally changing something that shouldn’t be changed.

However, const has its limitations. It *only* prevents us from reassigning to the constant variable. It won’t stop us modifying any mutable objects that are declared with const.

**const** song **=** {

title: 'Raspberry Beret',

artist: 'Prince'

};

console.log('song before mutation', song);

song.title **=** 'When Doves Cry';

console.log('song after mutation', song);

**const** songs **=** [

song,

'Happy Birthday',

'Hey Jude'

];

console.log('songs array before mutation', songs);

songs[1] **=** 'Call Me Maybe';

songs.pop();

console.log('songs array after mutation', songs);

const still prevents us accidentally reassigning the variable to a totally new object, so we still want to use it for mutable objects as well as [immutable](https://en.wikipedia.org/wiki/Immutable_object) values like numbers and strings.

**Global Scope**

If we don’t use one of the key words (var, let or const) when declaring a variable, it will be in global scope, and available everywhere.

**const** helloWorld **=** **function** () {

result **=** "Hello World!";

}

helloWorld();

console.log(result);

*// -> "Hello World!"*

Polluting the global scope in this way is bad practice. We should always use a key word when declaring a variable.

**Recap**

What is the difference between let and const?

Answer

What are the respective scopes of var, let and const declared varaiables?

Answer

What does const not prevent us from modifying?

Answer

**Conclusion**

Now we know the different behaviours and uses for var, let and const, we can make our code safer and less prone to mistakes. We can avoid vars unusual function scoping, and avoid accidental reassignments with const.

Broadly speaking, when defining a variable, use const. If you need to reassign that variable’s value, switch to let. var is only actually appropriate in rare circumstances. So we’re not going to see it much from this point on.

**Scope Homework: Who Dunnit**

**Learning Objectives**

* Understand function scope
* Know the difference in between the let and const keywords

**Brief**

Using your knowledge about scope and variable declarations in JavaScript, look at the following code snippets and predict what the output or error will be and why. Copy the following episodes into a JavaScript file and add comments under each one detailing the reason for your predicted output.

**MVP**

**Episode 1**

**const** scenario **=** {

murderer: 'Miss Scarlet',

room: 'Library',

weapon: 'Rope'

};

**const** declareMurderer **=** **function**() {

**return** `The murderer is ${scenario.murderer}.`;

}

**const** verdict **=** declareMurderer();

console.log(verdict);

**Episode 2**

**const** murderer **=** 'Professor Plum';

**const** changeMurderer **=** **function**() {

murderer **=** 'Mrs. Peacock';

}

**const** declareMurderer **=** **function**() {

**return** `The murderer is ${murderer}.`;

}

changeMurderer();

**const** verdict **=** declareMurderer();

console.log(verdict);

**Episode 3**

**let** murderer **=** 'Professor Plum';

**const** declareMurderer **=** **function**() {

**let** murderer **=** 'Mrs. Peacock';

**return** `The murderer is ${murderer}.`;

}

**const** firstVerdict **=** declareMurderer();

console.log('First Verdict: ', firstVerdict);

**const** secondVerdict **=** `The murderer is ${murderer}.`;

console.log('Second Verdict: ', secondVerdict);

**Episode 4**

**let** suspectOne **=** 'Miss Scarlet';

**let** suspectTwo **=** 'Professor Plum';

**let** suspectThree **=** 'Mrs. Peacock';

**const** declareAllSuspects **=** **function**() {

**let** suspectThree **=** 'Colonel Mustard';

**return** `The suspects are ${suspectOne}, ${suspectTwo}, ${suspectThree}.`;

}

**const** suspects **=** declareAllSuspects();

console.log(suspects);

console.log(`Suspect three is ${suspectThree}.`);

**Episode 5**

**const** scenario **=** {

murderer: 'Miss Scarlet',

room: 'Kitchen',

weapon: 'Candle Stick'

};

**const** changeWeapon **=** **function**(newWeapon) {

scenario.weapon **=** newWeapon;

}

**const** declareWeapon **=** **function**() {

**return** `The weapon is the ${scenario.weapon}.`;

}

changeWeapon('Revolver');

**const** verdict **=** declareWeapon();

console.log(verdict);

**Episode 6**

**let** murderer **=** 'Colonel Mustard';

**const** changeMurderer **=** **function**() {

murderer **=** 'Mr. Green';

**const** plotTwist **=** **function**() {

murderer **=** 'Mrs. White';

}

plotTwist();

}

**const** declareMurderer **=** **function** () {

**return** `The murderer is ${murderer}.`;

}

changeMurderer();

**const** verdict **=** declareMurderer();

console.log(verdict);

**Episode 7**

**let** murderer **=** 'Professor Plum';

**const** changeMurderer **=** **function**() {

murderer **=** 'Mr. Green';

**const** plotTwist **=** **function**() {

**let** murderer **=** 'Colonel Mustard';

**const** unexpectedOutcome **=** **function**() {

murderer **=** 'Miss Scarlet';

}

unexpectedOutcome();

}

plotTwist();

}

**const** declareMurderer **=** **function**() {

**return** `The murderer is ${murderer}.`;

}

changeMurderer();

**const** verdict **=** declareMurderer();

console.log(verdict);

**Episode 8**

**const** scenario **=** {

murderer: 'Mrs. Peacock',

room: 'Conservatory',

weapon: 'Lead Pipe'

};

**const** changeScenario **=** **function**() {

scenario.murderer **=** 'Mrs. Peacock';

scenario.room **=** 'Dining Room';

**const** plotTwist **=** **function**(room) {

**if** (scenario.room **===** room) {

scenario.murderer **=** 'Colonel Mustard';

}

**const** unexpectedOutcome **=** **function**(murderer) {

**if** (scenario.murderer **===** murderer) {

scenario.weapon **=** 'Candle Stick';

}

}

unexpectedOutcome('Colonel Mustard');

}

plotTwist('Dining Room');

}

**const** declareWeapon **=** **function**() {

**return** `The weapon is ${scenario.weapon}.`

}

changeScenario();

**const** verdict **=** declareWeapon();

console.log(verdict);

**Episode 9**

**let** murderer **=** 'Professor Plum';

**if** (murderer **===** 'Professor Plum') {

**let** murderer **=** 'Mrs. Peacock';

}

**const** declareMurderer **=** **function**() {

**return** `The murderer is ${murderer}.`;

}

**const** verdict **=** declareMurderer();

console.log(verdict);

**Extensions**

Make up your own episode!

# Constructor Functions & Prototype Objects

**Lesson Duration: 90 minutes**

### Learning Objectives

* Understand how the new operator works
* Be able to create objects using constructor functions
* Understand the benefits of attaching functions to a prototype
* Be able to attach functions to an object’s prototype
* Be able to require one module into another

## Intro

We’ve learned about the basic JavaScript language features, now it’s time to start putting them all together and considering how we might use them to structure an application.

Classes are the building blocks that we use to construct applications in object-oriented programming, so learning how to create them in JavaScript is the next logical step.

JavaScript is an object based language, but it isn’t strictly object-oriented like Ruby, Java or a lot of other languages. That doesn’t prevent us from applying the OO design principles that we know and love.

## “Classes” in JavaScript

Strictly speaking there is no class construct in JavaScript, although we can achieve something very similar using the constructor function pattern.

Constructor functions might look and behave slightly differently to a traditional class, but they allow us to achieve the same goal: creating objects.

## Constructor Function Notation

Let’s imagine that we want to create a Person object to model Shaggy. We can use a constructor function to achieve this in the same way that we might use a class in another language:

touch person.js

*// person.js*

**const** Person **=** **function** () {

}

**const** shaggy **=** Person();

console.log('shaggy:', shaggy);

*// -> shaggy: undefined*

The correct naming convention for constructor functions is PascalCase / UpperCamelCase.

A constructor function is just like any other function. Seeing as we are not returning anything from our function, the return value of the Person function will be undefined. As a result the value of shaggy is undefined.

## The new Operator

If we add a new operator before our function call, its behaviour changes. Instead of returning undefined, it will return an empty Person object.

**const** Person **=** **function** () {

}

**const** shaggy **=** **new** Person(); *// MODIFIED*

console.log('shaggy:', shaggy);

*// -> shaggy: Person {}*

shaggy knows that it was created using the Person constructor.

Instructor note: Ask the students not to code along here

If we were to manually return an object from the function, then we would get back a plain object instead. It wouldn’t know that it was created using the Person constructor.

**const** Person **=** **function** () {

**return** { name: 'Shaggy Rogers' }; *// NEW*

}

**const** shaggy **=** **new** Person();

console.log('shaggy:', shaggy);

*// -> shaggy: { name: 'Shaggy Rogers' }*

The new operator is now ignored and the value of shaggy is the plain object that we returned. It doesn’t even know that it was created via the Person constructor function.

This isn’t what we want…

The key thing to remember is that we never explicitly return anything from a constructor function because it prevents the new operator from working properly.

## Adding Properties to Objects

We can give objects properties (also known as attributes) to store information about them (their state). Let’s give our person a name property with a String value.

To do this we will need to access and modify the Person object as it’s being constructed. How can we do that?

In JavaScript there is a special keyword that we can use to refer to the object that we’re currently creating from inside our constructor function: this.

**const** Person **=** **function** () {

console.log('this:', **this**); *// MODIFIED*

*// -> this: Person {}*

}

**const** shaggy **=** **new** Person();

console.log('shaggy:', shaggy);

*// -> shaggy: Person {}*

They’re the same object! this refers to whichever Person we’re creating at the time.

### Task: (10 minutes)

* Add a name property with the value 'Shaggy Rogers' to the object that is returned from our constructor
* console.log only shaggy’s name property instead of the entire object

Example solution

Notice that we’re able to access shaggy’s name property directly. There are no access modifiers in JavaScript. There are ways to encapsulate data, effectively mimicking the private access modifier that you may have encountered when working with other languages, but this is a fairly advanced topic in JavaScript.

If a property should share the same initial value across all instances, we can hard code the value in the constructor as we did here with name.

People have different names, so in this case we should pass the value in to our constructor function as an argument to make our code more versatile or dynamic:

**const** Person **=** **function** (name) { *// MODIFIED*

**this**.name **=** name; *// MODIFIED*

}

**const** shaggy **=** **new** Person('Shaggy Rogers'); *// MODIFIED*

Now we can pass in a name when we create a Person object and as a result different people can have different names.

## Adding Methods To Objects

Due to the fact that functions are objects that can be stored in variables in JavaScript, we can attach methods to our objects in exactly the same way that we attach properties.

To do this we can use the this keyword, give our method a name and then assign an anonymous function as it’s value. We can then access the method using the dot (.) notation, just like we would if we were accessing a property, and invoke the method by adding brackets (()) for our argument list.

**const** Person **=** **function** (name) {

**this**.name **=** name;

**this**.greet **=** **function** () { *// NEW*

console.log(`Hi! My name is ${ **this**.name }`);

}

}

**const** shaggy **=** **new** Person('Shaggy Rogers');

shaggy.greet(); *// NEW*

*// -> Hi! My name is Shaggy Rogers*

We can use the this keyword to refer to the object that is calling our method too; in this case shaggy. This is called the context.

Now our object has some behaviour!

If we create another object using the Person constructor, it will also have an identical greet function attached to it.

**const** velma **=** **new** Person('Velma Dinkley');

velma.greet();

*// -> Hi! My name is Velma Dinkley*

We can also access the method on this new object, which is awesome, but there’s a better way to do this. We are essentially creating multiple identical copies of the greet method and attaching one to every instance.

console.log('shaggy:', shaggy);

console.log('velma:', velma);

*// -> shaggy: { name: 'Shaggy Rogers', greet: [Function] }*

*// -> velma: { name: 'Velma Dinkley', greet: [Function] }*

Instructor note: Ask the class…

Can you think of any disadvantages of storing the same method on multiple objects?

Answer

## Prototypes

Instead of adding methods in our constructor function, we can add our methods to the constructor’s prototype object.

### What is a Prototype Object?

Before we learn about prototype objects in JavaScript, let’s think about what the word prototype means in English.

The Oxford English Dictionary defines a prototype as:

“A first or preliminary version of a device or vehicle from which other forms are developed.”

In JavaScript a prototype object acts as a central store of information which all objects created via a particular constructor function can access.

### How do we do this?

When we create objects using a constructor function with the new operator, the constructor’s prototype object is assigned to it. Prototype objects are just objects with key-value pairs, like any other in JavaScript.

Let’s access the Person object prototype and add a greet method to it.

**const** Person **=** **function** (name) {

**this**.name **=** name;

}

Person.prototype.greet **=** **function** () { *// NEW*

console.log(`Hi! My name is ${ **this**.name }`);

}

We created a new key, greet, within the Person constructors prototype object and assigned a function as its value.

This method will now be stored only on the shared prototype object, instead of each Person instance.

**const** shaggy **=** **new** Person('Shaggy Rogers');

shaggy.greet();

*// -> Hi! My Name is Shaggy Rogers*

**const** velma **=** **new** Person('Velma Dinkley');

velma.greet();

*// -> Hi! My Name is Velma Dinkley*

console.log('shaggy:', shaggy);

console.log('velma:', velma);

*// -> shaggy: { name: 'Shaggy Rogers' }*

*// -> velma: { name: 'Velma Dinkley' }*

Notice that the greet function is no longer attached to each object, but they still have access to the method.

We can see these object’s prototypes using Object.getPrototypeOf.

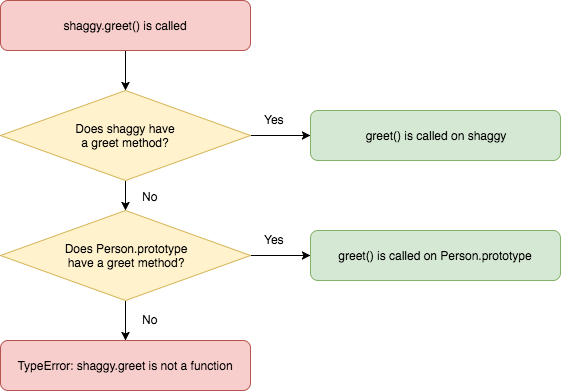
console.log("shaggy's prototype:", Object.getPrototypeOf(shaggy));

console.log("velma's prototype:", Object.getPrototypeOf(velma));

### How does this work?

If we try to call a method on an object, JavaScript first will check if the method exists on the object itself. If it doesn’t find it then it will check the object’s prototype.

Therefore when we call greet() on shaggy, JavaScript will first look for that method on shaggy. It won’t find it there so JavaScript will check the Person prototype and find it there. If the method was not found on the prototype object then JavaScript would conclude that shaggy.greet is not a function.



Calling a Method That is Defined on a Prototype Object

It is possible to create a prototypal inheritance chain but, like multiple inheritance in other languages, it is usually best to avoid it.

### Why do we do this?

Storing methods on a prototype, instead of the objects themselves, is more memory efficient. When we do this only one copy of each method exists in memory. If we were to store a copy of every method on every object that we created, then we would be using additional memory to store multiple copies of the exact same function.

### Task: (15 minutes)

1. Create a new file - pet.js
2. Create a Pet constructor
3. Assign values to the following properties in your Pet constructor via parameters:
   * name
   * species
4. Add an eat method to your Pet’s prototype. This method should accept a food parameter and output a string containing the name property and the food that was eaten. For example: 'Scooby Doo ate a Scooby Snack'.

Example solution

## Requiring and Exporting Modules

This is great so far. We can do just about anything that we could do with a class in another language. Now only one thing remains…

Let’s get our objects speaking to each other!

Note: Remove everything apart from the constructor functions and prototype methods from each file

First we’ll create another file:

touch app.js

This will be the entry point into our application where we will create objects using our constructors.

To do that we will need to require the constructors from the other files:

*// app.js*

**const** Pet **=** require('./pet.js');

**const** Person **=** require('./person.js');

The .js file extensions are optional when using require.

We’re using UpperCamelCase for Person and Pet here because we’re importing the constructor functions and storing them in variables.

Now that we have imported our constructors, let’s create an instance of Pet and Person and access their methods:

*// app.js*

**const** scooby **=** **new** Pet('Scooby Doo', 'Dog');

scooby.eat('Scooby Snack');

**const** shaggy **=** **new** Person('Shaggy Rogers');

shaggy.greet();

We aren’t done just yet. If we run app.js now we should hit an error.

node app.js

*# -> TypeError: Pet is not a constructor*

Pet is not a constructor. We didn’t tell JavaScript what we would like to export from each file, so we just got back empty objects.

Let’s tell JavaScript which objects we actually want to export from each file:

*// person.js*

module.exports **=** Person;

*// pet.js*

module.exports **=** Pet;

In this case we want to export our constructor functions.

We can think of require as a function that looks at whichever file we tell it to and gives us back the value of module.exports for that file.

### Task: (15 minutes)

* Assign a pet property in the Person constructor via a parameter
* Add a feedPet method to Person.prototype which:
  + Accepts a food parameter
  + Outputs a string containing the person and pet’s names and the food. For example: 'Shaggy Rogers fed Scooby Doo a Scooby Snack'
  + Invokes the pet’s eat method and passes the food to it

Example solution

## Recap

Instructor note: Ask the class…

Step by step: what happens when we invoke a function using the new operator?

Answer:

Which keyword can we use to refer to an object inside its constructor?

Answer:

What is the main advantage of storing methods on a prototype object?

Answer:

## Conclusion

Now we can use constructor functions to create objects with a similar set of properties and prototype objects to store methods for them.

This allows us to write much more DRY (Don’t Repeat Yourself) code than if we were to manually create these objects, as well as allowing us to ensure that all of objects of a similar type have the same set of properties and behaviours.

# NPM

## Learning Objectives

* Know what package managers are
* Understand why we use package managers
* Know what package-lock.json is
* Understand the purpose of package.json
* Understand Semantic Versioning
* Be able to install a package from npm

## Intro

We’ve learned a lot about the core language features of JavaScript. One thing that we’ve been missing so far, however, is unit testing. As with other programming languages, there are a plethora of testing frameworks available for JavaScript.

The easiest way to install and start using one of them is via a package manager.

## What is a Package Manager?

In simple terms, a package manager is a piece of software (usually a command-line interface) that allows us to easily install, update or uninstall other pieces of software. You might have used Homebrew on Mac, Chocolatey on Windows or APT/YUM on Linux distros.

## Why Do We Use Package Managers?

When developing software, package managers can be especially useful for installing libraries. A library is a package of code that someone has written to help solve a common problem. You might use a library to write a web server or to write and/or run unit tests. You might even write your own library to help complete common tasks.

Most programming languages have a widely used package manager that can be used to quickly and easily install libraries for that language. Ruby has RubyGems and Python has pip, for example.

The most widely used package manager for JavaScript is npm.

## What is npm?

npm is the world’s largest software registry. It’s used by over 7 million developers, amasses over 4 billion weekly downloads and is host to over half a million libraries!

npm also helps us to manage the dependencies, libraries that our code is dependent on, of our projects.

Imagine that two developers are working on a large project. What if one of them was to add a new dependency and forgot to tell the other? How would they know what they would have to download in order to get the project running again?

YouTube video: [npm Inc - get started with npm](https://youtu.be/x03fjb2VlGY" \t "_blank)

## Setting up a New Project

If we’re going use npm to install and manage the dependencies of a project, the first thing that we have to do is run npm init at the top level of our project.

npm init

npm will now ask you to set values for a few different options, describing your project. In the real world you would probably fill this out with real information, but it’s really not important here. We’ll just hit enter until npm asks us to confirm that everything is okay, then it will take us back to our command-line prompt.

Alternatively, we can use npm init -y to say ‘yes’ to the default options without having to hit enter to accept each one individually. We can always modify these options later.

When we run npm init npm creates a file called package.json for us.

ls

*# -> package.json*

### package.json

Right now this file only contains some metadata about our project, which is completely meaningless to us right now. It will be used by npm later on, however, to manage the dependencies of our project.

Let’s open the current directory up in Atom and take a quick look the package.json.

atom .

// package.json

{

**"name"**: "npm",

**"version"**: "1.0.0",

**"description"**: "",

**"main"**: "index.js",

**"scripts"**: {

**"test"**: "echo \"Error: no test specified\" && exit 1"

},

**"keywords"**: [],

**"author"**: "",

**"license"**: "ISC"

}

This is JSON, or JavaScript object notation. JSON looks just like a JavaScript object. All JSON is valid JavaScript, but not all JavaScript is valid JSON. The keys in a JSON object must be declared as double-quoted (") strings and their values cannot be functions.

## Installing a Package

The package that we’re going to use for the purposes of this lesson is [five](https://www.npmjs.com/package/five). Five is a simple (and fairly useless) library, which provides a set of methods which we can then use within our project.

We can install any package from the npm registry using the npm install command.

npm install five

When we use npm install npm generates three things for us: a directory called node\_modules, an updated version of our package.json and a file called package-lock.json.

ls

*# -> node\_modules*

*# -> package-lock.json*

*# -> package.json*

### node\_modules

The node\_modules directory is where npm stores the files that we have downloaded. These folders can become quite large in terms of file size, so it’s best practice to add them to your .gitignore file to prevent them from being checked in to git.

touch .gitignore

**# .gitignore**

**node\_modules/**

### package.json

If we take another look at our package.json in Atom, we should see that npm has added five as a dependency of our project.

**"dependencies"**: {

**"five"**: "^0.8.0"

}

### package-lock.json

The package-lock.json file is generated whenever npm modifies either our package.json or node\_modules. The purpose of this file is to ensure that npm will always be able to get the correct versions of our dependencies. This also applies to the dependencies of our dependencies, the version numbers of which aren’t tracked in our package.json.

If one of the packages that our project is dependent on was to be updated and introduce breaking changes, our code would stop working. In this case we might not want to update to the latest version, which is the default behaviour.

## Semantic Versioning

Let’s take a minute a to talk about the version numbers that we can see in the package.json and package-lock.json and what they mean.

**"version"**: "1.0.0"

The first thing that you might notice is that they’re split in to three distinct parts. We can refer to these parts (from left to right) as the major, minor and patch version numbers. These numbers are incremented each time a new version is released.

The patch release number is incremented if only backwards compatible bug fixes are included in this version.

The minor release number is incremented if a substantial amount of new functionality or improvements are added in this version. A minor release may also include patch level changes. The patch version will always reset to zero when a new minor version is released.

The major release number is incremented when backwards incompatible changes are released. A major release may also contain both minor and patch level changes. The patch and minor versions will always reset to zero when a new major version is released.

## Running an existing project

Imagine for a second that this was someone else’s project that we had cloned from GitHub. Their node\_modules were added to their .gitignore, so we need to install the dependencies of the project before we can run it.

Instructor note: Ask the students not to follow along here

rm -rf node\_modules

ls

*# -> package-lock.json*

*# -> package.json*

If we use the npm install command, without passing any arguments to it, npm will look at the dependencies in our package.json and install everything from the dependency list.

We can use npm i as shorthand for npm install

npm install

ls

*# -> node\_modules*

*# -> package-lock.json*

*# -> package.json*

## Using a Package

Now that we’ve learned about npm and installed a package, we can use its functionality in our project. The first thing that we’ll have to do is create a file to work in.

touch five\_play.js

We can access libraries that we have installed via npm in our code using require. If we don’t specify a relative path, ./ for example, then Node will know to look into our node\_modules directory for that package.

*// five\_play.js*

**const** five **=** require('five');

Now that we have imported the library into our codebase we can use any of the methods that it provides us with freely.

**const** five **=** require('five');

console.log(five.upHigh());

console.log(five.downLow());

console.log(five.tooSlow());

*// -> ⁵*

*// -> ₅*

*// -> 5*

## Recap

What is a package manager?

Answer

What is npm?

Answer

Where does npm store the list of our dependencies?

Answer

What does npm install do when not passed any arguments?

Answer

## Conclusion

Now that we can use npm, we have access to hundreds of thousands of packages that can help us to accomplish tasks that we would either be unable to complete on our own, or would take considerably more time.

These packages will be invaluable for things like unit testing, running a web server or connecting to a database throughout the rest of this module.

## Further Resources

npm - [package.json](https://docs.npmjs.com/files/package.json" \t "_blank)  
npm - [package-lock.json](https://docs.npmjs.com/files/package-lock.json)  
[Semantic versioning](https://semver.org/)

# Test-Driven Development Using Assert and Mocha

**Duration: 75 minutes**

## Learning Objectives

* Be able to write basic tests using Node’s Assert module
* Be able to write unit tests using Assert in combination with Mocha
* Be able to run test files with Mocha using an npm script

## Intro

We’ve learned the fundamentals of JavaScript and we’re able to use them to write some fairly robust code, but how can we be sure that our code works?

We could use console.log() to verify that our functions have the expected output, but then our code would be littered with redundant statements that aren’t relevant to its functionality.

We could remove those console.log()s once we’re satisfied that our code works, but then how would we test that our code still works if we were to refactor it?

Unit tests to the rescue!

## Assert

Node ships with a basic testing module out of the box which we can use to unit test our code. This module provides us with a set of assertion methods that can be used for testing.

These testing tools are quite primitive, in that they don’t offer much functionality, and are usually used in tandem with a testing framework like Mocha as a result.

We’ll begin by taking a look at Node’s Assert module on it’s own so that we can get a feel for how it works before we start using it with a testing framework.

## Writing Tests with Assert

The first thing that we’ll need to do is create a .js file to work in.

touch play.js

Node ships with the assert module out of the box, but if we want to use it then we have to require it.

*// play.js*

**const** assert **=** require('assert');

Now we’re ready to start writing some basic tests.

### assert.equal()

Now that we have access to the assert module, we can use any of the methods that come with it. We can use the equal() method to check if two values are the same, just like you might have done when unit testing in other languages.

assert.equal(**true**, **true**);

We can then run our tests in the same way that we would run any other .js file.

node play.js

Notice that we don’t get any feedback at all. This isn’t ideal. Node will only tell us if our tests are failing. Let’s break our test and see what happens.

assert.equal(**true**, **false**);

node play.js

*# -> AssertionError [ERR\_ASSERTION]: true == false*

Note: Assert will only show us one failing test at a time. Comment out any failing tests before moving on

This is better than nothing but it isn’t very descriptive or expressive.

Another potential issue that we could encounter when using equal() is that, as we can see from the above AssertionError, it uses JavaScript’s abstract equality operator (==).

If the two parameters are of different types then JavaScript will first try to find a common type for them before determining whether or not they are equal.

The following test will convert the string '1' to a numerical value before performing a strict comparison on them, deciding that they are equal and passing.

assert.equal(1, '1');

Just like using the abstract equality operator (==) elsewhere in our code this can lead to unexpected behaviours, such as the following test passing.

assert.equal([], **!**[]);

We should be as specific as possible when testing therefore avoid using equal().

### assert.strictEqual()

Assert gives us another method, strictEqual(), which uses JavaScripts strict equality operator (===). As a general rule of thumb we should use that instead of equal() so that we can be certain that our tests are passing or failing for the right reasons.

The following test will fail.

assert.strictEqual(1, '1');

node play.js

*# -> AssertionError [ERR\_ASSERTION]: 1 === '1'*

The two values are not strictly the same so this is typically the desired behaviour of our tests.

There is one more thing that we need to know when writing our assertions with Assert: when comparing objects using equal() or strictEqual() JavaScript will check if they are the same object.

The following test will fail as a result of this.

assert.strictEqual([1, 2, 3], [1, 2, 3]);

The arrays look the same but they aren’t physically the same array.

The following test, however, will pass because both variables refer to the exact same array.

**const** firstArray **=** [1, 2, 3];

**const** secondArray **=** firstArray;

assert.equal(firstArray, secondArray);

### assert.deepEqual() and assert.deepStrictEqual()

Assert gives us another set of methods; deepEqual() and deepStrictEqual(). These methods look at the values contained within the object and use those to determine if the objects are equal, rather than checking if the object are the same object.

The following test will pass because even though the first and second parameters are not the same array they contain values which are equal.

assert.deepEqual([1, 2, 3], [1, 2, 3]);

Similarly to equal(), deepEqual() uses JavaScript’s abstract equality operator (==). We can mix Strings and Numbers and the test will still pass.

assert.deepEqual([1, 2, 3], ['1', '2', '3']);

We can use deepStrictEqual(), which uses JavaScript’s strict equality operator (===) to avoid this behaviour. The following test will fail because the contents of the array are not strictly the same values.

assert.deepStrictEqual([1, 2, 3], ['1', '2', '3']);

node play.js

*# -> AssertionError [ERR\_ASSERTION]: [ 1, 2, 3 ] deepStrictEqual [ '1', '2', '3' ]*

Third-party assertion libraries are available if you need something more fully featured but Assert should be able to take care of all of our needs for the time being.

## Mocha

Node’s built in assert module is very basic as you can see. It gives us some methods that we can use to test our code but that’s all. It doesn’t even tell us if our tests are passing. If we only used assert to test our code then our test files could quickly become an unintelligible mess of calls to equal() unless we wrote our own testing framework to organise them.

Mocha has some nice features that help us to organise our tests and gives us much more readable and descriptive output.

## Writing Unit Tests With Mocha and Assert

Now that we’ve seen how some of the methods that we can get from Assert work, let’s create a model, employing TDD, and look at how we can use Mocha to better organise and run our tests.

Before we do, it’s worth mentioning that Mocha supports a number of different syntaxes. The type we’re going to use is Behaviour Driven Development, or BDD.

BDD is an extension of TDD which attempts to focus on the user, and the product. Tests written in a BDD style will follow the format “It should…”, and they should tie in closely to the user stories that you or your UX colleagues should have written.

(As a… I want to… So that…)

The first thing that we’ll need to do is create some files to work in. We’re going to model a taxi, so we’ll need a file for our Taxi model and corresponding spec file. We typically create a specs folder to keep our tests organised separately from our models and name our test files the same as our models with a \_spec suffix. For example, the test file for our taxi model will be called taxi\_spec.js.

touch taxi.js

mkdir specs

touch specs/taxi\_spec.js

We’re going to be using Mocha to write and run our tests, so we have to install it using npm.

We’ll use Mocha during development but our tests aren’t necessary to actually run our application, so we’ll save Mocha as a dev dependency.

If someone was to then clone our project with the intention of running it without modifying the code, they could use npm install --production to avoid installing our dev dependencies.

npm init

npm install --save-dev mocha

We have a test file and we’ve installed Mocha, so now we have everything that we need. How do we run our tests?

We want to use Mocha to run all of the files in our specs folder.

mocha specs

*# -> zsh: command not found: mocha*

We installed Mocha as a dependency of our project, but our Terminal has no idea what Mocha is.

There are a few ways that we could solve this problem: we could install Mocha globally, which would allow us to use this command, but then if another developer were to try to run our code then they would have to install Mocha too. Instead we can use an npm script to tell npm to run our tests with Mocha. We installed Mocha using npm, so npm knows what Mocha is.

We can create npm scripts by adding a name to refer to them by and the command that we want to execute as a key-value pair to the "scripts" object in our package.json.

npm expects us to have a "test" script, so it provides one by default. We can then update it’s value so that it runs our tests for us.

*// package.json*

"scripts": {

"test": "mocha specs" *// MODIFIED*

},

Now that we’ve added our npm script we can run it using npm test while we’re at the same level as our package.json.

npm test

*# -> 0 passing*

When we run npm test npm looks at the scripts that we have defined in our for the key "test", when it finds it executes the string value in Terminal for us; in this case mocha specs.

The script runs okay but we have 0 passing tests because we haven’t written any tests yet.

If we want to start writing tests then the first thing that we have to do is require Assert and the model that we want to test. We’ll be testing our Taxi model.

*// taxi\_spec.js*

**const** assert **=** require('assert');

**const** Taxi **=** require('../taxi.js');

### describe()

Before we write our first test let’s take a look at one of the organisational functions that Mocha gives us: describe().

Describe can be used to group similar tests. In this case we are going to use describe() to label our tests with the name of the relevant model. This will be displayed in the Terminal output when we run our tests. Later on when we have a lot of different models this will make the output of our tests a lot easier to read.

We’ll call Mocha’s describe() function and pass it two arguments:

1. The name of the model that we’re testing as a String, in this case 'Taxi'
2. A function which will contain all of the tests associated with the thing that we’re describing. This syntax might look a little bit strange right now but we’ll learn more about this very soon.

describe('Taxi', **function** () {

});

### it()

Inside of the function that we’re passing to describe() we can use the it() function for each of our test cases.

For the moment, we’re just going to pass a single argument to it(): a String describing the test case.

We’re going to add a manufacturer property to our Taxi so let’s describe our test accordingly.

describe('Taxi', **function** () {

it('should have a manufacturer');

});

When combined with it() our test names should be readable and expressive. For example:

* it('should have a name')
* it('can calculate the number of days until Christmas')

Ideally, these should come from the user stories that you or your colleagues have written. This will help to keep you focussed on your product, and your MVP.

This also makes the output of our tests meaningful and errors easier to interpret as a result. If we run npm test now we can see what our test output will look like.

Notice that this test is listed as being pending. Pending tests are just tests that we haven’t tackled yet - they’re neither passing, nor failing.

It can be quite useful to write a few pending tests at once, so that we can see what we have to do. Let’s add another:

describe('Taxi', **function** () {

it('should have a manufacturer');

it('should have a model');

});

Now, we should have two pending tests. Let’s get started on writing the body of the test.

In order to write the body of the test, just as we did with describe, we have to pass a function as the second argument to it().

describe('Taxi', **function** () {

it('should have a manufacturer', **function** () {

});

it('should have a model', **function** () {

});

});

This anonymous function will contain the setup for our test, and our assert.

### Arrange-Act-Assert

As with many other testing frameworks we can use the arrange-act-assert pattern here.

1. Arrange: Perform any setup that might be required for the test
2. Act: Perform the action that we want to test
3. Assert: Check that our action had the expected result

describe('Taxi', **function** () {

it('should have a manufacturer', **function** () {

**const** taxi **=** **new** Taxi('Toyota', 'Prius'); *// Arrange*

**const** actual **=** taxi.manufacturer; *// Act*

assert.strictEqual(actual, 'Toyota'); *// Assert*

});

it('should have a model', **function** () {

**const** taxi **=** **new** Taxi('Toyota', 'Prius');

**const** actual **=** taxi.model;

assert.strictEqual(actual, 'Prius');

});

});

When employing TDD we should only be working on one test at a time. We can mark any tests that we don’t want to run with xit(), which will tell Mocha to skip them. Let’s skip the second test, so that we can concentrate on the first.

xit('should have a model', **function** () { *// MODIFIED*

**const** taxi **=** **new** Taxi('Toyota', 'Prius');

**const** actual **=** taxi.model;

assert.strictEqual(actual, 'Prius');

});

We’ve written our test but it should fail because we haven’t written the code to make it pass yet. We should always run our test at this point so that we can see it fail.

If you’ve never seen a test fail then you can’t be sure that it’s a good test, or that it tests anything at all.

npm test

*# -> 1 failing*

*# -> TypeError: Taxi is not a constructor*

Now we know exactly where to start. Let’s create the constructor function for our Taxi, making sure not to forget our module.exports.

*// taxi.js*

**const** Taxi **=** **function** (manufacturer) {

**this**.manufacturer **=** manufacturer;

}

module.exports **=** Taxi;

Now we should be able to run our test again and it should pass.

npm test

*# -> 1 passing*

Next we’re going to add a model property to our Taxi so we have to tell Mocha not to skip the second test.

*// taxi\_spec.js*

describe('Taxi', **function** () {

*// ...*

it('should have a model', **function** () { *// MODIFIED*

**const** taxi **=** **new** Taxi('Toyota', 'Prius');

**const** actual **=** taxi.model;

assert.strictEqual(actual, 'Prius');

});

});

Again, we should run our test so that we can see it fail. If it passed already then we would know that we had made a mistake.

npm test

*# -> 1 passing*

*# -> 1 failing*

*# -> AssertionError [ERR\_ASSERTION]: undefined === 'Prius'*

taxi.model gives us back undefined, not the String that we want. That’s because we haven’t set that property in the constructor yet.

Let’s go ahead and add that model property to our Taxi.

*// taxi.js*

**const** Taxi **=** **function** (manufacturer, model) {

**this**.manufacturer **=** manufacturer;

**this**.model **=** model;

}

Now we should be able to run our tests again and see them both pass.

npm test

*# -> 2 passing*

### beforeEach()

So far, so good. There’s some repetition in our test cases though. We’re creating a new Taxi object in each test.

Mocha gives us some handy hooks which we can use to execute code at specific points during testing. In this case we want to create an object before each test, so we can use beforeEach().

*// taxi\_spec.js*

describe('Taxi', **function** () {

beforeEach();

*// ...*

});

In contrast to the Mocha functions that we’ve used so far beforeEach() doesn’t require a String. We’ll just pass it the function that we want to execute before each of our test cases.

**let** taxi;

beforeEach(**function** () {

taxi **=** **new** Taxi('Toyota', 'Prius');

});

We have to declare any variables outside of beforeEach() to prevent them from being scoped locally to that function.

We can now remove the Taxi objects that we created in each test and use the one that we’re creating before each test using the beforeEach() hook.

it('should have a manufacturer', **function** () {

*// const Taxi = new Taxi('Toyota', 'Prius'); REMOVED*

**const** actual **=** taxi.manufacturer;

assert.strictEqual(actual, 'Toyota');

});

it('should have a model', **function** () {

*// const Taxi = new Taxi('Toyota', 'Prius'); REMOVED*

**const** actual **=** taxi.model;

assert.strictEqual(actual, 'Prius');

});

### Task: 5 minutes

Add a driver property to our taxi. This should be a String containing the driver’s name.

* Write a test to ensure that our taxi has a driver.
* Add a driver property to the taxi.

Example solution

### Nested describe()s

describe()s can be nested within each other. This can be useful for grouping similar tests. We might have several tests cases to test the various outcomes of one piece of functionality, for example.

We’re going to add an array of passengers to our Taxi, so let’s use another describe() to group all of our tests relating to passenger functionality together.

*// taxi\_spec.js*

describe('Taxi', **function** () {

*// ...*

describe('passengers', **function** () {

});

});

Note: You can skip entire describe blocks using xdescribe.

We can use then use it() for any test cases relating to this functionality, just like we did earlier.

First let’s write a test to assert that we initially have an empty array of passengers.

describe('passengers', **function** () {

it('should start with no passengers');

});

Remember: if we want to assert that two array objects contain the same values then we have to use assert.deepEqual().

describe('passengers', **function** () {

it('should start with no passengers', **function** () {

**const** actual **=** taxi.passengers;

assert.deepStrictEqual(actual, []);

});

});

Let’s run our test and see it fail.

npm test

*# -> 3 passing*

*# -> 1 failing*

*# -> AssertionError [ERR\_ASSERTION]: undefined deepEqual []*

Our test fails because taxi.passengers is undefined and we want it to be an empty array. That’s because we haven’t defined it in our constructor.

Let’s add that empty array of passengers to our Taxi constructor now.

*// taxi.js*

**const** Taxi **=** **function** (manufacturer, model, driver) {

*// ...*

**this**.passengers **=** []; *// NEW*

}

Now our test should pass.

npm test

*# -> 4 passing*

Next we’re going to extend our Taxi, adding additional functionality for the passengers array.

### Task: 20 minutes

Employing TDD, add the following methods to your taxi:

* numberOfPassengers
* addPassenger
* removePassengerByName
* removeAllPassengers

A passenger should be represented as a String containing the passenger’s name.

Example Solution

## Recap

Instructor note: Ask the class…

Which methods did we use from Node’s Assert module?

Answers

What is the difference between equal() and strictEqual()?

Answer

What is the difference between equal() and deepEqual()?

Answer

How can we set up an npm script to run our tests for us?

Answer

Which Mocha function can we use to group similar tests?

Answer

## Further Resources

* [Assert Documentation](https://nodejs.org/api/assert.html)
* [Mocha Documentation](https://mochajs.org/)
* [Introducing BDD](https://dannorth.net/introducing-bdd/)

## Conclusion

We can now write descriptive and well organised unit tests, which will help us to determine whether or not our code is working. Providing that we write good tests they can even ensure that we write better code.

Once our unit tests are in place we can safely refactor without having to worry about breaking anything. If our tests still pass then we know that our code is still working.

Our code will be better and more maintainable as a result of unit testing.

# Lab: TDD Painter Decorator

**Suggested Lab Duration: 90 minutes**

### Learning Objectives

* Be able to write unit tests using Assert in combination with Mocha
* Be able to run test files with Mocha using an npm script
* Be able to create an application using multiple interacting objects

## Brief

Your task is to create an application that allows a painter decorator to manage paint stock and decorating jobs. You application should allow a user to calculate how much paint is required to paint a room, and whether they have enough paint in stock to paint the room.

**Note: Work on the assumption that one litre of paint covers one square meter.**

### MVP

A room should:

* have an area in square meters
* should start not painted
* should be able to be painted

A paint can should:

* have a number of litres of paint
* be able to check if it is empty
* be able to empty itself of paint

A decorator should:

* start with an empty paint stock
* be able to add a can of paint to paint stock
* be able to calculate total litres paint it has in stock
* be able to calculate whether is has enough paint to paint a room
* be able to paint room if has enough paint in stock

### Extensions

A decorator should:

* should be able to decrease amount of paint in stock when painting a room
* should be able to remove empty paint cans from stock

Hint: We want to avoid removing items from arrays while iterating over them. This is because iteration uses the index number to access the current item, and if you remove an item during one of the iterations, the position of each of the following items will be changed and lead to unexpected results.

### Planning

To setup your project you will need a models folder with a specs folder inside. You will also need to:

* Initialise npm with npm init
* Install mocha as a dev dependency with npm install -D mocha
* Update the npm test script in the package.json run your spec files with mocha using mocha models/specs
* Create a .gitignore file to ignore the node\_modules directory

### Considerations

Remember to use the appropriate assert method for the data types you are comparing in your tests

# Homework: Jurassic Park

### Learning Objectives

* Be able to use the fundamental JavaScript language features and data types
* Be able to create an application using multiple interacting objects

## Brief

John Hammond has tasked you with creating an app to help him manage his theme park. He needs to be able to track and manage information about the various dinosaurs in his park and the number of visitors that they will attract.

Use the start point provided, which contains the tested Dinosaur model and the uncompleted tests for the MVP Park model. (You will need to write your own tests for the extension tasks.)

### MVP

A dinosaur must have:

* A species
* A diet (e.g. carnivore, herbivore or omnivore)
* An average number of visitors attracted per day

A park must have:

* A name
* A ticket price
* A collection of dinosaurs

A park must be able to:

* Add a dinosaur to its collection of dinosaurs
* Remove a dinosaur from its collection of dinosaurs
* Find the dinosaur that attracts the most visitors
* Find all dinosaurs of a particular species
* Calculate the total number of visitors per day
* Calculate the total number of visitors per year
* Calculate the total revenue from ticket sales for one year

### Extensions

A park must be able to:

* Remove all dinosaurs of a particular species
* Provide an object containing each of the diet types and the number of dinosaurs in the park of that diet type  
  Example: { 'carnivore': 5, 'herbivore': 2, 'omnivore': 1 }

**Hint**: We want to avoid removing items from arrays while iterating over them. This is because iteration uses the index number to access the current item, and if you remove an item during one of the iterations, the position of each of the following items will be changed and lead to unexpected results. You will need to keep this principle in mind when removing all the dinosaurs of a particular species.

### Planning

By looking in the package.json you can see that the project already has mocha installed an a dependency. To get the package run npm install in the same directory as the package.json, which generates the node\_modules.

Looking at the package.json, we can also see that a script has been added to run our tests with mocha, so we can run npm test to run the tests.

### Considerations

Remember to use the appropriate assert method for the data types you are comparing in your tests.

# Homework: Jurassic Park

### Learning Objectives

* Be able to use the fundamental JavaScript language features and data types
* Be able to create an application using multiple interacting objects

## Brief

John Hammond has tasked you with creating an app to help him manage his theme park. He needs to be able to track and manage information about the various dinosaurs in his park and the number of visitors that they will attract.

Use the start point provided, which contains the tested Dinosaur model and the uncompleted tests for the MVP Park model. (You will need to write your own tests for the extension tasks.)

### MVP

A dinosaur must have:

* A species
* A diet (e.g. carnivore, herbivore or omnivore)
* An average number of visitors attracted per day

A park must have:

* A name
* A ticket price
* A collection of dinosaurs

A park must be able to:

* Add a dinosaur to its collection of dinosaurs
* Remove a dinosaur from its collection of dinosaurs
* Find the dinosaur that attracts the most visitors
* Find all dinosaurs of a particular species
* Calculate the total number of visitors per day
* Calculate the total number of visitors per year
* Calculate the total revenue from ticket sales for one year

### Extensions

A park must be able to:

* Remove all dinosaurs of a particular species
* Provide an object containing each of the diet types and the number of dinosaurs in the park of that diet type  
  Example: { 'carnivore': 5, 'herbivore': 2, 'omnivore': 1 }

**Hint**: We want to avoid removing items from arrays while iterating over them. This is because iteration uses the index number to access the current item, and if you remove an item during one of the iterations, the position of each of the following items will be changed and lead to unexpected results. You will need to keep this principle in mind when removing all the dinosaurs of a particular species.

### Planning

By looking in the package.json you can see that the project already has mocha installed an a dependency. To get the package run npm install in the same directory as the package.json, which generates the node\_modules.

Looking at the package.json, we can also see that a script has been added to run our tests with mocha, so we can run npm test to run the tests.

### Considerations

Remember to use the appropriate assert method for the data types you are comparing in your tests.

**Callback Functions & Enumeration**

**Lesson Duration: 120 minutes**

**Learning Objectives**

* Understand the implications of functions being first-class objects
* Be able to interrogate documentation
* Be able to pass functions to higher-order functions
* Be able to use built-in Array enumeration methods

**Introduction**

In this lesson we are going to look at the key language features of JavaScript, higher-order functions and callbacks.

A higher order function accepts or returns another function. A callback is a function that is passed to a function as an argument.

Using higher-order functions and callbacks allow us to write more dynamic code and form the foundations of the event-driven programming that we will be doing for the browser later in the course.

Today we are going to be using some of JavaScript’s built in higher-order functions that handle iterating over arrays; the enumeration methods.

**Higher-order Functions and Callbacks**

We have seen that in JavaScript, functions are first class objects. This means they can be stored in variables and data structures; and passed as arguments to other functions.

A higher order function is a function that takes a function as an argument or returns a function.

A callback is a function that is passed to another function as an argument.

Before we start writing our own higher-order functions, let’s look at using some that are built in JavaScript methods.

**Enumeration Methods**

We know we can iterate over an array using a for of loop, but there are a number of enumeration methods on the Array prototype that enable us to do the same and some offer us extra functionality. These enumeration methods are higher-order functions; they take in a callback which they invoke for each element of the array.

**forEach**

We have seen that we can loop through an array using the for of loop. Let’s use it to log out each element of an array.

touch enumeration.js

**const** myNumbers **=** [1, 2, 3, 4, 5];

**for** (**const** number **of** myNumbers) {

console.log(`This is number ${number}`)

}

We can achieve exactly the same result using the enumeration method forEach. forEach is a method on the Array prototype and is a higher-order function, in that it takes in a callback as an argument. forEach will loop over the array (myNumbers), and for each iteration will invoke the callback, passing the callback the current element (number).

*// ...*

myNumbers.forEach((number) **=>** {

console.log(`This is number ${number}`);

})

Note: We do not invoke the callback that we pass to forEach. We pass the whole function declaration and forEach is responsible for invoking it for each iteration of the loop, passing it the current number.

**Using the MDN docs**

Look at the documentation for the forEach method on the [MDN docs](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach).

In the documentation, any arguments in square brackets are optional. We can see that forEach takes a callback function, and that forEach will pass the callback function the current element of the array, and also optionally the current index position.

Let’s see how we would modify the message to make use of the element’s index position.

myNumbers.forEach((number, index) **=>** { *// MODIFIED*

console.log(`This is number ${number} at index position ${index}`); *// MODIFIED*

});

**Using forEach to manipulate an array**

We are going to write a function that iterates over the array of numbers and returns a new array with each number from the original array multiplied by two. To do this we are going to use forEach to handle the iterating.

We will start by declaring a function that takes in an array of numbers, creates a new array and returns it. This new array is the array we are going to populate with the multiplied numbers.

*// ...*

**const** multiplyByTwo **=** **function** (numbers) { *// NEW*

**const** result **=** [];

**return** result;

}

Now we are going to use forEach to iterate over the numbers array, multiply each number by two, and push the multiplied number into the new array. forEach takes a callback function, and it will invoke our callback for each iteration, passing the callback the current element from the array (number).

**const** multiplyByTwo **=** **function** (numbers) {

**const** result **=** [];

numbers.forEach((number) **=>** { *// NEW*

});

**return** result;

}

We can now multiply number by two and push it into the new array.

**const** multiplyByTwo **=** **function** (numbers) {

**const** result **=** [];

numbers.forEach((number) **=>** {

**const** multiplied **=** number **\*** 2; *// NEW*

result.push(multiplied); *// NEW*

});

**return** result;

}

console.log(multiplyByTwo(myNumbers)); *// NEW*

Now when we run the code, as we have logged out the return value of multiplyByTwo we can see we have returned an array with each number multiplied by two.

**Return Value of forEach**

Instructor note: Ask the class…

Looking at the docs again, what is the return value of forEach?

**Answer:**

forEach doesn’t return a value and we can’t return anything from the callback that we pass to it. If we try and return from the callback, we are returning into the forEach and as forEach is implemented in a way that it doesn’t do anything with the return value, for have no way of accessing that value. Instead we have to manually handle the value. In this case we have added the modified element into the previously declared empty array. In this way forEach is a direct replacement for a for loop.

**Task: (10 minutes)**

Using the forEach enumeration method, complete the following tasks:

1. Write a function called getEvens that takes an array of numbers as an argument and returns a new array that only contains the even numbers from the original array.
2. Write a function called sumElements that takes an array of numbers as an argument and returns the sum total of all the elements in the array.

**Hint:** If you get stuck, start by writing the function using a for of loop, then refactor, replacing the for loop with the forEach.

**Example Solution:**

**Other enumeration methods (map, filter, reduce)**

map, filter and reduce are three commonly used enumeration methods on the Array prototype. They are similar to forEach, in that they take a callback and iterate over the array invoking the callback for each element, passing in the element, but they each have some extra inbuilt functionality.

**Paired discussion (5 minutes)**

1. Look at the MDN docs and find out the return value of map, filter and reduce and think of a situation when you might want to use each of them.
2. Using what you found out in question one, decide which of these enumeration methods would be useful for which of our functions, multiplyByTwo, getEvens, sumElements.

**Answers:**

**map**

Let’s refactor multiplyByTwo to use map to handle the iterating. We know that the return value of map is a new array, so we don’t need to declare our own empty array. Instead let’s store the the array that map will return to us in a variable.

**const** multiplyByTwo **=** **function** (numbers) {

**const** result **=** numbers.map(); *// MODIFIED*

**return** result;

}

map takes a callback as an argument.

*// ...*

**const** multiplyByTwo **=** **function** (numbers) {

**const** result **=** numbers.map(() **=>** {}); *// MODIFIED*

**return** result;

}

Looking at the MDN docs, what parameter will our callback need, and what does it need to return? It needs a parameter to accept the current element of the array and it must return the value to be added to the new array.

**const** multiplyByTwo **=** **function** (numbers) {

**const** result **=** numbers.map((number) **=>** { *// MODIFIED*

**return** number **\*** 2; *// NEW*

});

**return** result;

}

When we run the code, we can see we have returned the array with the numbers multiplied by two.

Note: If we forget to return from the callback, we will get an array of undefineds of the same length as the original length.

**filter**

We are now going to refactor getEvens to use the enumeration method filter, to filter the original array and return a new array of only even numbers. As filter gives us a filtered array, we don’t need to declare an empty array. Instead let’s store the array that filter will return to us in a variable.

*// ...*

**const** getEvens **=** **function** (numbers){

**const** result **=** numbers.filter(); *// MODIFIED*

**return** result;

}

We can see from looking at the MDN docs that filter takes a callback. What parameter will our callback need, and what does the callback need to return?

1. The callback will need a parameter for the current element of the array (number)
2. The callback needs to return true if we want to keep the element, and false it we want to lose it. In this case we return true if the number is even, and false if not.

**const** getEvens **=** **function** (numbers){

**const** result **=** numbers.filter((number) **=>** { *// MODIFIED*

**return** number **%** 2 **===** 0; *// NEW*

});

**return** result;

}

We have now used the enumeration method filter to return an array of just the even numbers.

**reduce**

Lastly, we will refactor sumElements to use reduce to sum all the elements of the original array and return the total. Reduce can be used in many situations as it can keep track of a value while it iterates. In our case we want to keep track of the total.

We will use reduce to return the final total, so we don’t need to declare a starting total. Instead we will save the return value of reduce to a variable.

*// ...*

**const** sumElements **=** **function** (numbers) {

**const** total **=** numbers.reduce(); *// MODIFIED*

**return** total;

}

We can see from looking at the MDN docs that reduce takes a callback. What parameter will our callback need?

The callback will need two parameters:

1. An accumulator. This is the parameter that reduce uses to keep track of while iterating. In our case it will be the runningTotal.
2. The current item in the array (number).

**const** sumElements **=** **function** (numbers) {

**const** total **=** numbers.reduce((runningTotal, number) **=>** { *// MODIFIED*

});

**return** total;

}

What does the callback need to return? Whatever we return from the callback, will become the accumulator (runningTotal) on the next iteration. This is how reduce keeps track of a value while iterating. In this case, we want to keep track of the running total, so we will return the runningTotal plus the current number. When reduce has finished looping, it will finally return runningTotal.

*// ...*

**const** sumElements **=** **function** (numbers) {

**const** total **=** numbers.reduce((runningTotal, number) **=>** {

**return** runningTotal **+** number;

});

**return** total;

}

Reduce takes a second argument, which is how we set the accumulator (runningTotal) for the first iteration. In this case we want it to start as 0.

*// ...*

**const** sumElements **=** **function** (numbers) {

**const** total **=** numbers.reduce((runningTotal, number) **=>** {

**return** runningTotal **+** number;

}, 0); *// MODIFIED*

**return** total;

}

If we run the code, we can see we have now used reduce to sum the elements of the array.

We have seen that forEach can be used to achieve the same results as using these enumeration methods, but the benefits of using the enumeration methods with built-in functionality where appropriate include:

* cleaner, more readable and expressive code
* inversion of control - we don’t have to worry about the implementation details of how JavaScript maps, filters or reduces

**How are these higher-order functions implemented?**

We have used some of JavaScript’s enumeration methods, which we know are higher-order functions because they take in a callback. Let’s have a look at what they are doing with the callback that we pass to them.

If we were writing our own version of forEach, it would need to take in two arguments:

1. The array, because we are not writing this on the Array prototype we need to get access to the array by taking it in as argument.
2. The callback that will be invoked for each element in the array.

Then we will iterate over the array, invoking the callback, passing in the current element on each iteration.

**const** ourForEach **=** **function** (array, callback) {

**for** (**const** element **of** array) {

callback(element);

}

};

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

ourForEach(numbers, (number) **=>** {

console.log('the number is:', number);

});

**Recap**

What is a higher-order function?

**Answer:**

What is a callback?

**Answer:**

What are the benefits of using JavaScripts enumeration methods?

**Answer:**

What do all the enumeration methods take as an argument?

**Answer:**

Do we invoke the callback that we pass to the enumeration method?

**Answer:**

What happens if you try and return from the callback that you pass to forEach?

**Answer:**

What will be the return value of map be if we don’t return anything from the callback we pass it?

**Answer:**

**Conclusion**

We have written callbacks and passed them to a number of JavaScript enumeration methods, which are higher-order functions.

We started by using forEach to iterate over the array, manually creating the value we wanted to return from our function.

After refactoring the code to use map, filter and reduce we achieved the same results, but by using the functionality of these methods we were able to write cleaner more expressive code.

**Further Resources**

*JavaScript Enumeration Methods Table*

# Enumeration and Callbacks Lab: Cinema

**Lab Duration: 90 minutes**

### Learning Objectives

* Be able to pass functions to higher-order functions
* Be able to use built-in Array enumeration methods

## Brief

You have been given a project with two models, Cinema and Film, and their corresponding test files.

A Film has:

* a title
* a genre
* a year
* a length

A Cinema has:

* an array of Films

You should write the code to make the Cinema tests pass. You should use JavaScript’s built-in enumerator methods, only using forEach if you can’t find a way to use one of the other more appropriate methods.

Hint: You may want to explore other enumeration methods from the MDN documentation. For example, find works very similarly to filter, but rather than returning an array of items that match a condition, it returns the first item. Other useful methods include some and every.

### MVP

Cinema:

* should have a collection of films
* should be able to get a list of film titles
* should be able to find a film by title
* should be able to filter films by genre
* should be able to check whether there are some films from a particular year (true/false)
* should be able to check whether all films are over a particular length (true/false)
* should be able to calculate total running time of all films

### Extensions

Add a another test, 'Cinema should be able to filter films by year'.

We already have a method that filters films by genre, the functionality of which is very similar. We don’t want two separate methods as that wouldn’t be DRY. Your task is get the final test to pass by to writing a new method called filmsByProperty, which takes two arguments:

1. the name of the property
2. the value being search for

Once the final test is passing, modify the test 'Cinema should be able to filter films by genre' to use the new filmsByProperty method.

### Consideration

If you use reduce, remember that you will need to pass in the initial value of the accumulator as the second argument

# Enumeration Homework: Traveller

### Learning Objectives

* Be able to pass functions to higher-order functions
* Be able to use built-in Array enumeration methods

## Brief

You have been given a project with two models, Traveller and Journey, and their corresponding test files.

A Journey has:

* a start location
* an end location
* a mode of transport
* a distance in miles

A Traveller has:

* an array of Journeys

You should write the code to make the Traveller tests pass, without modifying the spec files. You should use JavaScript’s built-in enumerator methods, only using forEach if you can’t find a way to use one of the other more appropriate methods.

### MVP

Traveller:

* should have a collection of journeys
* should be able to get the journeys start locations
* should be able to get the journeys end locations
* should be able to get journeys by transport
* should be able to get journeys over a certain distance

### Extensions

Traveller:

* should be able to calculate total distance travelled
* should be able to get a unique list of modes of transport

Note: Remember to remove the x from xit() on the pending tests to run them.

# Enumeration Homework

### Learning Objectives

* Be able to pass functions to higher-order functions
* Be able to use built-in Array enumeration methods

## Brief

You have been given four projects, each containing a coding problem. You should write the code to make the tests pass, without modifying the spec files. You should use JavaScript’s built-in enumerator methods where appropriate.

You should attempt to make your code as clean as possible. You don’t have to do all the work in the function that’s being called in the test. Don’t be afraid to attach little helper functions to the provided prototype.

Note: For now, make sure you use arrow functions for your callbacks to avoid problems with context. We will be looking more at the differences between function declaration types soon.

## UPPERCASER

map an array of strings to a new array containing uppercase versions of each string.

## Pangram Finder

A pangram is a sentence or phrase which contains every letter of the alphabet. “The quick brown fox jumps over the lazy dog.” is probably the most notable pangram in English.

Given a sentence or phrase you should be able to determine whether or not every letter of the alphabet is included in it.

## Isogram Finder

An isogram is a word, phrase or sentence that does not contain any repeated letters. “Orange” is an isogram but “Banana” is not.

Given a word, phrase or sentence you should be able to determine whether or not it is an isogram. That is, you should be able to determine whether every letter is unique.

## Extension: Anagram Finder

An anagram is a word formed by rearranging the letters of another word. Listen is an anagram of silent, for example.

Given a word and an array of other words you should be able to filter the array, leaving only the anagrams of the word in question in the array.

**The Document Object Model**

**Lesson Duration: 90 minutes**

**Learning Objectives**

* Know what the DOM is
* Understand what the DOM allows us to do
* Understand what the window and document objects are
* Understand the purpose of Chrome Dev Tools Console
* Be able to use Chrome Dev Tools Elements to investigate the current state of the DOM
* Be able to query the DOM
* Be able to modify existing DOM elements
* Be able to create DOM elements
* Be able to insert elements into the DOM

**Intro**

In today’s world, JavaScript is used everywhere; to write desktop and mobile applications, to automate behaviours in software (such as Adobe PhotoShop or Google Sheets), on servers, in hardware, and even to give instructions to robots!

In this lesson we’re going look at JavaScript in the environment for which it was written, the web browser. We are also going to look at some of the tools we can use to debug our applications when working in the browser.

JavaScript enables us to create dynamic, interactive web applications that feel more like native software than websites. This is achieved by using JavaScript to update the page in real-time, rather than making additional requests to a server for new information. The DOM is the browser’s interface to developers, and it allows us to do just that.

**About the DOM**

The Document Object Model (DOM) consists of of the following:

**1. A node-tree**

The DOM’s node-tree is a representation of all the elements that make up the current page.

*DOM node-tree*

When the browser receives HTML in response to an HTTP request, it parses the HTML and creates the node-tree, which is the browser’s representation of the HTML that has been written.

**2. Methods and properties**

The DOM provides us with methods and properties that allow us to manipulate the node-tree and handle interaction using JavaScript.

The DOM allows us to change the structure, style, and content of a web page using JavaScript. These changes could include:

1. modifying the data being displayed
2. changing the styling of an element
3. adding behaviour in response to events
4. creating and displaying new elements

These changes can be made in response to events, for example, user interaction or when new data received.

Open [BBC Weather](https://www.bbc.co.uk/weather/0/8299621) in Google Chrome

On the BBC weather website, if we:

1. select a day of the week by clicking on a tab
2. click on an hour of the day to see more detail

we can see that JavaScript is being used to:

1. change the css to increase the tab size
2. update the data to the selected day’s weather
3. create and display the expandable detail section

**Inspecting the DOM**

Most web browsers come with a built-in DOM viewer. We’re going to take a closer look at Google Chrome’s in the devtools.

To open up the devtools, we can either right-click on the page and choose Inspect, or we can use the keyboard shortcut cmd + shift + C.

In the Elements tab we can see the DOM’s node-tree, with all the elements that make up the page. It has a <head> and <body> sections, with <div>s that have IDs and classes.

*Elements tab in devtools displaying the DOM’s node-tree of page elements*

**Manipulating the DOM**

We can update the DOM directly in the devtools.

1. Click on the select tool which enables us to to select an element from the page.
2. Click on a element on the page, which focusses the element in the devtools.
3. Change the text in the devtools. When we hit enter, the page is updated.

*Manipulating the DOM in the devtools*

It is important to understand that the DOM is not always the same as the HTML. The DOM is initially a representation of the HTML because the DOM is created by parsing the HTML, but once this process is complete, the DOM may be manipulated by JavaScript, at which point it differs from the HTML.

In the case of the BBC weather web page, we updated the DOM in devtools, but this only changed what the browser displays, not the HTML or CSS files themselves. The DOM viewer allows us to see the current state of the page.

Updating the DOM in the devtools in a neat party-trick, but usually we’d interact with the DOM programmatically, using JavaScript. Let’s look at how we would go about that.

**Manipulating the DOM with JavaScript in the Console**

Instructor note: Hand out the start point

To run the start point, open index.html in the browser. You will see in the devtool’s Elements tab that the browser has parsed the HTML to create the DOM, which is being used to display the text, ‘Hello World!’ in an <h1>.

**window and document**

We want to modify the DOM using JavaScript, so how do we do that?

The window object represents the browser’s window. The window object is known as the browser’s *global* object. It is the main context in which our code runs. When we use console.log, we’re actually using window.console.log.

Prior to working in the browser, Node was the environment our code was being run in. In addition to the functionality that the Node global object had, (like console.log) the window also has information about the browser environment.

We access the DOM via the window, with window.document. We can just write document however, because window is the global scope. The document object is the Document Object Model, the DOM.

Open the Console tab of devtools and type window and hit enter. This shows us the window object. You will see window has a property of the document object.

**document’s methods**

To manipulate the DOM with JavaScript, we can use the document’s methods. Let’s just update the h1 that says, “Hello World!” to say, “JavaScript says, Hello World!”.

We are going to look at these methods in more detail in the ‘DOM Manipulation’ section, but for now let’s get hold of the title by it’s tag type, h1 by using querySelector, which allows us to query the DOM to access its elements.

**const** title **=** document.querySelector('h1');

Now let’s overwrite it using the textContent property on the h1 element.

title.textContent **=** 'JavaScript says, Hello World!';

We have updated the DOM using JavaScript, the page has updated to reflect the new state of the DOM.

**Manipulating the DOM from a JavaScript file**

That’s great, but we want to write our JavaScript in our web application, not just in the console. How are we going to interact with the DOM using JavaScript written in our .js files?

In index.html’s <head> there is a script tag that links to a .js file.

**<script src="js/app.js"></script>**

When index.html is loaded by the browser, the above script tag tells the browser to make another request for the JavaScript file, app.js. Requests are also made for any other resources defined in <script> and <link> tags in the index.html’s <head> tag.

*HTTP Request/Response Cycle*

Once app.js has been loaded by the browser, it has access to the browser’s global object, the window.

We can confirm that by putting a log in app.js.

*// js/app.js*

console.log('app loaded', window);

When we refresh the page, we now see the window object in the console.

We have the window object available to us, and therefore we also have access to the document object and all its methods. So let’s go ahead and update the title as we did before.

However if we now try to overwrite the title as we did in the console we get an error:

**const** title **=** document.querySelector('h1');

title.textContent **=** 'JavaScript says, Hello World!';

*// -> Uncaught TypeError: Cannot set property 'textContent' of null*

This tells us title is null. That is because the JavaScript file is being run before the DOM has finished loading, so the h1 element isn’t available at the point querySelector is executed. We need to wait until the DOM content has finished loading before executing the code.

**DOMContentLoaded**

We want to wait until the DOM has finished loading before executing the code. addEventListener, which takes the name of an event and a callback as arguments, will listen for the event to be fired by the browsers, and then execute the callback.

By using addEventListener, when the document’s DOMContentLoaded event fires, we can be guaranteed to have access to the DOM elements. At this point, the DOM is ready to be manipulated.

document.addEventListener('DOMContentLoaded', () **=>** { *// NEW*

**const** title **=** document.querySelector('h1');

title.textContent **=** 'JavaScript says, Hello World!';

});

Now we have seen that we can manipulate the DOM from inside our JavaScript applications, let’s look in more detail at the document’s methods that allow us to make the modifications to the DOM.

**DOM Manipulation**

We are going to learn how to do the following:

* Access existing DOM elements in our JavaScript
* Modify the existing DOM elements - e.g. change an element’s text / give an element a class.
* Create new DOM elements and display them on the page.

**Accessing DOM Elements**

**querySelector**

To modify an element that is already in the DOM, we first need to access it in our JavaScript.

We’ve already used the document object’s [querySelector](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector) method to access the title (h1) from the DOM before we updated its text (const title = document.querySelector('h1')). To access the h1, we passed querySelector a query string, and the document returned to us an object representing the h1 element on our page.

We can use document.querySelector to target any single element by querying its ID, class or the tag name unsing the CSS selector syntax:

1. An ID selector is prefixed with a # - querySelector('#my-id')
2. A class selector is prefixed with a . - querySelector('.my-class')
3. A tag selector has no prefix - querySelector('p')

**Task (5 minutes)**

1. Use document.querySelector to access the paragraph with the ID ‘welcome-paragraph’ from the DOM.
2. What is the difference in output when you use console.log and console.dir?

**Solution**

querySelector will return only the **first** element on the page matching the specified query. If we look inside the Elements tab in Chrome’s dev tools, we can see two elements in our page have the class “red”, so when we access the element with the class “red”, we only get the paragraph element, as it comes before the red list-item in the page.

*// ...*

**const** redElement **=** document.querySelector('.red');

console.log(redElement);

*// -> <p class=red>A list:</p>*

If we wanted to access the list-item (li) with the class “red”, querySelector can be used with combined selector types for increased specificity. Let’s access the li with the class “red”.

*// ...*

**const** redListItem **=** document.querySelector('li.red');

console.dir(redListItem);

*// -> li.red*

Combining queries in this way, we can be specific about the element we want to access.

**querySelectorAll**

If we want to get back a collection of multiple elements then we can use querySelectorAll. Like querySelector, this method accepts a CSS selector, but it will always return a collection of all of the elements that match our query. For example, if we wanted to get back all the list items.

*// ...*

**const** allRedElements **=** document.querySelectorAll('li'); *// NEW*

console.dir(allRedElements); *// NEW*

*// ->*

*// NodeList(3)*

*// 0: li.blue*

*// 1: li.green*

*// 2: li.red*

*// length: 3*

});

querySelectorAll returns a [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList), which is a collection similar to an Array.

Note: There are other, older methods to access DOM elements, but querySelector and querySelectorAll do everything these do and more. They are: [document.getElementById](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById), [document.getElementsByClassName](https://developer.mozilla.org/en-US/docs/Web/API/Element/getElementsByClassName) and [document.getElementsByTagName](https://developer.mozilla.org/en-US/docs/Web/API/Element/getElementsByTagName)

**Modifying DOM Elements**

Now we have accessed existing elements from the DOM. Now we have access to DOM elements, there are many ways in which we can modify them and we are going to look at a couple of them.

**textContent**

We can the change an element’s text value using textContent, or change which CSS rules apply to the element by modifying the ID or the classes.

We have seen that we can update the text of an element with textContent when we updated the title (title.textContent = 'JavaScript says, Hello World!').

Let’s update the text of the list item with the class “red”.

**const** redListItem **=** document.querySelector('li.red');

redListItem.textContent **=** 'RED!!'; *// NEW*

If we refresh the page, we can see the text has been updated.

**Element.classList**

We can also modify a DOM element by adding a new class to it’s list of classes. If we look in styles.css we can see we have styles for a class “bold”. Let’s give our redListItem this class, using the add method on the element’s classList object.

**const** redListItem **=** document.querySelector('li.red');

redListItem.textContent **=** 'RED!!';

redListItem.classList.add('bold'); *// NEW*

console.dir(redListItem);

*// -> li.red.bold*

If we refresh the page and look at the Elements tab in the dev tools, we can see that the list item has a new class.

Note: The classList object, attached to all DOM Elements has other [methods](https://developer.mozilla.org/en-US/docs/Web/API/Element/classList#Methods), such as remove and toggle, which all take a string as an argument.

**Creating New DOM Elements**

**document.createElement**

We can also create brand new DOM elements by using [document.createElement](https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement).

Let’s create a new list item and add it to the list. In style.css we have some styling that targets the class ‘purple’, so we can create a new list item with the text “purple” and assign it the class “purple”.

**const** newListItem **=** document.createElement('li'); *// NEW*

newListItem.textContent **=** 'Purple'; *// NEW*

newListItem.classList.add('purple'); *// NEW*

If we refresh the page, we don’t see our new element. That’s because it only exists as a JavaScript object right now, we still need to actually put it into the DOM. To do this, we need an element which is already in the DOM, that we can append our newly created element to.

**Appending DOM Elements**

We want to add our new list item to the list of colours. So first we need to access the list (ul), using querySelector. Then we can use another of the document’s methods - appendChild - to add the new list item to the list.

**const** list **=** document.querySelector('ul'); *// NEW*

list.appendChild(newListItem); *// NEW*

If we look inside the Elements tab, we can see that the new list item has been appended to the list. The fact that list items are indented tells us they are the child of the list. Whenever we append one element to another with appendChild it creates this parent child relationship.

**Recap**

What type of element is returned by document.querySelector and is created by document.createElement?

**Answer**

Published with [GitHub Pages](https://pages.github.com/)

**Advanced Browser Devtools**

**Lesson Duration: 20 minutes**

**Learning Objectives**

* Be able to display data in the browser’s console with console.log and console.table
* Be able to use Chrome Dev Tools Network to view assets being loaded by a webpage
* Be able to use Chrome Dev Tools Debugger to step through running code

To help us with our JavaScript development, we should familiarise ourselves with some of the more advanced features of the development tools that are built in to our web browser.

**console**

Let’s start by taking a look at the browser’s console.

We have seen that we can pass more than one argument to console.log. The console will concatenate them into a single string, with a space in-between them.

*// js/app.js*

document.addEventListener("DOMContentLoaded", **function**() {

**const** answer **=** 1 **+** 1;

console.log("The answer is ", answer);

});

We can also format arrays and objects nicely, using console.table().

*// js/app.js*

document.addEventListener("DOMContentLoaded", **function**() {

*// ...*

**const** fruits **=** ["apple", "orange", "banana"];

console.table(fruits);

**const** person **=** {

name: "Jane",

age: 40

};

console.table(person);

});

**Breakpoints**

While console.loging information can be useful, it is often preferable to be able to step through our code line-by-line, checking the state of the program along the way.

We can also use our developer tools as a debugger, just as we’ve done in other programming languages.

Let’s say that we have some complex logic we want to step through:

*// js/app.js*

document.addEventListener("DOMContentLoaded", **function**(){

*// ...*

**let** number1 **=** 5;

number1 **+=** 10;

**const** number2 **=** 20 **+** number1;

**const** number3 **=** number2 **/** 10;

});

We can step through this program line by line, by using the debugger statement.

*// js/app.js*

document.addEventListener("DOMContentLoaded", **function**(){

*// ...*

**debugger**; *// Added*

**let** number1 **=** 5;

number1 **+=** 10;

**const** number2 **=** 20 **+** number1;

**const** number3 **=** number2 **/** 10;

});

Now, when the program executes, we should see the browser window appear to “freeze” when it hits the debugger; line.

We can use the arrows to step through our code line by line. Notice that the values of the variables in the Scope pane change as we step through. We can also execute code in the console, and when we do, we’ll have access to the same variables that we would have at that point in our program.

**DOM Breakpoints**

Sometimes, it can be hard to tell *what* is modifying the DOM, especially as our programs grow in size. We can set DOM breakpoints to give us more information about why the DOM is changing.

Let’s set a DOM breakpoint by right-clicking the <body> tag within our devtools. Next, we’re going to select “Break on…” > “Attribute Modifications”.

Now, we’re going to add the following code to our JavaScript file. This should add a new class to the <body> tag after three seconds:

*// js/app.js*

document.addEventListener("DOMContentLoaded", **function**() {

*// ...*

setTimeout(**function**(){

document.body.classList.add("lightblue");

}, 3000);

});

When this code is about to execute, the breakpoint kicks in, allowing us to tell what’s going on inside our programs. We can also break when the element’s children would be added, changed, or removed (“subtree modification”), or when the element itself would be removed (“node removal”).

To remove a DOM breakpoint, right-click on the same element, and remove the breakpoint you set.

**Network Tab**

Finally, let’s look at another developer tool that can be useful when we’re debugging problems, the Network tab. The Network tab displays information about all of the resources that our page needs to display itself.

**Loaded Resources**

We can see information about all the individual requests and responses for our images, CSS, and JavaScript files, among others.

In the Network tab we can see that index.html, app.js and style.css. If we commented out the <script> tag that loads app.js in index.html, the JavaScript wouldn’t be run.

**<!-- index.html -->**

**<!-- ... -->**

**<!-- <script src="js/app.js"></script> --> <!-- MODIFIED -->**

We can filter the type of request by clicking on the relevant tab or typing in the ‘filter’ field.

This tab can be really helpful to diagnose whether files are loading or not. For example, if you aren’t seeing any styles applied, you can check to make sure that the response code for your stylesheet is 200, as it should be. If there is a problem with one of your files, it should be highlighted in red.

Notice that we can click on any individual request to get more information about it, or the related response.

**Further Resources**

1. [Chrome Devtools Overview](https://developer.chrome.com/devtools)
2. [Video: debugging using devtools](https://developers.google.com/web/tools/chrome-devtools/javascript/)

**Conclusion**

We have learnt a number of debugging tools including:

1. Outputting data to the browser console with console.log and console.table
2. Chrome Dev Tools, which give us a range of debugging tools, such as breakpoints
3. the network tab, where we can view more information about requests and loaded files

# Events

**Lesson Duration: 75 minutes**

### Learning Objectives

* Understand what an event-driven environment is
* Be able to add behaviour to an event
* Know common event types
* Understand delayed execution of behaviour within callback functions
* Understand default behaviours in events, and know how to prevent it

## Intro

As a language, JavaScript follows a number of programming styles, or paradigms. One these paradigms is known as event-driven programming.

Event-driven programming has its roots in the earliest days of graphical user interfaces. We use it to listen for and react to events in our applications. Sometimes these events will be triggered by the system itself (for example, by a temperature sensor, or by a web page loading in the browser), but more commonly, events are triggered by user-input, for example, a button click, a key press, or a form submission.

When an event is triggered, a specific piece of code is executed. When writing JavaScript, we can use the power of callback functions to specify which piece of code is executed when an event is dispatched.

## Adding callbacks to events

Instructor note: Hand out start point. Ask students to open the index.html in the browser and to check for the output ‘JavaScript has loaded’ in the browsers console.

In order to react to an event, we need to ask three questions:

1. Which element will be dispatching the event?
2. Which event should we listen for?
3. Which piece of code should run when the event is dispatched?

We’ve actually already been adding a callback to an event. We do it every time we write:

*// app.js*

document.addEventListener('DOMContentLoaded', () **=>** {

console.log('JavaScript has loaded');

});

1. The element that will be dispatching the event is the document.
2. The event that is being listened for is DOMContentLoaded.
3. The code that will be run when the event is dispatched is the callback containing the console.log.

addEventListener() is used to add the callback to the event.

We call the same method, addEventListener(), on DOM elements to add behaviour to their events. addEventListener() takes two arguments: The event as a string and a callback to be invoked when the event is fired.

Other examples include:

* to listen for a select’s change event, in order to render details about the selected option to the page
* to listen for a form’s submit event, in order to makes a post request

## Manipulating the DOM with Events

Looking at the start code, we can see that there are various html elements defined in index.html (a button, an input, a select and a form). At the moment these elements don’t do anything when interacted with in the browser. We are going to add event listeners to these elements, so that when a user interacts with them, the page is updated.

If we look at the MDN docs, we can see a list of events and their targets listed down the left-hand side: <https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener>

If you click on the click event, it tells us the target of this event can be any DOM element, so let’s add a behaviour to our button.

### Buttons

We want the button to listen for a click event and update the paragraph below it from being, “That button has not been clicked.” to “That button has definitely been clicked.”

Let’s use the three steps we listed before to plan the event listener:

1. Element - button
2. Event - click
3. Callback - responsible for updating the paragraph

We will start by querying the DOM to get the button element. Then we will add the event listener to the button, passing in the event as a string and a callback that we haven’t written yet.

document.addEventListener('DOMContentLoaded', () **=>** {

**const** button **=** document.querySelector('#button'); *// MODIFIED*

button.addEventListener('click', handleButtonClick); *// NEW*

});

Now let’s write the callback handleButtonClick, that will be responsible for updating the paragraph text. We can begin by making sure that handleButtonClick has been added to the button’s click event by logging an output.

*// ...*

**const** handleButtonClick **=** **function** () { *// NEW*

console.log('Button has been clicked');

};

If you refresh the page and click the button you will see the output in the console.

The handleButtonClick will query the DOM for the paragraph element and set it’s textContent.

**const** handleButtonClick **=** **function** () {

**const** resultParagraph **=** document.querySelector('#button-result'); *// NEW*

resultParagraph.textContent **=** 'That button has definitely been clicked.'; *// NEW*

};

Refresh the browser and see the paragraph text update when the button is clicked.

### Text Inputs

We want the input to listen for a [input](https://developer.mozilla.org/en-US/docs/Web/Events/input) event and update the paragraph below it from being, “Nothing has been typed yet.” to “You typed:” followed by the text that has been typed into the input.

Planning for the event listener:

1. Element - input
2. Event - input
3. Callback - responsible for getting the value from the input and updating the paragraph

This will follow the same pattern as handling the button click, but this time we will need to get the value from the input (the text that’s been typed) so it can be displayed. Let’s start by adding the event listener and passing in a callback that we haven’t written yet.

document.addEventListener('DOMContentLoaded', () **=>** {

*// ...*

**const** textInput **=** document.querySelector('#input'); *// NEW*

textInput.addEventListener('input', handleInput); *// NEW*

});

As before, we can now write the callback handleInput and again we will start by logging an output to make sure the function has been added to the input’s ‘input’ event.

*// ...*

**const** handleInput **=** **function** () { *// NEW*

console.log('Input has been typed in');

};

Refresh the page and type something into the input. You should see that the ‘input’ event is fired every time you type a character by looking in the browsers console.

Let’s complete the handleInput function. It is going to be responsible for getting the value from the input and updating the paragraph text below.

**const** handleInput **=** **function** () {

*// 1. Get value from Input*

*// 2. Update paragraph text*

};

So how do we get the value back from the input?

### The Event Object

When we pass a callback to addEventListener, addEventListener passes an event object to our callback that contains various information about the event that was dispatched. Let’s start by loging the event and having a look at it.

*// ...*

**const** handleInput **=** **function** (event) {

console.log(event);

};

Now when we type something into the input in the browser, we see the event object in the console. It has a property called target, which is the element that dispatched the event (in this case the text input). target has a property called value which contains what text that has been typed into the input.

### Task: (5 minutes)

Complete the callback so that it updates the paragraph text to be “You typed:” followed by the text that has been typed.

Example Solution

### Selects

When an option is selected from the select, we want to update the paragraph below it from being, “No choice has been made.” to being “You went with:” followed by the value of the selected option.

Planning for the event listener:

1. Element - select
2. Event - change
3. Callback - responsible for getting the value from the select and updating the paragraph

### Task: (5 minutes)

1. Add an event listener to the select element that listens for the change event, passing in the callback handleSelectChange, which you haven’t written yet.
2. Write the callback handleSelectChange so that it updates the paragraph text below the select to be “You went with:” followed by the value of the selected option.

**Hint: log the event object to check how to get the value from the select**

Example Solution

### Forms

When the form is submitted, we want to update the paragraph below it from being, “Who’s it going to be?” to being “Your name:” followed by the value of the first and last name inputs.

Planning for the event listener:

1. Element - form
2. Event - submit
3. Callback - responsible for getting the values from the form inputs and updating the paragraph

Let’s start by adding the event listener to the form element and writing the callback as we have been doing previously. To get input values back from a form, we use the ID of the input. So in this case we will log event.target.first\_name.value to get the value of the first name.

**const** form **=** document.querySelector('#form');

form.addEventListener('submit', handleFormSubmit);

**const** handleFormSubmit **=** **function** (event) {

console.log(event.target.first\_name.value);

};

But wait, we have a problem. When we try to submit the form there is nothing displayed in the console. That’s because by default a html form, when submitted, sends a post request to the current URL. This means our page has been refreshed and with it, the state of the application has been lost. Because we want to handle our requests with javascript, we are going to prevent this default behaviour by using a method called preventDefault on the event object.

**const** form **=** document.querySelector('#form');

form.addEventListener('submit', handleFormSubmit);

**const** handleFormSubmit **=** **function** (event) {

event.preventDefault();

console.log(event.target.first\_name.value);

};

Great. Now we can see the event.target.first\_name.value in the console.

### Task: (5 minutes)

Complete the callback so that it updates the paragraph text below the form to be “Your name:” followed by the first and second names submitted by the form.

Example Solution

## A Note on Context

What is the value of `this` inside the callbacks we have written?

In the callbacks event.target refers to the DOM element that dispatched the event. this also refers to the DOM element, so we could replace event.target with the this keyword when accessing the inputs values.

**const** handleFormSubmit **=** **function** (event) {

event.preventDefault();

**const** resultParagraph **=** document.querySelector('#form-result');

resultParagraph.textContent **=** `

Your name:

${**this**.first\_name.value}

${**this**.last\_name.value}

` *// MODIFIED*

};

If we had used an arrow function expression to define the callbacks, this would retain its definition context, and therefore would not refer to the DOM element.

## Recap

What are the three things we need in order to react to an event?What values and methods did we use from the `event` object and how were these useful to us?

## Conclusion

Using the event-driven paradigm, we have added behaviour to events that get triggered by user inputs, enabling us to build an interactive JavaScript app. We used the method addEventListener to add the behaviour to events. We also looked at a number of DOM elements and their events, such as button clicks, select changes and form submits.

The event object is passed as an argument to our callback by addEventListener and gives us information about the event that has just been dispatched, including the value of inputs and selects. It also gives us a method, preventDefault, which enables us to override the default behaviour of events, such as the request made on the submission of a form.

# Events Lab

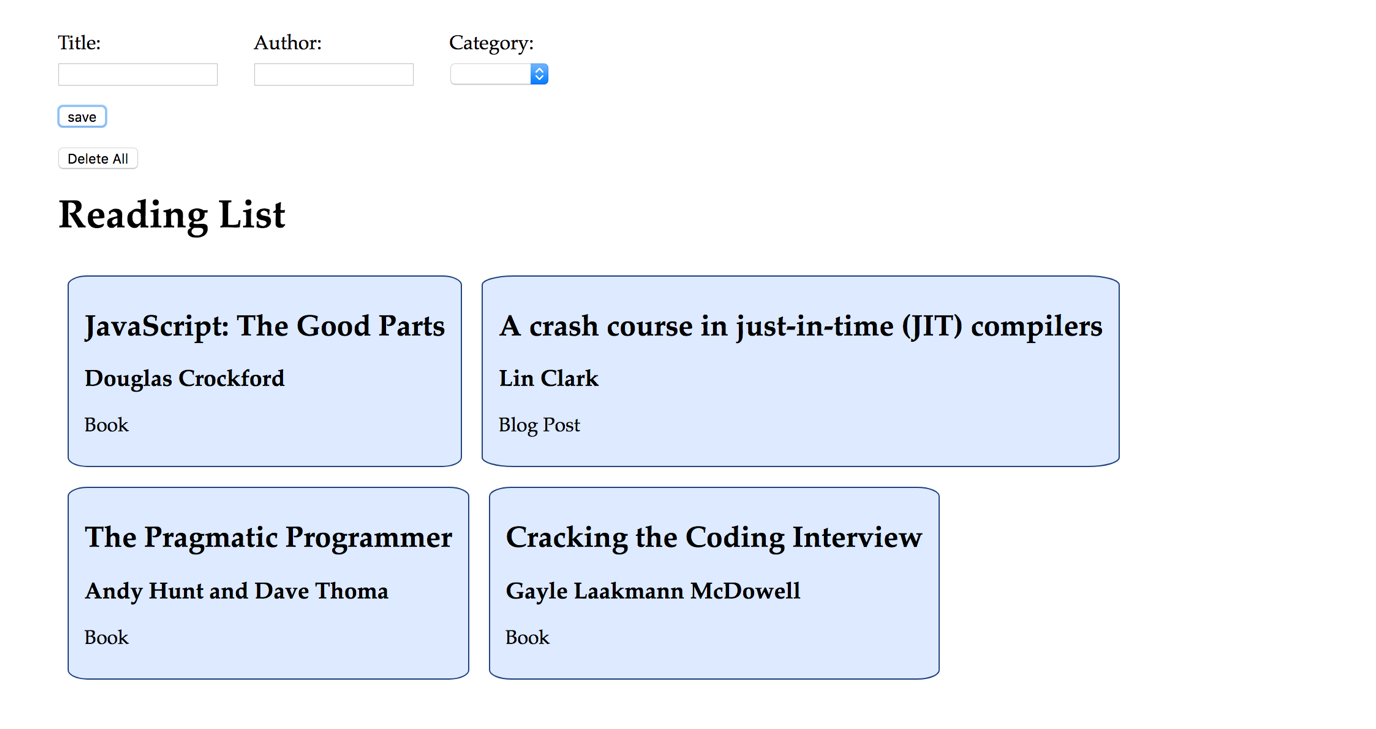
**Lab Duration: 120 minutes**

### Learning Objectives

* Be able to use a variety of event listener types
* Be able to access values from form inputs on submit
* Be able to attach events to a variety of DOM elements
* Be able to read and write to the DOM

## Brief

Your task is to create a Reading List app that allow a user to submit items they intend to read later and view them on the page. Using the start code, which has the html and css already provided, you will need to handle the form’s submission and display the submitted details on the page.



Screenshot of Example Result

### MVP

The provided html form has the following fields:

1. Title - text input
2. Author - text input
3. Category (e.g. book, article, blog post) - select

Once the user has submitted the form, the reading list item’s details should be displayed below and the form’s fields should reset to empty. Each time the user submits the form, the new reading list item should be added to the display.

**Hint: Research form.reset()**

### Considerations

When we want to get the values from inputs that are contained within a form, we do that by handling the form’s submit event, rather than handling the individual input’s event separately.

There’s no need modify the body of the HTML. If you need to create an element do so in JS.

Don’t worry too much about how our example looks above. Remember we should first choose semantic elements, then style them however we like.

### Planning

1. Add a script tag to index.html so that the browser knows to load the JavaScript
2. Add behaviour to the DOM elements’ events

### Extensions

1. Add a delete all button, that clears the list of submitted reading list items.
2. Add flexbox layout to the reading list so that items are displayed in a responsive grid.

# Homework: Manipulating the DOM

### Learning Objectives

* Be able to attach event listeners to DOM elements to respond to events
* Be able to access data from the event object
* Be able to use the document object to add information to the page by manipulating the DOM

## Brief

Your task is to create a front-end JavaScript application that allows users to input items and see them displayed in a list. Choose a theme for the application, such as a list of endangered Animals / favourite Sports Stars / any other theme of your choice. Users should be able to enter values for different properties relating to the theme (Name, Species and Continent, for example) and see them displayed below.

### MVP

* Create a form in HTML with inputs for relevant data
* When the form is submitted, access the data from the form in the form’s submit event object
* Create a list in HTML
* Append the submitted data to the list
* Add a “Delete All” <button> which clears all of the list items from the list

Note: If you are at all unclear about the process of reading and writing to the DOM, complete the MVP, writing all the code in one function. Understanding the document’s methods is the main learning for this homework. Abstracting code into helper functions is secondary.

### Extensions

* Refactor your code, extracting helper functions with single responsibilites to keep each function small and readable
* Experiment with adding a new form input and experimenting with a new <input> type eg. Add a [radio button](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/radio) input
* Style your application with CSS
* Add any other extension functionality that you want!

### Considerations

When you want to display a new element on the page, you will need to:

1. Access an element that is already in the DOM (querySelector)
2. Create a new element in the JavaScript and populate it with values (createElement)
3. Append the new element you have created in the JavaScript, to the element that is already in the DOM (appendChild)

### PDA Reminder:

Remember to gather evidence for your PDA this week. This should only take 5 minutes:

* Go to your [PDA Checklist](https://github.com/codeclan/pda/tree/master/Evidence%20Gathering%20Portfolio)
* Submit your PDA evidence (screenshots, etc.) to your own PDA repo