# The Ultimate JavaScript Master Series

# Part 1

## 01 - Getting Started - 01 Welcome

This will teach the fundamentals of programming and Javascript.

## 01 - Getting Started - 02 What is JavaScript?

What is JavaScript

What can you do with it?

Where does JavaScript code run?

What’s the difference between JavaScript vs ECMAScript?

What is JavaScript?

It is one of the most popular and widely used programming languages in the world. It is growing faster that any other programming languages. (Ah… Java and JavaScript are not the same. Der).



Big companies like Netflix, Walmart, and Paypal build Entire applications around JavaScript. Average salary is $72,000 per year. (According to glassdoor.com?)

You can work as a front end developer, or a back end developer, or a full-stack Developer.

What can you Do with JavaScript? For a long time, javascript was only used in browsers to build interactive webpages. “Some developers refer to javascript as a toy language. But those days are gone because of huge community support and investments by large companies like facebook and google.”

You can build full blow web or mobile apps, as well as real-time networking apps like chats and video streaming services, command-line tools, or even games.

Where does JavaScript Code run? It was originally designed to run only in browsers. Every browser has a “JavaScript Engine” that can execute javascript code. E.g., the javascript engines in firefox and chrome are Spidermonkey and v8. In 2009 an engineer named Ryan daul, took the opensource JavaScript Engine in chrome, and embedded it inside a C++ program. He called the program Node.

Node is a C++ program that includes googles V8 JavaScript engine. Using this, we can run JavaScript code Outside a browser. We can pass our JavaScript code to Node for execution. This means, with JavaScript, we can build the back end for our web and mobile applications.

Javascript code can be run inside a browser or in node. They both provide a ‘runtime?’ environment for our javascript code.

What is the difference between JavaScript and ECMAScript?

ECMAScript is just a Specification. JavaScript is a programming Language that Conforms to this specification.

We have a programming language called ECMA, which is responsible for defining standards. They take care of this ECMAScript Specification.

The first version of ECMAScript was released in 1997. In 2015, ECMA has been working on annual releases of a newer specification. E.g., in 2015, they release ES2015/ES6. This specification defined Many new features for JavaScript.

Every browser has a JS engine, and we can write code here without an other tools. Let’s inspect a chrome window.

Select the Console Tab. This is our JavaScript Console. “We can write any Valid JS code here”.

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

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

*VM451:1 Hello World*

*undefined*

We can see the hello world Message on the console. (The VM451:1 is an artifact).

We can also write mathematical expressions here.

*2+2*

*2 +2*

*4*

alert(‘yo’)



## 01 - Getting Started - 03 - Setting Up the Development Environment

VSCode, Sublime Text, and Atom are all code Editors. Mosh prefers VSCode.

We will also install Node, from Nodejs.org. You don’t Need Node to execute JS, because you can execute it inside a browser. But it’s good to have node, because you can use it to install Third Party Libraries.

Create a folder, e.g., js-basics, and drag and drop in VSCode.

Let’s add a new file:



“Now you don’t really need to know html in order to take this course, but if you want to be a front end developer, you should know your html well.”

Make a boilerplate doc !. We’ll use this as a host for our JS code. Save.

Open index.html with live server.

In order to write JS, we need a script element. There are two places where we can add this. In the head section, or the body section.

The best practice is to put the script element at the end of the body section, After all the ‘existing?’ elements.

Why is this a best practice? One reason is that the browser parses this file from top to bottom. If you put the script element in the head, there would be a lot of JS code there, and your browser may get busy parsing and executing that JS code and it won’t be able to render the Content of the page. This will create a bad user experience. The user sees a white or blank webpage while your browser is busy parsing and executing your javascript code.

The second reason is that almost always the code between script elements needs to talk to the elements on this web page. For example we may wish to show or hide some elements. So by adding the code here at the end of the body section we’ll be confident that all these elements will be rendered by the browser.

There are exceptions to this rule. Sometimes you are using third party code that has to be placed in the head section. But these are exceptions. As a best practice, you should add your JavaScript code at the end of the body section.

Let’s write some code:

 <body>

    <h1>Hello World</h1>

    <script>

console.log('Hello World');

    </script>

  </body>

The highlighted code is a statement. A statement is a piece of code that expresses an action to be carried out. In this case, we want to *log* a message *‘Hello World’*, on the *console*.

All statements in JavaScript should be terminated by a semicolon ;

console.log('Hello World');

What we have here in between single “code?” is called a string. A string is a sequence of characters.

In JavaScript we also have this notation: // We can add two slashes and this represents a comment.

<script>

        // Comment

        console.log('Hello World');

    </script>

Here we can add some description to our code and this description is ignored by the JavaScript engine. It is not executed. It is purely for documenting the code when you want to explain to other developers why you have written the code this way. You don’t want to explain what the code does because that should be clear in the code itself. (Mosh highlights the console statement). We want to explain Why’s and Hows.

  <h1>Hello World</h1>

    <script>

      // This is my first JS code.

      console.log("Hello World");

    </script>

Let’s open our console again the browser. alt ctrl i

## 01 - Getting Started - 05 - Separation of Concerns

In real world applications we have thousands or even millions of lines of code. Therefore, writing in the script element is oftentimes not practical. We don’t want to write all the code inline here. We want to extract and separate our JS code from our html code.

Mosh utilizes a metaphor in which we recognize that bedrooms stores your bed and your clothes. We don’t store are clothes in the kitchen. This is what we call the Separation of Concerns. We want to separate HTML--which is all about content--from JavaScrip, which is all about behavior.

How should your webpage behave? What should happen when we hover our mouse over a given element? Maybe something should pop up or be hidden. We’ll use JavaScript to implement behavior.

In VSCode let’s start a new file will call index.js. Let’s cut our JS code from our html file, and paste it in index.js. In this application we have a single file, a single JavaScript file. In a real world application we have hundreds or even thousands of JavaScript files. We’ll eventually learn how to combined these files into a bundle and ‘serve’ that bundle to a ‘client’.

Now that all are JavaScript code is in a separate file, we need to reference that file here.

Let’s add an attribute here (in our HTML document):

<script></script>

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

This tells the browser that are JavaScript code is in index.js.

When we open our browser, we note that the Hello World message is still up, which indicates that our code is still working. Huzzah!

## 01 - Getting Started - 06 - JavaScript in Node

We navigate to our js-basics folder. Then we type (in command prompt… not in node?)

*node index.js*

*C:\Users\Mr. Artifice\Desktop\js-basics>node index.js*

*Hello World*

We get the same messages on the consol. As we can see, node is a program that includes google’s V8 JavaScript engine. We can give it a piece of JavaScript code and it will execute that code for us just like we can in a browser. (This works in both command prompt and the node command prompt). So, node is a runtime environment for executing JavaScript code.

Here’s a tip from Mosh:

VSCode includes an integrated terminal, so you don’t have to open up a separate terminal window. Under view, you’ll find the “Terminal” option. Note that our terminal is pointing to the same folder where we created our files… You don’t have to explicitly navigate to this folder. (Make sure you have the index selected).

Here you can type node index.js and get the same output. (I have a big error message from my previous shell manipulations).



In this course we will no longer work with node. Node is a separate topic for which mosh has devoted a course with 14 hours of content.

## 02 - Basics - 01 - Variables - 5.36

Variables are one of the most fundamental concepts in JavaScript or any other programming language.

In programming we use a variable to store data temporarily in a computer’s memory.

We store our data somewhere, and give that memory location a name:

Memory

|  |  |  |  |
| --- | --- | --- | --- |
| Variable |  |  |  |

Variable Name.

With this Name, we can read the data at the given location in the future.

Metaphor!!!

Imagine putting items in various boxes, and labeling each box. Now, you can readily find your stuff. A variable is like this box. What we put inside this box is the value we assign to a variable; that’s the data. And the label that we put on the box is the Name of our Variable.

In index.js let’s declare a variable. In the old days before ES6 we used the *var* keyword to declare a variable.



However, there are issues with VAR as we will discover later in the course.

After ES6, the best practice is to use the *let* keyword to declare a variable.

let

Let’s give this variable a name or an identifier. This is like the label we put on a box. We’ll call it *name* and terminate it with a semicolon.

let name;

Let’s add this on the console and see what happens.

*console.log(name);*

let name;

console.log(name);

In the console, we see *undefined.* (I also have loads of error messages, for whatever reason).



By default variables that we define in JavaScript… Their value is undefined.

We can optionally initialize this variable:

let name;

console.log(name);

= a string, which is a sequence of characters.

let name = 'Mosh';

console.log(name);

We can use single or double quotes. It is more common to use single quotes for declaring strings in JavaScript.



Now we see Mosh on the console.

let name = 'Mosh';

console.log(name);

We have declared a variable called *name* and we have set it to this (‘Mosh’) value, to this string.

There are a few rules for naming the variables.

They cannot be a reserved keyword. For example we can’t use the keyword *let*. If we tried to use one of these names, we will get an error.

Note this red underline:



This is indicating that this is not a valid identifier.

Our second rule is that the name should be meaningful. The name should give some clue as to what the purpose of the variables are. What kind of data are we storing at that memory location. Always use meaningful and descriptive names.

The third rule is that they cannot start with a number.

The fourth rule is that they cannot contain a space or a hyphen.

For example, *let firstName .* Mosh is using camel notation, so the first letter of the first word is lowercase, and the first letter of every word after should be uppercase.

Camel notation is the convention used in JavaScript to name are variables.

The fifth rule for variable names is that they are case sensitive. E.g., the following variables are different:

*let firstName;*

*let FirstName;*

The sixth rule is that if you want to declare multiple variables there are two ways to do this. You can declare them on one line and separate them using a comma… e.g.,

*let firstName, lastName;* (In this case Mosh has not initialized either of these variables. They are both undefined).

We can optionally initialize one or both of them. E.g.,

*let firstName = ‘Mosh’, lastName;* (lastName is undefined) or… Up

*let firstName =’Mosh’, lastName = ‘Hamedani’;*

But the modern, best practice is to declare each variable on a single line. Like so:

*let firstName =’Mosh’;*

*let lastName = ‘Hamedani’;*

## 02 - Basics - 02 - Constants

Let’s make a variable called interest rate:

*let interestRate = 0.3.;*

This is the initial value; we can always change it later.

let interestRate = 0.3;

interestRate = 1;

If we log this on the console, we’ll see the new value… right?



There it is. 1 on the console.

However, there are real world situations in which we don’t want the value of that variable to change… Because otherwise it’s going to cause all kinds of bugs in the application. So, we use a Constant instead of a variable.

The value of the variable changes, but the value of a constant.

So, let’s change *let*, to *const*:

const interestRate = 0.3;

interestRate = 1;

console.log(interestRate);

When we save these changes, we’ll see an error in the console on line two.



If we click the error circled above:



We can see the line in code where this error occurred. So… we cannot reassign a constant.

The best practice is that if you don’t need to reassign, constant is your best choice. If you need to reassign a variable, use let.

## 02 - Basics - 03 - Primitive Types

What are the kind of values we can and assigned to a variable? We have seen strings… but we have more types.

In JavaScript we have two categories of types:

1. Primitives AKA Value Types
2. Reference Types.

For Primitives, we have:

* Strings
* Number
* Boolean
* undefined
* null

E.g.,

let name = 'Mosh';

We have a variable called name, which is set to a string. This: “*‘Mosh’*” is what we call a String literal. This is a fancy name for a string.

Let’s declare a variable and set it to a number.

let age = 30;

This is what we call a number literal.

Let’s declare a Boolean. A Boolean can be either true or false.

Let is approved to be true:

let isApproved = true;

We use Boolean in situations where we want to have some logic. E.g., If the order is approved, it needs to be shipped. So, the value of Boolean variable can be true or false. (Both true and false are reserved keywords, so they cannot be variable names.)

(Are undefined variables ones that aren’t initialized? What is initialized?).

Ah, yes. If we do not initialize a variable, then by default it is undefined.

let firstName;

We can also explicitly set the variable to undefined:

let firstName = undefined;

However, that is not very common. In contrast, we have another key word, *null*.

let lastNames = null;

We use *null* in all situations that we want to Explicitly clear the value of a variable. E.g., We might want to present the user with a list of colors. If the user has no selection, we want to set the:

*selectedColor* variable to null:

let selectedColor = null;

In the future… if the user selects a color, we will reassign this variable to a color like ‘red’.

let selectedColor = 'red'

But then, if the user clicks red again, we want to remove the selection and set this back to null. We use null in situations where we want to clear the value of a variable.

These are the examples of primitives/value types.

let name = 'Mosh';  //This is a String Literal

let age = 30;  //Number Literal

let isApproved = true;  // Boolean Literal

let firstNames = undefined;  //undefined

let lastNames = null; //null

## 02 - Basics - 07 - Functions

In the category of reference types we’ve learned about objects and arrays.

A function is a set of statements that performs a task or calculates a value.

Let’s declare function using the *function* key word. Let’s give function name, *greet*. Add parenthesis… that’s part of the syntax for declaring functions. () Then {}. What’s inside the curly braces is referred to as the body of the function. This is where we had all the statements to define some kind of logic in our application.

E.g., the logic in this function should be to display a message on the console.

function greet () {

    console.log('Hello World');

}

Note that we terminate our statement with a ;. But when we are declaring a function, we do not need to add a ; at the end, because we’re not declaring it like a variable like this: *let number = 1;*

function greet () {

    console.log('Hello World');

}; this semicolon is unnecessary.

We can call a function like so:

greet();

function greet() {

    console.log('jimbob');

}

greet();

We add the name of the function *greet* with parenthesis *()* and a semicolon to indicate that this is a statement.

**So, in my terms… the function *greet ()* is all one unit, both the word and the parenthesis. Inside {}, we execute the log command. So… all you have to do is add a *greet()* [[with a semicolon]] and you are executing the function by making a statement. So… empty parenthesis are part of the function, along with the name.**

function greet () {

    console.log('Hello World');

}

greet();

((Also, we are mere executing a console log here. Copying and pasting the console log outside the function would give the same value)).

Our function can have inputs, and these inputs can change how the function behaves.

Let’s say we’d like to display the name of a person instead of hello world. We can add a variable in between parentheses. We refer to this variable as a parameter. ^^

function greet(name) {

    console.log('Hello World');

}

greet();

Thus, our greet function has one parameter called name. “essentially name is like a variable that is only meaningful inside of this function. So inside of this function we can work with this name variable but it will not be accessible outside of this function”

[[[I.e., this variable will not contaminate data other code outside the function]]]

Name is an input to this function. So, instead of displaying hello world, we can delete world and add a plus to concatenate two strings… e.g., name:

function greet(name) {

    console.log('Hello' + name);

}

greet();

“When calling the greet function, we need to pass a value for the name variable, or name parameter more accurately”.

[[[When I don’t pass a value for name, the variable is undefined. See below:]]]



So, we can pass John between the brackets. We refer to this as an argument.

function ***greet (name)*** {

    console.log('Hello' + **name**);

}

greet('John');

“So, John is an argument *to* the greet function. And, name is a parameter *of* the greet function.” ^^ “That’s one of the things that a lot of programmers don’t know. They don’t know the difference between a parameter and an argument.”

[[[I explain function thusly: John is an argument that defines the name variable/parameter. Newly defined name variable/parameter equals **undefined name variable**, which is inside the body of the function]]]

“So a parameter is what we have here at the time of ***declaration***. The argument is the actual value we supply for that parameter.

function ***greet (name) {***

    console.log('Hello' + name);

}

greet('John');



Why does Mosh have a space?



Ah haha. Add a space after hello.

We can reuse this function, with a different input. We can copy the argument line, paste it, and add a name.

function greet (name) {

    console.log('Hello ' + name);

}

greet('John');

greet('Mary');

Now we have two different messages on the console.



A function can have multiple parameters. We can add more parameters by using a comma. E.g., lastName.

function greet (name, lastName) {

    console.log('Hello ' + name);

}

greet('John');

greet('Mary');

Now we can add this new parameter to console.log. Note that we added an extra space in the quotes. Also, note that parameters aren’t put into quotes. Presumably only the strings? Also, we can easily add spaces to strings, but we cannot do that with parameters. We had to add empty quotes to incorporate spaces.

function greet (name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John');

greet('Mary');

“when calling this greet function, we should pass another argument for the last name.” Let’s see what happens if we don’t do this.

((I got a different error message previously because last name was not capitalized. Remember that this stuff is case sensitive))



The default value of variables in JavaScript is undefined which is why hello john and hello Mary are undefined. “so because we did not pass a value for the last name, by default it’s undefined.”

For greet, let’s pass another argument. We’ll add a comma, and a last name in quotes. “we don’t need the second call to the greet function”. We’re deleting the second call to the greet to function, which is Mary. ((apparently values are assigned two parameters in consecutive order))

/\*

function greet () {

    console.log('Hello World');

}

greet();

function greet (name) {

    console.log('Hello World');

}

greet();

Here we concatenated two strings (hello and name) with a plus, and provided an arguement/function Name

function greet (name) {

    console.log('Hello ' + name);

}

greet('John');

Here we copied our function, and put in a different argument.

function greet (name) {

    console.log('Hello ' + name);

}

greet('John');

greet('Mary');

Here we add a last name, parameter, but we did not pass an argument (assign a value) to it, which is why it is (and displays as) literally 'undefined' in the console.

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John');

greet('Mary');

Lastly, below, we have passed another argument, "Smith", which is assigned to the parameter lastName

\*/

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John', 'Smith');

## 02 - Basics - 04 - Dynamic Typing

Something that separates JavaScript from other programming languages is that java script is a dynamic the language.

There are two types of languages:

* Static (statically-typed)
* Dynamic (Dynamically-typed).

In static languages, when we declare a variable, the Type of that variable is set and it cannot be changed in the future: *string name = ‘John’;*

Whereas in a dynamic language, the type of a variable can change at runtime: *let name = ‘John’;*

Let’s examine our code:

let name = 'Mosh';  //This is a String Literal

let age = 30;  //Number Literal

let isApproved = true;  // Boolean Literal

let firstName = undefined;  //undefined

let lastNames = null; //null

At the top, we have declared (let?) this name variable, and we have “set that to a string”. So, the type of name is currently a string, but it can change in the future.

Let’s go to our console, and execute some JavaScript code. We have a typeof operator, which we can use to check the type of variable.

So, we type *typeof* followed by our name variable *name*

*typeof name*



If we reassign name to a different value, like a number (*name = 1;*) and check it’s type:



The type is now changed to number. *‘number’*

This is what we call a dynamic language. Unlike static languages, the type of these variables will be determined at run time, based on the values we assign to them.

Let’s look at some more examples of the typeof operator. (typeof using other reserve key word).

(Command^^) To clear the console press ctrl L.

*typeof age*

‘number’

Type of age is a number.

Let’s change age to a floating point number in the console:

*age = 30.1*

*30.1*

Note that when we look at typeof, age is still a number.



In JavaScript, unlike other programming languages, we don’t have two types of numbers: we don’t have floating point numbers and integers. All numbers are *oftype* number.

*typeof isApproved*

*‘boolean’*

*typeof firstName*

*‘undefined’*

“That’s funny, because the value of this variable (highlights the word undefined) is undefined, but its type is Also undefined”:

let firstName = undefined;

[Now it seems to me that the typeof is *not* undefined. Surely this is part of the following list of types?   
  
**JavaScript types**

* Boolean type.
* Null type.
* Undefined type.
* Number type.
* BigInt type.
* String type.
* Symbol type.

So then, why is this type undefined?]

“What does this mean? Well, earlier I told you that we have two categories of types. Primitives/Value Types, and Reference Types.

In the Primitive/Value Types we have:

* String
* Number
* Boolean
* undefined
* null

So… undefined is actually a Type… but it is also a Value.

let firstName = undefined;

“In this example, *because* we have set *firstName* to *undefined* as a value, it’s type is also undefined.”

((If we renamed firstName to the string ‘icecream’, string would be the type, but icecream the value… I think)).

How about *selectedColor*?

let selectedColor = null;

key Variable type

word

*typeof selectedColor*

*'object'*

The type of this Variable, is an Object.

## 02 - Basics - 05 - Objects

Reference Types

* Object
* Array
* Functions

An object in JavaScript and other programming languages is like an object in real life. For example, a person has a name, age, address, etc. These are the Properties of a person. The same concept exists and JavaScript.

When we’re dealing with multiple related variables, we can put these variables inside an object.

E.g., here we have two variables: name and age.

let name = 'Mosh';

let age = 30;

They are highly related; they are part of the representation of a person. So instead of declaring two variables, we can declare a person Object. Then instead of referencing these two variables, we can simply reference the person object. It makes for cleaner code.

[[[https://www.freecodecamp.org/news/javascript-object-keys-tutorial-how-to-use-a-js-key-value-pair/

*const desk = {*

*height: "4 feet",*

*weight: "30 pounds",*

*color: "brown",*

*material: "wood",*

*};*

*An object contains properties, or key-value pairs. The desk object above has four properties. Each property has a name,* ***which is also called a key,*** *and a corresponding value.*

*For instance, the key height**has the value "4 feet".* ***Together, the key and value make up a single property.***

*height: "4 feet",]]]*

Let person = an object literal, i.e., curly braces.

*let person =* {};

The curly braces above are what we call an object literal.

'Between these curly braces, we add one or more key value pairs. The keys are what we call the properties of this object. In this case, we want the person object to have two properties, or two keys: name and age.

let name = 'Mosh';

let age = 30;

let person = {

};

Name [this is the key]: [after that, we set the value] ‘Mosh’ [add a comma], [another key value pair] [the key ‘age’] age: [the value] 30

“So we add name here… that’s the key. Then we add a colon, and then after that we set the value… Mosh.”

let name = 'Mosh';

let age = 30;

let person = {

    name: 'Mosh',

    age: 30

};

  name: 'Mosh',

Key Value

“Now we have a person object with two **properties**, or two **keyvalue pairs** ((Highlights name: ‘Mosh’)): name and age.” “And with that, we don’t need those two variables”

let name = 'Mosh';

let age = 30;

let person = [{

**name: 'Mosh',**

**age: 30**

};

Now, let’s log person on the console.

let person = {

    name: 'Mosh',

    age: 30

};

console.log(person);



We see our person object {name: “Mosh”, age: 30}. “Note the object literal syntax” - ((He means the curly braces)).

We have a couple key value pairs. ((Mosh wave his mouse over both age and its value, and name and its value)) They ((the key value pairs?)) are the Properties of the person object.

*age:30 Name: ‘mosh’*

Key:value Key: Value

Key Value Pair Key Value Pair

Property Propery

“There are two ways to work with these properties. Let’s say we want to change the name of this person. We’ll need to access the name property.” One way is to utilize Dot Notation. We add the name of our object (person), by typing *person dot*. … and now its properties are displayed in the popup list: It shows the age and the name properties.



Once we see the property that we wish to alter, we can enter it and do the following:

//Dot Notation

person.name = 'John';

let person = {

    name: 'Mosh',

    age: 30

}

person.name = 'John';

console.log(person);

“We can also use the dot notation to read the value of a property.” “Instead of logging the person object we can log it’s name property.” Now, in the console, it just says “John”.

let person = {

    name: 'Mosh',

    age: 30

}

person.name = 'John';

console.log(person.name);

((In other words, we can reference or display specific properties-key:values in an object)).

The other way to access a property is bracket notation. So instead of dot, we use square brackets… and we pass a string that determines the name of the target property. The name of the target property is name. So… we can change that to, say, Mary”

let person = {

    name: 'Mosh',

    age: 30

}

person.name = 'John';

person['name'] = 'Mary';

console.log(person.name);

“When reading that, we can use the dot notation, or the bracket notation. If we save the changes, now we get Mary”



Which notation is better? Dot notation or bracket notation?

“Dot notation is more concise, so that should be your default choice.

*[[So… person is the object, with a couple of properties. Dot notation allows us to select one (or more?) of those properties, and display (probably) or alter it]]*

“…However, bracket notation has its own uses. Sometimes you don’t know the name of the target property until the runtime. E.g., in our user interface, the user might be selecting the name of the target property. In that case, at the time of writing code, we don’t know what property we are going to access. That is going to be selected at runtime by the user.”

“So, we might have another variable somewhere else like “let selection = ‘name’;”. *Let selection*  =’name’ and that can change at runtime.”

let person = {

    name: 'Mosh',

    age: 30

}

person.name = 'John';

let selection = 'name';

person['name'] = 'Mary';

console.log(person.name);

“With this [[person [] = ‘Mary’;]] we can access that property using the bracket notation in a dynamic way. We pass selection here… and we get the same result. [[[Mary on the console]]]

let person = {

    name: 'Mosh',

    age: 30

}

person.name = 'John';

let selection = 'name';

person[selection] = 'Mary';

console.log(person.name);

/\* let name = 'Mosh';

let age = 30;

The Object below allows us to eliminate the two variables above

let person = {

    name: 'Mosh',

    age: 30

};

then add

console.log(person);

Now both of the person object properties display in the console.

//Dot Notation

person.name = 'John';

console.log(person);

This displays just John in the Console window, as we have defined the person object (displayed by the console) as merely the name property (using dot notation), which we also redefine.

Next, we also employ bracket notation to define the properties that will be displayed by the console, and its altered values.

let person = {

    name: 'Mosh',

    age: 30

};

//Bracket Notation

person ['name'] = 'Mary';

console.log(person.name);

So... the object is person.

let person = {

    name: 'Mosh',

    age: 30};

Object has a name property.

name: 'Mosh',

The the variable Selection is created, with a value of name.

let selection = 'name';

The console will open the person object's name propery.

console.log(person.name);

The person object is defined with bracket notation as selection, with selection being set with the value of Mary.

The object Person = the value Name,

The variable Selection = the value Name,

The object Person = Selection,

Selection = Mary

The Object display's the name Mary in the console.

\*/

let person = {

    name: 'Mosh',

    age: 30

};

//Bracket Notation

let selection = 'name';

person [selection] = 'Mary';

console.log(person.name);

## 02 - Basics - 06 - Arrays - 4.18

Sometimes in your applications you might be dealing with a list of objects. For example, the list of products in a shopping cart. Or, the list of colors the user is selected. We utilize arrays to store such lists.

*let selectedColors*  Note that we aren’t using an indecipherable abbreviation like SC. We have a meaningful name. Let’s initialize this:

*let selectedColors =*

We’ll set this to an empty array. The square brackets are what we call *array literal*. They indicate an empty array.

*let selecteColors = [];*

We can initialize this array and add a couple of items.

let selectedColors = ['red', 'blue'];

Now, let’s log this on the console:

console.log(selectedColors)

let selectedColors = ['red', 'blue'];

console.log(selectedColors)



Note that our array has two elements. Also note that each element has an index, and that determines the position of that element in the array.

To access an element in an array, we use that index…

For example, suppose we want to display the first element in the array. We can use the square brackets, and then specify the index. Observe:

let selectedColors = ['red', 'blue'];

console.log(selectedColors[0])



Now, only the first item in the index displays.

The lengths of our arrays and the type of objects in them are dynamic, they can change. For example, we can add other elements besides red and blue and expand our array.

let selectedColors = ['red', 'blue'];

selectedColors[2] = 'green';

console.log(selectedColors)



Now we have an array with three elements. Thus, the length is dynamic. It is changeable. Also, the type of objects in this array is also dynamic. Most programming languages do not support mixing different types of objects in the same array.

Here, we will change one of the elements in the array from a colour to a number.

let selectedColors = ['red', 'blue'];

selectedColors[2] = 1;

console.log(selectedColors)



Now we have two strings, and a number.

So, the element type in the array as well as the size of the array is dynamic.

Technically an array is an object. Just like the personal object we previously defined, the array has a bunch of key value pairs or Properties that we can access using the dot notation.

If we examined the already in the console with the typeof function, this is our output:

*typeof selectedColors*

*‘object’*

Note, that this array is indeed an object.

In VS code, we can examine the properties of this array/object using the dot notation. VSCode displays a number of properties to choose from for an array in JavaScript.

“Every time we declare an array using square brackets that array will automatically receive these properties. That we did not explicitly define them… they are simply magically inherited from somewhere else.” (We will learn about this later when we examine prototypes.)



Let’s examine the length property.

let selectedColors = ['red', 'blue'];

selectedColors[2] = '1';

console.log(selectedColors.length);

This property returns the number of items or elements in an array. See below.



An array is a data structure that we used to represent a list of items.

/\*

02 - Basics - 06 - Arrays

let selectedColor = []

The above Square brackets are called a literal array

let selectedColors = ['red', 'blue'];

console.log(selectedColors)

This displays both elements

let selectedColors = ['red', 'blue'];

console.log(selectedColors[0])

This displays the First element.

let selectedColors = ['red', 'blue'];

selectedColors[2] = 'green';

console.log(selectedColors)

This displays a Third element in the array.

\*/

let selectedColors = ['red', 'blue'];

selectedColors[2] = '1';

console.log(selectedColors.length);

## 02 - Basics - 08 - Types of Functions

Mosh disapproves of all the concatenations listed below. But he says we’ll worry about fixing them later with template literals.

console.log('Hello ' + name + ' ' + lastName);

The function below is performing a task. “It’s task is to display something on the console”.

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John', 'Smith');

Let’s do a function that calculates a value. “Let’s call this function square. This function should take a parameter. We’ll call it number”.

function square(number) {

}

“We need to calculate the square that number. That is number \* number”.

function square(number) {

number \* number;

}

“we need to *return* this **value** to whoever is calling this function. For that we use the return key word” We cannot have another variable called return because it is another reserved key word.

function **greet**(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

function square(number) {

    return **number \* number**;

“Instead of calling to **greet** function, let’s call the **square** function. Square… and we pass 2. This returns a Value… so we can use that value to initialize a variable.”

function square(number) {

    return *number* \* *number*;

}

square(2);

*[[[No, 2 is an argument to the Square function. It Defines the variable, of the function named Square. Number is a variable/parameter of the Square function. It defines the undefined number variable, in the body]]]*

“E.g., we can declare another variable called number, and set it to square of 2.

let number = square(2);

function square(number) {

    return number \* number;

}

let number = square(2);

Now we can display this on the console.

console.log(number);

function square(number) {

    return number \* number;

}

//We were calling the greet function.  Now we are calling the square function.

let number = square(2);

console.log(number);

[[[Sooo. Console.log log will display Number. But what is Number? Number equals the output of the square function. And the Square Function receives an argument, or we “declare” (*let number*) “a variable” and we “pass a Value” to the Square function. That value is then executed inside the body of the function. I just don’t know if the variable we declare is passing a value, or an argument to the function. ]]]

Soo... we'll call the function Square.  Square has a number parameter.  We will return number times number.

Our variable says that number will equal square function (N times N), with the number parameter being 2.



“In this particular example we don’t necessarily have to declare a second variable, if all we want to do is display the square of two (square(2)) on the console. We can exclude this variable declaration…”

function square(number) {

    return number \* number;

}

let number = square(2);

“…and simply pass square of two to console.log”

function square(number) {

    return number \* number;

}

console.log(square(2));

**((Here, the log function square has an argument value of 2, which it will pass to the function square with a parameter value of two, which undergoes the function 2X2, and this value will be passed to console.log))**

“So, when the java script engine executes this code, first it will call this function [[square(2)]], it will get a value, and then pass that value to console.log.”



“How many function calls do we have in this code?”

function square(number) {

    return number \* number;

}

console.log(square(2));

“We have two function calls”. (((What is a function call? Duuuur. It’s the thing that Activates the function))). “Square of two is one function call.

function square(number) {

    return number \* number;

}

console.log(square(2));

“But console dot log is also Another function call, because here we have parenthesis”.

function square(number) {

    return number \* number;

}

console.log();

“We are calling the log function which is defined somewhere, and passing an argument ((between the brackets)).

function square(number) {

    return number \* number;

}

console.log()

“We could pass a simple string like hello:”

function square(number) {

    return number \* number;

}

console.log('Hello');

“Or we can pass an expression. That expression can be a call to Another function… like square of 2”

function square(number) {

    return number \* number;

}

console.log(square(2))

“For now, all I want you to take away is that a function is a set of statements that either performs a task for calculates and returns a value.”

/\*

This function is Performing a task.

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

greet('John', 'Smith');

function square(number) {

    return number \* number;

}

//We were calling the greet function.  Now we are calling the square function.

let number = square(2);

console.log(number);

Soo... we'll call the function Square.  Square has a number parameter.  We will return number times number.

Our variable says that number will equal square function (N times N), with the number parameter being 2.  There is an order of operations aspect.  The same term can have different values on the page... because sometimes that term is being acted upon, and other... things are receiving that finished output

function square(number) {

    return number \* number;

}

let number = square(2);

console.log(number);

\*/

//Performing a task

function greet(name, lastName) {

    console.log('Hello ' + name + ' ' + lastName);

}

function square(number) {

    return number \* number;

}

console.log(square(2));

## 03 - Operators - 01 - JavaScript Operators - .40

We utilize operators along with variables and constants to create expressions. With expressions we can implement logic and algorithms.

Kinds of operators:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Bitwise operators

## 03 - Operators - 02 - Arithmetic Operators

We use arithmetic operators for performing calculations, just like in mathematics. Let’s examine some of them.

We use them for performing calculations, just like in mathematics.   
  
Here’s an example of the addition operator:

let x = 10;

let y = 3;

console.log(x + Y);

Arithmetic operators usually take two operands (like x & y) and then produce a new value.

An expression is something that produces a value. E.g., X + Y.

Comment out - Mosh - 03 - Arithmetic Operators - Highlight a group of text in VSCode and control and slash. It turns text into this:

More Operators:

* // console.log(x + Y);
* // console.log(x - Y);
* // console.log(x \* Y);
* // console.log(x / Y);
* // console.log(x % Y); Remainer of division
* // console.log(x \*\* Y); Exponential? X to the power of Y
* // console.log(x + Y);
* // console.log(x + Y);

Increment operators are indicated by two plus signs (++). Depending on where we put the plus signs this operator will behave differently.

Console.log(x) ((log of X)). “X is initialized to 10” ((Highlights 10 above)).

10 displays on console.

let x = 10;

let y = 3;

console.log(++x);

Displays 11.

This is where things get Weird. If we put this operator After X like so:

let x = 10;

let y = 3;

console.log(x++);

The value of X displays first. And Then the value of X will be incremented by one. So, if we do a Second log:

let x = 10;

let y = 3;

console.log(x++);

console.log(x)

10, Then 11 displays on the console.

/\*

let x = 10;

let y = 3;

console.log(x);

Outputs 10

let x = 10;

let y = 3;

console.log(++x);

Outputs 11

let x = 10;

let y = 3;

console.log(x++);

Outputs 10 for the First console, but X Has been incremented by one.

let x = 10;

let y = 3;

console.log(x++);

console.log(x)

Displays 10, and 11 on console, because x is incremented by one after first console.log

Decrement works in the same manner.

\*/

let x = 10;

let y = 3;

console.log(x++);

console.log(x)

// console.log(x + Y);

// console.log(x - Y);

// console.log(x \* Y);

// console.log(x / Y);

// console.log(x % Y);  Remainer of division

// console.log(x \*\* Y);  Exponential?  X to the power of Y

// Increment (++)

// Decrement (--)

## 03 - Operators - 03 - Comparison Operators - 2.01

We use comparison operators to compare the value of a variable with something else.

Here we have the x set to 1.

*let x = 1;*

Let’s look at the first comparison operator that is greater than. *console.log(x > 0);*

“So we want to check of X’s greater than zero”

let x = 1;

console.log(x > 0);

The console displays this as true.

“So the result of an expression that includes a comparison operator is a Boolean… It’s true or false”.

console.log(x >= 1);



We also have less than, and less than or equal to:

let x = 1;

console.log(x > 0);

console.log(x >= 1);

console.log(x < 1);

console.log(x <= 1);



We refer to the above operators as relational operators.

We also have equality operators. We can check and see if X is equal to one:

console.log(x === 1);

If we want to see if x is not equal to a given value, replace the equal sign with an exclamation mark.

((From [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_inequality) - This is a Strict Inequality Operator. “The strict inequality operator ( !== ) **checks whether its two operands are not equal, returning a Boolean result**. Unlike the inequality operator, the strict inequality operator always considers operands of different types to be different.”))

console.log(x !== 1);



These are all the comparison operators in JavaScript.

let x = 1;

//Relational

console.log(x > 0);

console.log(x >= 1);

console.log(x < 1);

console.log(x <= 1);

//Equality

console.log(x === 1); Is x equal to 1.

console.log(x !== 1); Is x Not equal to 1.

## 03 - Operators - 05 - Equality Operators

Previously we learned about the equality operator. E.g.,

// Strict Equality Operator (Type + Value)

console.log(1 === 1);

In JavaScript we have another equality operator that is indicated by two equals signs.

// Lose Eqaulity Operator

console.log(1 == 1);

The strict equality operator ensures that the values on either side of this operator have the same type and value.

For example on both sides of this operator there are two numbers. Both their type and their value are equal.

console.log(1 === 1);

Suppose we change one of the numbers to a string, like so:

console.log('1' === 1);

This expression is going to evaluate to false. We are comparing a string to a number. The types do not match.

Whereas if we compare a lose equality operator that features a number and a string like so:

// Lose Eqaulity Operator

console.log(1 == 1);

console.log('1' == 1);

The result is still true.

In this case, the operator ((the two equals signs)) looks at the value on the left side, observes that it is a string, and converts the numeral on the right to a string as well.

What if on the left side we have a Boolean?

console.log(true == 1);

The operator will automatically convert the value on the right side to a Boolean.

The console will also read this as true.

“Here’s what you need to take away: the strict equality operator ensures that both values have the same type and the same value. The lose equality operator does not care about the types matching; if the types don’t match it will convert that type on the right side to match the type on the left side. Then, it will only check if the values are equal”

Generally, you will use the strict equality operator because it is more precise and accurate.

/\*

// Strict Equality Operator (Type + Value)

console.log(1 === 1);

console.log('1' === 1);

// Lose Eqaulity Operator

console.log(1 == 1);

\*/

// Lose Eqaulity Operator

console.log(1 == 1);

console.log('1' == 1);

console.log(true == 1);

## 03 - Operators - 06 - Ternary Operator

Let’s examine the Ternary or conditional operator, which is incidentally one of Mosh’s favorites.

Consider the following problem:

// If a customer has more than 100 points,

// they are a 'gold' customer, otherwise,

// they are a 'silver' customer.

Let’s start by declaring a variable to keep track of the number of points.

let points = 110;

Now let’s declare another variable called type, that will represent the type of customer.

let points = 110;

let type =

Let’s start with a condition. We want to compare the number of points with 100.

So, we use the comparative operator here:

let points = 110;

let type = points > 100

This expression produces a Boolean. The result of this expression is either true or false.

Depending on the result, we’ll set the type variable to a different value. Let’s add a question mark:

*let type = points > 100 ?*

If this expression evaluates to true, that means this is a gold customer… So we want to set type to gold.

let points = 110;

let type = points > 100 ? 'gold' :

If the expression evaluates to false up, we want to set the type to:

let points = 110;

let type = points > 100 ? 'gold' : 'silver';

Let’s log type on the console.

let points = 110;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

The console displays gold.

let points = 90;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

Now the console displays silver.

let points = 110;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

Start with a condition ((using a conditional operator)). If that condition evaluates to true, then the type will receive the value gold. If false, silver.

/\*

let points = 110;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

\*/

// If a customer has more than 100 points,

// they are a 'gold' customer, otherwise,

// they are a 'silver' customer.

let points = 90;

let type = points > 100 ? 'gold' : 'silver';

console.log(type);

## 03 - Operators - 07 - Logical Operators - 5.30

We use logical operators to make decisions based on multiple conditions. In JavaScript we have three kinds of logical operators:

* Logical and
* Logical or
* Not

We’ll start with *Logical AND*, which is indicated with two ampersands: &&.

Here is the basic rule of thumb: logical and returns TRUE if both operands are TRUE. Let’s see an example.

console.log(true && true)

We have two operands, and they are both true. So, the result of evaluating this expression will be true.

If either of these are false, the result will be false.

console.log(false && true)

The console outputs false. The console output false if one or both are false.

What is a real world to use case for this operator? Suppose we want to build an application for approving loans. We want to see if the applicant has high income and a good credit score.

Let’s declare a couple variables:

let highincome = true;

let goodCreditScore = true;

Here we are dealing with two conditions. We need to know that the applicant is true for both high income and credit score. That’s where we use the logical And.

Let’s declare another variable:

let eligibleForLoan =

This is where we use the logical and operator:

let eligibleForLoan = highIncome && goodCreditScore;

Now let’s log this on the console:

let eligibleForLoan = highIncome && goodCreditScore;

let highIncome = true;

let goodCreditScore = true;

let eligibleForLoan = highIncome && goodCreditScore;

console.log(eligibleForLoan);

The console outputs true.

Now let’s examine the logical or.

Logical OR is indicated by two vertical lines. This returns true if one or both of the operands are TRUE.

Let’s edit the logical and, and turn it into a logical or, and change one of the operands to false.

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

console.log(eligibleForLoan);

Lastly, let’s look at the NOT operator. The not operator is indicated by an exclamation mark.

If the applicant is not eligible for loan, we want to consider their application as refused.

Let’s declare another variable:

*let applicationRefused*

Let’s use the NOT operator:

*let applicationRefused = !*

We apply it on eligibleForLoan:

let applicationRefused = !eligibleForLoan;

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log(eligibleForLoan);

If the eligibleForLoan is true, the not operator will convert that to false. So, whatever we give it, it will give us the opposite. So if eligibleForLoan is true, this will be converted to False, which means applicationRefused will Also be false.

In this scenario, applicationRefused is always the opposite of eligibleForLoan. That’s where we use the not operator.

So… let’s try changing both conditions to False:

let highIncome = false;

let goodCreditScore = false;

let eligibleForLoan = highIncome || goodCreditScore;

console.log(eligibleForLoan);

//NOT (!)

let applicationRefused = !eligibleForLoan;

And… let’s add a label to console.log:

console.log('Eligible', eligibleForLoan);

((When I test this in the console, it gives the result “Eligible false”, instead of just false.))

Let’s do another console.log, and add another label:

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log('Application Refused')

And we log the applicationRefused variable:

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log('Application Refused', applicationRefused);

let highIncome = false;

let goodCreditScore = false;

let eligibleForLoan = highIncome || goodCreditScore;

console.log('Eligible', eligibleForLoan);

//NOT (!)

let applicationRefused = !eligibleForLoan;

c



/\*

//Logical AND (&&)

//Returns TRUE if both operands are TRUE

console.log(true && true)

We have two operands, and they are both true.  So, the result of evaluating this expression will be true.

console.log(false && true)

The console outputs false.  The console output false if one or both are false.

let highIncome = true;

let goodCreditScore = true;

let eligibleForLoan = highIncome && goodCreditScore;

console.log(eligibleForLoan);

This outputs as true.

// Logical OR (||)

// Returns TRUE if one or both of the operands is TRUE

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

console.log(eligibleForLoan);

let highIncome = false;

let goodCreditScore = true;

let eligibleForLoan = highIncome || goodCreditScore;

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log(eligibleForLoan);

This will return the opposite value it receives.  In the above case, the applicant Is eligible for the loan, which means eligibleForLoan returns false, which means applicationRefused is Also false.

\*/

// Logical OR (||)

// Returns TRUE if one or both of the operands is TRUE

let highIncome = false;

let goodCreditScore = false;

let eligibleForLoan = highIncome || goodCreditScore;

console.log('Eligible', eligibleForLoan);

//NOT (!)

let applicationRefused = !eligibleForLoan;

console.log('Application Refused', applicationRefused);

/\* Basically, when highIncome and goodCreditScore are false, then eligibleForLOad is false.

And the console.log will display the value, along with a label.

If !eligibleforloan has a false input value, it will flip it to true, and applicationRefused will be true.

Console.log will show the label string, and that applicationRefused is True. \*/

## 03 - Operators - 08 - Logical Operators with Non-booleans - 5.53

In JavaScript unlike many other programming languages we can use logical operators with Non-booleans values.

false or true, aka

false || true

The above is a true expression.

How about:

false || ‘Mosh’



How about:

false || 1

We get an output of 1.

What we realize is that the result of a logical expression is not necessarily true or false. That depends on the value of the operands we have.

In the first example our second operands is true, which is why we get true back.

In our second example, our second Operand is a string, which is why we get a string back.

In the third example our second Operand is a number which is why we get a number back.

So when our JavaScript engine tries to evaluate this expression it looks at each operand. If that operand is not a Boolean true or false, it will try to interpret it as what we call truthy or falsy.

Falsy is not the same as Boolean false.

The values of falsy are:

* Undefined
* Null
* The number zero 0
* Boolean false
* Empty string ‘’
* Not a number NaN groups

Not a number is a special value in JavaScript. Mathematical calculations that do not produce a valid number, this value, not the number, is returned.

If we use any of these Falsy values in a logical expression, they will be treated as Falsy, which is kinda like a Boolean false, but it is not exactly the same.

Anything that is not Falsy, is truthy.

In our second example, our second operand is a string with four characters. ((‘Mosh’)). It’s not an empty string. It’s not falsely. So it’s truthy. Because with the logical/or operator, the result will be true if One of the operands is true. The first operands is false (false || ‘Mosh’), and so the JavaScript engine examines the second operand, which it finds to be not falsy, and therefore truthy.

The same goes for the third example. (false || 1) The 1 is not a Boolean true… It’s truthy. That’s why the value of that operands is returned.

What if we have a triple expression?

false || 1 || 2

The console outputs 1.

With the logical/or operator, it will return a result as soon as it finds an operand that is truthy. The first truthy operand discovered is the one the console will output. You could have a dozen more operands… it will only return the first truthy operand it encounters. This is called Short-Circuiting.

When do we actually use non-Boolean operators?

Suppose we are building an application, and somewhere the user has to pick a color, or were going to use a default color. E.g., the color of the T-shirt they want to buy.

Let’s declare a few variables:

let userColor = 'red';

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

Let’s log this on the console:

let userColor = 'red';

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

The console output is “red”. Because, our user has selected a color.

If the user has not selected a color, then you get the obvious result:

let userColor = undefined;

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

Console output is ‘blue’.

“This is the power of using the logical or operator between non-Boolean’s. With this technique we can provide default values.”

/\*

Falsy (false) :

Undefined

null

0

boolean false

empty string ''

Not a Number NaN

Anything that is not Falsy -> Truthy

E.g.:

 false || ‘Mosh’

returns: 'Mosh'

false || 1

We get an output of 1.

false || 1 || 2

The console outputs 1.

With the logical/or operator, it will return a result as soon as it finds an operand that is truthy.

You could have a dozen more operands… it will only return the first truthy operand it encounters.  This is called Short-Circuiting.

let userColor = 'red';

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

The output is Red, because user color is selected

\*/

let userColor = undefined;

let defaultColor = 'blue';

let currentColor = userColor || defaultColor;

console.log(currentColor);

## 03 - Operators - 09 - Bitwise Operator

These operators are more on the theoretical side and not something we'd use on a daily basis. This section of the lecture is for the enthusiasts, a.k.a. autists. ((Mosh encourages us to skip this lesson if we like)).

We humans use the decimal system to represent numbers. “12345”. In computers these numbers are stored in the binary format.

E.g., the number one in the decimal system is represented by seven zeros “00000001”. We have eight digits, each digit is what we call a bit. Above we have eight bits which represent one byte of information.

Number two equals six zeros and then one zero. “00000010”.

“There are mathematical formulas behind these for converting numbers between decimal and binary systems”. If you don’t want to learn the mathematics, simply Google decimal to binary converters.

“Bitwise operators in JavaScript or any other programming languages are similar to logical operators but they work on the individual bits of a number.”

E.g.:

console.log(1 | 2);

Note that we have a single vertical line, that represents a bitwise or. ((1 or 2, 1 | 2)) ((Logical ors have double vertical lines)).

“When we apply the bitwise or between one and two, this is what’s going to happen…”

“I add R as the result here (R =). Now, this operator here is going to look at each of these bits in a vertical way. ((Like adding sums?)). If either of these bits is one, the result will be one. Otherwise it will be zero”

1 = 0000001

2 = 0000010

R = 0000011

“If you convert this binary number (00000011, aka the sum of 0000001 and 00000010) to a decimal number, the result will be three”.

The console displays 3.

Bitwise and is similar. We use a single ampersand instead of a vertical line.

If both numbers are one, one will be returned, otherwise the result will be zero.

1 = 0000001

2 = 0000010

3 = 0000011

R = 0000000

console.log(1 | 2); // Bitwise Or

console.log(1 & 2); // Bitwise And

The console outputs 3 and 0.

Suppose you want to implement an access control system. The user can have the following permissions “Read, Write, Execute”.

We can use one byte of information (or eight bits) to determine the kind permission of a user can have.

E.g., we can use five zeros 00000 (the first 5 bits are irrelevant), for the last three bits, if the user has permission we will use 1, otherwise we will use 0.

E.g., read only permission would look like this: 00000100.

00000100 Read

00000010 Write

00000001 Execute

This is where we use bitwise operators.

Let’s implement an access control system using bitwise operators.

Let’s define a constant called read permission, and set it to a decimal number that is equivalent to a binary number 00000100, i.e. 4.

00000100 Read

00000010 Write

00000001 Execute

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

Let’s declare a variable like my permission and initially set that to 0.

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

Now let’s give ourselves extra permissions.

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

((Ah ha. So, my permission equals 0, **OR** it equals readPermission, aka 4, or writePermission, aka 2.

But… what is the point of adding the numbers together? I don’t get it. Ah, if it’s six, that means read and write?))

Let’s log this:

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

console.log(myPermission);

6 is our output.

“Technically we don’t care about this decimal number. We can use the bitwise and operator to see if I have a given permission.”

For example:

Let’s declare a variable message.

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message =

And use the ternary/conditional operator.   
  
*For refreshers:*

let type = points > 100 ? 'gold' :

“We start with a condition”: = **()** ((When working with ternary/conditional operators above, no parenthesis were used. We had a greater than > ))

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message = ()

“We take myPermission”

*Let message = (myPermission)*

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message = (myPermission)

“And use the bitwise & operator”

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message = (myPermission &)

“Along with the readPermission”

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message = (myPermission & readPermission)

“If this evaluates to true” *let message = (****myPermission & readPermission****)* “that means I have the read permission.”  
  
***((How does any of this evaluate as true or false? If they just aren’t zero? Ah. So, my permission is zero, and read permission can be 0 or 4. If four, then true, and the message displays?))***

“Let’s say we want to display a message like yes:

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message = (myPermission & readPermission) ? 'yes'

“Otherwise, we want to display no”

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message = (myPermission & readPermission) ? 'yes': 'no';

“Let me break this up into multiple lines so we can see clearly”

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message =

(myPermission & readPermission) ? 'yes': 'no';

“Here’s our Ternary Operator” Referring to the bottom half I believe. “We have a condition; if that evaluates to true, this value will be used. Otherwise, this other value will be used”.

“Finally, let’s log this message on the console.

Console.log(message);

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

Console outputs: yes

((Interesting. When you remove the yes: no, the console outputs 4. But, when you add that “?” and the yes: no [[gold: silver in the example way above]], it now hinges on the truth of the condition. If it evaluates to true, the first value (yes) will be used. So, perhaps the ? : make it a conditional/ternary… thing))

((If we remove readPermission from the myPermission variable, then the *let message* variable will output as false, and therefore no)).

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

“So, here’s what I want you to take away. With the bitwise or operator we can add permissions. And with the bitwise & operator we can check to see if we have a given permission.”

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

“Of course this is just one real world use case for using the bitwise operators. There are other use cases, but as I told you before they are really not that common. So if this lecture was confusing, don’t worry about it.”

/\*

Example of a bitwise or with a single vertical line: console.log(1 | 2);

(Logical ors have double vertical lines ||)

// Bitwise Or  If Either of the bits are 1, the result will be 1.

console.log(1 | 2);

1 = 0000001

2 = 0000010

R = 0000011

// Bitwise And - If both numbers are one, one will be returned.  Otherwise, the result will be zero:

console.log(1 & 2);

1 = 0000001

2 = 0000010

3 = 0000011

R = 0000000

console.log(1 | 2);

console.log(1 & 2);

The console outputs 3 and 0.

These are the fundamentals.

//Read, Write, Exectute  (User features)

//00000 (the first five bits are irrelevant).  The remaining bits indicate that the user has permission if the value is one.

00000100 Read

00000010 Write

00000001

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission =0;

myPermission = myPermission | readPermission | writePermission;

console.log(myPermission);

((console displays 6))

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | readPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

Outputs to yes

\*/

const readPermission = 4;

const writePermission = 2;

const executePermission = 1;

let myPermission = 0;

myPermission = myPermission | writePermission;

let message =

 (myPermission & readPermission) ? 'yes': 'no';

 console.log(message);

## 03 - Operators - 10 - Operators Precedence

Complex expressions require you to consider the precedence of the operators.

E.g.,

let x = 2 + 3 \* 4;

Let’s ask the console for the answer:

console.log(x);

14

Of course, the multiplication expression is evaluated before the addition expression.

It is hard to memorize which operators have higher or lower precedence. So when working with complex expressions, you can use parenthesis to determine how these operators are applied. E.g.,

let x = (2 + 3) \* 4;

console.log(x);

Now our output is 20.

/\*

let x = 2 + 3 \* 4;

console.log(x);

The output is 14

let x = (2 + 3) \* 4;

console.log(x);

This output is 20.

\*/

let x = (2 + 3) \* 4;

console.log(x);

## 03 - Operators - 11 - Drill

1 - What is the result of “true && false”?

This statement uses the double ampersand, logical and operator, where the statement is true if both operands are true. Boolean?

Statement is therefore False.

2 - What is the result of “(true && false) || true”?

The Logical and statement is false, and is considered first because it is in parentheses, according to the order of operations. The double vertical lines indicate logical or, where only one of the operands need to be true. The statement is therefore true.

3 - What is the value of y?

Let x = 10;

Let y = (x > 10) ? 1 : 0;

Using the comparison operator, we find that 10 is not greater than 10. The ternary, or conditional operator is indicated by the question mark. If true, I believe the value is 1. If false, 0.

Y = 0.

4 - What is the value of x?  
  
let x = (2 + 3) \* (4 + 5);

X = 5\*9

X = 45

5 - Which expression returns true?

A - ‘1’ === 1;

B - 1 == 1

C - 1 === 1

The above involve Equality operators. Strict equality operators are indicated by triple equal’s signs ===, and require both the Value And the type to be the same. **A** evaluates as false, as ‘1’ is a string, and a different type.

The Lose Equality Operator requires that the value, but not the type must be the same. It will convert the type on the right to match the left, and then evaluate. **B** evaluates as true, with either operator.

**C** evaluates as true, as both the type and value are the same for a Strict Equality Operator.

6 - What is the value of x?

Let x = (1 == true);

With the Lose Equality Operator, the type on the left is converted to the type on the right, and only then evaluated. Mind you… What numeral is it converted to? Seems to be 1.

So, 1 == true turns into 1 == 1. Which is a True statement. Therefore, x = true.

7 - What is the value of y?

Let x = 10

Let y = (x > 5) && (x < 15);

Let y = (10 > 5) && (10 < 15);

Let y = true && true

Let y = true.

&& indicates Logical &, where the value is true if both operands are true.

8 - What is the value of x?

Let x = 5;

X += 3;

This was not covered in the course. This is an Addition Assignment Operator. It adds the value of the Right operand to a variable, and “assigns the result (of the addition?) to the variable”.

5 += 3 or 5+3=8

X = 8  
  
  
10 - What is the value of y?

Let x = 10;

Let y = x++;

Incremental operators are indicated by two plus signs ++ raise the value of a number by one when assessed Before the number.

When displayed After a number, the increment is only counted when logged a Second time.

So, if this were a double log console.log(x++) and console.log(x), the x would be 11. Instead, it is only 11 in potentia.

X = 10

10 - What is the value of y?

Let x = 1;

Let y = x !==2;

This is not a pairing of the Loose Equality operator and the Logical Not as I thought. This is the Strict Inequality Operator. It checks to see if the operands are not equal, and returns a Boolean. It considers operands of different types to be different.

Y = 1 !==2;

1 is unequal to 2. True.

Y is True

## 03 - Operators - 12 - Swapping Variables

Let’s do a simple programming exercise. Let’s declare two variables:

let a = 'red';

let b = 'blue';

console.log(a)

console.log(b)

We are supposed to add some code to make A post red, and B post blue. I tried this:

let a = 'red';

let b = 'blue';

a = 'blue';

b = 'red';

console.log(a)

console.log(b)

Which works, but is apparently the wrong answer.

Mosh is talking about a 3rd variable. Can we employ logical not?

Huh. We use a third variable as a ‘backup’. Probably, because you’d have a circle if you just tried to make a and b equal each other.

C is assigned the value of A.

A is overwritten by b.

B is overwritten by C.

But the A that is overwritten by b, is Not the b overwritten by C.

let a = 'red';

let b = 'blue';

let c = a;

a = b; //A is overwritten by b

b = c; //C, which has A's value, is overwriting b

console.log(a)

console.log(b)

## 04 - Control Flow - If…else

We’re going to learn how to use expressions and operators along with conditional statements to implement interactivity in our applications.

JavaScript has two kinds of conditional statements:

* if and else - If…else
* switch and case - Switch…case

Suppose we want to get the current hour and depending on its value, we’ll greet the user with a different message.

So if the hour is between 6 AM and 12 PM, we’ll display a “good morning!” message.

If between 12 PM and 6 PM we’ll display good afternoon.

Otherwise we will display good evening.

If our *condition* evaluates to true, the following *statement* will be executed.

if (a **Condition**, which executes statement if true)

statement

(If we have multiple statements put them between curly braces. This creates a block of code).

if (condition) {

    statement

}

We can also have Another condition. If this condition is true, the other statement will be executed.

if (condition) {

    statement

}

else if (anotherCondition)

statement

We can add more curly braces if we want to add more statements.

if (condition) {

    statement

}

else if (anotherCondition) {

    statement

}

As a convention, when we want to define a block of code we put the first curly brace on the same line as displayed above.

if (condition) {

    statement

}

else if (anotherCondition) { //Goes on this line not below

    statement

}

else if (yetAnotherCondition)

statement

If none of the above statements evaluate to true (note we’ve added more else ifs) we can use else to execute one or more other statements.

if (condition) {

    statement

}

else if (anotherCondition) {

    statement

}

else if (yetAnotherCondition)

statement

else

statement

The first statement we want to execute is good morning. We will declare a variable to simulate the time.

let hour = 10;

if (condition) {

    statement

}

else if (anotherCondition) {

    statement

}

else if (yetAnotherCondition)

statement

else

statement

Let’s use 24 hour time format to keep things simple for ourselves.

We are using two comparison operators and two **Expressions**, and applying logical and && in between them. If both of these expressions evaluate to true, then the result of the *full expression* will also be true.

let hour = 10;

if *(****hour >= 6*** *&&* ***hour <  12*** *)* {

    statement

}

else if (anotherCondition) {

    statement

}

else if (yetAnotherCondition)

statement

else

statement

Let’s add the good morning statement, using console.log.

let hour = 10;

if (hour >= 6 && hour <  12 ) {

    console.log('Good morning');

}

else if (anotherCondition) {

    statement

}

else if (yetAnotherCondition)

statement

else

statement

Mosh suggested finishing on our own. So, I came up with two solutions to the problem of 6PM to 12AM, and 12 AM to 6 AM being in the same statement. So, I could either use a Logical Or ||, or just make two consoles. Let’s test it.

let hour =  22;

if (hour >= 6 && hour < 12 ) {

    console.log('Good morning');

}

else if (hour >= 12 && hour < 18) {

    console.log('Good Afternoon!');

}

else if ((hour >= 18 && hour < 24) || (hour > 0 && hour < 6)) {

console.log('Good Evening!');

}

Works fine.

Let’s try splitting up the statement. This also works:

let hour =  4;

if (hour >= 6 && hour < 12 ) {

    console.log('Good morning');

}

else if (hour >= 12 && hour < 18) {

    console.log('Good Afternoon!');

}

else if (hour >= 18 && hour < 24) {

console.log('Good Evening!');

}

else if (hour > 0 && hour < 6) {

    console.log('Good Evening!');

    }

But Both are wrong!

Apparently else covers everything else.

let hour = 4;

if (hour >= 6 && hour < 12 ) {

    console.log('Good morning');

}

else if (hour >= 12 && hour < 18) {

    console.log('Good Afternoon!');

else

console.log('Good evening');

Mosh says to get rid of the brackets, because there aren’t Multiple statements inside the brackets.

let hour =  4;

if (hour >= 6 && hour < 12 )

    console.log('Good morning');

else if (hour >= 12 && hour < 18)

    console.log('Good Afternoon!');

else

console.log('Good evening');

/\*

Ff the hour is between 6 AM and 12 PM, we’ll display a “Good morning!” message.

If between 12 PM and 6 PM we’ll display Good afternoon!

Otherwise we will display Good evening!

let hour =  22;

if (hour >= 6 && hour < 12 ) {

    console.log('Good morning');

}

else if (hour >= 12 && hour < 18) {

    console.log('Good Afternoon!');

}

else if ((hour >= 18 && hour < 24) || (hour > 0 && hour < 6)) {

console.log('Good Evening!');

}

Works fine.  Let’s try splitting up the statement.

This also works:

let hour =  4;

if (hour >= 6 && hour < 12 ) {

    console.log('Good morning');

}

else if (hour >= 12 && hour < 18) {

    console.log('Good Afternoon!');

}

else if (hour >= 18 && hour < 24) {

console.log('Good Evening!');

}

else if (hour > 0 && hour < 6) {

    console.log('Good Evening!');

    }

let hour =  4;

if (hour >= 6 && hour < 12 ) {

    console.log('Good morning');

}

else if (hour >= 12 && hour < 18) {

    console.log('Good Afternoon!');

}

else

console.log('Good evening');

\*/

//Mosh says get rid of ugly brackets.

let hour =  4;

if (hour >= 6 && hour < 12 )

    console.log('Good morning');

else if (hour >= 12 && hour < 18)

    console.log('Good Afternoon!');

else

console.log('Good evening');

## 04 - Control Flow - 02 Switch…Case

Let’s declare a variable called role. We’ll imagine that this represents the role of the current user

let role;

Suppose we want to determine the role of the current user i.e. guest, moderator, or an admin. We could implement this logic using *if and else* statements, but there’s another way. Switch and case.

Start with a switch statement:

switch

Now let’s add a variable (role) instead of condition:

switch (role) {

}

Now let’s add one or more **case statements**. Case statements compare the value of the variable with ‘something’.

Here’s our first case. [[I.e., case (in switch in case) means, “Switch means X. In that case, do Y”]]

let role;

switch (role) {

case

}

If this role equals guest:

let role;

switch (role) {

case 'guest'

}

(We are comparing the value of role with guest).

Then we add a colon and one or more statements.

let role;

switch (role) {

case 'guest':

    console.log('Guest');

}

Now we need to add the break statement, to jump us out of this switch block… Otherwise the other statements ‘below’ will be executed.

let role;

switch (role) {

case 'guest':

    console.log('Guest');

break;

}

E.g. let’s add another case statement. If this user is a moderator, we want to display on console.log ‘moderator user’. Then add a break.

let role;

switch (role) {

case 'guest':

    console.log('Guest');

break;

case 'moderator':

    console.log('Moderator User');

    break;

}

So, if we don’t add the first break statement above, and the role equals guest, both the console.log statements will be executed. Adding a break allows us to jump out of the switch block.

let role;

switch (role) {

case 'guest':

    console.log('Guest');

break;

case 'moderator':

    console.log('Moderator User');

    break;

    default:

        console.log('Unknown User');

}

If none of these cases are matched, we can have a default statement. We can use it to execute one or more statements. Like console.log unknown user.

We don’t need a break for the default statement “because at this point the control would automatically get out of the switch block”.

The console is outputting unknown user because we have not initialized *role* and its default value is undefined, and it doesn’t match any of our case statements.

let role;

switch (role) {

case 'guest':

    console.log('Guest User');

break;

case 'moderator':

    console.log('Moderator User');

    break;

    default:

        console.log('Unknown User');

}

Whereas, if we define role, we will output guest.

let role; ‘guest’;

switch (role) {

case 'guest':

    console.log('Guest User');

break;

case 'moderator':

    console.log('Moderator User');

    break;

    default:

        console.log('Unknown User');

}

With switch in case, we can compare the value of a variable against multiple other values. They don’t have to be strings, they can be numbers or even Boolean’s… though using Boolean’s here is less common… Because if you want to compare the value of a variable with true and false it makes more sense to use an if statement.

We can also implement this exact logic using if and else statements.

if (role === 'guest') console.log('Guest User');

else if (role === 'moderator') console.log('Moderator User');

else console.log('Unknown User');

Note that the if and else implementation is cleaner and shorter. We don’t have the break statements or default or the curly braces. Some people still use switch and case though they are somewhat outdated. “And look a little bit ugly in my personal opinion.” However, sometimes they are more explicit and it comes down to your own personal preference.

/\*

Suppose we want to determine the role of the current user i.e. guest, moderator, or an admin.

let role;

switch (role) {

case 'guest':

    console.log('Guest');

break;

case 'moderator':

    console.log('Moderator User');

    break;

    default:

        console.log('Unknown User');

The console is outputting unknown user because we have not initialized role and its default value is undefined, and it doesn’t match any of our case statements.

\*/

let role = 'guest';

switch (role) {

case 'guest':

    console.log('Guest User');

break;

case 'moderator':

    console.log('Moderator User');

    break;

    default:

        console.log('Unknown User');

}

//We can also implement this exact logic using if and else statements.

if (role === 'guest') console.log('Guest User');

else if (role === 'moderator') console.log('Moderator User');

else console.log('Unknown User');

## 04 - Control Flow - 03 - For

Suppose we wanted to repeat an action a number of times. For example, we might wish to say hello world five times. Don’t do it like this:

console.log('Hello World');

console.log('Hello World');

console.log('Hello World');

console.log('Hello World');

console.log('Hello World');

To achieve a better result, use loops. Loops repeat an action a number of times.

Loops:

* For
* While
* Do…while
* For…in
* For…of

These loops all do basically the same thing. But there are subtle differences between how they start and end.

Let’s examine the for loop first.

Now we need three statements. The first statement is the initialExpression.

for (initialExpression)

Here we declare and initialize a variable. We use *let* to declare the variable *i*, and set it to *zero*.

for (let i = 0)

I is short for index and is a common convention to use in for loops. This is what we call the loop variable.

After initializing this to zero, we terminate the statement with a semicolon.

for (let i = 0;)

The second part of the for loop is what we call a **condition**.

for (let i = 0; condition)

Here we want to add a condition and compare the value of i with something else. This loop will run as long as is condition evaluates to true.

So, if we want the loop to run five times, we’d compare I with five.

for (let i = 0; i < 5; )

The third part is what we call incrementExpression. Often times will use the increment operator like so, to increment the value of I with one.

for (let i = 0; i < 5; i++)

After this *for* statement, we can add one or more statements. (If you have multiple statements you need to put them in a curly braces code block).

for (let i = 0; i < 5; i++) {

    console.log ('Hello World');

}



So, how does this all work?

As mentioned before, this is what we call an Initial Expression. We are initializing I to 0. ((I has a value of zero)).

for (let i = 0; i < 5; i++) {

    console.log ('Hello World');

}

This loop will execute

for (let i = 0; i < 5; i++) {

    console.log ('Hello World');

}

so long as this condition is true:

for (let i = 0; i < 5; i++) {

    console.log ('Hello World');

}

As long as I is less than five, the log statements will be executed.

After the first iteration, i is incremented by one. Then the i < 5 condition is evaluated again, and one is less than five.

After the second iteration, i is incremented by one, and now were in the third iteration.

To see this in action, let’s output on the console:



At the end of the fifth iteration, i will four, and when we increment that by one I will be five, and the condition will evaluate to false.

So, there are two ways to repeat an action using the for loop. We can initialize I to zero and check to see if it’s less than five.

Alternately, I to 1, and see if it is less than or equal to five.

for (let i = 1; i <= 5; i++) {

    console.log ('Hello World', i);

}

((Note that we are displaying 1 to 5 instead of 0 to 4))



Suppose we only wanted to display the odd numbers between 1 to 5.

We can make an *if* statement check the remainder of the division of I by two.

for (let i = 1; i <= 5; i++) {

if (i % 2 !== 0)

}

((Even numbers Don’t have remainders… i.e., they equal zero)).

So, if the remainder of the division of I by two is not zero, that means I is an odd number.

Let’s displayed on the console.

for (let i = 1; i <= 5; i++) {

if (i % 2 !== 0)

}



We can also work backwards; instead of going from 1 to 5, we can go from 5 to 1.

for (let i = 5; i >= 1; i--) {

   if (i % 2 !== 0) console.log(i);

}

This outputs odd number in the reverse order.

The previous form is more common… initializing I to zero or 1. But for certain problems, you want to use the for loop in the reverse order.

/\*

console.log('Hello World');

console.log('Hello World');

console.log('Hello World');

console.log('Hello World');

console.log('Hello World');

Don't be dumb.

this is how you do it.

for (let i = 0; i < 5; i++) {

    console.log ('Hello World');

}

note the value of the statement displayed on conosole

for (let i = 0; i < 5; i++) {

    console.log ('Hello World', i);

}

We can start i witha value of 1

for (let i = 1; i <= 5; i++) {

    console.log ('Hello World', i);

}

\*/

for (let i = 5; i >= 1; i--) {

   if (i % 2 !== 0) console.log(i);

}

## 04 - Flow Control - 04 - While

Loops:

* For
* While
* Do…while
* For…in
* For…of

In the previous lecture we wrote this for loop to display all odd numbers between zero and five.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

Let’s do the same thing with the while loop.

In for loops, the loop variable is part of the loop itself.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

In while loops, we have to declare this variable externally.

Let’s declare a variable like i, and set it to zero:

*//for (let i = 0; i <= 5; i++) {*

*//    if (i % 2 !== 0) console.log(i);*

*//}*

let i = 0;

With the for loop, the variable was only meaningful and accessible inside the for loop. It’s called **scope**. “These two variables are completely different, even though they are named the same.”

//for (let i = 0; i <= 5; i++) {

//    if (i % 2 !== 0) console.log(i);

}

let i = 0;

Now let’s add a while statement, and in parenthesis will add our condition. Our condition is that I is less than or equal to five.

let i = 0;

while (i <= 5)

Next will add our statements.

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

}

Lastly, at the end of our while block we need to increment i.

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

This is a direct translation from a for loop into a while loop.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

Let’s review the code below. Initially, I is zero.

In the why loop, first this condition is evaluated. If this condition is true, then the body of the while loop will be executed.

In the next iteration the condition is evaluated again. If true, the statements in the while block will be executed, otherwise the while block will terminate.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

((Also, This does not work. Not sure why:

let i = 0;

while (i <= 5; i++;) {

if (i % 2 !== 0) console.log(i);

}

))

## 04 - Control Flow - 05 - Do…While

Loops:

* For
* While
* Do…while
* For…in
* For…of

Let’s rewrite this as a do…while loop.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

let i = 0;

Mosh points out that i has already been declared in this document as part of the while loop. It is *not* incompatible with i in the for loop as that is… contained.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

let i = 0;

So we comment out those lines pressing control/.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

//let i = 0;

//while (i <= 5) {

//if (i % 2 !== 0) console.log(i);

//i++;

//}

let i = 0;

Let’s rewrite the logic above.

Like while loops, we have to declare our loop variable externally. We declare i and initialize it to zero.

Now we add the “do statement” and a code block.

for (let i = 0; i <= 5; i++) {

    if (i % 2 !== 0) console.log(i);

}

//let i = 0;

//while (i <= 5) {

//if (i % 2 !== 0) console.log(i);

//i++;

//}

let i = 0;

do {

}

In this block we should have our statements. Let’s copy them from the while loop.

*for (let i = 0; i <= 5; i++) {*

*if (i % 2 !== 0) console.log(i);*

*}*

*//let i = 0;*

*//while (i <= 5) {*

*//if (i % 2 !== 0) console.log(i);*

*//i++;*

*//}*

let i = 0;

do {

    if (i % 2 !== 0) console.log(i);

     i++;

}

At the end of this block, we add the while statement, along with our condition in parentheses. That’s I less than or equal to five.

*for (let i = 0; i <= 5; i++) {*

*if (i % 2 !== 0) console.log(i);*

*}*

*//let i = 0;*

*//while (i <= 5) {*

*//if (i % 2 !== 0) console.log(i);*

*//i++;*

*//}*

let i = 0;

do {

    if (i % 2 !== 0) console.log(i);

     i++;

} while (i <= 5);

What is the difference between a while loop and a do while loop?

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

let i = 0;

do {

    if (i % 2 !== 0) console.log(i);

     i++;

} while **(i <= 5)**;

Do while loops are always executed at least once even if this **condition** is evaluated to false.

Let’s examine our old while loop.

let i = 0;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}



If we were to let i equal nine, with the old while loop…

let i = 9;

while (i <= 5) {

if (i % 2 !== 0) console.log(i);

i++;

}

the console will display nothing. Because the first time we try to execute the while loop, this condition evaluates to false. The following statements are never executed. So, in while loops this condition is evaluated before the following code block, at the beginning of *every iteration*.

Whereas in do while loops

let i = 0;

do {

    if (i % 2 !== 0) console.log(i);

     i++;

} while (i <= 5);

this condition is evaluated at the end. Thus, the middle statements are always executed at least once, even if the while condition is false.

So, if we set this while loop to nine, we get nine on the console.



“Realistically you’re not going to use this do while a lot in programming. There are situations in which you may want to use this, but in practical terms most of the time you’ll be using a four or a while loop. Just be aware of the difference between a while loop and a do while loop. ”

The// for (let i = 0; i <= 5; i++) {

//     if (i % 2 !== 0) console.log(i);

// }

// let i = 9;

// while (i <= 5) {

// if (i % 2 !== 0) console.log(i);

// i++;

// }

let i = 9;

do {

    if (i % 2 !== 0) console.log(i);

     i++;

} while (i <= 5);

## 04 - Control Flow - 06 - Infinite Loops

Loops:

* For
* While
* Do…while
* For…in
* For…of

An infinite loop executes infinitely. If you create one of these loops, you will crash your browser or your computer.

Let’s declare a variable and set it to zero. We’ll put it in a while loop, where I is less than five, and then console.log.

let i = 0;

while(i < 5) {

    console.log(i);

}

If we forget the increment I (i++), we end up creating an infinite loop.

In the first iteration by zero and is less than five, so we will display i on the console. In all the subsequent iterations I will still be zero.

This log ran up on the console the 6500 times before freezing.

Another example of an infinite loop:

while (true) {

}

You can also create an infinite do while loop.

do {

} while (true);

Or another, where we fail to increment x:

let x = 0;

do {

***// x++;***

} while (x < 5);

We can also create an infinite for loop.

for (let i = 0; i > 0: i++)

or

for (let i = 0; i < 10; )

## 04 - Control Flow - 07 - For…in

Loops:

* For
* While
* Do…while
* For…in
* For…of

With the first three loops we can repeat an action a number of times. We use the other kind of loops to “iterate over the properties of an object or elements of an array”.

Iterate: perform or utter repeatedly. "the bird's call is a monotonously iterated single note"

Suppose we have an object called person with two properties: name and age:

const person = {

    name: 'Mosh',

age: 30

};

Suppose we want to display all the properties of this person object. That’s when we use the *for-in* loop.

Unlike the *for loop*, we **don’t** have an initial expression; a condition; and in increment)

*for (initialExpression; condition; increment)*

const person = {

    name: 'Mosh',

age: 30

};

for (initialExpression; condition; increment)

let key in person:

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

So, in every iteration the key variable in the loop will hold the name of one of the properties in this person object. [[[I.e., the key variable will ***represent***one of the properties of the object. So, in the first iteration it displays name, in the second iteration, it displays the age, etc. Note, that person is the ‘variable’? that key is connected to]]]

Let’s add a console.log of the key variable.

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key);



In the first iteration key is name, in the second-generation key is age.

Suppose we wanted to display the value of each property next to it?

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key);

To access the properties of an object we could use dot notation:

person.name

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key);

person.name

Or, we could utilize bracket notation. Person, square brackets, and the name of the target property as a string.

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key);

person.name

person ['name']

We use the bracket notation when we don’t know ahead of time at the time of writing code what property we are going to access. The name of the target property might be calculated at run time. [[Does this mean with multiple iterations, you don’t know Which property of the object will be retrieved?]]]

E.g. “When we **iterate over** [[iterate *through*, one after another,might be a better way of putting it]] the properties of the person object, in each **iteration**, the value of key is going to be different.”

const person = {

    name: 'Mosh',

age: 30

};

**for (let key in person)**

console.log(key);

person.name

person ['name']

So here we can’t ((I think Mosh says *can’t*)) use the dot notation to display the value of this property. I.e., “we can’t do something like this”:

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person.key);

person.name

person ['name']

“because we don’t have a property called key in the person object”. [[Key refers to the Object person, not to the Properties of person.]]

So that’s when we use the bracket notation. So, we add square brackets and pass key as the name of the target property (as a ‘string’). [[So, instead of saying Name or Age, we use ‘pass’ or pretend that Key is the target property]]

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);



This is the for in loop. We use it to iterate over [[through]] the properties of an object. We can also use these to iterate over [[through]] an array, but it’s not the ideal way. Let’s see an example anyway.

Let’s define an array called colors.

const colors = ['red', 'green', 'blue'];

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

const colors = ['red', 'green', 'blue'];

Now let’s use the for..in loop to iterate over [[go through]] this array.

for (let index in colors)

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

const colors = ['red', **'green'**, **'blue'**];

for (let index in colors)

We named the loop variable *index* because in each iteration, the index variable will be set to the index to one of the elements in this array. So, it’s going to be zero, **one** and **two**.

Let’s have a look, using console.log:

console.log(index);

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

const colors = ['red', 'green', 'blue'];

for (let index in colors)

console.log(index);



If you want the ***element*** at the given index, once again, we use the square bracket notation. Colors of index:

console.log(index, colors[index]);

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

const colors = ['red', 'green', 'blue'];

for (let index in colors)

console.log(index, colors[index]);



Now we can see each element of our colors array.

Starting from ecma script 6, or es6, which is the modern version of javascript, we have a new kind of loop, called for-of loop. This is the ideal way to iterate over [[through]] arrays.

/\*

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key);

//this displays "name" and then "age" in the second iteration

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person.key);

//dot notation ((e.g., person.key)) does not work.  There is no property called key in //the person object.

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

//this 'passes' key as the name of the target property.  I think this means key stands in for or pretends to be the property of the person object, and it runs/iterates through them.  Now both the property, And the property's value is displayed.

\*/

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

const colors = ['red', 'green', 'blue'];

for (let index in colors)

console.log(index, colors[index]);

//with console.log(index), the numerical values, i.e., the index varialbe is set to the index of one of the elements in the array... will be displayed.  Whereas, when colors[index] is included, the Element at a given index will be displayed

## 04 - Control Flow - 08 - For…of

“Starting from ecma script 6, or es6, we have a new way to iterate over arrays. And that is using the *for…of* loop.”

This is the ideal way to iterate over [[through]] arrays.

“The for-of is very similar to the for-in loop but instead of the in keyword, we use the *of* keyword.”

E.g.,

for (let color of colors);

*const person = {*

*name: 'Mosh',*

*age: 30*

*};*

*for (let key in person)*

*console.log(key, person[key]);*

const colors = ['red', 'green', 'blue'];

*for (let index in colors)*

*console.log(index, colors[index]);*

for (let color of colors);

Now we can simply log this color on the console:

console.log(color);

*const person = {*

*name: 'Mosh',*

*age: 30*

*};*

*for (let key in person)*

*console.log(key, person[key]);*

const colors = ['red', 'green', 'blue'];

*for (let index in colors)*

*console.log(index,* ***colors[index])****;*

for (let color of colors)

console.log(color);

“With this new four-loop, we don’t have to deal with this (above) index. And we don’t have to access the element at the **given index.**  So in each iteration, this color, which is our loop variable will hold one of the items in this array.”

((Zack. Hence, we are utilizing the color variable, rather than the value of the key/index.))



To summarize:

We use the for-in loop to iterate over the properties of an object.

And the for-of loop to iterate over the elements, or items in an array.

//for-in

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

//for-of

const colors = ['red', 'green', 'blue'];

for (let color of colors)

console.log(color);

/\*

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

const colors = ['red', 'green', 'blue'];

for (let index in colors)

console.log(index, colors[index]);

for (let color of colors)

console.log(color);

//with the for of loop, the color variable will 'hold' an item in the array as it iterates through.  It won't also display the index number, like with you use a for-in loop.

\*/

//for-in

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);

//for-of

const colors = ['red', 'green', 'blue'];

for (let color of colors)

console.log(color);

//We use the for-in loop to iterate over the properties of an object.

//And the for-of loop to iterate over the elements, or items in an array.

## 04 - Control Flow - 09 - Break and Continue

With all the loops we’ve learned about in this section, the two keywords break and continue they can change how the loop behaves.

Let’s use a while loop as an example.

Let’s declare a variable:

let

We’ll call it i:

let i

And *initialize* to zero:

let i *= 0;*

Now will put this in a while loop:

while (i <= 10)

let i = 0;

while (i <= 10)

And display I on the console:

console.log(i);

let i = 0;

while (i <= 10) {

    console.log(i);  
}

And increment it:

 i++;

let i = 0;

while (i <= 10) {

    console.log(i);

    i++;

}



Sometimes, you want to jump out of a loop for some reason that may happen at runtime.

E.g., an *if* condition

if

let i = 0;

while (i <= 10) {

    if

    console.log(i);

    i++;

}

And an if statement ()

  if ()

let i = 0;

while (i <= 10) {

    if ()

    console.log(i);

    i++;

}

If i = 5…

 if (i === 5)

let i = 0;

while (i <= 10) {

    if (i === 5)

    console.log(i);

    i++;

}

…we want to jump out of the loop.

This is where we use the *break* keyword.

 if (i === 5) break

let i = 0;

while (i <= 10) {

    if (i === 5) break

    console.log(i);

    i++;

}



Now let’s examine the continue keyword. Let’s write a new *if* statement.

If i is an even number. i, modulus 2 equals zero. (Modulus is another term for Absolute Value).

If (i % 2 === 0)

 if (i % 2 === 0)

let i = 0;

while (i <= 10) {

   // if (i === 5) break

   if (i % 2 === 0)

    console.log(i);

    i++;

}

If so, increment I, and continue:

if (i % 2 === 0) {

    i++;

    continue;

let i = 0;

while (i <= 10) {

   // if (i === 5) break

   if (i % 2 === 0) {

    i++;

    continue;

   }

    console.log(i);

    i++;

}



E.g., when i is 2, it is an even number, and then we increment i… so i will be 3. When the javascript engine sees the keyword continue… it will jump to the beginning of the loop…

while (i <= 10) {

…and continue execution of the next iteration.

At this point, i is 3, and the if statement is not executed. That is why we see i logged on the console as 3. [[[In other words, if the *if* statement conditions are met, then it continues to the beginning, and does not reach the console.log]]]

Mosh does not use continue that often… it is old, legacy code. But you may see it in projects that you are working on. “It is an ugly way of writing code.”

With the break keyword we jump out of a loop, and with the continue keyword, jump to the next iteration [[[at the top]]].

## 04 - Control Flow - Exercise - 10 - Max of Two Numbers

“I want you to write a function that takes two numbers and returns the maximum of the two.

Call that function, give a different argument, and make sure it is working properly”.

OK, first attempt, while yielding the correct answer, is Not a function. Mosh does indeed want a function.

if (10>20) {console.log(10)}

else

console.log(20)

I see this much from his lesson, so maybe I can finish:

*function maximum (number1, )*

This is a fail:

function maximum (number1, number2) {

    if (number1 > number2) {console.log(number1)

    else

console.log(number2)}

}

let number1 = 10

let number2 = 20

Well, let’s play with our if…else just for fun:

if (number1 > number2) {console.log(number1)}

else

console.log(number2)

let number1 = 10

let number2 = 20



Ah… need to input the values Before executing the code.

let number1 = 10

let number2 = 20

if (number1 > number2) {console.log(number1)}

else

console.log(number2)

Ah, that works.

This does not:

let number1 = 10

let number2 = 20

function maximum(number1, number2) {

    if (number1 > number2) {console.log(number1)}

    else

console.log(number2)}

}

This does not work:

function maximum(1, 5) {

    if (1 > 5) {console.log(1)}

    else

console.log(5) }

Well, this doesn’t throw an error message:

function maximum(number1, number2) {

    return number1 > number2;

}

Nor does this… I thought I saw that it did… hmm.

function maximum(number1, number2) {

    return number1 > number2;

    return number1 < number2;

}

This just returns False:

function maximum(number1, number2) {

    return number1 > number2;

    return number1 < number2;

}

console.log(maximum(10, 20));

This returns true:

function maximum(number1, number2) {

    return number1 > number2;

    return number1 < number2;

}

console.log(maximum(10, 5));

Well… I can’t just log the output of maximum, because I have two outputs.

Ok, well this worked:

function maximum(number1, number2) {

    if (number1 > number2) return number1

    else return number2

}

console.log(maximum(10, 20));

So did this:

function maximum(number1, number2) {

    if (number1 > number2) {return number1}

    else return number2

}

console.log(maximum(50, 20));

Perhaps this is better?

function maximum(number1, number2) {

    if (number1 > number2) {return number1;}

    else return number2;

}

console.log(maximum(50, 20));

Well… it may all be nonsense.  
  
  
Hmmm, better still?

function maximum(number1, number2) {

    if (number1 > number2) {return number1;

    }

    else return number2;

}

console.log(maximum(50, 20));

Now, let’s follow along with Mosh:

Let’s start by defining a function, will call it Max. We will add two parameters: number one, and number two.

function max(number1, number2)

Or we can use shorter names like a and B. Mosh says A and B are kind of self-explanatory in this case, so we can go ahead and use these undefined names which he normally warned against.

“We want to have some logic, we want to compare the value of a with B, so were going to use an if statement”

If a is greater than be, we want to return a.

if (a > b) return a;

Else, we want to return b.

function max(a, b) {

if (a > b) return a;

else return b;

}

“This is the simplest implementation, it’s not the best”.

Let’s declare a variable called number, and set it to max of one and two.

let number = max(1, 2);

let number = max(1, 2);

function max(a, b) {

if (a > b) return a;

else return b;

}

Now we’ll display the number on the console.

let number = max(1, 2);

console.log(number);

function max(a, b) {

if (a > b) return a;

else return b;

}

(Mosh notes that we “called this function, with different arguments. Different test cases” I.e., different numbers.

Let’s clean up this code. Let’s remove the *else* keyword. Mosh says that if a is true, we will return A and jump out of the function, and the subsequent code won’t be executed. If A is untrue, we should just return b.

return b;

let number = max(1, 2);

console.log(number);

function max(a, b) {

if (a > b) return a;

return b;

}

We can use the conditional operator to make this even simpler. We have a (condition) in parenthesis, a ?, and if this condition evaluates to true, we use one value, otherwise we use the other.

(condition) ? a : b.

Jumping out of the lesson. Hmmm. How can I implement this?

This comes back as undefined:

let number = max(1, 2);

console.log(number);

function max(a, b) {

(a > b) ? a : b

}

Ah, this did it:

return (a > b) ? a : b

let number = max(1, 2);

console.log(number);

function max(a, b) {

return (a > b) ? a : b

}

Back to Mosh:

We use the conditional operator. For the condition, is A > B. If this condition is true ? return A. Otherwise return B.

a > b) ? a : b

let number = max(1, 2);

console.log(number);

function max(a, b) {

(a > b) ? a : b

}

All we need is a return statement, and to terminate with a semicolon.

return (a > b) ? a : b;

let number = max(1, 2);

console.log(number);

function max(a, b) {

return (a > b) ? a : b;

}

## 04 - Control Flow - Exercise - 11 - Landscape or Portrait

function isLandscape(width, height) {

}

Implement this function. Function, isLandscape, and it takes two parameters. Width, and height of an image.

Returns true if the image is landscape, meaning width is greater than height. Otherwise, it returns false. This is similar to the previous exercise, but Mosh has a tip.

So, we want to return a Boolean instead of a numerical value, I guess. I think I did that accidentally earlier…

OK, this did not work… wait, it did. I just had to reload the browser

let boolean = isLandscape(5, 2);

console.log(boolean)

function isLandscape(width, height) {

    if (width > height) return true

   return false

}

We could pare this down:

let boolean = isLandscape(5, 2);

console.log(boolean)

function isLandscape(w, h) {

    if (w > h) return true

    return false

}

Let’s see if we can do the same thing with our conditional operator.

Well, this didn’t work.

let boolean = isLandscape(5, 2);

console.log(boolean)

function isLanscape(w, h) {

    (w > h) ? true : false

}

Nor did this, when we added return:

let boolean = isLandscape(5, 2);

console.log(boolean)

function isLanscape(w, h) {

return (w > h) ? true : false

}

Ah ha… this Does work. The one above had a Typo in isLandscape. Duuuur.

let boolean = isLandscape(1, 2);

console.log(boolean);

function isLandscape(w, h) {

    return (w > h) ? true : false;

}

Interestingly enough, this did not work… when I capitalized False:

let boolean = isLandscape(1, 2);

console.log(boolean);

function isLandscape(w, h) {

    return (w > h) ? true : False;

}

Though it did work when I turned it into a string:

let boolean = isLandscape(1, 2);

console.log(boolean);

function isLandscape(w, h) {

    return (w > h) ? 'true' : 'False';

}

Now, we do it Mosh’s way.

function isLandscape(width, height) {

    if (width > height) return true;

    return false;

}

Using the conditional operator:

function isLandscape(width, height) {

    return (width > height) ? true : false;

}

However, the code above “looks very amateurish. You don’t want to return true or false explicitly. This is very ugly”.

The reason being that we can delete the conditional operator, “and simply return the value of this expression”.

function isLandscape(width, height) {

    return (width > height) ? true : false;

}

If width is greater than height, the expression will be evaluated to true. If width is less than height, this expression will evaluate to false. “So there is no need to explicitly return true and false here. That’s a poor way of writing code”.

Let’s do a console.log. “Simply call this function here, isLandscape”

console.log(isLandscape)

function isLandscape(width, height) {

    return (width > height)

}

Let’s pass the dimensions 800 x 600:

console.log(isLandscape(800, 600));

function isLandscape(width, height) {

    return (width > height)

}

let boolean = isLandscape(5, 2);

console.log(boolean)

function isLandscape(w, h) {

    if (w > h) return true

     return false

}

let boolean = isLandscape(1, 2);

console.log(boolean);

function isLandscape(w, h) {

    return (w > h) ? true : false;

}

function isLandscape(width, height) {

    if (width > height) return true;

    return false;

}

function isLandscape(width, height) {

    return (width > height) ? true : false;

}

function isLandscape(width, height) {

    return (width > height)

}

\*/

console.log(isLandscape(300, 600));

function isLandscape(width, height) {

    return (width > height)

}

## 04 - Control Flow - Exercise - 12 - FizzBuzz

This question is a popular interview question called the fizz buzz algorithm.

So, we have this function called fizzBuzz. We give it an input, and it returns a string.

function fizzBuzz(input) {

}

Let’s declare a constant called output… here we *call* fizzBuzz, and pass an input, like 3

const output = fizzBuzz(3);

function fizzBuzz(input) {

}

Now, log the output:

const output = fizzBuzz(3);

console.log(output);

function fizzBuzz(input) {

}

Mosh’s console outputs Fizz. Mine does not. It just says undefined. I tried switching from edge to chrome, no difference.



“So if the number that we pass is divisible by three, we get Fizz.”

If the number that we pass is divisible by 5, we get Buzz.

If the number we pass is divisible by 3 and 5, like 15, we get fizzBuzz

If the number is not divisible by either 3 or 5, we get the same number… e.g., 7.

if (hour >= 6 && hour < 12 )

    console.log('Good morning');

else if (hour >= 12 && hour < 18)

    console.log('Good Afternoon!');

else

console.log('Good evening');

Let’s aim for outputting just one variable. Switch and case seems another way.

Hmmm. Looks like Mosh used if. We’ll try switch and case too.

const output = fizzBuzz(3);

console.log(output);

function fizzBuzz(input) {if 10 > 1 return true; }

So my problem is, I’m not sure how to output anything.

Ah. Had to put this in parenthesis.

const output = fizzBuzz(3);

console.log(output);

function fizzBuzz(input) {if (10 > 1)

    return true; }

This outputs an error.

const output = fizzBuzz(6);

console.log(output);

function fizzBuzz(input) {if (i % 3 !== 0)

    return true; }

This outputs true

const output = fizzBuzz(6);

console.log(output);

function fizzBuzz(input) {if (fizzBuzz % 3 !== 0)

    return true; }

returns fizz

const output = fizzBuzz(6);

console.log(output);

function fizzBuzz(input) {if (fizzBuzz % 3 !== 0)

    return 'fizz'; }

ah… returns true, even if the number Isn’t divisible by 3.

const output = fizzBuzz(2);

console.log(output);

function fizzBuzz(input) {if (fizzBuzz % 3 !== 0)

    return 'fizz'; }

Ahhh. The !== means Not divisible? === outputs false

const output = fizzBuzz(2);

console.log(output);

function fizzBuzz(input) {if (fizzBuzz % 3 %== 0)

    return 'fizz';

else return false }

Changing fizzbuzz to 5 yields the same result:

const output = fizzBuzz(5);

console.log(output);

function fizzBuzz(input) {if (5 % 2 !== 0)

    return 'fizz';

else return false }

Ah… this needed to be changed to input… durr.

const output = fizzBuzz(4);

console.log(output);

function fizzBuzz(input) {if (input % 2 !== 0)

    return 'fizz';

else return false }

Ok, so this works for 3:

const output = fizzBuzz(3);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0)

    return 'fizz';

else return false }

Logical And && washed out…

const output = fizzBuzz(15);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return fizzBuzz

    (input % 3 === 0)

    return 'fizz';

else return false }

So did this:

function fizzBuzz(input) {if ((input % 3 === 0) && (input % 5 === 0))

    return fizzBuzz

    (input % 3 === 0)

    return 'fizz';

else return false }

Removing the 3 code produces a weird error:

const output = fizzBuzz(15);

console.log(output);

function fizzBuzz(input) {if ((input % 3 === 0) && (input % 5 === 0))

    return fizzBuzz

else return false }



This gives us a regular error:

const output = fizzBuzz(15);

console.log(output);

function fizzBuzz(input) {if ((input % 3 === 0) && (input % 5 === 0));

    return fizzBuzz

else return false }

Hey dummy… did you remember to pass a string? This works:

const output = fizzBuzz(30);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    {return 'fizzBuzz';}

else return false }

We can pare this back down… I had tried a few things:

const output = fizzBuzz(30);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

else return false }

Ok… now to try not divisible by either:

const output = fizzBuzz(9);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 3 !== 0 && input % 5 !==)

    return (input)

else return false }

This kinda works

const output = fizzBuzz(11);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 3 !== 0)

    return (input)

else return false }

Hmmm. 10 returns false… maybe I need to write up something for 5 now:

const output = fizzBuzz(10);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input)

else return false }

Ok, this works great:

const output = fizzBuzz(101);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input)

else return 'false' }

This doesn’t work. Error message.

const output = fizzBuzz(dog);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input)

else return 'Not a number' }

If we input the string, we get back the string… instead of not a number.

const output = fizzBuzz('dog');

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input)

else return 'Not a number' }

Hmmm. How do we get the return input to exclude non numbers? I tried making input divisible by 1… also tried input plus 1

const output = fizzBuzz('dog');

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input % 1 === 0)

else return 'Not a number' }

Haha. Input plus zero is thus: (outputted dog0)

const output = fizzBuzz('dog');

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input + 0)

else return 'Not a number' }

Ok, dog input % 1 is false.

const output = fizzBuzz('dog');

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input % 1 === 0)

else return 'Not a number' }

Non 3, 5 and either numbers output to true. Strings output to ‘Not a number’. False, outputs to fizzbuzz. True, outputs to true.

const output = fizzBuzz(17);

console.log(output);

function fizzBuzz(input) {if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 1 !== 0)

    return 'Not a number'

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input % 1 === 0)

}

Ok, so instead of input % 1, we did input/1. Plus some rearranging. Everything seems to work.

const output = fizzBuzz(14);

console.log(output);

function fizzBuzz(input) {if (input === true)

    return 'Not a number'

    else if (input === false)

    return 'Not a number'

    else if (input % 1 !== 0)

    return 'Not a number'

    else if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 3 !== 0  && input % 5 !== 0)

    return (input/1)

}

Wait… why would we need input/1? Just input. Must have been bad formatting before:

const output = fizzBuzz(false);

console.log(output);

function fizzBuzz(input) {if (input === true)

    return 'Not a number'

    else if (input === false)

    return 'Not a number'

    else if (input % 1 !== 0)

    return 'Not a number'

    else if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 3 !== 0 && input % 5 !== 0)

    return (input)

Let’s see Mosh’s solution. Ahh. Here is what we wanted.

const output = fizzBuzz(false);

console.log(output);

function fizzBuzz(input) {

    if (typeof input !== 'number')

   return 'Not a number'

}

“Here we’re dealing with a complex expression. To make this code more readable, I would like to put each expression in parenthesis.”

const output = fizzBuzz(false);

console.log(output);

function fizzBuzz(input) {

    if (typeof input !== 'number')

   return 'Not a number';

   if (input % 3 === 0)

   return 'Fizz';

   if (input % 5 === 0)

   return 'Buzz';

   if (input % 3 === 0 && input % 5 === 0)

}

Select the parenthesis, type ( and it will automatically enclose it ^^ tips and tricks.

const output = fizzBuzz(false);

console.log(output);

function fizzBuzz(input) {

    if (typeof input !== 'number')

   return 'Not a number';

   if (input % 3 === 0)

   return 'Fizz';

   if (input % 5 === 0)

   return 'Buzz';

   if ((input % 3 === 0) && (input % 5 === 0))

}

Ah, ha. Mosh pulled a fast one on us. He is showing that order of operation matters. Also, to move code, select it, push the alt key, and the arrows ^^. Tips and tricks.

console.log(output);

function fizzBuzz(input) {

    if (typeof input !== 'number')

   return 'Not a number';

   if ((input % 3 === 0) && (input % 5 === 0))

   return 'FizzBuzz';

   if (input % 3 === 0)

   return 'Fizz';

   if (input % 5 === 0)

   return 'Buzz';

   return input;

}

In JS, there is a special value called Not A Number. NaN

const output = fizzBuzz(dog);

console.log(output);

function fizzBuzz(input) {

    if (typeof input !== 'number')

   return NaN;

   if ((input % 3 === 0) && (input % 5 === 0))

   return 'FizzBuzz';

   if (input % 3 === 0)

   return 'Fizz';

   if (input % 5 === 0)

   return 'Buzz';

   return input;

}



NaN returns as a number. Weird. But it isn’t a valid mathematical number.

/\*

divisible by 3, => Fizz

divisibile by 5 => Buzz

divisible by both 3 and 5 => FizzBuzz

Not divisible by 3 or 5 => input

not a number => 'Not a number"

 (input % 3 === 0)

    return 'fizz';

    const output = fizzBuzz(false);

console.log(output);

function fizzBuzz(input) {

const output = fizzBuzz(7);

console.log(output);

function fizzBuzz(input) {if (input === true)

    return 'Not a number'

    else if (input === false)

    return 'Not a number'

    else if (input % 1 !== 0)

    return 'Not a number'

    else if (input % 3 === 0 && input % 5 === 0)

    return 'fizzBuzz'

    else if (input % 3 === 0)

    return 'fizz'

    else if (input % 5 === 0)

    return 'Buzz'

    else if (input % 3 !== 0 && input % 5 !== 0)

    return (input)

}

\*/

const output = fizzBuzz(false);

console.log(output);

function fizzBuzz(input) {

    if (typeof input !== 'number')

   return NaN;

   if ((input % 3 === 0) && (input % 5 === 0))

   return 'FizzBuzz';

   if (input % 3 === 0)

   return 'Fizz';

   if (input % 5 === 0)

   return 'Buzz';

   return input;

}

## 04 - Control Flow - Exercise - 13 - Demerit Points

Mosh says this is more difficult than the earlier exercises.

checkSpeed(50);

function checkspeed(speed) {

}

This function takes a parameter that’s the speed of a car. Speed Limit is 70 kph.

So, if a car is driving under or at the speed limit, output OK.

For every 5 above the speed limit, they get 1 point.



If you are at 72… still output to OK.

“As part of calculating the point here, you’ll have to use one of the built in functions in JS. Math.floor.

Math.floor

checkSpeed(50);

function checkspeed(speed) {

}

We can give this function a floating point number like 1.3… or 1.3 points… and this will convert that to the greatest integer.

Math.floor(1.3)

checkSpeed(50);

function checkspeed(speed) {

}

If we pass 80, we get 2 points. If we do 180, it outputs License suspended. So… if a driver gets more than 12 points, license suspended.

Same trouble as usual. How do I get this out output anything?

Math.floor(1.3)

checkSpeed(50);

function checkspeed(speed) {console.log(checkspeed)

}

Math.floor(1.3)

checkSpeed(50);

function checkspeed(speed) {console.log(speed)

}

Math.floor(1.3)

checkSpeed(50);

function checkspeed(speed) {return console.log(speed)

}

Ah ha… we first discover a Typo! Now we are outputting. Displays 50.

checkSpeed(50);

function checkSpeed(speed) {console.log(speed);

}

Trying to log checkSpeed, does this recursive thing, displaying the entire function:

checkSpeed(50);

function checkSpeed(speed) {console.log(checkSpeed);

}



Ok, now we’ll add a condition. And over 75, we don’t display.

checkSpeed(50);

function checkSpeed(speed) { if (speed < 75) console.log(speed);

}

Now, we try to output the string OK, and we fail… oh, duh, because we are over 75.

checkSpeed(90);

function checkSpeed(speed) { if (speed < 75) console.log('OK');

}

Durrr.

checkSpeed(40);

function checkSpeed(speed) { if (speed < 75) console.log('OK');

}

Ok, if speed is less than or equal to 74… we return OK.

checkSpeed(40);

function checkSpeed(speed) { if (speed <= 75) console.log('OK');

}

So, 12 times 5 is 60. Max speed is 130

So, subtracting 75 from speed, and then dividing by 5 seems like the answer… just need to output it.

checkSpeed(80);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if console.log((speed) - 75)

}

Ah… here we are.

checkSpeed(80);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75)

console.log(speed -75)

}

OK… better still:

checkSpeed(80);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75)

console.log((speed -75)/5)

}

Now, here’s the string:

checkSpeed(80);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75)

console.log('Point: '+ (speed -75)/5)

}

Ah… but 75 is Zero points.

checkSpeed(75);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75)

console.log('Point: '+ (speed -75)/5)

}

OK… 75 is 1 point now:

checkSpeed(75);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75)

console.log('Point: '+ ((speed -75)/5 + 1) )

}

Ah ha… 89 comes out to 3.8:

checkSpeed(89);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75)

console.log('Point: '+ ((speed -75)/5 + 1) )

}

Ah ha… here is Math.floor, changing a 91 to a whole number:

checkSpeed(91);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75)

console.log('Point: '+ (Math.floor((speed -75)/5 + 1)) )

}

OK… now all we have to do is make sure this only applies to under 130.

checkSpeed(140);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75  && speed < 130)

console.log('Point: '+ (Math.floor((speed -75)/5 + 1)) )

 return console.log('License Suspended')

}

However, if Under 130, we still get points displaying:

checkSpeed(100);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75  && speed < 130)

console.log('Point: '+ (Math.floor((speed -75)/5 + 1)) )

 return console.log('License Suspended')

}



Ah… just have to add else

checkSpeed(90);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75  && speed < 130)

console.log('Point: '+ (Math.floor((speed -75)/5 + 1)) )

else return console.log('License Suspended')

}

OK… problem solved.

Lets see Mosh’s Solution.

“The first thing I want to improve here is to turn this magic number into a constant. Because someone else looking at this code may not know what 70 represents here. So you should avoid using magic numbers in your code. Always use constants to describe them”

function checkSpeed(speed) {

    if (speed < 70)

    console.log('OK');

}

We establish a constant speedLimit, and then change (speed < 70) to (speed < speedLimit).

function checkSpeed(speed) {

**const speedLimit = 70;**

    if (speed < speedLimit)

    console.log('OK');

}

“With this change, now our code is more expressive, and also if we want to use this number somewhere else in this function, we don’t have to repeat it multiple times. We have defined it **here** once, and we can reuse it in multiple places.

If tomorrow, the speed limit changes to, let’s say, 75 or 65, there is only a single place we have to change.”

“Once again, we don’t want to use this magic number here… it’s not descriptive.”

function checkSpeed(speed) {

    const speedLimit = 70;

    if (speed < speedLimit)

    console.log('OK');

    else {

         (speed - speedLimit) / 5

    }

}

Instead, we define another constant. Kilometer per point:

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         (speed - speedLimit) / kmPerPoint

    }

}

“With this we get the number of points”. And Mosh changes this to: *let points = (speed - speedLimit) / kmPerPoint;*

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         let points = (speed - speedLimit) / kmPerPoint;

    }

}

The result of this expression can be a floating point number.

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         let points = (speed - speedLimit) / kmPerPoint;

    }

}

So, let’s call Math.floor… and as an argument, we pass the highlighted expression:

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         let points = Math.floor((speed - speedLimit) / kmPerPoint);

    }

}

“Now we should check to see if the driver gets more than 12 points. So, we need another if statement. If points is greater than or equal to 12, we want to suspend their license.”

[[[I’d note that this is basically a *let* *if* statement. Interesting. ]]]

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         let points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

    }

}

“Otherwise, we want to display the number of points on the console… and as a second argument, we pass this points variable” [[[The first is string, the second is a value]]]

   else

         console.log('Points', points);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         let points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

“Now technically here I could also use a constant, and that’s a better practice to make sure I don’t accidentally modify this points here”

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         let points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

So, we’ll change this to a constant:

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         const points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

And if we try to modify the points like so, we’ll get an error:

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         const points = Math.floor((speed - speedLimit) / kmPerPoint);

         points++;

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

“Const should be your default choice. If you want to reassign a variable, that’s when you use the let keyword.”

“Let’s call this function, *checkSpeed* and pass 50.

checkSpeed(50);

checkSpeed(50);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         const points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

If we drive exactly at the speed limit, we get “Points 0”, instead of the OK message. So there is a bug in our condition here. Let’s change it to less than or equal to. [[[I’m not repasting that small change.]]]

checkSpeed(70);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         const points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

Ok… now if you’re going 71, you get the “Points 0” message instead of OK. So, we still have a bug.

Let’s change

 if (speed <= speedLimit)

to:

    if (speed < speedLimit + kmPerPoint)

[[[kmPerPoint is 5… so it’s basically like adding 5. Then we change it to less than]]]

checkSpeed(70);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed < speedLimit)

    console.log('OK');

    else {

         const points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

“So our function is working. I just want to show you one more tip to make our code cleaner.

Here, we have this else block, and this has caused indentation here[[[ here being, say, the space before const]]]. “Now this is not a terribly bad thing, but it is better to avoid indentation if we can. Because otherwise we have to scroll to the left and right to see what’s happening here.”

checkSpeed(70);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed <= speedLimit + kmPerPoint)

    console.log('OK');

    else {

         const points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

“So, if this condition is true, we want to display ok on the console. To get rid of this else block…”

checkSpeed(70);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed <= speedLimit + kmPerPoint)

    console.log('OK');

    else {

         const points = Math.floor((speed - speedLimit) / kmPerPoint);

         if (points >= 12)

         console.log('License suspended')

         else

         console.log('Points', points);

    }

}

“…We can add a block here” and stick console.log inside it. And then add a return after: “With this we will jump out of this function, and none of the code **here** will be executed”

 if (speed <= speedLimit + kmPerPoint) {

        console.log('OK');

        return;

    }

checkSpeed(70);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed <= speedLimit + kmPerPoint) {

        console.log('OK');

        return;

    }

**else {**

**const points = Math.floor((speed - speedLimit) / kmPerPoint);**

**if (points >= 12)**

**console.log('License suspended')**

**else**

**console.log('Points', points);**

    }

}

Now we can remove the else statement and the code block, and remove the indentation:

checkSpeed(70);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed <= speedLimit + kmPerPoint) {

        console.log('OK');

        return;

    }

    const points = Math.floor((speed - speedLimit) / kmPerPoint);

    if (points >= 12)

    console.log('License suspended')

    else

    console.log('Points', points);

}

/\*

My solution:

checkSpeed(80);

function checkSpeed(speed) { if (speed <= 74) console.log('OK');

else if (speed >= 75  && speed < 130)

console.log('Point: '+ (Math.floor((speed -75)/5 + 1)) )

else return console.log('License Suspended')

}

\*/

checkSpeed(70);

function checkSpeed(speed) {

    const speedLimit = 70;

    const kmPerPoint = 5;

    if (speed <= speedLimit + kmPerPoint) {

        console.log('OK');

        return;

    }

    const points = Math.floor((speed - speedLimit) / kmPerPoint);

    if (points >= 12)

    console.log('License suspended')

    else

    console.log('Points', points);

}

## 04 - Control Flow - 14 - Exercise 5 - Even and Odd Numbers

Write a function called show numbers that takes a parameter called limit. So, we pass a number here like 10, and when we call this function, we should see all the numbers from zero to the number we supplied as the limit.

showNumbers(10);

function showNumbers(limit) {

}



“Next to each number, if that number is even, you should display the even label. Otherwise, you should display odd”

Mosh says this is easy. Looks like a loop of some kind. Let’s see if we can make something display:

Fail.

showNumbers(10);

function showNumbers(limit) {

    for (let i = 0; i < limit)

    console.log(i)

}

OK… this displays.

showNumbers(10);

function showNumbers(limit) {

    let i = 0;

    console.log(i)

}

But this does Not work:

showNumbers(10);

function showNumbers(limit) {

    (let i = 0;)

    console.log(i)

}

This does:

showNumbers(10);

function showNumbers(limit) {

    let i = 0;

    i <= limit;

    console.log(i)

}

This only iterates once:

showNumbers(10);

function showNumbers(limit) {

    let i = 0;

    i <= limit;

    i++

    console.log(i)

}

Huh. This works… why didn’t the earlier expressions? Maybe they did

showNumbers(10);

function showNumbers(limit) {

    for (let i = 0;

    i <= limit;

    i++)

    console.log(i)

}

Well, no replacing for with let, willy nilly. This didn’t work:

showNumbers(10);

function showNumbers(limit) {

    if (let i = 0;

    i <= limit;

    i++)

    console.log(i)

}

This only displays even numbers, 0 to 10:

showNumbers(10);

function showNumbers(limit) {

    for (let i = 0;

    i <= limit;

    i++)

    {if (i % 2 === 0) console.log(i + ' ' + "EVEN")}

Well… this is no good. Just an error message:

showNumbers(10);

function showNumbers(limit) {

    for (let i = 0;

    i <= limit;

    i++)

    {if (i % 2 === 0) console.log(i + ' ' + "EVEN") || (i % 2 !== 0) console.log}

Hmm. This was an error:

showNumbers(10);

function showNumbers(limit) {

    for (let i = 0;

    i <= limit;

    i++)

    {if (i % 2 === 0) console.log(i + ' ' + "EVEN")}

    {if (i % 2 !== 0) console.log(i + ' ' + "ODD")}

}

Ok… so this works with an *if else*.

showNumbers(10);

function showNumbers(limit) {

    for (let i = 0;

    i <= limit;

    i++)

    {if (i % 2 === 0) console.log(i + ' ' + '"EVEN"')

    else if (i % 2 !== 0) console.log(i + ' ' + '"ODD"')

}

}

I note that inputting true gives me weird results.



What was that return only numbers thing?

function fizzBuzz(input) {

    if (typeof input !== 'number')

   return 'Not a number'

}

Ok… this is my output:

showNumbers('dog');

function showNumbers(limit) {

    if (typeof limit !== 'number')

    return

    for (let i = 0;

    i <= limit;

    i++)

    {if (i % 2 === 0) console.log(i + ' ' + '"EVEN"')

    else if (i % 2 !== 0) console.log(i + ' ' + '"ODD"')

}

}

Ok, I got rid of some Pluses by odd and even:

showNumbers(100);

function showNumbers(limit) {

    if (typeof limit !== 'number')

    return

    for (let i = 0;

    i <= limit;

    i++)

    {if (i % 2 === 0) console.log(i + ' "EVEN"')

    else if (i % 2 !== 0) console.log(i + ' "ODD"')

}

}

So, let’s see Mosh’s solution.

“Here we need a *for* loop. Set our i variable to zero. As long as i is less than or equal to limit, we’re going to run this loop. And in every iteration, we’ll increment i”.

showNumbers(10);

function showNumbers(limit) {

for (let i = 0; i <= limit; i++) {

}

}

“Here we need to check if i is an even number or not. If i modulus 2 equals zero, we display console.log of i, and as a second argument we pass ‘EVEN’.” (((Ah, he used a coma instead of +)))

showNumbers(10);

function showNumbers(limit) {

for (let i = 0; i <= limit; i++) {

    if (i % 2 === 0) console.log(i, 'EVEN');

}

}

“Otherwise, we do a console.log of i and odd.” (((Ah, no need for any conditions))).

showNumbers(10);

function showNumbers(limit) {

for (let i = 0; i <= limit; i++) {

    if (i % 2 === 0) console.log(i, 'EVEN');

    else console.log(i, 'ODD');

}

}

“Instead of having two separate console.log statements, we can declare a constant. Call it Message. And here we can use the conditional operator.”

showNumbers(10);

function showNumbers(limit) {

for (let i = 0; i <= limit; i++) {

const message = (i % 2 === 0) ? 'EVEN' : 'ODD';

console.log(i, message);

}

}

Mosh thinks the turnery operator is cleaner and less noisy, but the first is fine as well.

My solution

showNumbers(100);

function showNumbers(limit) {

    if (typeof limit !== 'number')

    return

    for (let i = 0;

    i <= limit;

    i++)

    {if (i % 2 === 0) console.log(i + ' "EVEN"')

    else if (i % 2 !== 0) console.log(i + ' "ODD"')

}

}

first mosh solution

showNumbers(10);

function showNumbers(limit) {

for (let i = 0; i <= limit; i++) {

    if (i % 2 === 0) console.log(i, 'EVEN');

    else console.log(i, 'ODD')

}

}

\*/

showNumbers(10);

function showNumbers(limit) {

for (let i = 0; i <= limit; i++) {

const message = (i % 2 === 0) ? 'EVEN' : 'ODD';

console.log(i, message);

}

}

## 04 - Control Flow - 15 - Exercise 6 - Count Truthy - 4.33

“Takes an array, and returns the number of truthy elements in this array”.

E.g,. “If we have an if statement, we may pass a Boolean true or false as the condition here:

if ()

function countTruthy(array) {

}

“E.g., we might have a constant called isactive… we set this to true.

const isActive = true;

if ()

function countTruthy(array) {

}

“When we use t his constant as the condition here, we get this hello message on the console”.

const isActive = true;

if (isActive) console.log('Hello');

function countTruthy(array) {

}

“But sometimes what we have in our *if* statement is not necessarily a Boolean value. It could be a string, it could be a number, it could be an object… so the JS engine tries to convert that value to something that is Like a Boolean but is Not a Boolean.”

“Let’s say we have a constant called name, and set it to the string Mosh.”

const isActive = true;

const name = 'Mosh';

if (isActive) console.log('Hello');

function countTruthy(array) {

}

“If I pass name here… obviously name is a string, it’s not a Boolean true or false. So, the JS engine tries to interpret this as truthy or falsy. When we run this, we still see the hello message, because we are still dealing with a string of characters. This is what we call a truthy value. It’s not a Boolean true… its truthy.

const isActive = true;

const name = 'Mosh';

if (name) console.log('Hello');

function countTruthy(array) {

}

“In contrast, if we had an empty string, JS would see this as falsy” “Now when we run this code, we no longer see the hello message.”

const isActive = true;

const name = '';

if (name) console.log('Hello');

function countTruthy(array) {

}

Here’s the list of falsy values in JS. “NaN is something I forgot to mention when I talked about all the falsy values in JS.”

* Undefined
* Null
* Empty string ‘’
* Boolean false
* 0
* NaN

“Your job is to create a function called count truthy that gets an array, and returns the number of truthy values in this array.”

“E.g., if I declare this array ((( const array = []))) and set it to [1, 2, 3]… so here we have 3 truthy values.

const array = [1, 2, 3];

function countTruthy(array) {

}

“We call count truthy, pass this array, and immediately pass this result to console.log” We get 3.

const array = [1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {

}

If you add zero, you still get three, because zero is a falsy value.

const array = [0, 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {

}

If you add null, undefined, empty string, you still get 3:

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {

}

Well, this is a tricky one. The console.log is exterior to the function. So, I have to somehow exclude falsy elements inside the array.

This doesn’t work:

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) { if (array = truthy)}

I mean… I could just run a number filter, so only numbers displayed? And also exclude 0?

First of all… how do we output anything?

This returns undefined.

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {}

This is no good

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {for (let array = [])

if [] = truthy}

Ok, I googled how to call an array with a function… and I got some strange results that aren’t quite applicable. Let’s see what I can figure out.

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {let i = array

return (i)

}

This is displayed on the console:

(7) [0, null, undefined, '', 1, 2, 3]

This returns false:

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {let i = array

return (i === typeof number)

}

This returns *[Array(7)]*.

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {let i = array

return ([i])

}

Ah… maybe I should be paying more attention to the error messages.  
*Identifier 'i' has already been declared*

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {let i = array

let i = (truthy || falsy;

}

Ok… this returns *[Array(7)]*

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {return([array])

}

Returns *(7) [0, null, undefined, '', 1, 2, 3]*

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {return (array)}

*NaN*

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {return (array - 0)}

Just returns number of elements in the array:

const array = [0, null, undefined, '', 1, 2, 3, 4];

console.log(countTruthy(array));

function countTruthy(array) {

if (typeof array !== 'number')

return array.length

OK, I give up.

Mosh says, *let count = 0;*

const array = [0, null, undefined, '', 1, 2, 3, 4];

console.log(countTruthy(array));

function countTruthy(array) {

    let count = 0;

}

Now we’re supposed to use a *for of* loop, to iterate through the array.

So… I was an idiot… I reviewed Arrays, but did not continue my word search to watch other array videos. Duuuuuuuuuuuurrrrrr. Duuuur.

Ah ha… so, I am pulling from the for of video, as *for in* is outmoded. Here’s is the code I pulled from:

const colors = ['red', 'green', 'blue', 3, 5];

for (let color if colors)

console.log(color);

I have come up with this, so far.

const colors = ['red', 'green', 'blue', 3, 5];

for (let color of colors)

if (typeof color == 'number')

console.log(color.length)

Time travel. OK, something like This is what I was looking for, though Mosh says this is bad. “You don’t want to have five conditions here for every falsy value” Also, I couldn’t make it work.

const array = [0, null, undefined, '', 1, 2, 3, 4];

console.log(countTruthy(array));

function countTruthy(array) { for

    let (count = 0;)

    for (let value of array)

    if (value !== false || value !== undefined)

    count++;

    return count;

“We simply pass that value here. Now that value might be a Boolean or a non Boolean. If it is Not a Boolean, the JS engine tries to interpret it as truthy or falsy. So if the **result** is truthy, then count will be incremented. Simple as that.

const array = [0, null, undefined, '', 1, 2, 3, 4];

console.log(countTruthy(array));

function countTruthy(array) {

    let count = 0;

    for (let value of array)

**if (value)**

    count++;

    return count;

}

Incidentally, I can’t get an output from the ‘wrong’ way:

const array = [0, null, undefined, '', 1, 2, 3, 4];

console.log(countTruthy(array));

function countTruthy(array) {

    let bob = 0;

    for (let value of array)

    if (value === 'number' || value !== false || value !== 0 || value !== null || value !== undefined || value !== '')

    bob++;

    return bob;

}

const array = [0, null, undefined, '', 1, 2, 3];

console.log(countTruthy(array));

function countTruthy(array) {

return (array.length,)

}

const array = [0, null, undefined, '', 1, 2, 3, 4];

console.log(countTruthy(array));

function countTruthy(array) {

    let count = 0;

}

const person = {

    name: 'Mosh',

    age: 30

};

for (let bob in person)

console.log(person[bob]);

const colors = ['red', 'green', 'blue', 3, 5];

for (let dave in colors)

console.log(colors[dave]);

for (let color of colors)

if (typeof color == 'string')

console.log(color)

for (let color if colors)

console.log(color);

I have come up with this, so far.

const colors = ['red', 'green', 'blue', 3, 5];

for (let color of colors)

if (typeof color == 'number')

console.log(color.length)

\*/

const array = [0, null, undefined, '', 1, 2, 3, 4];

console.log(countTruthy(array));

function countTruthy(array) {

    let bob = 0;

    for (let value of array)

    if (value === 'number' || value !== false || value !== 0 || value !== null || value !== undefined || value !== '')

    bob++;

    return bob;

}

## 04 - Control Flow - Exercise 7 - String Properties

Create a function called showProperties. Pass an object here… and this function show display all the properties of this object that are of type string.

function showProperties(obj) {

}

“For example, we define an object like movie, and use the object literal syntax to initialize this [[[curly braces]]]. Here’ we’re going to add a few properties like *title*. We set it to a simple string… that value doesn’t really matter. A is enough. ReleaseYear… we set it to a number, like 2018.

Rating… again, a number. And then director… we set it to another string.

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

function showProperties(obj) {

}

“Now if we call this function, and pass this movie object, see what we get… the string properties of our movie object… title is a, and director is b.”

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {

}

OK. Let’s hope we have a better time than last lesson.

We start out with regurgitating code from the object lesson. We predictably output the key value pairs, the properties of the movie object. (Freecodecamp says key and value together make a property).

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) { for (let key in movie)

    console.log(key, movie[key])

}

Letting key equal string just gives me the name/key, and undefined values:

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) { for (let key in movie)

    console.log(key, movie[key = 'string'])

}



Same result for typeof

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) { for (let key in movie)

    console.log(key, movie[key ==  typeof string])

}

This gives us Just the values:

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {

    for (let key in movie)

    if (key, movie[key])

    console.log(movie [key])

}

Ok… so we try dot notation:

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {  movie.title = 'pig';

    console.log(movie.title)

}

Interesting… we try loose equality operator, and the value *a* is displayed. Same thing happens with an empty string.

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {  movie.title == 'pig';

    console.log(movie.title)

}

Ah… the console.log returns the empty string for numbers.

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {  movie.title == '';

movie.releaseYear == ''

movie.rating == ''

movie.director == 'bob'

    console.log(movie.title,

        movie.releaseYear)

}

Ok, I pulled some new code from online… and it did allow me to just display the object values… in an array? Not very useful, but an interesting trick all the same:

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {

    console.log(Object.values(movie))

}



Ok, I give up. Mosh says to use the *for in* loop, to iterate over the properties of the object. So, in each iteration, this key will hold the name of one of **these** properties.

const movie = {

**title**: 'a',

**releaseYear**: 2018,

**rating**: 4.5,

**director**: 'b'

};

showProperties(movie);

function showProperties(obj) {

for (let key in obj)

}

Console.log of key:

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) { for (let key in obj)

    console.log(key);

}



“Now we need to get the value of each of these properties, and then check the type of that value. If the type is a string, then we’ll display the property and its value”.

“To get the value of this property, we use the bracket notation. So, object of key”

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {

    for (let key in obj)

    obj[key]

}

To check the type of this value, use the typeof operator…

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {

    for (let key in obj)

    typeof obj[key]

}

“And then put this in an if statement. So, if the type of object of key equals string, then, display the key, as well as object of key”

showProperties(movie);

function showProperties(obj) {

    for (let key in obj)

    if (typeof obj[key] === 'string')

    console.log(key, obj[key]);

}

Mosh says, why not use curly braces after the for statement. That’s because there’s only a single statement, under the for statement. “The console dot log statement is a single statement that belongs to our *if* statement.” But curly braces also work.

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) {

    for (let key in obj)

    if (typeof obj[key] === 'string')

    console.log(key, obj[key]);

}

const movie = {

    title: 'a',

    releaseYear: 2018,

    rating: 4.5,

    director: 'b'

};

showProperties(movie);

function showProperties(obj) { for (let key in movie)

   if (typeof key movie[key] === 'string')

   console.log(key, movie[key])

}

Well, I wasn’t super far off. I simply couldn’t get the syntax right. Also, obj can be called movie, and it still works. A couple of my attempts are below… though I think I got closer.

function showProperties(obj) { for (let key in movie) if (key, movie[key == 'string'])

unction showProperties(obj) {

    for (let key in movie) if (key, movie[key == 'string']) return console.log

}

   console.log(key, movie[key ==  typeof string])

## 04 - Control Flow - 17 - Exercise 8 - Sum of Multiples of 3 and 5

Mosh says this is another easy one. Thanks Mosh. You know, that hard one a while back wasn’t so hard. These ‘easy’ lessons kinda suck. You know… maybe you’re just a bad teacher… had you thought about that? Maybe you should incorporate some review lessons, before throwing us into the deep end, and demoralizing the slower members of the class like myself. Maybe… you’re just a Sadist. You came from Iran. I’m thinking… ex intelligence? Interrogator… Teacher. They’re almost the Same thing when you think about it. I’m thinking… you used coercion to extract information like secret codes… and now you’re laughing at all of us… as you still try to extract code from your students… code that aren’t in a position to give you. And you watch us sweat. You know… hell is real, Mosh. It’s full of pirates, and lawyers, and Assholes.

This function will return all the multiples of 3 and 5 from zero up to this limit.

console.log(sum(10));

function sum(limit) {

}

What are the multiples of 3 and 5 between zero and ten? Multiples of 3 are 3, 6 and 9. Multiples of 5, are 5 and 10… including the number that is passed as a limit. If you add all these numbers together… the result will be 33.

Ok, so we have to iterate through a number until we hit the limit… that’s easy enough, I guess. And then we have to do the same for 5. And then add the numbers together.

So… which loop is best to use?

Well… *Do while* loops seem to execute at least Once, not matter what. And *for in* and *for of* may be made specifically for objects and arrays, which we are not ((turn the timer back on!)) using.

The While loop is similar, except that the loop variable is outside the loop.

Alright, my usual outputting woes. Nothing but errors. Ah… first, is unexpected token ‘;’ at the end of the for line.

console.log(sum(10));

function sum(limit) {

for (let i = 0; i <= limit; i++;);

return

}

Ok, now we are undefined.

console.log(sum(10));

function sum(limit) {

for (let i = 0; i <= limit; i++);

}

So… do we avoid using magic numbers? Do we define a couple variables?

Ok… maybe we need an internal console.log, so we can at least see what we are outputting.

So… with an internal console.log, nothing posts, because there is nothing to call our function:

function sum(limit) {

for (let i = 0; i <= limit; i++);

console.log(i)

}

Uncaught ReferenceError: i is not defined

at sum (index.js:126:13)

at index.js:119:1  
So… does this mean that console.logs inside functions don’t work?

sum(10)

function sum(limit) {

for (let i = 0; i <= limit; i++);

console.log(i)

}

Ok, back to original code. Return 5, gives us 5 on the console.

console.log(sum(10));

function sum(limit) {

return 5

}

Ok… finally something on the console:

console.log(sum(10));

function sum(limit) {

    let i = 0;

    while ((i \* 3) <= limit) {console.log(i++)}

}

0

1

2

3

index.js:119 undefined

This is an infinite loop.

console.log(sum(10));

function sum(limit) {

    let i = 0;

    while ((i \* 3) <= limit) {console.log(i)}

Another infinite loop:

console.log(sum(10));

function sum(limit) { let i = 1;

while ((3 \* i) <= limit)

console.log(sum)

i++; }

Ok… defining sum returns sum. Can we make sum the sum of two separate statements?

console.log(sum(10));

function sum(limit) { sum = 22;

    return sum

}

Ok, this returns 27.

function sum(limit) {

    let fiver = 7

    let threes = 20

    let sum = (fiver + threes);

    return sum

}

OK… this takes me up to 99 on the console. But I don’t want to add 1, I want to add the sum of bob and bob.

for (let bob = (5 + 5); bob < 100; bob++)

console.log(bob);

OK. Instead of console.loging i, we logged the (i \*3) expression, and got 3, 6, 9 on the console.

console.log(sum(10));

function sum(limit) {

    for (let i = 1; (i \* 3) <= limit; i++)

    console.log((i \* 3))

}

Ok… added a second for statement, and got interesting results… I think the second i \* 5 statement, truncated the i \* 3 iterations.

console.log(sum(10));

function sum(limit) {

    for (let i = 1; (i \* 3) <= limit; i++)

    for (let i = 1; (i \* 5) <= limit; i++)

    console.log((i \* 3))



OK. This is another interesting result:

console.log(sum(10));

function sum(limit) {

    for (let i = 1; (i \* 3) <= limit; i++)

    for (let b = 1; (b \* 5) <= limit; b++)

    console.log((i \* 3) + (b \* 5))

}

3 and 5 make 8. 8 and 5 make 15. Next loop. 6 & 5 make 11. 11 and 5 make 16. Third loop 9 and 5 make 14. 14 and 5 make 19.



So, if we use logical or, we get 8 and 16 as the output:

console.log(sum(10));

function sum(limit) {

    for (let b = 1; (b \* 3) <= limit && (b \* 5) <= limit; b++)

    console.log((b \* 5) + (b \* 3))

Returning limit gives us:

console.log(sum(10));

function sum(limit) {

    for (let b = 1; (b \* 3) < limit; b++)

   console.log(limit)

}



OK… so I eventually gave up and googled the answer.

https://www.codegrepper.com/code-examples/javascript/For+loop+sum+in+javascript

*sp: output loops into a sum js*

The problem is, we need some kind of Container as we iterate through the numbers. The answer is to define a variable as zero before the *for* statement (the function doesn’t work if you define if after the *for* statement… because it becomes subservient to the *for* statement?), and then after the *for* statement, you redefine the variable as *variable equals variable plus* Second *variable*. Because it is *after* the *for* statement, perhaps the for statement is Acting upon it.

console.log(sum(10));

function sum(limit) {

let threez = 0;

    for (let b = 1; ((b \* 3) <= limit); b++)

   threez = threez + (b \* 3)

I still had some trouble incorporating the Second statement… they kept interacting with each other weirdly, even if I assigned separate variables. But it occurred to me that the same set of rules may apply. So, I defined the next variable, and vola.

console.log(sum(10));

function sum(limit) {

let threez = 0;

    for (let b = 1; ((b \* 3) <= limit); b++)

   threez = threez + (b \* 3)

   let fiver = 0;

   for (let i = 1; ((i \* 5) <= limit); i++)

   fiver = fiver + (i \* 5)

   let sum = (threez + fiver)

    return sum

}

After looking ahead at Mosh’s solution, I see a couple interesting things. If we remove the *x = x + y*, and simply output *x + y*, the console outputs to zero:

console.log(sum(10));

function sum(limit) {

let threez = 0;

    for (let b = 1; ((b \* 3) <= limit); b++)

    threez + (b \* 3)

   let fiver = 0;

   for (let i = 1; ((i \* 5) <= limit); i++)

    fiver + (i \* 5)

   let sum = (threez + fiver)

    return sum

}

But… if we output +=, aka the addition assignment operator (it adds the value of the Right operand to the variable), we can simplify the code.

console.log(sum(10));

function sum(limit) {

let threez = 0;

    for (let b = 1; ((b \* 3) <= limit); b++)

    threez += (b \* 3)

   let fiver = 0;

   for (let i = 1; ((i \* 5) <= limit); i++)

    fiver += (i \* 5)

   let sum = (threez + fiver)

    return sum

}

Let us see Mosh’s fancy solution.

Mosh starts with a for loop. Let i equal zero. So long as i is less than or equal to limit… increment i. Let’s see if we can run with that. Nope.

“Now we need to check to see if i is a multiple of 3 or 5.”

function sum(limit) {

    for (let i = 0; i <= limit; i++)

    if (i % 3 === 0 || i % 5 === 0)

}

“Now we need to take i and add it to our sum. Here we need to declare a variable, let sum, set it to zero”.

console.log(sum(10));

function sum(limit) {

let sum = 0;

    for (let i = 0; i <= limit; i++)

    if (i % 3 === 0 || i % 5 === 0)

}

“if i is a multiple of 3 or 5, we need to add 5 to sum. And finally, return sum”.

console.log(sum(10));

function sum(limit) {

let sum = 0;

    for (let i = 0; i <= limit; i++)

        if (i % 3 === 0 || i % 5 === 0)

            sum += i;

    return sum;

}

“Now pay attention to how I have formatted this code. I have added these extra vertical line breaks to separate the initialization, from the actual logic, from the return value.”

console.log(sum(10));

function sum(limit) {

    let sum = 0;

for (let i = 0; i <= limit; i++)

        if (i % 3 === 0 || i % 5 === 0)

            sum += i;

    return sum;

}

“If we didn’t have these vertical breaks, the code would look a little bit squashed… a little bit hard to read. As a best practice, it is always good to separate the last line… the return statement.”

“In this case, we can have all these lines together, because they are Highly related.”

console.log(sum(10));

function sum(limit) {

let sum = 0;

    for (let i = 0; i <= limit; i++)

        if (i % 3 === 0 || i % 5 === 0)

            sum += i;

    return sum;

}

“But some developers prefer to separate the initialization, so you could add a line break here as well.

“But what I want you to take away is that lines that are highly related should be together, and they should be separate from other lines”

console.log(sum(10));

function sum(limit) {

    let sum = 0;

for (let i = 0; i <= limit; i++)

        if (i % 3 === 0 || i % 5 === 0)

            sum += i;

    return sum;

}

Ah… an elegant solution. He just divided all the numbers under 10 by 3 or 5, and added it to a sum. He didn’t have to perform any multiplications.

## 04 - Control Flow - 18 - Exercise 09 - Grade - 6.32

We want to calculate the grade of a student. We pass the array to this function,

If the average is 1-59: F

60-69: D

70-79: C

80-89: B

90-100: A

“Spend 5 or 10 minutes on this exercise.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

}

This seems like a fine place for if else, or some kind of equivalent. Well, first up though, let’s see if we can get a sum.

This returns 210 on the console which is good.

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let average of marks)

    sum += average

    return sum

}

Ah… the next part Seems simple. Finding the average by counting and dividing does work. But of course, it needs to function automatically. So, we have to count the array, and then divide by it.   
  
Regardless, this does return correctly:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let average of marks)

    sum += average

    if (sum / 3 <= 59)

    {return 'D'}

}

This is a bit queer.

We can add sum + marks.length, and get 213 on the console:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    marks.length

    for (let average of marks)

    sum += average

    return sum + marks.length

}

But if we try to divide sum by marks.length, it doesn’t work. Maybe because it’s in parenthesis? E.g., let’s add sum plus marks.length… it returns undefined.

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    marks.length

    for (let average of marks)

    sum += average

    if ((sum + marks.length) <= 59)

    {return 'D'}

}

Huh… I think it was just some kind of syntax issue. Because when we take away the parenthesis, and add the equals back in, it returns just fine. Huh**. Wait, no it doesn’t**. This just returns Failed no matter what the input:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    marks.length

    for (let average of marks)

    sum += average

    if (sum += marks.length <= 59)

    {return 'FAILED!!!!!!!!!!!!!!!!!!!!!'}

    else if ((sum += marks.length > 60 )

    {return 'Dumb'}

}

So… this seems to work.

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    marks.length

    for (let average of marks)

    sum += average

    if (marks.length  <= 2)

    {return 'FAILED!!!!!!!!!!!!!!!!!!!!!'}

    else if (sum += marks.length > 60 )

    {return 'Dumb'}

}

This is getting weird. Ok… so return sum gives me 210:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    marks.length

    for (let average of marks)

    sum += average

    {return sum}

}

But adding sum to marks.lenbth gives me 211. What is being added? Ah… if the number is 3 or less… it doesn’t add 1. So, only marks.length is being compared. This must be some order of operations issue.

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    marks.length

    for (let average of marks)

    sum += average

    if (sum += marks.length <= 59)

    {return sum}

}

Ok… so sum / marks.length Does return 70…

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    marks.length

    for (let average of marks)

    sum += average

    {return sum / marks.length}

}

OK… so we are getting the same result when we just translate this to number. So, the if statement is the problem:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let average of marks)

    sum += average

    if ( (210 / 3) < 50)

    {return sum / marks.length}

}

OK… to activate the if statement, sum must be += to any number… zero. Huh. But then I can’t add numbers to sum inside brackets. This is a pickle.

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    marks.length

    for (let average of marks)

    sum += average

    if (sum <= 100)

    {return 'Bob'}

}

Ah… this seems to negate that previous statement?

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let average of marks)

    sum += average

    if (sum < 1000)

{return sum}

}

So… I was wondering of parenthesis acted as some kind of function call or something… regardless, using square brackets works… though I don’t know why. Returns bob, whereas 2 + 999 is undefined. [[[I can no longer repeat this. Devil if I know why]]]

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let average of marks)

    sum += average

    if ( [2 + 995]  < 1000)

{return 'Bob'}

}

OK… here is the first working solution:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let average of marks)

    sum += average

    if ( [sum / marks.length]  < 59)

    {return 'F'}

    else if (   [sum / marks.length] >= 60  && [sum / marks.length] <= 69  )

    {return 'D'}

    else if ( [sum / marks.length] >= 70 && [sum / marks.length] <= 79 )

    {return 'C'}

    else if (  [sum / marks.length] > 80 && [sum / marks.length] <= 89 )

    {return 'B'}

    {return 'A'}

}

Let’s see if we can shorten it, but somehow declaring sum /marks.length. Kitty suggested something like const points = (sum / marks.length). So, that gets us away from a let statement that didn’t work.

I ended up using points = (sum / marks.length). ((I later changed it to a let statement, which I swore didn’t work earlier. Huh)). Works:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let average of marks)

    sum += average

     points = (sum / marks.length)

    if ( [sum / marks.length]  < 59)

    {return 'F'}

    else if (   [sum / marks.length] >= 60  && [sum / marks.length] <= 69  )

    {return 'D'}

    else if ( [points] >= 70 && [points] <= 79 )

    {return 'C'}

    else if (  [sum / marks.length] > 80 && [sum / marks.length] <= 89 )

    {return 'B'}

    {return 'A'}

Here is the shortened version:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let average of marks)

    sum += average

    let points = (sum / marks.length)

    if ( points  < 59)

    {return 'F'}

    else if (points >= 60  && points <= 69  )

    {return 'D'}

    else if (points >= 70 && points <= 79 )

    {return 'C'}

    else if (points > 80 && points <= 89 )

    {return 'B'}

    {return 'A'}

}

Let’s hear Mosh’s version.

“Here first, we need to calculate the average mark. The calculate the average we need to declare a variable like sum and initialize it to zero”.

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

}

“Then we need to use the for of loop to iterate this array of marks and calculate the sum of all these marks. For… let mark of marks”.

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let mark of marks)

}

“We get Mark, and add it to sum. So sum plus equals mark. This is the total.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let mark of marks)

        sum += mark;

}

“Now to get the average, we need to divide some by the number of marks. That is marks.length.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let mark of marks)

        sum += mark;

    let average = sum / marks.length;

}

“One way to calculate the grade is like this. If average is greater than or equal to zero and average is less than or equal to 59, we’re going to return F.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let mark of marks)

        sum += mark;

    let average = sum / marks.length;

    if (average >= 0 && average <= 59) return 'F';

}

“But this code is a little bit ugly. There is so much noise in this code. Instead of having to conditioners along with a logical and, we can simplify this expression to something like this: if average is less than 60. Because the range for F is 0 to 59.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let mark of marks)

        sum += mark;

    let average = sum / marks.length;

    if (average < 60) return 'F';

}

“If average is less than 70, return ‘D’. Because if we get to this point that means average is at least 60 or higher. Now I’m just checking the upper limit. So the upper limit should be 69. So if the averages between 60 and 69, we return D”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let mark of marks)

        sum += mark;

    let average = sum / marks.length;

    if (average < 60) return 'F';

    if (average < 70) return 'D';

    if (average < 80) return 'C';

    if (average < 90) return 'B';

    return 'A';

}

“We can make this code even better. We can break this function into two smaller functions, each focusing on one thing. That is what we call single responsibility principle. I use a metaphor. Think of a restaurant; people who work in a restaurant have different roles. So the chef is in the kitchen responsible for cooking. The chef doesn’t come out and take orders. So he is responsible only for one thing. The same goes for the manager. The manager is responsible only for one thing; managing the restaurant at a high level.”

“We had the same concept in programming. So we should have functions that are small and are focused only on one thing.”

“In this function we are basically doing two different things. The first part of this function is all about calculating the averages of numbers in an array. In the second part is about mapping that average to a grade. So we can extract these few lines…”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    let sum = 0;

    for (let mark of marks)

        sum += mark;

    let average = sum / marks.length;

    if (average < 60) return 'F';

    if (average < 70) return 'D';

    if (average < 80) return 'C';

    if (average < 90) return 'B';

    return 'A';

}

“and put them in a separate function. Function calculate average. We give it an array. This array can be anything. It can be an array of marks or it can be an array of temperatures… anything. We can reuse this function in the future whatever we need to calculate the average of a bunch of numbers.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    if (average < 60) return 'F';

    if (average < 70) return 'D';

    if (average < 80) return 'C';

    if (average < 90) return 'B';

    return 'A';

}

    function calculateGrade(marks) {

        let sum = 0;

        for (let mark of marks)

            sum += mark;

        let average = sum / marks.length;

}

“Here we’re working with marks, but the parameter of this function is called array. So, put the cursor here and press f2 to rename these two array. Also, we need to rename our loop variable from mark to value”:

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    if (average < 60) return 'F';

    if (average < 70) return 'D';

    if (average < 80) return 'C';

    if (average < 90) return 'B';

    return 'A';

}

 function calculateAverage(array)  {

        let sum = 0;

        for (let value of array)

            sum += value;

        let average = sum / array.length;

}

“So now we have a generic function. It’s not tied to marks. It can be reused in different situations. And finally, we can return the average here.”

Or, we can make this code even simpler. So, instead of storing the average in a variable, and then returning it, we can simply return the result of this expression.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    if (average < 60) return 'F';

    if (average < 70) return 'D';

    if (average < 80) return 'C';

    if (average < 90) return 'B';

    return 'A';

}

 function calculateAverage(array)  {

        let sum = 0;

        for (let value of array)

            sum += value;

        let average = sum / array.length;

        return average;

}

“So, return sum divided by array dot length.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    if (average < 60) return 'F';

    if (average < 70) return 'D';

    if (average < 80) return 'C';

    if (average < 90) return 'B';

    return 'A';

}

 function calculateAverage(array)  {

        let sum = 0;

        for (let value of array)

            sum += value;

        return sum / array.length;

}

“And finally we can reuse this function here. So, we declare a constant average, and set it to calculate average of marks. With this change we now have two very simple functions. Each function is responsible for only one thing.”

const marks = [80, 80, 50];

console.log(calculateGrade(marks));

function calculateGrade(marks) {

    const average = calculateAverage(marks);

    if (average < 60) return 'F';

    if (average < 70) return 'D';

    if (average < 80) return 'C';

    if (average < 90) return 'B';

    return 'A';

}

 function calculateAverage(array)  {

        let sum = 0;

        for (let value of array)

            sum += value;

        return sum / array.length;

}

## 04 - Control Flow - 19 - Exercise 10 - Stars - 2.33

showStars(10);

function showStars(rows) {

}

OK… so the first thing I try is using the truthy function to return a value. If i, iterate value, then log value.

This returns 10.

showStars(10);

function showStars(rows) {

    let count = 0;

    for (let i = 0; i <= rows; i++)

        if (i)

        count++

        console.log(count)

}

showStars(10);

function showStars(rows) {

    let count = 0;

    for (let i = 0; i <= rows; i++)

        if (i)

        console.log('\*')

        count++

        console.log(count)

}



Ok… this gives us 10 stars on the console:

showStars(10);

function showStars(rows) {

    let count = 0;

    let star = '\*';

    for (let i = 0; i <= rows; i++)

        if (i)

       star += '\*'

        console.log(star)

}



This yields a star for every iteration:

showStars(10);

function showStars(rows) {

    let count = 0;

    let star = '\*';

    for (let i = 0; i <= rows; i++)

        console.log(i, star)

}



Now we’re cooking with bacon!

showStars(10);

function showStars(rows) {

    let count = 0;

    let star = '\*';

    for (let i = 0; i <= rows; i++)

        console.log(i, star = star + '\*')

}



We make star = 0 instead of ‘\*’ and we get the correct star output.

showStars(10);

function showStars(rows) {

    let count = 0;

    let star = 0;

    for (let i = 0; i <= rows; i++)

        console.log(i, star = star + '\*')

}



OK, next we remove *let star = 0;* and just change it to an empty string. And we use the += on console.log instead of *star = star + ‘\*’.*

showStars(10);

function showStars(rows) {

    let star = '';

    for (let i = 0; i <= rows; i++)

        console.log(i, star += '\*')

}



So, the string seems to be coming from down below, not from the star variable.

So… we did it. Here it is. I realized that if you log i, it will log numbers on the console until you reach your set limit. So, instead of posting i, we just piggy back on i’s output, and every time i outputs, we post star on console.log

showStars(20);

function showStars(rows) {

    let star = '';

    for (let i = 1; i <= rows; i++)

       if(i)

       console.log(star += '\*')

}

Ah… there’s an even simpler solution:

showStars(20);

function showStars(rows) {

    let star = '';

    for (let i = 1; i <= rows; i++)

       console.log(star += '\*')

}

Mosh’s solution seems more complicated. It utilized two separate functions. Every time function one iterates, the internal function iterates by counting the external function, and outputs a += to a value… which is then logged on the console.   
  
Perhaps this helps avoid performing complicated behavior in the console itself, or the code remaining intact even if you don’t output console.log.

“What we have here, is what we call a nested loop.” “You may see this pattern in a lot of algorithms.”

showStars(20);

function showStars(rows) {

    for (let row = 1; row <= rows; row++) {

     let pattern = '';

     for (let i = 0; i < row; i++)

        pattern += '\*';

        console.log(pattern);

    }

}

For my edification. Below utilizes a term, instead of some kind of term and operation in the console.log. When I put both *pattern* And *console.log* in parenthesis after the *for* statement, both get executed.

showStars(20);

function showStars(rows) {

     let pattern = '';

     for (let i = 0; i < rows; i++)

        (pattern += '\*',

        console.log(pattern, i));

}



In fact, all we Really need to do, is remove the parenthesis, but keep the comma after pattern statement, it works the same.

showStars(20);

function showStars(rows) {

  let pattern = '';

     for (let i = 0; i < rows; i++)

        pattern += '\*',

        console.log(pattern, i);

}

But… if we add a semicolon (or nothing) instead of the comma, the i is no longer defined at all.

showStars(20);

function showStars(rows) {

     let pattern = '';

     for (let i = 0; i < rows; i++)

        pattern += '\*';

        console.log(pattern, i);

}



Comma gives us this:

showStars(20);

function showStars(rows) {

     let pattern = '';

     for (let i = 0; i < rows; i++)

        pattern += '\*',

        console.log(pattern);

}



No comma gives us this:

showStars(20);

function showStars(rows) {

     let pattern = '';

     for (let i = 0; i < rows; i++)

        pattern += '\*'

        console.log(pattern);

}



## 04 - Control Flow - 20 - Exercise 11 - Prime Numbers - 8.24

Takes a limit, and shows all the prime numbers up to this limit. Prime numbers have a factor of one and itself.

showPrimes(20);

function showPrimes(limit) {

}

Ok, here’s a basic structure. Weird returns:

showPrimes(20);

function showPrimes(limit) {

for (let i = 1; i <= limit; i++)

if (limit % i)

console.log(i)

}



Ok, what do we need to do? We every number between 1 and limit be divided by every number less than or equal to that number.

showStars(20);

function showStars(rows) {

    for (let row = 1; row <= rows; row++) {

     let pattern = '';

     for (let i = 0; i < row; i++)

        pattern += '\*';

        console.log(pattern);

    }

}

This was no good:

showPrimes(20);

function showPrimes(limit) {

for (let i = 1; i <= limit; i++)

{for (let b = 2; b <= limit && b < i; b++)

 console.log(i);

}

}



Well… adding a colon makes sure it only returns once… oh, what happened to my %? Got to add that back in.

showPrimes(20);

function showPrimes(limit) {

for (let i = 1; i <= limit; i++)

{for (let b = 2; b <= limit && b < i; b++);

 console.log(i);

}

}

Ok, so this shows all non primes:

showPrimes(20);

function showPrimes(limit) {

for (let i = 1; i < limit; i++)

{for (let b = 2; b < i; b++)

    if (i % b === 0)

 console.log(i);

}

}

This is dumb… it would only display non primes, and it doesn’t load:

showPrimes(55);

function showPrimes(limit) {

for (let i = 1; i < limit; i++)

    {for (let b = 2; b < i; b++)

{for (let c = 2; c < b; c++)

    if (b \* c = i)

    console.log(i)

}

Maybe we can do this by counting factors.

OK… I gave up. I was correct in all my assumptions. It all boils down to how to say, “Not (i % b === 0)”. And that, turned out to be using a Boolean. You set a variable to true. And then you define false as (i % b === 0).

showPrimes(20);

function showPrimes(limit) {

for (let i = 1; i < limit; i++)

{for (let b = 2; b < i; b++)

    if (i % b === 0)

 console.log(i);

}

}

Mosh’s solution:

showPrimes(15);

function showPrimes(limit) {

    for (let i = 2; i <= limit; i++)

    }

For each number, we need to check to see if this number is prime or not. Check from 2 to the current number, which is i.

Mosh notes that using letters is confusing. So, using more meaningful variable names is helpful. We change i to number, and the other variable to factor. (Highlight i, press f2, change it to number).

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

    }

    }

The inner loop is for finding the factors of the number. “This number has a factor of a number other than one and itself. That’s why I have not included one in this loop, because every number can be divided by one. For the same reason I have not included the number itself in this loop [[[no equals]]], because again we know that each number can be divided by itself evenly.”

“In this **inner loop**, we want to see if we have a factor other than the one and itself. If we do, that number is not a prime number”

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

**for (let factor = 2; factor <** number; factor++)

        if (number % factor === 0)

    }

    }

“What we can do here is declare a variable… we can assume this number is a prime number. But if this condition is true, we need to set isPrime to false.”

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        let isPrime = true;

        for (let factor = 2; factor < number; factor++) {

        if (number % factor === 0)

        isPrime = false;

        }

    }

    }

“If we find a factor for this number, that means this number is not a prime number for there is really no need to continue running this loop. Because this is just wasting our computer’s processing power”.

“So, here we can add a code block. If we get to this point we can use break to jump out of this loop.”

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        let isPrime = true;

        for (let factor = 2; factor < number; factor++) {

        if (number % factor === 0) {

        isPrime = false;

        break;

        }

    }

    }

“When we get to this point, is prime is either true or false. If it is true, we log this number on the console”

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        let isPrime = true;

        for (let factor = 2; factor < number; factor++) {

        if (number % factor === 0) {

          isPrime = false;

          break;

          }

        }

if (isPrime) console.log(number);

    }

    }

Mosh states that the function can easily be split. I was a bit perplexed, so here’s his explanation.

“This logic does not necessarily have to be part of our showPrimes function.

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        let isPrime = true;

        for (let factor = 2; factor < number; factor++) {

        if (number % factor === 0) {

          isPrime = false;

          break;

          }

        }

if (isPrime) console.log(number);

    }

    }

“We can have a separate function called is prime that takes a number and simply tells us this number is prime or not. Then we can use this function inside of show prime’s function or somewhere else. It’s a reusable function; it’s a reusable piece of code.”

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        let isPrime = true;

        for (let factor = 2; factor < number; factor++) {

          if (number % factor === 0) {

            isPrime = false;

            break;

          }

        }

 if (isPrime) console.log(number);

    }

    }

    function isPrime(number) {

    }

­

“I’m going to extract all these lines… move them to our new function”.

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        if (isPrime) console.log(number);

    }

}

function isPrime(number) {

    let isPrime = true;

    for (let factor = 2; factor < number; factor++) {

      if (number % factor === 0) {

        isPrime = false;

        break;

      }

    }

}

“We can make this code cleaner and more elegant. If we get to this point, instead of breaking from the loop, we can simply return false. In other words there is no need to ~~break~~, and then ~~return isPrime;~~ “

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        if (isPrime) console.log(number);

    }

}

function isPrime(number) {

    let isPrime = true;

    for (let factor = 2; factor < number; factor++) {

      if (number % factor === 0) {

        isPrime = false;

~~break;~~ return false;

      }

    }

~~return isPrime~~

}

“We can do a shortcut here. If we get to this point, we simply return false. That means we don’t need to set ~~isPrime to false~~. We don’t even need to declare ~~is prime~~. We run this loop; we try to find a factor for this number. If we find a factor we immediately return false. Otherwise, if we get to this point, that means we didn’t find any factors for this number, so return true.

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        if (isPrime) console.log(number);

    }

}

function isPrime(number) {

~~let isPrime = true;~~

    for (let factor = 2; factor < number; factor++) {

      if (number % factor === 0) {

~~isPrime = false;~~

        return false;

      }

    }

    return true;

}

“Also, we can get rid of all these curly braces, because our for statement… our four loop has a single statement, and our if statement also has a single statement. So let’s get rid of all these extra curly braces, getting in the way, annoying me”

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        if (isPrime) console.log(number);

    }

}

function isPrime(number) {

    for (let factor = 2; factor < number; factor++) {

      if (number % factor === 0) {

        return false;

      }

    }

    return true;

}

“Finally, in showPrimes; here we have only a single loop. We are iterating the numbers from two to this limit. We simply call isPrime function and pass our number. If the number is prime, we log it on the console. This also means we don’t need these curly braces.”

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        if (isPrime(number)) console.log(number);

    }

}

function isPrime(number) {

    for (let factor = 2; factor < number; factor++)

      if (number % factor === 0)

        return false;

    return true;

}

“With these changes, we have two very simple functions. If you give this function to the dumbest person in the world they will understand it. Also we don’t have nested loops. Generally speaking nested loops are bit hard to understand. So whenever you have a nested loop it’s probably an indication that you can extract the logic in the inner loop and put it somewhere else in a different function.”

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++)

        if (isPrime(number)) console.log(number);

}

function isPrime(number) {

    for (let factor = 2; factor < number; factor++)

      if (number % factor === 0)

        return false;

    return true;

}

­­­­

Haven’t finished taking notes yet, but I thought I’d include this. Splitting this function was confusing to me. The findPrime function was originally isPrime. And value of the isPrime function was originally number. (Clearly, there’s no relation to the isPrime in the let statement. And we can reassign the number value in the second function).

The call to the findPrime function takes place in an *if* statement. In the *if* statement I wondered why the value for the findPrime function call couldn’t be zzz. But, of course, the *(findPrime(number))* is a call to and an argument to the findPrime function… we need to tell the findPrime function that its value is number, so the second function can divide number.

In a previous lesson, the function call was performed as a constant. Here it is performed as an *if* statement.

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++)

 if (findPrime(number)) console.log(number);

    }

    function findPrime(zzz) {let isPrime = true;

        for (let factor = 2; factor < zzz; factor++)

          if (zzz % factor === 0)

            return false;

         return true;

        }

/\*

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++) {

        let isPrime = true;

        for (let factor = 2; factor < number; factor++) {

          if (number % factor === 0) {

            isPrime = false;

            break;

          }

        }

 if (isPrime) console.log(number);

    }

    }

\*/

showPrimes(15);

function showPrimes(limit) {

    for (let number = 2; number <= limit; number++)

        if (isPrime(number)) console.log(number);

}

function isPrime(number) {

    for (let factor = 2; factor < number; factor++)

      if (number % factor === 0)

        return false;

    return true;

}

## 05 - Objects - 01 - Basics - 4.53

“Objects are collections of key value pairs. If we have properties that are highly related, we want to encapsulate them inside an object.”

“Suppose we’re building an application for drawing different kinds of shapes. Like circles, rectangles, and so on. So, we could declare multiple variables around circles… e.g., let radius, we set that to 1. Then x, and y (((equal 1))). We’re defining multiple variables. That all these variables are highly related. They represent a circle.”

let radius = 1;

let x = 1;

let y = 1;

“A better approach is to put these variables inside an object. Now we can send that object anywhere in our program; we can pass that to any function. And all these variables will be available in that object.”

“I’ll define an object. We could use let or const. In this case, it doesn’t really matter. Let’s define a circle object using the object literal syntax.”

let radius = 1;

let x = 1;

let y = 1;

const circle = {};

“So, we add these curly braces and inside we add one or more key value pairs. The first key is radius and the value is one. Now this value we have here can be any type in JavaScript. It can be a number, a string, a Boolean, null, undefined; it can even be another object. Or an array, or a function.”

let radius = 1;

let x = 1;

let y = 1;

const circle = {

    radius: 1

};

“E.g. So, instead of defining two other key value pairs as X and Y, I’m going to add a key called location and set its value to another **object**. Now in this object we can have two ***keyvalue pairs, or properties.*** The first one is X and the second one is Y. We can also have another property here like isVisible, and set that to a Boolean, true or false. So, the purpose of an object is to group related variables.”

let radius = 1;

let x = 1;

let y = 1;

const circle = {

    radius: 1,

**location: {**

***x: 1,***

***y: 1***

    },

    isVisible: true

};

“But it’s not just grouping related variables. Quite often we have functions that should operate on these variables. E.g., we can have a function like draw for drawing a circle. Or we could have another function for moving a circle.”  
  
“These functions are highly related to these variables we have defined here. So, instead of defining these functions in a standalone way, it’s better to put these functions inside the circle object. With this, whenever we have a circle object inside our program, we have access to all its properties and functions.”

let radius = 1;

let x = 1;

let y = 1;

const circle = {

    radius: 1,

    location: {

        x: 1,

        y: 1

    },

    isVisible: true

};

function draw() {}

function move() {}

Let’s see how we can add this draw function inside the circle object. “We add another key value pair. The key is ‘draw’, and the value is a function. So, I told you the value of a key value pair can be anything in JavaScript. Here the value is a function. For simplicity, we’ll do a console.log here, and log ‘draw’ on the console.”

let radius = 1;

let x = 1;

let y = 1;

const circle = {

    radius: 1,

    location: {

        x: 1,

        y: 1

    },

    isVisible: true,

    draw: function() {

        console.log('draw');

    }

};

function draw() {}

function move() {}

“So, now with this circle object, we no longer need these independent variables and functions. All of these are now part of a circle object.”

~~let radius = 1;~~

~~let x = 1;~~

~~let y = 1;~~

const circle = {

    radius: 1,

    location: {

        x: 1,

        y: 1

    },

    isVisible: true,

    draw: function() {

        console.log('draw');

    }

};

~~function draw() {}~~

~~function move() {}~~

“So, we can access them using the dot notation. Circle. Look, all of them are here.”



“We can simply call the draw function like this. You see the draw message here.” (((Draw is displayed on the console))). “What you see here is what we refer to as object oriented style of programming, also abbreviated as **OOP. Object oriented programming** is basically a style of programming where we see a program as a collection of objects that talk to each other to perform some functionality.”

const circle = {

    radius: 1,

    location: {

        x: 1,

        y: 1

    },

    isVisible: true,

    draw: function() {

        console.log('draw');

    }

};

circle.draw();

“So here we have a circle object, and this object has a few properties, and a function. In object oriented programming terms, if a function is part of an object, we call that function a **method** ^^.”

…

“So, here more accurately, instead of saying were calling the draw function of the circle object, we say were calling the draw method of the circle object. That’s the difference between a function and a method. If a function is part of an object, in object oriented programming terms, we refer to that function as a method.”

const circle = {

    radius: 1,

    location: {

        x: 1,

        y: 1

    },

    isVisible: true,

    draw: function() {

        console.log('draw');

    }

};

circle.draw();

“Using this object literal syntax is an easy way to create an object. But as our applications get more complex, we need a different way to create objects.

const circle = {

    radius: 1,

    location: {

        x: 1,

        y: 1

    },

    isVisible: true,

    draw: function() {

        console.log('draw');

    }

};

circle.draw();

## 05 - Objects - 02 - Factory Functions - 5.45

“In the last lecture we learned how to use the object literal syntax to create an object. But there’s a problem.”

“Imagine we want to create two circle objects. So, let’s copy this code *here* and call this second circle object *circle2*. The problem we have here is that we have duplicated the implementation of the draw method. Right now, it’s a simple method, there’s only a single line of code. But what if we had 10 lines of code **here**. We wouldn’t want to repeat all these lines here, because if we had a bug in this method, then we have to fix it in multiple places.”

“Also take into account that our circle object currently has only a single method. What if we had 10 other methods here? We don’t want to duplicate or repeat all that logic. ***So, if our objects have logic, we need a different way to create objects***. That’s when we use factory or constructor functions.”

*const circle = {*

*radius: 1,*

*location: {*

*x: 1,*

*y: 1*

*},*

*isVisible: true,*

*draw: function() {*

*console.log('draw');*

*}*

*};*

const circle2 = {

    radius: 1,

    location: {

        x: 1,

        y: 1

    },

    isVisible: true,

**draw: function() {**

**console.log('draw');**

    }

};

“Just like a factory produces products, these factory functions produce objects.”

“We define a function and call it createCircle. Then we move this definition of the circle object inside of our factory function.”

function createCircle() {

    const circle = {

        radius: 1,

        location: {

            x: 1,

            y: 1

        },

        isVisible: true,

        draw: function() {

            console.log('draw');

        }

    };

}

“So, we have a circle object here. Finally, we need to return this. One way is to return it like this. But we don’t really need this circle constant defined, because we are not going to reference it anywhere. We only want to return it.”

function createCircle() {

    const circle = {

        radius: 1,

        location: {

            x: 1,

            y: 1

        },

        isVisible: true,

        draw: function() {

            console.log('draw');

        }

    };

    return circle;

}

“So, we can make this code shorter by removing the circle constant and simply returning this object. So, whenever we call the create circle function, we’ll get a circle object. However, we have hardcoded these values here, so every circle that is created by this function, its radius will be one. That’s not what we want. Maybe we want to have a circle with a bigger radius. So, we want to pass radius as a parameter here.”

function createCircle(………………………………………………..….) {

    return {

        radius: 1,

        location: {

            x: 1,

            y: 1

        },

        isVisible: true,

        draw: function() {

            console.log('draw');

        }

    };

}

“So, radius, and then, instead of hard coding 1, we want to set the value to this ‘radius’ argument that we supply it when calling this function.”

function createCircle(radius) {

    return {

        radius: radius,

        location: {

            x: 1,

            y: 1

        },

        isVisible: true,

        draw: function() {

            console.log('draw');

        }

    };

}

“Similarly, we can add another **parameter** here like location. And instead of hard coding this object here, we can simply set location to this location argument. (((Basically he’s saying, location is the location argument above))).

function createCircle(**radius, location)** {

    return {

        radius: radius,

        location: location, ~~{~~

~~x: 1,~~

~~y: 1~~

~~},~~

        isVisible: true,

        draw: function() {

            console.log('draw');

        }

    };

}

“However, for simplicity, we’re going to remove these two properties, *location* and *isVisible* so we can focus on the core of factory function. We don’t need the location parameter either.”

function createCircle(radius, ~~location~~) {

    return {

        radius: radius,

~~location: location,~~

~~isVisible: true,~~

        draw: function() {

            console.log('draw');

        }

    };

}

“We can make this code a little bit shorter. In modern JavaScript, if our key and value are the same, we can make our code shorter by removing the value and simply adding the key. So that is exactly equivalent to this code” (((radius: radius, = radius,)))"

function createCircle(radius) {

    return {

        radius~~: radius~~,

        draw: function() {

            console.log('draw');

        }

    };

}

“Next, we have our draw method. There is also a shorter syntax to define this method. So, instead of defining it as a key value pair, we can define it like this: Draw parenthesis and code block. This is similar to how we define a function outside of an object”  
…  
“So, if you wanted to define a function, this is how we define it, right? So, we have the function keyword, the name of the *function*, parenthesis, and code block”

…

“Now when we put this inside of an object, basically we are ~~dropping~~ the function keyword, and basically adding this inside of the object. You can see this syntax is a little bit shorter than what we have here.”

function createCircle(radius) {

    return {

        radius,

        draw() {

        }

        draw: function() {

            console.log('draw');

        }

    };

}

~~function~~ *draw*() {

}

“Let’s move our console.log statement into our draw method”

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

~~draw: function() {~~

~~}~~

    };

}

“So now we have a factory function. We can simply call this to create a circle object.”

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

“We’ll define this constant, circle1. And call the createCircle function, and pass 1 as the radius. Now, let’s log this circle object on the console… circle1”

function createCircle(radius, location) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

const circle1 = createCircle(1);

console.log(circle1);

“Radius is set to 1. We also have this draw method (((points to purple draw line)))”



“We can call circle1.draw(), and here’s our draw message on the console”.



“The beauty of this factory function is that we have defined our logic in one place. So, we can call this function with different values or different arguments, we get different circle objects, but we have defined the draw method only in one place. So, if there is a bug in this method that we need to fix in the future, there is a single place that we need to modify”.

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

const circle1 = createCircle(1);

console.log(circle1);

“So, now we’ll create another circle object; circle2, create circle, and pass 2 as the radius. Let’s log this on the console… circle2. Save the changes. Look, we have two different circle objects (((refers to console))) and a single definition of the draw method”

const circle2 = createCircle(2);

console.log(circle1);

(((I note, that the point of creating consts that equal our function, is partially for a separation of concerns, so we are not passing values in our console.log maybe. We can load different values with different console.logs… but I imagine it gets confusing)))

function createCircle(radius, location) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

const circle1 = createCircle(1);

console.log(circle1);

const circle2 = createCircle(2);

console.log(circle2);

Durr… think I had a typo.



“This is all about factory functions. But factory functions are not the only way to create objects. We can also use constructor functions”.

## 05 - Objects - 03 - Constructor Functions - 5.48

Constructor functions are another pattern for creating objects. “Just like the factory function we’re going to create a function, and the job of this function is to construct or create an object.”

“However, the naming convention we use for constructor functions is different. The naming convention we have for factory functions is what we call camel notation. The first letter of the first word is lowercase. But the first letter of every word after is uppercase. This is what we call camel notation.” oneTwoThreeFour… i.e., camel humps.

“In contrast we have another notation that is called Pascal notation. In this notation the first letter of every word should be uppercase. OneTwoThreeFour. When naming constructor functions we should use Pascal notation by convention. Because that something that other JavaScript developers expect when they read your code”

“Note we don’t call this create circle”

//factory function

function createCircle(radius, location) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

function Circle

“Just like our factory function, here we need to add a parameter, radius. However, instead of returning an object, we’re going to use a different approach to initialize an object.”

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

function Circle(radius) {

}

“In JavaScript we have a keyword called ‘this’. This is a reference to the object that is executing this piece of code. Imagine for now that this references an empty object.

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

function Circle(radius) {

this

}

“You know that with dot notation we can access properties of an object, we can read a property, or we can set a property. So, on this new empty object, we want to add a property called radius and we set that to this radius argument that we receive here.”  
…  
“So, in JavaScript our objects are dynamic. Once we create them, we can always add additional properties or methods to them. So, here were adding a new property to an empty object.”

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

function Circle(radius) {

    this.radius = radius;

}

“Similarly, we’re going to use this approach to add a new draw method to this empty object. This, dot draw… we set this to a function. And in the body of this function, we simply do a console.log of ‘draw’”

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

“Finally, in order to create a circle object using this constructor function, we’re going to define a constant called circle and here we’ll use another key word… that is new circle and pass one is the radius.”

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

// Constructor Function

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const circle = new Circle(1);

What is happening here? When we use this new operator here, three things happen. This new operator creates an empty JavaScript object. Something like this: constant x set to an empty object *const x = {}*. That is happening under the hood, but you don’t see that. Next it will set this to point to this new ***empty*** object.”

“So, in this code, we have access to this new empty object. And we set this radius property as well as the draw method in this new object. Finally, this new operator will return ***this*** new object from this function. So, it looks like this. Return this. We don’t have to explicitly add this statement here in this function. This will happen under the hood.”

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

~~return this;~~

}

const circle = new Circle(1);

const x = *{}*

“To recap. When we use the new operator three things happen. This operator first creates an empty object. Then, it will set this to point to this object. And finally, it will return that object from this function”.

“So, what we get ***here*** is that new object, and we simply set circle to point to that object.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const circle = *new Circle(1);*

const x = {}

“Now let’s see the difference between factory and constructor functions. With factory functions, we create an object like this: constant my circle… here we call createCircle, and pass an argument. *const myCircl = createCircle(1);* So, with factory functions, we simply **call a function** and in this function we **return a new object**.” (((Build your function, and then name a new const, and then say it equals the function we built. Can reuse function forever. Also… we are simply **Returning the new object**))).

“In contrast, with constructor functions we use the new operator, and instead of returning an object, we use the this keyword. Also, in terms of naming convention, with constructor functions we use that the Pascal naming convention but in factory functions we use the camel notation.”

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

const myCircle = **createCircle(1);**

// Constructor Function

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const circle = ***new Circle(1);***

Which one should we use to create new objects? Both these patterns are equally good for creating new objects. The construction function pattern is familiar to developers who have some experience programming in languages like C# or Java. So, with *this* pattern, you can see we are creating a new circle.

Now, if you don’t have any experiences in languages like C# or Java, you might want to go for a factory function. There is really no difference between these two patterns. However, there are some discussions on line by some strongly opinionated developers comparing these two patterns. My suggestion to you is do not get hung up on these discussions they are just a waste of time. Pick one pattern and stick to that.

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

const myCircle = createCircle(1);

// Constructor Function

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const circle = new Circle(1);

//factory function

function createCircle(radius) {

    return {

        radius,

        draw() {

            console.log('draw');

        }

    };

}

const myCircle = createCircle(1);

// Constructor Function

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const circle = new Circle(1);

## 05 - Objects - 04 - Dynamic Nature of Objects - 2.03

“Objects in JavaScript are dynamic which means once you create them you can always add new properties or methods, or remove existing ones.”

“So, here we havea circle *object*with a single property that is radius.”

**const circle = {**

***radius*: 1**

**};**

console.log(circle);

“We can add another property like color and set it to yellow. circle.color = ‘yellow’; . Now when we log the circle on the console, we can see we have two properties. Color and radius.”

const circle = {

    radius: 1

};

circle.color = 'yellow';

console.log(circle);



“Similarly, we can add a new method here. Circle.draw. Set it to new function. *circle.draw = function() {} .* And now, when we log this on the console, we can see our circle object has three members. Two properties, color and radius. And one method.”

const circle = {

    radius: 1

};

circle.color = 'yellow';

circle.draw = function() {}

console.log(circle);



“We can also delete existing properties or methods. So before logging the circle, we can use the delete operator to delete a member from a circle object. That member can be a property or a method. So, delete *circle.color*. We can also delete the draw method. So, *delete circle.draw*. Now our circle object has only the radius property that we had at the time of creating this object.”

const circle = {

    radius: 1

};

circle.color = 'yellow';

circle.draw = function() {}

delete circle.color;

delete circle.draw;

console.log(circle);



“Now, something that some developers find confusing is that here we’ve use the const keyword to define a new constant. However, we have modified the circle object. We’ve added a property and then removed that property. So, what kind of constant is that?”

const circle = {

    radius: 1

};

circle.color = 'yellow';

circle.draw = function() {}

delete circle.color;

delete circle.draw;

console.log(circle);

“When we use constant here, that means we cannot reassign this variable. So, this **variable** is more accurately a constant. So, we cannot reset circle to new object.” (((The variable is a const… but the value is flexible))).

const circle = {

    radius: 1

};

circle.color = 'yellow';

circle.draw = function() {}

delete circle.color;

delete circle.draw;

console.log(circle);

“If we do that we get this error. *Assignment two constant variable*. So, we cannot reassign this **constant**, but we can always change the circle **object** by adding or removing properties. ”

const circle = {

    radius: 1

};

circle = {};

circle.color = 'yellow';

circle.draw = function() {}

delete circle.color;

delete circle.draw;

console.log(circle);



const circle = {

    radius: 1

};

circle = {};

circle.color = 'yellow';

circle.draw = function() {}

delete circle.color;

delete circle.draw;

console.log(circle);

## 05 - Objects - 05 - Constructor Property - 2.24

“Every object in JavaScript has a property called constructor. And that references the function that was used to construct or create an object.”

“So, here we have two objects: circle and another. Let’s look at their constructor property.”

…

“On the console, another.constructor. (((You have to push enter twice. Also, another is a Constructor Function. ))). As you see, this returns our circle function that we used to create this object.” (((E.g., .constructor shows the function under the hood))).

// Factory Function

function createCircle(radius) {

    return {

        radius,

        draw: function() {

            console.log('draw');

        }

    }

}

const circle = createCircle(1);

// Constructor Function

function Circle(radius) {

    this.radius = radius;

    this.draw = function () {

        console.log('draw');

    }

}

const another = new Circle(1);



(((Trying to figure out which Mosh is referring to throughout the lesson. Remember, the Console is case sensitive… which helps narrow things down. I also note that **uppercase** Circle.constructor for the Constructor Function returns the following:

***Circle.constructor***

***ƒ Function() { [native code] }***

I have verified this, by isolating the Constructor Function in its own VSCode window… this is Not pulling from the factory function.))).

function Circle(radius) {

    this.radius = radius;

    this.draw = function () {

        console.log('draw');

    }

}

const another = new Circle(1);

“Now let’s look at circle.constructor. (((The Factory Function))) What is this? Well, we can tell this is a function because here we have this blue f. As we can see, the first letter of this function is uppercase. (((Object))). So, this is a built-in constructor function in JavaScript. **When we create an object using the object literal syntax, internally the JavaScript engine uses this constructor function**.”

*circle.constructor*

*ƒ Object() { [native code] }*



// Factory Function

function createCircle(radius) {

    return {

        radius,

        draw: function() {

            console.log('draw');

        }

    }

}

const circle = createCircle(1);

// Constructor Function

function Circle(radius) {

    this.radius = radius;

    this.draw = function () {

        console.log('draw');

    }

}

const another = new Circle(1);

“Let me show you. Let’s temporarily delete all this code. Let’s define an object like this.”

let x = {};

“When we use this syntax, object literal, JavaScript engine will translate that to something like this: *Let X equals new object*.” ((( [***ƒ Object() { [native code] }***])))

let x = {};

// let x = new *Object()*;

“So, the circle object, we create it and return it from our factory function. (((We make an object in our function, and then use Return))). And because we use the object literal syntax, internally it was created using this object constructor function.”



***circle.constructor***

***ƒ Object() { [native code] }***

(((Factory functions are function that contain an object… above which is the return command. Use a function-call to *call* the function and *return* the object in the function… so, return plays a pretty big role. But the object can only be created utilizing the object literal syntax, and That triggers the use of the constructor function: ***ƒ Object() { [native code] }*** )))

[See f Function ()](#ffunction)

“In JavaScript we have a few other built-in constructors. For example, we have string, for creating strings.”

new String();

“But quite often we use string literals. Single quote or double quote or backtick. Using these literals is cleaner and simpler than using the constructor.

new String();  // '', "", ``

“We also have Boolean… But again, we don’t use this. We either use true or false. So, we refer to these as Boolean literals.”

new String();  // '', "", ``

new Boolean(); // true, false

“We also have number. But instead, we use number literals like 123 whatever.”

new String();  // '', "", ``

new Boolean(); // true, false

new Number(); // 1, 2, 3, ...

“This is what I want you to take away. Every object has a constructor property and that references the function that was used to create that object.”

*another.constructor*

*ƒ Circle(radius) {*

*this.radius = radius;*

*this.draw = function() {*

*console.log('draw');*

*}*

*}*

(((For Factory Function:

*circle.constructor*

*ƒ Object() { [native code] }*

*createCircle.constructor*

*ƒ Function() { [native code] }*

// Factory Function

function createCircle(radius) {

  return {

      radius,

      draw: function() {

          console.log('draw');

      }

  }

}

const circle = createCircle(1);

For Constructor Function:

*another.constructor*

*ƒ Circle(radius) {*

*this.radius = radius;*

*this.draw = function() {*

*console.log('draw');*

*}*

*}*

*Circle.constructor*

*ƒ Function() { [native code] }*

// Constructor Function

function Circle(radius) {

  this.radius = radius;

  this.draw = function() {

      console.log('draw');

  }

}

const another = new Circle(1);

/\*

new String();  // '', "", ``

new Boolean(); // true, false

new Number(); // 1, 2, 3, ...

\*/

// Factory Function

function createCircle(radius) {

    return {

        radius,

        draw: function() {

            console.log('draw');

        }

    };

}

const circle = createCircle(1);

// Constructor Function

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const another = new Circle(1);

## 05 - Objects - 06 - Functions are Objects - 4.47

“One of the confusing concepts in JavaScript is that here, functions are objects. So, this circle function we have here is actually an object.”

function Circle(radius) {

    this.radius = raidus;

    this.draw = function() {

        console.log('draw');

    }

}

const another = new Circle(1);

“See? Circle. These are all the members of the circle function or the circle object. The purple icons are methods, like call, bind, apply. And the blue icons are properties. So, let’s take a look at a few of these members on the console.”

function Circle(radius) {

    this.radius = raidus;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.

const another = new Circle(1);



“Here in the console, circle.name . That returns the name of this function. Circle.length returns the number of arguments.”



“Now, earlier in this last lecture I told you that every object in JavaScript has a constructor property and that references the function that was used to create an object. Here’s the interesting part. Who do we think created this object? Let’s have a look.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const another = new Circle(1);



“Here, we have another built-in constructor called function. When we declare function using this syntax, internally JavaScript engine will use this function constructor to create this object.”

***Circle.constructor***

***ƒ Function() { [native code] }***

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

const another = new Circle(1);

“Let me show you. We’ll define a constant called Circle one, to separate from our circle. Set it to new function. Here, our function requires one parameter. Radius. So, we add that here as a string. Now, as a second argument we’re going to pass the code inside of this function.”

function Circle(radius) {

    this.radius = raidus;

    this.draw = function() {

        console.log('draw');

    }

}

const Circle1 = new Function('radius')

const another = new Circle(1);

“So, to break it up into multiple lines, I’m going to use the backtick character… And simply copy all this code, and put it here.”

function Circle(radius) {

    this.radius = raidus;

    this.draw = function() {

        console.log('draw');

    }

}

const Circle1 = new Function('radius', `

this.radius = raidus;

this.draw = function() {

    console.log('draw');

}

`);

const another = new Circle(1);

**“So, when we declare a function, *internally* it’s represented like this**. Now, we can call this *circle one,* just like calling our circle function”

function Circle(radius) {

    this.radius = raidus;

    this.draw = function() {

        console.log('draw');

    }

}

const *Circle1* = new Function('radius', `

this.radius = raidus;

this.draw = function() {

    console.log('draw');

}

`);

const another = new Circle(1);

(((I.e., let x = constructor function)))

“We can create a circle object by newing up Circle one, and pass one as the radius”

function Circle(radius) {

    this.radius = raidus;

    this.draw = function() {

        console.log('draw');

    }

}

const Circle1 = new Function('radius', `

this.radius = raidus;

this.draw = function() {

    console.log('draw');

}

`)

const circle = new Circle1(1);

const another = new Circle(1);

“Save the changes, now in the console, lets log circle. Look, it’s a real circle object with these two members. (((Refers to draw and radius))).”



“Now let’s take a look at a couple of methods that are available in our functions. Circle. Here we have this call method.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.call

const another = new Circle(1);

“…And with this, we can call a function. Look at the arguments. The first argument is “this” argument. (((Points to this at the top. I assume this indicates this.radius?))). ”



“Here we need to pass an empty object, and *this* will reference this object that we pass here.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.call({})

const another = new Circle(1);

“After that, we add our arguments explicitly. So, here we have one argument… we pass 1. If we had multiple arguments, we pass them explicitly like this: 1, 2, 3, 4.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.call({}, 1)

const another = new Circle(1);

“So, ***this*** expression is exactly like the expression here. When we use the *new* operator, this *new* operator will internally create an empty object {} and pass that as the first *argument* {} to the call method (). And this object, will determine the context for this. So, this will reference this object.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

***Circle.call({}, 1)***

const another = new Circle(1);

“Earlier I told you if you don’t use the ~~new~~ operator, this by default, will point to the global object, which is window.” (((Word search does not turn up global object, but I think that means outside the function or object))) So, if I want to rewrite this expression in this way…

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.call({}, 1)

const another = ~~new~~ Circle(1);

“…instead of passing an empty object, I’d pass window. Of course, we don’t want to do this… I’m just explaining what happens under the hood.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.call(window, 1)

const another = new Circle(1);

“So, the first argument here specifies the target of this. Now, we have another method called *apply*. It’s exactly like the call method. With apply we can call a function, but instead of passing all the arguments explicitly (1, 2, 3, 4), we pass them in an array. So, this is useful if you already have an array somewhere else in your application, and you want to pass that array as the second argument to the apply method

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.call({}, 1);

Circle.apply({}, [1, 2, 3]);

const another = new Circle(1);

“So, this is the take away. In JavaScript, functions are objects.”

function Circle(radius) {

    this.radius = radius;

    this.draw = function() {

        console.log('draw');

    }

}

Circle.call({}, 1);

Circle.apply({}, [1, 2, 3]);

const another = new Circle(1);

## 05 - Objects - 07 - Value vs Reference Types - 5.49

In JavaScript, we have two categories of types. Value types, aka primitives. And reference types.

Primitive or Value Types

* Number
* String
* Boolean
* Symbol (New in es6)
* Undefined
* null

‘Reference Types

* Object
* Function
* Array

“In the last lecture we learned that functions are also objects. The same is true about arrays. In a nutshell, in JavaScript we have primitives and objects. In this lecture were going to see how primitives and objects behave differently, because this is absolutely important for you to understand before we move on to the next section where we talk about prototypes.”

“Let’s define two primitives. X & Y.”

let x = 10;

let y = x;

“Now, I’m going to change the value of x to 20. What I want you to note here is that X and Y are two independent variables. Save, and go back into the console. X is 20, Y is 10. They are independent. So, when we work with primitives, this value is stored inside of this variable. When we copy that variable, that value that is stored in the variable is copied into this new variable. So, they are completely independent of each other.”

let x = 10;

let y = x;

x = 20;



(((Note, that the last x above, does not have a *let* statement. Also, with this formula, Y equals 20:

let x = 10;

x = 20;

let y = x;

))).

“Now let’s see what happens if we use a reference type or an object here. So, I’m going to change this to an object that has a property called value. And then, instead of setting X to 20, we’ll set X.value to 20”

let x = { value: 10 };

let y = x;

x.value = 20;

When we log x and y on the console, “we can see the value property of Y is also 20.”



“So, this is the take away. When we use an object, that object is not stored in this variable. That object is stored somewhere else in the memory… And the address of that memory location is stored inside that variable.”

let x = { value: 10 };

let y = x;

x.value = 20;



“So, then when we copy X, into Y, it is the address or the reference that is copied. In other words, both X and Y are pointing to the same object in memory. And when we modify that object using either X or Y, the changes are immediately visible to the other variable.”

let x = { value: 10 };

let y = x;

x.value = 20;



“So, here’s the conclusion. Primitives are copied by their **value**. Objects are copied by their **reference**.”

“Let’s define a function called *increase* that takes a number. And here we simply increase the number by one. Let’s declare a number variable and set it to 10. And then call increase and pass this number.”

let number = 10;

function increase(number) {

    number++;

}

increase(number);

“If I log this number on the console, what do we think we’ll see? We see ten. But didn’t we increase the number? Well, when we call increase and pass this number variable, its value is copied into this parameter that is local in this function. So, this number variable here is completely independent of this Other number variable.   
  
“Here in this function, we increment number by one, so it will be 11. But, after this function, this number is going to go out of this scope. So, when we log this number on the console, we’re essentially dealing with this first number.”

“So, I told you, primitives are copied by their **value**. So, here we are dealing with two independent copies. That’s why we see 10 on the console.”

let number = 10;

function increase(number) {

    number++;

}

increase(number);

console.log(number);

“Now, let’s change this to a reference type or an object. So, I’m going to change 10 to an object that has a value property. Let’s rename this variable to object. And similarly, we’ll rename the parameter in this function to obj. And then increment object.value."

“Now, when we log this on the console, we’ll see 11. The reason for this is when we ***call*** increase, and pass this obj, this object is passed by it’s **reference**. (((Reference meaning address in memory))). So, this local parameter that we have here, will point to the same object that we define here. So, in this case we are not dealing with two independent copies. We have two variables that are pointing to the same object. So, any changes that we make to this object will be visible to the other variable.”  
{value: 11}

let obj = { value: 10 };

function increase(obj) {

    obj.value++;

}

***increase(obj);***

console.log(obj);

“In JavaScript we have value types, also called primitives as well as reference types which are objects. Our primitives are number string Boolean symbol undefined and null. Primitives or value types are copied by their value. Reference types are objects are copied by their reference.

/\*

let x = { value: 10 };

let y = x;

x.value = 20;

let number = 10;

function increase(number) {

    number++;

}

increase(number);

console.log(number);

\*/

let obj = { value: 10 };

function increase(obj) {

    obj.value++;

}

increase(obj);

console.log(obj);

## 05 - Objects - 08 - Enumerating Properties of an Object - 5.09

“So, here we have this circle object with the radius property and the draw method. We’ve learned about the *for in* and the *for of* loops to iterate over the properties of an object. In this lecture, we’ll review this one more time”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

“First, let’s take a look at *the for in* loop. For let key in circle. With this we can iterate over all the properties and methods of an object. Now we can log the key on the console. So, we have radius and draw.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key);



“If you want to get the value of a property, use the bracket notation here. So, as a second argument here we pass *circle of key*. So, we’re using the bracket notation to get the value of this key or this property. We can see the value of radius is one and the value of draw is a function. And here is the implementation of our draw function or draw method.” (((Referring to the console output for the function))).

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);



“We also have this new *for of* loop. *For let key of circle*. Simply log it on the console… We see an error. Circle is not iterable. The *for of* loop can only be used with iterables such as arrays and maps. An object is not iterable. So, we cannot iterate it using a for of loop.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let key of circle)

    console.log(key);



“However, we have this method object.keys of circle. With this, we can get all the keys in our circle object, and this will return an array. And since arrays are iterable, we can use the *for of* loop to iterate them. Save… we get radius and draw. Now, let’s take a closer look at this method here.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let key of Object.keys(circle))

    console.log(key);



“Earlier, we learned that this object is a built-in constructor function. So, somewhere we have this constructor function like this, and whenever we create an object using the object literal syntax… internally *that* is translated into a call to this constructor function.”

“So, when we create an object using the object literal syntax like this--value is 1--internally, that is translated to a call to this object constructor function. So, that looks like this… new object. Right?” (((Basically. X = { value: 1} really means X = new object() and value: 1 goes inside.)))

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let key of Object.keys(circle))

    console.log(key);

    function Object() {}

const x = { value: 1 }

    const x = new Object();

“Also, you learned, that all functions in JS are objects. So, they have properties and methods that we can access using the dot notation.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let key of Object.keys(circle))

    console.log(key);

    function Object() {}

“So, when we type object. … we can see all the properties and methods defined in this object.”

   Object.



“So, here we’re using the keys method, and this method returns a string array, which contains all the properties and methods in this object.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let key of Object.keys(circle))

    console.log(key);

   Object.keys()

“We have another similar method to object.keys. That other method is called entries. So, instead of returning the keys as a string array, it returns each key value pair as an array.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let key of Object.keys(circle))

    console.log(key);

for (let key of Object.entries(circle))

    console.log(key);

“Let’s rename key to entry” Use f2 “Save the changes.”

“Each entry is an array. The first element in the array is a key(((radius))), and the second element is the value (((1))). So, that is also another way to get access to all the properties and methods in an object.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let entry of Object.entries(circle))

    console.log(entry);



“Sometimes you want to see if a given object has a given property or method. To do that, you can use the in operator. E.g., if radius as a string in circle, we’ll display a message like yes. With the in operator, we can see if a given property exists in a given object.” We get yes on the console, unless we change radius to something like color. Does not log no, btw.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let entry of Object.keys(circle))

    console.log(entry);

    for (let entry of Object.entries(circle))

    console.log(entry);

    if ('radius' in circle) console.log('yes');



“The simplest way to enumerate the properties of an object is to use the for in loop. But we can also use the for of loop along with object.keys and object.entries. And finally, to see if given property or method exists in an object, we use the in operator.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let entry of Object.keys(circle))

    console.log(entry);

    for (let entry of Object.entries(circle))

    console.log(entry);

    if ('radius' in circle) console.log('yes');

/\*

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let key of circle)

    console.log(key);

\*/

const circle = {

    radius: 1,

    draw() {

        console.log('draw')

    }

};

for (let key in circle)

    console.log(key, circle[key]);

for (let entry of Object.keys(circle))

    console.log(entry);

    for (let entry of Object.entries(circle))

    console.log(entry);

    if ('radius' in circle) console.log('yes');

## 05 - Objects - 09 - Cloning an Object - 4.21

“In the last lecture we learned how to enumerate the properties of an object. Now, using this technique, we can get all the properties in an object and copy them into another object.”

“So, let’s say we want to create another circle object which is a copy of the circle object here. I’m going to call that another. … initially we’ll set it to an empty object. *Const another* *const another = {}*”

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

const another = {};

“Now we can use the for in loop to iterate over all the properties in this object, and copy them into this new object.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

const another = {};

“For let key in circle.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

const another = {};

for (let key in circle)

“Now we use the bracket notation to access a property with a given key. So, we set another of key to circle of key. So, basically here in the first iteration, key will be radius. So, this code will be equivalent to this: Another of radius. So, we’re sending the radius property of another object, to circle of radius.” (((Basically saying another’s key property… is the same as circle’s key property)))

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

const another = {};

for (let key in circle)

    another[key] = circle[key];

another['radius'] = circle['radius']

(((This is Not how we access both Key **And** Value in the for in loop lesson. We log both the key in person, and then person[key].

const person = {

    name: 'Mosh',

age: 30

};

for (let key in person)

console.log(key, person[key]);



)))

“Here on the right side of the assignment operator, we are reading the radius property of the circle… the value of this property is one. So, we get one, and put it in the radius property of the another object.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

const another = {};

for (let key in circle)

    another[key] = circle[key];

another['radius'] = circle['radius']

“Now let’s log another on the console. Our radius property is set to one, and here is our draw method.” (((So, I thought that bracket notation was only pulling value, but it also seems to be pulling the key))).

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

const another = {};

for (let key in circle)

    another[key] = circle[key];

console.log(another);



“So, if we call another.draw, everything works as we expect. Beautiful.”



“But this approach for copying or cloning an object is a little bit old. In modern JavaScript, we have better ways to achieve the same thing.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

const another = {};

for (let key in circle)

    another[key] = circle[key];

console.log(another);

“One way is to use the object.assign method. We have this object that we’ve seen before. This object has a method called assign. Here as the first argument, we can pass a target object which can be an empty object or an existing object.”

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

Object.assign({})

console.log(another);

“Then, we can pass one or more source objects. Let’s say, circle. What this method does, is it takes all the properties and methods in the source object, and then copies them into this new object, and finally returns the result here. (((See next window))).

const another = Object.assign({}, circle);

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

Object.assign({}, circle);

console.log(another);

“So, this line of code is exactly equivalent to these three lines.”

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

const another = Object.assign({}, circle);

console.log(another);

“If we save the changes, we get the exact same result”.



“Now, this target object that we pass here doesn’t have to be an empty object. It can be an existing object. It can have one or more properties or methods.”

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

const another = Object.assign({}, circle);

console.log(another);

“For example, let’s add a color property here. We set that to yellow. Now save the changes. This new object has let color property that we initially had here, as well as the members of the circle object. So, this is the object.assign method.”

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

const another = Object.assign({

    color: 'yellow'

}, circle);

console.log(another);



“Let’s see yet another, simpler, and more elegant way to clone an object. We can use the spread operator. So, *another*. Here we set this to a new object. And then use the spread operator, which is three dots to spread this circle object.”

“Basically, what this operator does, is that it takes all the properties and methods in this object and puts them here, in between these curly braces. So, this is the simplest way to clone an object. Save the changes… we get the exact same circle object as before”

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

// const another = Object.assign({}, circle);

const another = { ...circle };

console.log(another);



“So, here’s what I want you to take away. **Object.assign** copies the properties and methods from one or more source objects into a target object. And, we can use that to clone an object or, combine multiple objects into a single object.”

“And, the **spread operator** is used to spread an object which basically means getting all its properties and methods and putting them into another object.

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

const another = Object.assign({}, circle);

const another = { ...circle };

console.log(another);

/\*

const circle = {

    radius: 1,

    draw() {

        console.log('draw');

    }

};

const another = {};

for (let key in circle)

    another[key] = circle[key];

another['radius'] = circle['radius']

console.log(another);

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

const another = Object.assign({

    color: 'yellow'

}, circle);

console.log(another);

\*/

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

// const another = {};

// for (let key in circle)

//     another[key] = circle[key];

// const another = Object.assign({}, circle);

const another = { ...circle };

console.log(another);

## 05 - Objects - 11 - Math - 2.56

“Let’s take a look at a few of the built in objects in JS. The first one we’ll look at is the math object”

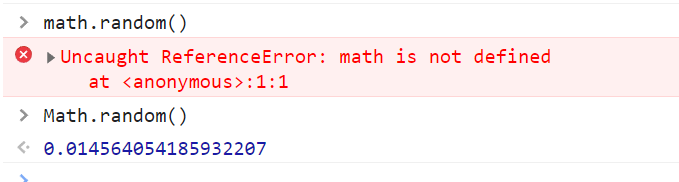
“Here in Google we’ll search for JavaScript math. The first link is developer.mozilla.org. This is your first reference for learning about all the objects in JS.”

The Math object is one of the built-in object that has properties and methods for mathematical constants and functions.

Scrolling down, you can see useful properties like like Math.PI. And then below that you have methods, like absolute, which returns the absolute value of a number. So, if you give it a negative number, it returns its positive value.”  
  
“All I want you to know is that we’ve got this math object built into JS. So, in your applications, if you ever have to deal with mathematical calculations, just come back to this page and see what methods we can use.”

“For each of these methods, we have comprehensive documentation. Along with examples… really, really easy.”

“So, here in the console, let’s call the random method. Every time we call this method, we get a new random number, between 0 and 1. If you want to map this to a given range, you need a simple mathematical formula.”



“Back to the documentation for the random method, if you scroll down the list of examples you can see this example. Getting a random number between two values. You need a function like this, that takes a min and max and generates a random number between these two values.”

function getRandomArbitrary(min, max) {

return Math.random() \* (max - min) + min;

}

“Another useful method is math.round. So, you can give it 1.9, and it returns 2.”  
  
“We also have Math.max. We can give it a bunch of arguments (((1, 2, 3, 4, 5))), and it will return the largest number in the list. Later in the course, this is going to be one of your exercises. So, you’re going to write a function like the max method here that takes a varying number of arguments, and returns the largest value.”  
  
“We also have Math.min(1, 2, 3, 4, 5)… which returns the smallest number in this list.”

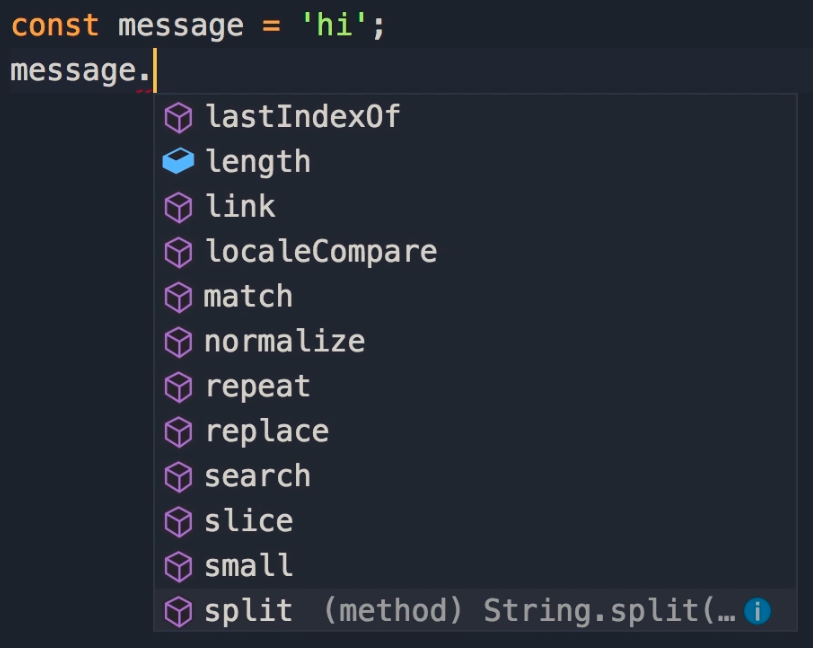
## 05 - Objects - 12 - String - 6.25

The next thing we are going to look at is the string object.

“I’m going to define a constant message, and set it to a string.”

const message = 'hi';

“Now, look at this. Message dot… what’s going on here? It looks like we have a bunch of properties and methods. But earlier in the course, I told you that string is a primitive type. Primitive types don’t have properties and methods. Only objects do. So why is it that we see these properties and methods in the string?”



“The reason for this is that in JavaScript we have two kinds of strings. This is what we call a string primitive”

// String primitive

const message = 'hi';

“But we also have a string object. So, we have this constructor function--string()”

// String primitive

const message = 'hi';

// String object

String()

“And we can use that to create a new string object. So, we can pass the same string here.”

// String primitive

const message = 'hi';

// String object

String('hi')

“Now, because this is a constructor function we need to apply the new operator...”

// String primitive

const message = 'hi';

// String object

new String('hi');

“And new we have another string.”

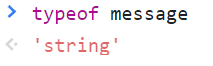
// String primitive

const message = 'hi';

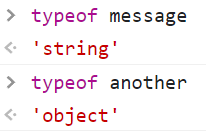
// String object

const another = new String('hi');

“Now let’s take a look at the type of each of these constants. So, *typeof* message. That’s a string”



“But, type of another is an object.”



“So, the first constant is a string primitive, the second one is an object. However, when we use the dot notation with a string primitive, JavaScript engine internally wraps this with a string object. We don’t see that… but we can work with this like a string object.”   
  
“Now just like the math object if you want to learn about all these methods, it’s best to look at the documentation. So, simply search for JavaScript string. Once again on developer.Mozilla.org. On this page, you can see all the properties and methods of the string object. In this lecture, I’m going to show you a few of these methods, but I strongly recommend you look at the documentation once. Just have a quick look to see what methods are there, in case you need them.”

“So, back to our code, let's change the string to this is my first message.”

// String primitive

const message = 'This is my first message';

// String object

const another = new String('hi');

(((on console))) “Here we have the length property, which returns the number of characters in a string.”

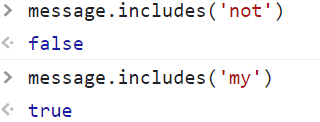


“This is particularly useful in situations where you want to make sure the user types at least a certain number of characters in an input field. Or, maybe you want to put a limit. You don’t want the user to type in more than 100 characters.”

“Now, if you want to access a character at a given index, you can use square brackets. So, message of zero returns T. Message of one returns h”



“If you want to see if the string has a special word, you can use the *includes method*. So, does this string have my? Yes it does… but it doesn’t have not. ”



“We also have another method, startsWith. So, this string startsWith this. But if you pass a t here, you get false. So, note that these searches are case sensitive.”



“We have a similar method: endsWith. So, message.endsWith(‘e’). So, we can see the last character here is e.”



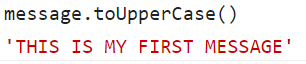
“If you want to find the index of a given character or a given string inside this string, you can use the indexOf method. So, what is the index of my? So, my starts at index eight.”



“We can also replace a part of the string. So, replace… Let’s say we want to replace first with second pretty easy. Note, that this returns a new string and does not modify the original one.”



“We also have a couple useful methods to uppercase. Message.toUpperCase()”



“Similar to this method we have to lowercase. Another useful method is trim. Let’s add a couple white spaces here, before and after our message.

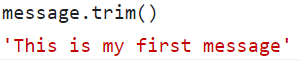
// String primitive

const message = '  This is my first message  ';

// String object

const another = new String('hi');

“Now, if we call the trim method, it gets rid of all the white space before and after our message.”



“And, of course this method has variations. For example, we have trim left, which only removes the white space at the beginning of the string. We have trim right. Etc”

“Another important concept you need to know in JavaScript is escape notation. (((Now called escape sequences))). So, if you look at the documentation for the string object, you can see in this table under escape notation, we have these special characters. So, if you want to use these, you need to encode them using the escape notation.”  
  
“For example, let’s say you want to have a single quote in your string. In this example we have defined our string with a single quote.”

// String primitive

const message = '  This is my first message  ';

// String object

const another = new String('hi');

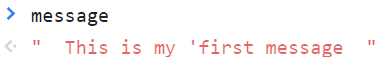
“So, if you want to have a single quote inside of the string… … You need to prefix it with a\. And now, this character is escaped, it’s encoded. So, when we log the message, you can see the single quote is actually part of the string”

// String primitive

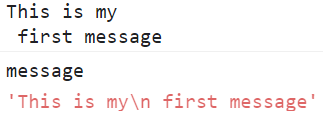
const message = '  This is my \'first message  ';

// String object

const another = new String('hi');



“Another useful escape character is \n which represents a new line. So, if we add a backslash n after my, this will add a new line.” (((For me, I can console.log this, but it won’t display on the console when typing message. Mosh’s does)))



“Another very useful method, is the split method. Message.split. With this, we can split a string based on a given character. So, here I’m going to pass a white space, and see what we get. *message.split(‘ ‘).* We get an array of five items, and each item in this array is a word in our message ”



/\*

\*/

// String primitive

const message = 'This is my\n first message';

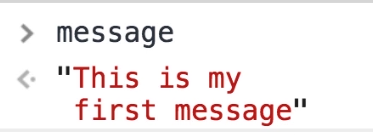
// String object

const another = new String('hi');

## 05 - Objects - 13 - Template Literals - 4.52

“In the last lecture, we learned that \n adds a new line to the string. Now, what I don’t like about this approach is that it does not allow us to visualize what the output looks like. In other words, if we log message on the console, we can see this is what the output looks like. So, on the first line we have ‘this is my’, and on the second line we have ‘first message’. But when writing code, our string looks different. (((I can console.log and get the below result, but I cannot enter message on the console, and have it display thus))).

const message = 'This is my\n first message';



“Now, if you want to make this string look similar do what the output looks like, we have to do something like this. So, we break this string into two parts and concatenate them like this:”

const message = 'This is my\n' + 'first message';

“And then we add the second part on a new line.”

const message = 'This is my\n' +

'first message';

“We can also put the first part on a new line and now this kinda looks similar do what we have in the output. But, we still have this \n getting in the way, creating some noise in the code.”

// String primitive

const message =

'This is my\n' +

'first message';

“To make the matter worse, we want to surround first we’ll have to escape the single quote character like this. Again, this is very, very ugly. Very, very ugly and noisy.”

// String primitive

const message =

'This is my\n' +

'\'first\' message';

“Here’s where template literals come into the picture. So far, we have learned about different kinds of literals in JavaScript. We have object literals that are indicated by curly braces. We have Boolean literals which are true or false. We have string literals that are indicated by single or “”. Now, starting from ES six we have template literals… Which are indicated by the backtick character.”

“This backtick character is the character before number one on your keyboard. So, let us see how template literals help us write cleaner code.”

// Object {}

// Boolean true, false

// String '', ""

// Template ``

“We’ll define another constant, but this time we’ll use a template literal instead of a string literal.”

// String primitive

const message =

'This is my\n' +

'\'first\' message';

// Object {}

// Boolean true, false

// String '', ""

// Template ``

const another =

“So, we add the backtick character. This is my… Now to add a new line we don’t need /n here. We can simply add a line break, and then type first message. So, we can format our string, the way we wanted to look like.”

// String primitive

const message =

'This is my\n' +

'\'first\' message';

// Object {}

// Boolean true, false

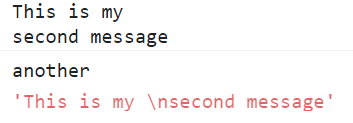
// String '', ""

// Template ``

const another = `This is my

first message`;

“Now, let’s log another… See? What we have in code looks very similar to what we have in the output.” (((Mine still displays the occult /n, which is apparently still hidden in the code, unless I console.log it. Mosh’s does not.))).



“We can also put this on a new line.”

// String primitive

const message =

'This is my\n' +

'\'first\' message';

// Object {}

// Boolean true, false

// String '', ""

// Template ``

const another =

`This is my

second message`;

“Now, if we want to surround first in quotes, we can simply add them here. There is no need to escape them… Because we have used a different character to define our string.”

// String primitive

const message =

'This is my\n' +

'\'first\' message';

// Object {}

// Boolean true, false

// String '', ""

// Template ``

const another =

`This is my

'second' message`;

“Compare this code, with the former code. This is particularly useful if you want to send out email messages in your application.”

// String primitive

const message =

'This is my\n' +

'\'first\' message';

// Object {}

// Boolean true, false

// String '', ""

// Template ``

const another =

`This is my

'second' message`;

“For example, let’s say you want to send an email to the user like this.

Hi John,

Thank you for joining my mailing list.

Regards,

Mosh

“So, we can format our string exactly the way we want to look like. There is no need for any of this plus or \n characters.”

// String primitive

const message =

'This is my\n' +

'\'first\' message';

// Object {}

// Boolean true, false

// String '', ""

// Template ``

const another =

`Hi John,

thank you for joining my mailing list.

Regards,

Mosh`;

“Another benefit of using template literals is that here we can add placeholders. So, let’s say we want to add the name dynamically. With a regular string, we have to do something like this.”

“Message, then set it to hi. Then we have to concatenate this with some variable or constant.”

const message = 'Hi '

const another =

`Hi John,

thank you for joining my mailing list.

Regards,

Mosh`;

“Let me define name here, set it to John. Now, will have to concatenate this message with name, then add a comma, then \n… This is really, really ugly.” (((my name is crossed out in JSCode)))

const name = 'John';

const message = 'Hi ' + name + ',\n';

const another =

`Hi John,

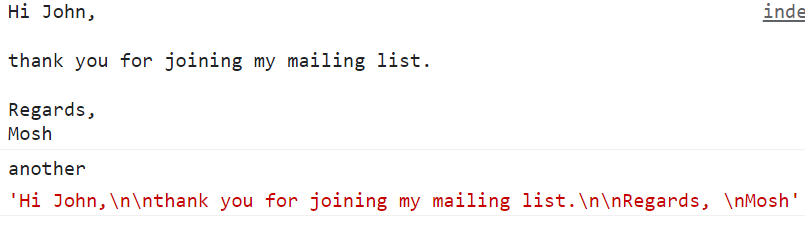
thank you for joining my mailing list.

Regards,

Mosh`;

console.log(another)

“Let me show you a better way. With template literals, we can add a placeholder using a $ and curly braces. Now, inside these curly braces we add an expression. We can pass the name of a variable or a constant. Now, let’s save the changes.” (((Note the console.log vs logging on the console gives different results.”)))



“So, this is a placeholder for our name constant. Now, here we can add any kind of expression. We can also add a simple mathematical expression, like this.”

const name = 'John';

const message = 'Hi ' + name + ',\n'

const another =

`Hi ${name} ${2 + 3}

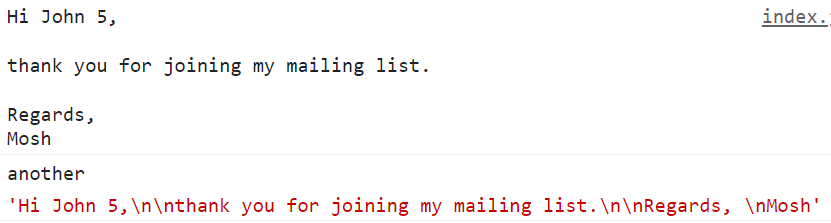
,

thank you for joining my mailing list.

Regards,

Mosh`;

console.log(another)



“So, technically we can add any expression that produces a value in between the curly braces. We can also call a function that returns a value, and that’s perfectly valid. So, these are the benefits of using template literals.”

const name = 'John';

const message = 'Hi ' + name + ',\n'

const another =

`Hi ${name} ${2 + 3},

thank you for joining my mailing list.

Regards,

Mosh`;

console.log(another)

const name = 'John';

const message = 'Hi ' + name + ',\n'

const another =

`Hi ${name} ${2 + 3},

thank you for joining my mailing list.

Regards,

Mosh`;

console.log(another)

## 05 - Objects - 14 - Date - 4.00

“The last built-in object we’re going to look at in this section is the date object. Just to clarify, the objects that we looked at in this section… They are not the only built-in objects in JavaScript. There’s more, and were going to learn about them as we go through the course.”

“Let’s see different ways to create a date object. We can create date one and set it to new date.. So date is instructor function.

const date1 = new Date()

“Now when open parenthesis here, you can see this number here… two of six. That shows different versions of this date constructor. You can use the up and down arrows to cycle through these. So, you can use the date constructor without any parameters and that returns the current date time.”

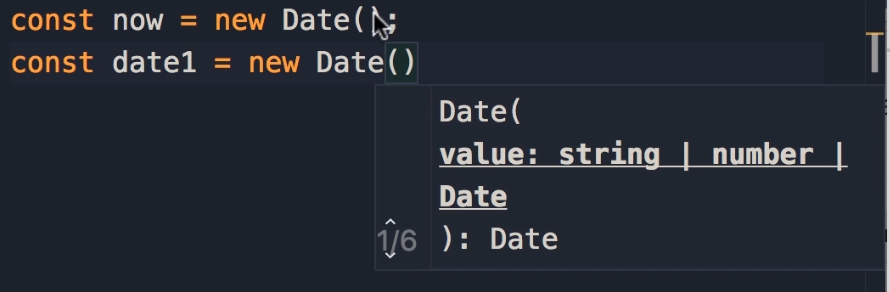


“

“So, let’s rename this to now.”

const now = new Date();

“Let’s create another date object, new date. We have another constructor with one parameter called value… Which can be a string or a number… Which is the number of milliseconds from January 1, 1970. Or, another date object.”



“Here I’m going to pass a string. Say, May 11, 2018. 9 AM. Now, this format is not the only format that you can pass here. If you want to find about all possible formats that are supported, simply search for JavaScript date” Under datestring you can read about various formats that are supported.

const now = new Date();

const date1 = new Date('May 11 2018 09:00');

“Finally, another way to create a date object is by passing numbers. So, here we have another constructor with year, which is a number, month, which is a number and so-on. So, will pass 2018. Now, the confusing thing about these date objects is that month is zero-based. So, zero represents January, and 11 represents December. So, we are going to use for for May. The third parameter is the date parameter, and that’s the day of the month. So, 11. Next parameter is hour, so 9 AM. And minutes is zero.”

“Now, we can exclude this argument because all the other arguments are initialize to zero by default.”

const now = new Date();

const date1 = new Date('May 11 2018 09:00');

const date2 = new Date(2018, 4, 11, 9, 0);

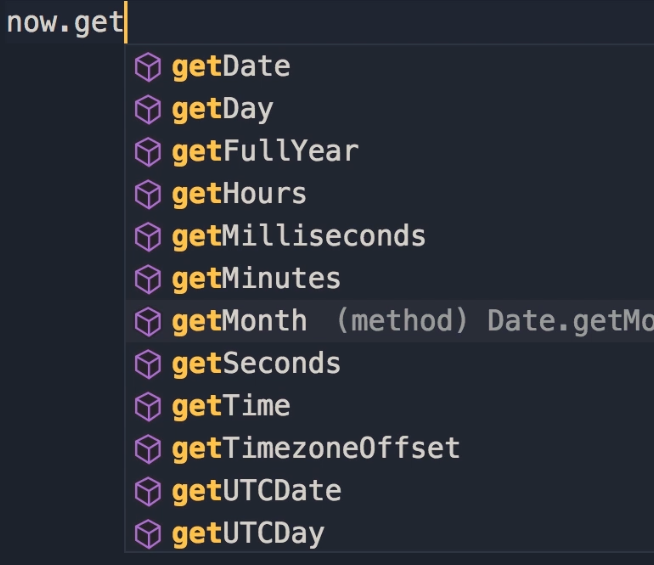
“So, this is how we create a date object. Now all these date objects have a bunch of get and set methods. For example, now.get. So, get date returns the day of the month, get hours, milliseconds, minutes, and so on”.

const now = new Date();

const date1 = new Date('May 11 2018 09:00')

const date2 = new Date(2018, 4, 11, 9, 0)

now.get



“We also have set methods. So, let’s call set full year, and change the year to 2017”

const now = new Date();

const date1 = new Date('May 11 2018 09:00')

const date2 = new Date(2018, 4, 11, 9, 0)

now.setFullYear(2017);

“We also have set date, hours, milliseconds, minutes, and so on.”

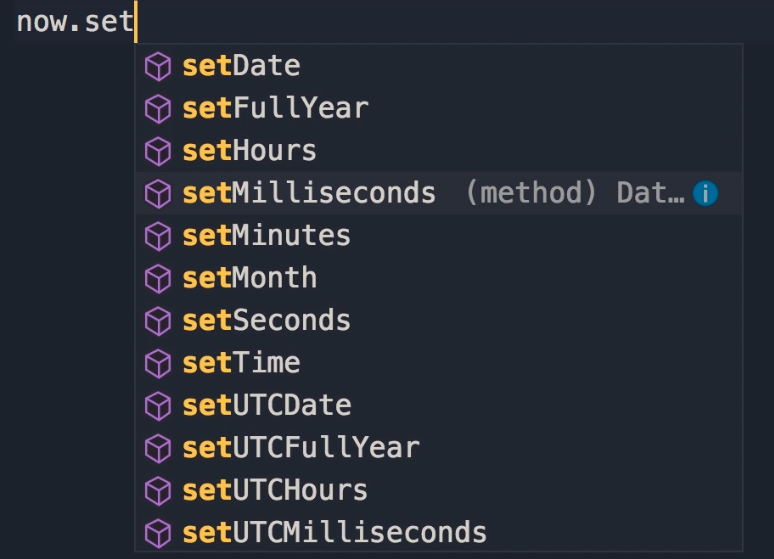
const now = new Date();

const date1 = new Date('May 11 2018 09:00')

const date2 = new Date(2018, 4, 11, 9, 0)

now.setFullYear(2017);

now.set

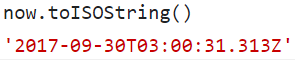


“Now, let’s see how we can display this on the console. All these date objects have a few methods for converting them to a string. We can call now.toDateString(). And we get a string like this.”  


“We also have now.toTimeString(), which returns the time component of this date object.”



“Another useful method is now.toISOString(), which returns a string like this. So, we have the date, then T, and time. This is a standard ISO format and is commonly used in web applications. So, if you’re building a web or a mobile app that talks to a backend, this is the format you commonly use to transfer date between the client and the server.”



## 05 - Object - 15 - Exercise 01 - Address Object - 1.41

“Here’s the first exercise for the section. We need to create an address object with three properties. Street, city, and zipcode. When you do that, then create a function called showAddress that takes an address object and displays all the properties in this object along with their value.”

Well, I had too many ideas as to what Mosh might want. I wasn’t sure what values to put into the address, and he just put a, b, c.

I thought maybe he wanted me to clone an object or something… turns out, all I had to do was put the object as an argument to my function. I saw this much. Let’s see Mosh’s solution.

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c',

}

function showAddress(address) {

 for (let key in address)

 console.log(key, address[key]);

}

console.log(showAddress(address))

“I’m going to create an address object. We can use the let or const in keyword. In this case it doesn’t really matter. Here were using the object literal syntax to initialize this object. ”

let address = {

};

“So, we have a street property. We set that to some value.”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c',

}

“Next we need to create this function. showAddress. That takes an address object”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c',

}

function showAddress(address) {

}

“And here to enumerate the properties of this object were going to use the for in loop. Let key in address. Then we do a console.log of key and address of key. So, here were using the bracket notation to get the value of the given key.”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c',

}

function showAddress(address) {

}

function showAddress(address) {

 for (let key in address)

 console.log(key, address[key]);

}

“And finally, we call the function showAddress, and pass our address object” (((Huh. Thought I tried this))).

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c',

}

function showAddress(address) {

 for (let key in address)

 console.log(key, address[key]);

}

showAddress(address);

/\*

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c',

}

function showAddress(address) {

 for (let key in address)

 console.log(key, address[key]);

}

console.log(showAddress(address))

\*/

// street

// city

// zipCode

// showAddress(address)

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c',

}

function showAddress(address) {

 for (let key in address)

 console.log(key, address[key]);

}

showAddress(address);

## 05 - Objects - 16 - Exercise 02 - Factory and Constructor Functions - 3.02

(((I got the order of operations wrong, and incorrectly labeled this #12)))

“So, and a last exercise we use the object literal syntax to initialize an address object. (((Nuh, uh))). In this exercise, I want you to initialize an address object using a factory function, and then using a constructor function.”

“So, your exercises to write two different functions. One is a factory function, the other is a constructor function to initialize an address object.”

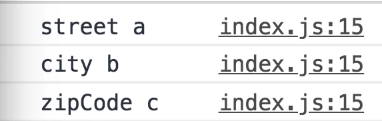
let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};



I did not observe in the video, that to the left on the console, the desired output was displayed. Durrr. This does simplify things a bit. Regardless, I’ll paste the code I was playing with in the beginning.

function map() {return {

    address: {

        street: 'a',

        city: 'b',

        zipCode: 'c'

    }

};

}

console.log(map())

1. *{address: {…}}*
   1. **address**: {street: 'a', city: 'b', zipCode: 'c'}
   2. [[Prototype]]: Object

I wonder if we could make the code above display in the Mosh format.

“Factory functions simply return a new object. Here we can call this function createAddress.”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

function createAddress()

“We should give it 3 parameters. Street, city, and zipCode.”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

function createAddress(street, city, zipCode) {

}

“Here we simply return a new object with three properties”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

function createAddress(street, city, zipCode) {

    return {

    }

}

“Street, which we set to the street argument we receive.”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

function createAddress(street, city, zipCode) {

    return {

        street: street

    }

}

“Now, as I told you before, if Key and Value are the same, we can make our code cleaner by dropping the value.”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

function createAddress(street, city, zipCode) {

    return {

        street~~: street~~

    }

}

“So, in this object we’re going to have these three properties. That’s it. So, that’s our factory function.”

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

function createAddress(street, city, zipCode) {

    return {

        street,

        city,

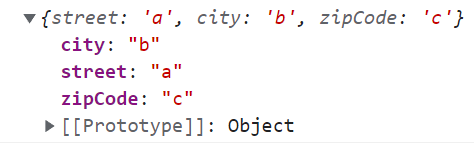
        zipCode

    };

}

“And, we can simply create a new address like this. So, instead of using the object literal syntax, we createAddress, and give it these three values: *let address = createAddress(‘a’, ‘b’, ‘c’)*”

“Now, log this address on the console to make sure everything is working. Here we have a new object, with these three properties.”



let address = createAddress('a', 'b', 'c');

console.log(address)

function createAddress(street, city, zipCode) {

    return {

        street,

        city,

        zipCode

    };

}

(((I note that the below works the same:)))

console.log(createAddress('a', 'b', 'c'))

function createAddress(street, city, zipCode) {

    return {

        street,

        city,

        zipCode

    };

}

Ok… given that there’s no exotica going on, I did the Constructor Function the same way:

function CreateAddress(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

const address = new CreateAddress('a', 'b', 'c');

console.log(address)

“Instead of camel notation that we use in factory functions, here we use Pascal notation.”

“Were going to call this function address. Again, were going to pass three parameters.

function Address(street, city, zipCode) {

}

“Now, here instead of returning an object, we’re going to use the this keyword to initialize this new object. This.street… we set that to the street argument. Then this . city. And finally this . zipCode”

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

“Now, to create a new address object using this constructor function, here, we use the *new* operator and call our constructor function. (((I believe the parenthesis was the constructor function… not sure))). Once again, we pass these values. (a,b,c,). And, log this address object on the console. ”

let address = new Address('a', 'b', 'c');

console.log(address);

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

“Save the changes, and we get the exact same object.”

“So, the purpose of this exercise, was to help you get familiar with the syntax for creating factory and constructor functions.”

/\*

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

function createAddress(street, city, zipCode) {

    return {

        street,

        city,

        zipCode

    };

}

let address = {

    street: 'a',

    city: 'b',

    zipCode: 'c'

};

function CreateAddress(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

const createAddress = new CreateAddress('a', 'b', 'c')

console.log(CreateAddress[street])

///this also works

console.log(createAddress('a', 'b', 'c'))

function createAddress(street, city, zipCode) {

    return {

        street,

        city,

        zipCode

    };

}

function CreateAddress(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

const address = new CreateAddress('a', 'b', 'c');

console.log(address)

\*/

///let address = createAddress('a', 'b', 'c');

let address = new Address('a', 'b', 'c');

console.log(address)

//factory function

function createAddress(street, city, zipCode) {

    return {

        street,

        city,

        zipCode

    };

}

//constructor function

function CreateAddress(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

## 05 - Object - 17 - Exercise - 03 - Object Equality - 3.59

“Were going to use the constructor function that we created in the last exercise, to create two address objects.”

let address = new Address('a', 'b', 'c');

//Constructor Function

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

“Here’s the first object. We’ll call this address1. Now, we’ll duplicate this line to create another address, which we’ll call address2.” (((alt shift))).

let address1 = new Address('a', 'b', 'c');

let address2 = new Address('a', 'b', 'c');

//Constructor Function

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

“Now, in this exercise we need to create two functions. One is called areEqual. So, we pass two address objects, address1 and address2. And we check to see if these two objects are equal. Now, as I told you before objects are reference types, so we can have two objects with the exact same properties… But these objects are in different memory locations.”

let address1 = new Address('a', 'b', 'c');

let address2 = new Address('a', 'b', 'c');

//Constructor Function

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

function areEqual(address1, address2) {

}

“So, when we use the equality operator, we are essentially checking to see if these objects have the same reference. Now, to properly check to see if two objects are equal we need to check the properties. We need to make sure that all their properties are equal. That’s the job of this function, areEqual. If all the properties of these two objects are equal, this function should return true, otherwise it should return false.”

“Now the other function I want you to write is areSame(address1, address2). This function simply tells us if address one and address two are pointing to the exact same object. In other words, they’re referencing the same object.”

(((Again, not really sure what the goal was, so I just threw up my hands))).

“We’ll start by implementing the second function because that’s easier. To check to see if two variables are referencing the same object, use the strict equality operator. Return address one equals address two. So, if they are referencing the same object, we’ll return true. Otherwise, we’ll return false.”

let address1 = new Address('a', 'b', 'c');

let address2 = new Address('a', 'b', 'c');

//Constructor Function

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

function areEqual(address1, address2)

function areSame(address1, address2) {

    return address1 === address2;

}

“Now, with the other function, we need to check individual properties. So, here we need to return a Boolean that is the result of evaluation of three conditional statements. Address1.street should be equal to address2.street. Here we use the logical and. Next we add address1.city… this should also equal address 2.city. Once again we use the logical and, and finally address1.zipCode should equal address2.zipcode”

let address1 = new Address('a', 'b', 'c');

let address2 = new Address('a', 'b', 'c');

//Constructor Function

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

function areEqual(address1, address2) {

    return address1.street = address2.street && address1.city === address2.city && address1.zipCode === address2.zipCode;

}

function areSame(address1, address2) {

    return address1 === address2;

}

“Now let’s call these functions and see the results. We’ll do a console.log and call are equal. Pass address1 and address 2. And then we’ll duplicate this and change areEqual to are same.”

“So, we can see that these two objects are equal. Their individual properties are equal. But they are not the same objects. They’re two different objects in memory.”

let address1 = new Address('a', 'b', 'c');

let address2 = new Address('a', 'b', 'c');

console.log(areEqual(address1, address2));

console.log(areSame(address1, address2));

//Constructor Function

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

function areEqual(address1, address2) {

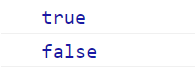
    return address1.street = address2.street && address1.city === address2.city && address1.zipCode === address2.zipCode;

}

function areSame(address1, address2) {

    return address1 === address2;

}



“In contrast, if we declare another variable here, let’s say address3, and we set that to address1. Now, both these variables are pointing or referencing the same object in memory.

let address1 = new Address('a', 'b', 'c');

let address2 = new Address('a', 'b', 'c');

let address3 = address1;

console.log(areEqual(address1, address2));

console.log(areSame(address1, address2));

//Constructor Function

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

function areEqual(address1, address2) {

    return address1.street = address2.street && address1.city === address2.city && address1.zipCode === address2.zipCode;

}

function areSame(address1, address2) {

    return address1 === address2;

}

“So, if we call areSame, and pass address1, and address3 were going to get true, because they are pointing to the same object in memory.”

let address1 = new Address('a', 'b', 'c');

let address2 = new Address('a', 'b', 'c');

let address3 = address1;

console.log(areEqual(address1, address2));

console.log(areSame(address1, address2));

console.log(areSame(address1, address3));

//Constructor Function

function Address(street, city, zipCode) {

    this.street = street;

    this.city = city;

    this.zipCode = zipCode;

}

function areEqual(address1, address2) {

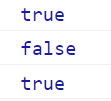
    return address1.street = address2.street && address1.city === address2.city && address1.zipCode === address2.zipCode;

}

function areSame(address1, address2) {

    return address1 === address2;

}



## 05 - Objects - 18 - Exercise 04 - Blog Post Object - 2.15

We are creating a blog post object with these properties.

// title

// body

// author

// views - represents the number of times this post has been viewed

// comments - each comment should have a couple properties

// (author, body)

// isLive - that can be either true or false.

“So, once again, you should use the object literal syntax to create and initialize a blogpost. Give each property some value… the actual value doesn’t really matter. I just want you to get comfortable with the object literal syntax.”

Again, not sure what Mosh is after. The only complexity I saw, was potentially putting an object in an object. I suppose some kind of placeholder is called for, for the viewcount, comments author, and commenter body. Seems like using the same value for property and the value (key and value) makes sense. Not going to overthink this.

let blog = {

    title: 'Revenge of the Nuns',

    body:

    `Four score, and seveny wimples ago,

    our sisters in Christ were massacred

    by vicious vampire tax attorneys.

    Now, vengence, nay Justice, will

    be ours!`,

    author: Mother Superior Charlemagne,

    views: viewcount,

    comments: {author: placeholder,

        body: body

    },

    isLive

}

“Here’s the solution. Let's create a post object, using the object literal syntax.”

let post = {

};

“We set the title to a. Body to b. Author to c. So, all these three properties are strings.”

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

};

“Now views should be a number. We can set that to 10.”

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

};

“Comments should be an array, because we can have multiple comments. So, we set that to an array.” (((Yes, I suppose I did only allow for one comment))).

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

    comments: []

};

“Now, each comment should have two properties. So, each comment should be an object. So here, once again, we use the object literal syntax to initialize a comment object.”

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

    comments: [

        { }

    ]

};

“We set the author to a, and body to b”.

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

    comments: [

        { author: 'a', body: 'b' }

    ]

};

“We can have another comment in this array. Let’s duplicate that, with different values.”

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

    comments: [

        { author: 'a', body: 'b' },

        { author: 'c', body: 'd' },

    ],

};

“And finally, we should add, the isLive property. So, isLive… we set that to true.” (((Haha, missing comma after array, and it would not load on the console)))

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

    comments: [

        { author: 'a', body: 'b' },

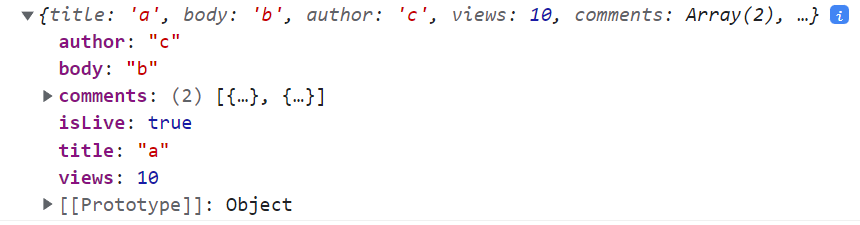
        { author: 'c', body: 'd' },

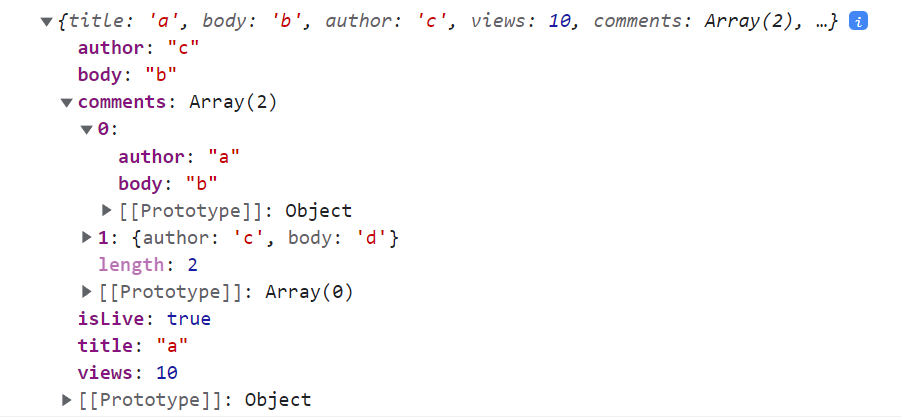
    ]

    isLive: true

};

“Let’s do a console.log of post. Inspect this on the console. So, we can see the comments property is an array with two objects. If we expand this (((comments))) we can see the properties of each object.”



“We’ve got author, and body”.

## 05 - Objects - 19 - Exercise - 05 - Constructor Function - 2.52

“In this exercise, once again we’re going to use a constructor function to create a post object. But this constructor function is a little bit different from the constructor function you wrote in an earlier exercise.”  
  
“So, here’s the scenario. Imagine we’re building a blogging engine. The user is drafting a post, but they have not published it yet. What do you think that constructor function should look like?”

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

    comments: [

        { author: 'a', body: 'b' },

        { author: 'c', body: 'd' },

    ],

    isLive: true

};

Using console.log, I realize I do not know how to even return an object. I seem to be only returning the function itself.

function Post('a', 'b', 'c',) {

    this.title;

    this.body;

    this.author;

};

Post.call({});

console.log(Post)

Peaking ahead. Ah, so this is a tale of two posts. If I post the function, I Get the function

function Post(title, body, author) {

  this.title = title;

  this.body = body;

  this.author = author;

};

let post = new Post('a', 'b', 'c');

*console.log(Post)*

*ƒ Post(title, body, author) {*

*this.title = title;*

*this.body = body;*

*this.author = author;*

*}*

But if I post a Call to the function (and pass values in this case) I **Run** the function. Duh. Posting the function posts the function, posting function calls Runs the function.

function Post(title, body, author) {

  this.title = title;

  this.body = body;

  this.author = author;

};

let post = new Post('a', 'b', 'c');

console.log(post)

*Post {title: 'a', body: 'b', author: 'c'}*

Also worth noting… I can pass values as an argument to a function like so:

let post = new Post('a', 'b', 'c');

But if I try to directly input those values as parameters, I just get an error. The parameters have to match the properties of the function.

function Post('a', 'b', 'c') {

  this.title = title;

  this.body = body;

  this.author = author;

};

let post = new Post('a', 'b', 'c');

console.log(post)

Let’s see if we can pass an argument to a function with the below method. And… I gave up. There.

function Post(title, body, author) {

  this.title = title;

  this.body = body;

  this.author = author;

};

Post.call({});

let post = Post('a');

console.log(Post.title)

Mosh:

“Let’s start, by creating this constructor function. We’re going to call that post… and note the capital P here.

function Post()

“Now, what parameters do we need here? Well, obviously we need the title, and the body, and the author… but we don’t need the number of views. Because the first time we create a post, views should be set to zero, and every time we view that post, we increment that value.

“So, views is not something we want to pass here. We want to use a default value of zero, when initializing this post object.”

“The same is true for the comments property. For a new post that is not been published yet, we don’t have any comments. So, we don’t want to add a comments parameter here. The same is true for the isLive property. We want to set this to false by default, so there is really no need to add an extra parameter in this function.

“So, you should aim to have functions with fewer parameters. The more parameters of function has, the harder it gets to use it.”

function Post(title, body, author)

“We’re going to set this to title to the title argument. The same is true for the body property and author.”

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

    comments: [

        { author: 'a', body: 'b' },

        { author: 'c', body: 'd' },

    ],

    isLive: true

};

function Post(title, body, author) {

    this.title = title;

    this.body = body;

    this.author = author;

}

“Now, for the views I’m going to initialize this to zero. Similarly, we’re going to initialize comments to an empty array. So, in the future we can add new objects to this array. And finally, we set is live to false.”

let post = {

    title: 'a',

    body: 'b',

    author: 'c',

    views: 10,

    comments: [

        { author: 'a', body: 'b' },

        { author: 'c', body: 'd' },

    ],

    isLive: true

};

function Post(title, body, author) {

    this.title = title;

    this.body = body;

    this.author = author;

    this.views = 0;

    this.comments = [];

    this.isLive = false;

}

“Now, instead of initializing this post object ~~here~~…”

let post = ~~{~~

~~title: 'a',~~

~~body: 'b',~~

~~author: 'c',~~

~~views: 10,~~

~~comments: [~~

~~{ author: 'a', body: 'b' },~~

~~{ author: 'c', body: 'd' },~~

~~],~~

~~isLive: true~~

~~};~~

function Post(title, body, author) {

    this.title = title;

    this.body = body;

    this.author = author;

    this.views = 0;

    this.comments = [];

    this.isLive = false;

}

“We are going to call our constructor function. Pass the title, body, and author. And finally, let’s log this on the console.”

“We can see author, body, and title properties are initialized based on the values that we passed here. But the other properties have default values. So, comments is initialized to an empty array. isLive, is false and views is zero.

let post = new Post('a', 'b', 'c')

console.log(post)

function Post(title, body, author) {

    this.title = title;

    this.body = body;

    this.author = author;

    this.views = 0;

    this.comments = [];

    this.isLive = false;

}



Old examples:

*“To recap. When we use the new operator three things happen. This operator first creates an empty object. Then, it will set this to point to this object. And finally, it will return that object from this function”.*

*“So, what we get* ***here*** *is that new object, and we simply set circle to point to that object.”*

*function Circle(radius) {*

*this.radius = radius;*

*this.draw = function() {*

*console.log('draw');*

*}*

*}*

*const circle = new Circle(1);*

*const x = {}*

*function Circle(radius) {*

*this.radius = radius;*

*this.draw = function() {*

*console.log('draw');*

*}*

*}*

*Circle.call({}, 1);*

*function createCircle(radius, location) {*

*return {*

*radius,*

*draw() {*

*console.log('draw');*

*}*

*};*

*}*

*const circle1 = createCircle(1);*

*console.log(circle1);*

*const circle2 = createCircle(2);*

*console.log(circle2);*

let post = new Post('a', 'b', 'c')

console.log(post)

function Post(title, body, author) {

    this.title = title;

    this.body = body;

    this.author = author;

    this.views = 0;

    this.comments = [];

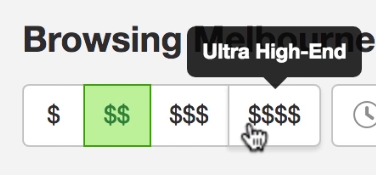
    this.isLive = false;

}

## 05 - Objects - 20 - Exercise 06 - Price Range Object - 3.37

“On yelp.com, we have these price range buttons. Inexpensive, moderate, pricey and so on. If we want to implement this concept using objects, what properties do you think those objects should have? That’s the focus of this exercise. I want you to create an array of objects. Each object is what we call a price range object. It’s one of these objects here.”

“Now here we have four price range objects. In this exercise, will create only three price range objects. Think about the properties that each object should have.”



Again, not bothering. Hmmm, maybe some more HTML and CSS experience would help here. So, looking ahead, we’re looking at some basic stuff here. We’re not looking at a price range, beyond the labels.

Mosh displayed as far as tooltip. We’ll see what he might have wanted.

let gizmo = [

    { label: '$', tooltip: 'Message', Color: Highlight, Box: size  },

    { label: '$', tooltip: 'Message', Color: Highlight, Box: size  },

    { label: '$', tooltip: 'Message', Color: Highlight, Box: size  }

]

“We’ll create an array called price ranges.”

let priceRanges = [

];

“In this array were going to have three objects. Each object should have four properties. One is the label that we display to the user. Like a $.

let priceRanges = [

    { label: '$'}

]

“The second property is a tooltip. So here, when I hover my mouse over each button, we can see a tooltip. So, for each price range object, we need this tooltip property. We’ll call this inexpensive.”

let priceRanges = [

    { label: '$', tooltip: 'Inexpensive', }

]

“Now, we need to more properties, and with this we can determine the minimum and maximum price of a meal per person. We can call them minPerPerson: 0, and maxPerPerson: 10. So, for this object I’m assuming that the average price of a meal per person falls between zero and $10.”

let priceRanges = [

    { label: '$', tooltip: 'Inexpensive', minPerPerson: 0, maxPerPerson: 10 }

“And this is important for filtering. Because if we’re building an application like yelp, somewhere we need to store the list of restaurants. So, we might have another array here let restaurants… And in this array we’re going to have restaurant objects. And each restaurant object should have this property… averagePerPerson. Let’s say the averagePerPerson is 5 dollars.”

let priceRanges = [

    { label: '$', tooltip: 'Inexpensive', minPerPerson: 0, maxPerPerson: 10 }

]

let restaurants = [

    { averagePerPerson: 5 }

]

“Now, when it comes to filtering let’s imagine the user only wanted to see inexpensive restaurants. So, this is the object that we are going to work with. And in this object we can see the price per person is between zero and $10.”

let priceRanges = [

    { label: '$', tooltip: 'Inexpensive', minPerPerson: 0, maxPerPerson: 10 }

]

let restaurants = [

    { averagePerPerson: 5 }

]

“Now, we want to filter the list the list of restaurants and pull out those where their average price per person is between these values. So, that’s the whole point of having these two properties.

let priceRanges = [

    { label: '$', tooltip: 'Inexpensive', minPerPerson: 0, maxPerPerson: 10 }

]

let restaurants = [

    { averagePerPerson: 5 }

]

“Similarly, we can create two more objects. We just need to change the values of these properties.

let priceRanges = [

    { label: '$', tooltip: 'Inexpensive', minPerPerson: 0, maxPerPerson: 10 }

    { label: '$$', tooltip: 'Moderate', minPerPerson: 11, maxPerPerson: 20 }

    { label: '$$$', tooltip: 'Expensive', minPerPerson: 21, maxPerPerson: 50 }

]

let restaurants = [

    { averagePerPerson: 5 }

]

## 06 - Arrays - 01 - Introduction - .30

In this section, will take a detailed look at arrays. We’re going to learn all kinds of operations that we can perform on arrays, such as:

* Adding new elements to an array
* Finding elements
* Removing elements
* Splitting arrays
* Combining arrays

“These operations are extremely important in programming, especially if you are starting out.”

## 06 - Arrays - 02 - Adding Elements - 3.35

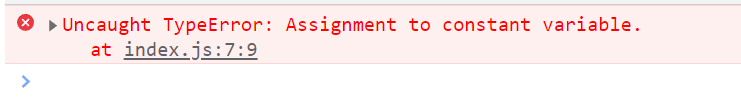
“Let’s declare a constant called numbers. And we set this to an array with two elements, 3 and 4.”

const numbers = [3, 4];

“Note that here we’ve declared numbers as a constant. That means we cannot reassign numbers to something else. If we do this, we get this error, assignment to constant variable. However, it’s perfectly fine to modify the content of this array. We can add new elements, or remove existing elements.”

const numbers = [3, 4];

numbers = [];

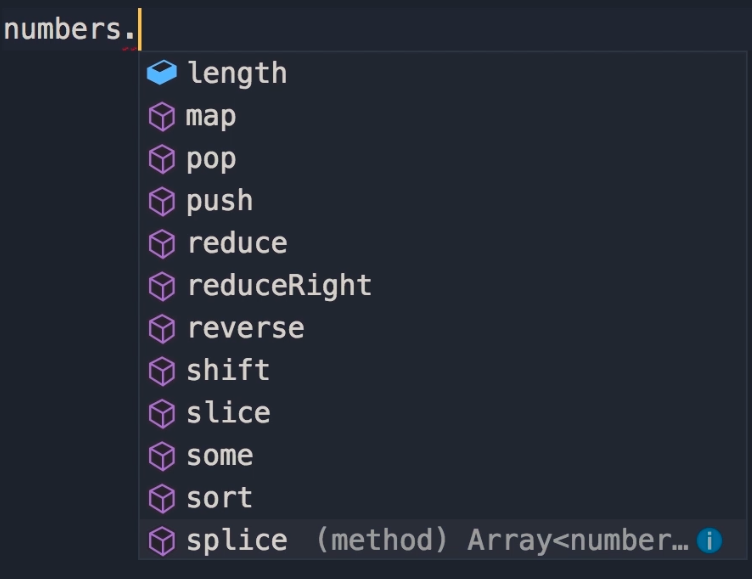


“So, constant does not stop us from modifying the content of an array.”

const numbers = [3, 4];

numbers = [];

“Earlier we have learned that arrays are objects, so using the dot notation, we can look at all the properties and methods defined in arrays. In this lecture, were going to look at three of these methods. To add new elements to the end, beginning, or middle of an array.”



const numbers = [3, 4];

// End

// Beginning

// Middle

“The first method we’re going to use is the push method. numbers.push() We can pass one or more arguments, and these arguments will be placed at the end of this array. So, let’s pass five and six, and do a console.log of numbers.”

const numbers = [3, 4];

// End

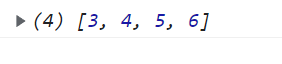
numbers.push(5, 6);

console.log(numbers);

// Beginning

// Middle

“We had three and four, now we have five and six at the end.”



“If you want to add elements to the beginning of an array, you use the unshift method. numbers.unshift… This basically pushes the elements in this array to the right and add new elements at the beginning. So, again here we can pass one or more arguments. We’re going to pass one and two.”

const numbers = [3, 4];

// End

numbers.push(5, 6);

console.log(numbers);

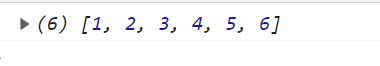
// Beginning

numbers.unshift(1, 2);

// Middle

I note, that if we log this on the console now, our changes don’t show up.

“Now, let’s bring this console.log down. To do this, we can hold down the alt key. Now we have one or two at the beginning of this array.”



“And finally, if you want to add elements to the end of an array you use the splice method. numbers.splice() . With this method we can go to a given position and add new elements or remove existing elements.”

const numbers = [3, 4];

// End

numbers.push(5, 6);

// Beginning

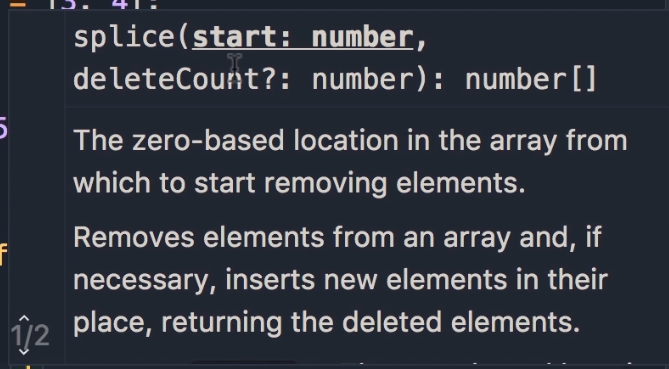
numbers.unshift(1, 2);

// Middle

numbers.splice()

console.log(numbers);

“Look at the parameters of this method. The first parameter is *start*, which is a number. So, that’s our starting position.”



“So, let’s say in this array, after one and two, between two and three, we want to add a new element. So, our starting position is here. What is the index of this element? Well, you know arrays are zero indexed.”

*(6) [1, 2, 3, 4, 5, 6]*

“So, our starting position, our starting index is two”

const numbers = [3, 4];

// End

numbers.push(5, 6);

// Beginning

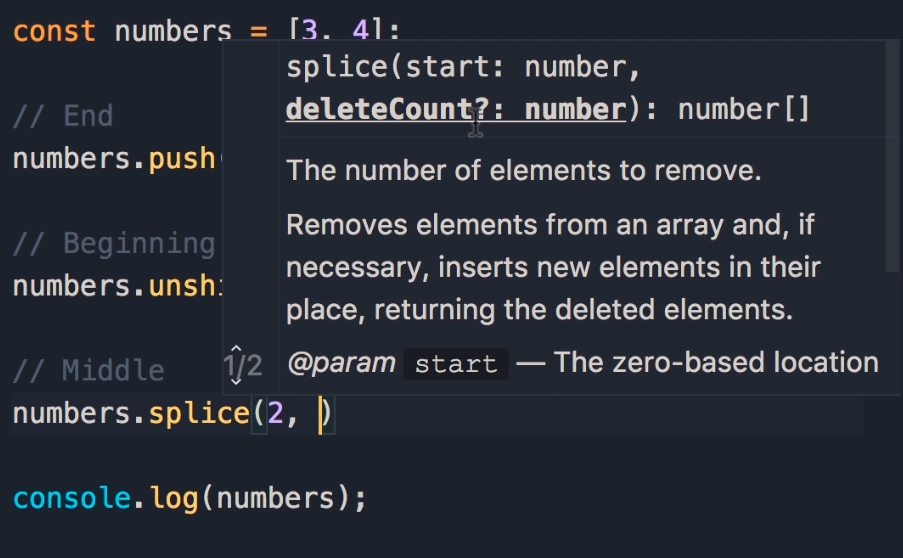
numbers.unshift(1, 2);

// Middle

numbers.splice(2)

console.log(numbers);

“Now, look at our second parameter here. That’s delete count, which is a number. How many elements do we want to delete? In this lecture, none.”



“So, we’ll pass zero.”

const numbers = [3, 4];

// End

numbers.push(5, 6);

// Beginning

numbers.unshift(1, 2);

// Middle

numbers.splice(2, 0, )

console.log(numbers);

“And then look at the third parameter. That’s the items we want to add. So, to make the stand out, we’re going to add to characters here. A and b.”

const numbers = [3, 4];

// End

numbers.push(5, 6);

// Beginning

numbers.unshift(1, 2);

// Middle

numbers.splice(2, 0, 'a', 'b');

console.log(numbers);

“Note that a and B have been placed after one and two.”

[1, 2, 'a', 'b', 3, 4, 5, 6]

const numbers = [3, 4];

// End

numbers.push(5, 6);

// Beginning

numbers.unshift(1, 2);

// Middle

numbers.splice(2, 0, 'a', 'b');

console.log(numbers);

## 06 - Arrays - 03 - Finding Elements (Primitives) - 3.32

“Let’s see how we can find elements in an array. Finding elements really depends on if we are storing primitives or reference types in an array. We’re going to start with primitives because they are easier, and then we’ll see how to find reference types in an array.”

“So, let’s say we have an array of numbers with four elements *const numbers = [1, 2, 3, 4];”*

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

“Here we have a method called indexOf.”

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

numbers.indexOf()

“We pass the element that we are looking for, and if that element exists in the array, this method will return the index of that element in the array. If it doesn’t exist, it will return -1. Let’s see a few different examples. ”

“First we will pass the character a. Obviously, we don’t have this element.”

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

numbers.indexOf('a')

“So, the result that we will see will be -1. So index of returns the index of the given element in this array. ”

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

console.log(numbers.indexOf('a'));

-1

“However, if we change this to one, we get zero because the index of this element is zero.”

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

console.log(numbers.indexOf(1));

0

“Note, that the type of this element matters. So, if I pass one as a string here, again we get -1, because we don’t have one as a string in this array. We have it as a number. ”

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

console.log(numbers.indexOf('1'));

“Similar to indexOf, we have another method called last index of. That will return the last index of the given element, or -1 if it does not exist. So, to demonstrate this, we’ll add another one here. And we’ll do another console.log. Numbers.lastIndexOf(1)

“So, the last index of one is three, because it’s here, in the index of this element is three.”

*-1*

*3*

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

console.log(numbers.indexOf('1'));

console.log(numbers.lastIndexOf(1));

“So, basically to see if a given element exists in an array, we can do something like this. Console.log(numbers.indexOf(1) is not -1). If this expression returns true, that means this element exists in this array. But this is a little bit ugly. We have a new method in JS for achieving the same thing. ”

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

console.log(numbers.indexOf('1'));

console.log(numbers.lastIndexOf(1));

console.log(numbers.indexOf(1) !== -1);

*-1*

*3*

*true*

“Console.log(numbers.includes(1)). This simply returns true if a given element exists in the array.”

*-1*

*3*

*true*

*true*

“All these methods have a second parameter which is optional, and that is the starting index. For example, with indexOf, let’s ~~change~~ this to one as a number. Save. We can see the index of 1 is zero.

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

console.log(numbers.indexOf(~~'~~1~~'~~));

console.log(numbers.lastIndexOf(1));

console.log(numbers.indexOf(1) !== -1);

console.log(numbers.includes(1));

*0*

*3*

*true*

*true*

“However, we can pass a second argument here that argument is called fromIndex. That’s the index at which the search will begin. So, we can pass 2 here. Let’s see what we get. We get 3, which is the index of the second one in this array.”

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

console.log(numbers.indexOf(1, 2));

console.log(numbers.lastIndexOf(1));

console.log(numbers.indexOf(1) !== -1);

console.log(numbers.includes(1));

*3*

*3*

*true*

*true*

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

console.log(numbers.indexOf(1, 2));

console.log(numbers.lastIndexOf(1));

console.log(numbers.indexOf(1) !== -1);

console.log(numbers.includes(1));

## 06 - Arrays - 04 - Finding Elements (Reference Types) - 5.46

“… Finding primitives is different than finding reference types. Let’s see why.”

“We’ll start by declaring a new array called courses. In this array, were going to have a couple of course objects.”

const courses = [

    {}

]

“Here’s the first one, with two properties, id and name.”

const courses = [

    { id: 1, name: 'a'}

]

“ … Now we can duplicate this line by holding down shift, alt and the down arrow.”

const courses = [

    { id: 1, name: 'a'}

    { id: 2, name: 'b'}

]

“Change the values to id 2, and name b. And don’t forget the semicolon”

const courses = [

    { id: 1, name: 'a'}

    { id: 2, name: 'b'}

];

“So, we have two course objects here. Let’s see if we have a course with the name a in this array. The includes method that you learned in the last lecture will not help us here. Let me show you.”

“courses.includes… here we pass the object that we’re looking for… course with id 1, and name a.”

const courses = [

    { id: 1, name: 'a'}

    { id: 2, name: 'b'}

];

courses.includes({ id: 1, name: 'a' })

“Now, let’s log this on the console.”

const courses = [

    { id: 1, name: 'a'}

    { id: 2, name: 'b'}

];

console.log(courses.includes({ id: 1, name: 'a' }));

Haha, this doesn’t load because we forgot to add our comma’s after each object… durr.

“We get false. The reason for this, is because these two objects… that is the object that we are passing to the includes method, and the object that we have in the courses array… These are two different objects. They have two different references. They are in two different locations in memory.”  
  
“And, in the last section we learn that objects are reference types; so, when we check them for their equality, their references are checked.” (((Ah, reference types… as in, they have reference numbers, instead of their data stored on site))). In this case, there are two different references. That is why we get false.”  
  
“So, if you have an array with reference types, you need to use the find method.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

console.log(courses.includes({ id: 1, name: 'a' }));

*false*

“We’ll ~~delete~~ this line” And now “courses.find() . Now, look at the parameter of this method. What we see here is way too complicated.”

const courses = [

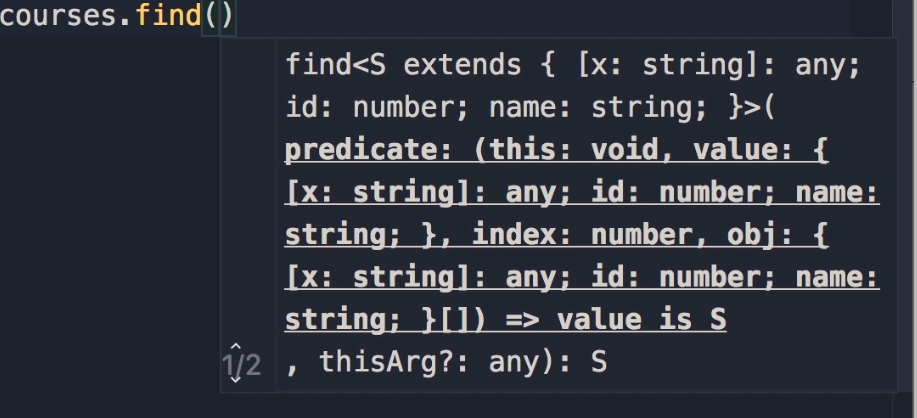
    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

courses.find()

c~~onsole.log(courses.includes({ id: 1, name: 'a' }));~~



“So, whenever you want to learn about an object or a function in JavaScript, simply search for that. ‘JavaScript array find’. (((MDN)))… So, that’s a good reference. Here, we can find a good description of what this method does. And, a very simple example of using that method.”

“So, here we have an array with a few numbers. We’re calling the find method. Note that as an argument to this method, we have to pass a function.” *(((Ah, here things deviate. In my version, a function is not required… the code is simpler. Ahh. But, I see the my code is labeled an “Arrow Function”. Still, I will show both versions, I guess.)))* “We call this function a predicate, and we use that to determine if the given element exists in an array or not.”

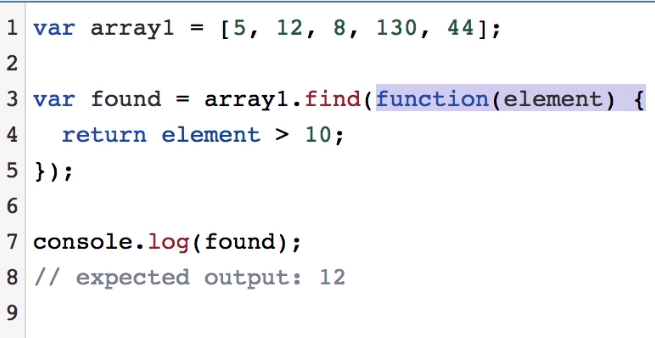
“So, this function takes a parameter that’s an element in this array. And, in the body of this function we should return a Boolean.”

*const array1 = [5, 12, 8, 130, 44];*

*const found = array1.find(element => element > 10);*

*console.log(found);*

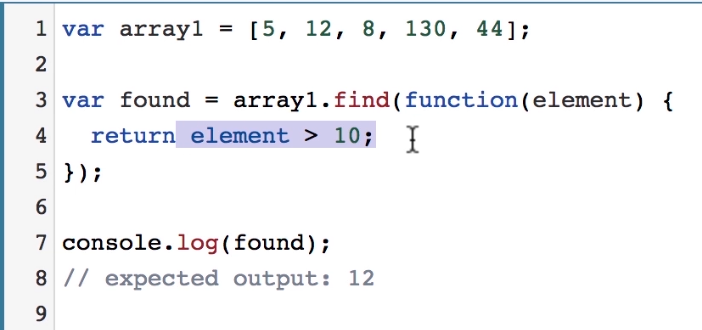
*// expected output: 12*



“So, here we have a simple expression. If this element is greater than 10, we’ll return true; otherwise we will return false. So, this is what happens when we call the find method and pass this function ((( *function(element)* ))). This function is executed once for the first element in this array. So, in this case five will be passed here ((( (*element*) )))… Five is not greater than 10, so we will return false, and the search will continue.”

“Now, this function ((( *function(element)* ))) will be called for the second element in this array. So, 12 will be passed as an argument to this function ((( (*element*) ))), and because it’s greater than 10, we will return true, and the search will stop here.”

“So now, what we’ll get here ((( found ))) as a result of calling the find method (((array1.find))), is the first element that Matches ((( element ))) that matches this criteria (((element > 10 ))). So, if we log this found variable on the console, we’ll get 12.”



“Now, in contrast, if there are no elements in this array that matches this criteria (((element > 10 ))), we’ll get undefined. Let me show you…”

“Back to our example, let’s say we want to see if we have a course with the name a in this array. So, we pass a function here. We call this a predicate or a call back function. Because this function is called back as part of finding an element in this array.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

courses.find(function())

“So, here we should have a parameter. We can call that element or course. Because each element in this array is a course object. So, it’s better to be more explicit; that makes her code more readable and understandable.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

courses.find(function(course))

“So, what is our criteria. We are looking for a course with the name equal to a.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

courses.find(function(course) {

    course.name === 'a';

}

“So, we simply return this…”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

courses.find(function(course) {

    return course.name === 'a';

})

“And, let’s store the result in a constant called course.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'a';

});

“Finally, let’s log this on the console. Here’s the course object.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'a';

});

console.log(course);

“Here’s the course object that we have in this array.”

1. *{id: 1, name: 'a'}*
   1. **id**: 1
   2. **name**: "a"
   3. [[Prototype]]: Object

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'a';

});

console.log(course);

“Now, if we change the search criteria do something like this, xyz. Obviously, we don’t have a course with this name. So, when we save the changes, we get undefined.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'xyz';

});

console.log(course);

“So, here’s our find method. It returns the first element that matches this criteria. We have a similar method called find index. It works exactly the same. But, instead of returning the actual object… like the course object, it will return its index.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'xyz';

});

console.log(course);

“We’ll change this to find index. In this case, we don’t have a course with this name, so when we save the changes we should see -1 on the console.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.findIndex(function(course) {

    return course.name === 'xyz';

});

console.log(course);

“If we change this back to a, and save the changes, we get zero, because the index of the first element that matches this criteria is zero.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.findIndex(function(course) {

    return course.name === 'a';

});

console.log(course);

/\*

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'a';

});

console.log(course);

\*/

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.findIndex(function(course) {

    return course.name === 'a';

});

console.log(course);

## 06 - Arrays - 05 - Arrow Function - 1.34

“In the last lecture we learned about this predicate function that we pass to the find method. In ES six, there is a shorter and cleaner way to write the same code using arrow functions.”

“So, whenever you want to pass a function, as a callback function or as an argument to a different method, you can use the arrow function syntax.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'a'

});

console.log(course);

“So, you remove the ~~function~~ keyword…”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(~~function~~(course) {

    return course.name === 'a'

});

console.log(course);

“…and to separate the parameters of this function from its body, put a fat arrow in between them. This is what we call an arrow function.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find((course) => {

    return course.name === 'a'

});

console.log(course);

“Now, if your function has a single parameter, we can also get rid of the parenthesis. And this makes your code cleaner; you’re removing the noise in the code.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(course => {

    return course.name === 'a'

});

console.log(course);

“If you don’t have any parameters, you have to pass an empty parenthesis.”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(() => {

    return course.name === 'a'

});

console.log(course);

“In this case, we Do have a single parameter, so we’ll bring that back. And finally, if your function is a single line of code, and it is returning a value… we can make this code even shorter.”  
  
“First, we get rid of the ~~return~~ keyword. And then, you remove the ~~curly~~ braces…”

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find((course) => ~~{~~

~~return~~ course.name === 'a'

~~}~~);

console.log(course);

“And finally, we can put everything on one line… so here’s the end result. We read this expression as course… goes to… course.name equals a. So, basically we are finding a course with a name equal to a.

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(course => course.name === 'a');

console.log(course);

/\*

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(function(course) {

    return course.name === 'a'

});

console.log(course);

\*/

const courses = [

    { id: 1, name: 'a'},

    { id: 2, name: 'b'},

];

const course = courses.find(course => course.name === 'a');

console.log(course);

## 06 - Arrays - 06 - Removing Elements - 3.03

“Let’s see how we can remove elements from an array. So, start by declaring a new array… numbers… and we set this to one, 2, 3, 4.”

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

“Earlier in this section, we learned about 3 methods to add new element to an array. We learned about the push method ((( numbers.push() ))), which we use to add something to the end of an array. We learned about the unshift method ((( numbers.unshift() ))) that we use for adding elements at the beginning of an array, and the splice method that we use for adding an element somewhere in the middle.”

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

numbers.push()

numbers.unshift()

numbers.splice()

numbers = [1, 2, 3, 4]

numbers.push()

numbers.unshift()

numbers.splice()

“Now, when it comes to removing an element… again, we can remove an element from the end, from the beginning, and from the middle.”

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

// End

numbers.push()

// Beginning

numbers.unshift()

// Middle

numbers.splice()

“But, instead of the push method, we use the pop method. This will remove the last element in this array, and return it. So, let’s store that in a constant called last. Here I’ll do a console.log of numbers so we can see the array after we pop the last element. And also, I’m going to show you this last element.”

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

// End

const last = numbers.pop();

console.log(numbers);

console.log(last);

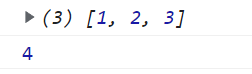
// Beginning

numbers.unshift()

// Middle

numbers.splice()

“Here’s our array. 1, 2, 3… and we can see that the last element is 4.”



“Now, if you want to remove an element from the beginning you use the shift method. Similar to the pop method, this will return the element that was removed. So, we can restore that in first.” And log on the console. “So, the first element is 1. Now, if we log numbers, we only have 2 and 3.”

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

// End

const last = numbers.pop();

// Beginning

const first = numbers.shift();

console.log(first);

// Middle

numbers.splice()

“And finally, the splice method that we learned about earlier. If you want to remove an element somewhere in the middle, you pass the index of that element. So, let’s say we want to remove the three here. The index of that element is two.” “

“So, we pass two here and as the second argument we pass the number of elements we want to delete. We can remove one element.” Console.log.

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

// End

const last = numbers.pop();

// Beginning

const first = numbers.shift();

// Middle

numbers.splice(2, 1);

console.log(numbers);

“Actually, the result is confusing… so let’s comment out these two lines, so we work with our original array.”

*(2) [2, 3]*

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

// End

// const last = numbers.pop();

// Beginning

// const first = numbers.shift();

// Middle

numbers.splice(2, 1);

console.log(numbers)

“So, note that three is removed. We have one, two, and four.”

(3) [1, 2, 4]

“Now, if you want to delete multiple elements, you pass a value greater than one. So, we can delete two elements starting from index two. So, three and four are removed, and we only have one and two.”

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

// End

// const last = numbers.pop();

// Beginning

// const first = numbers.shift();

// Middle

numbers.splice(2, 2);

console.log(numbers)

(2) [1, 2]

“So, to recap use the pop method to remove the last element. The shift method to remove the first element and the splice method to remove an element in the middle of an array.”

/\*

// End

const last = numbers.pop();

console.log(numbers);

console.log(last);

// Beginning

numbers.shift();

console.log(first);

// Middle

numbers.splice()

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

// End

const last = numbers.pop();

// Beginning

const first = numbers.shift();

console.log(first);

// Middle

numbers.splice()

\*/

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

// End

const last = numbers.pop();

// Beginning

const first = numbers.shift();

// Middle

numbers.splice(2, 2);

console.log(numbers)

## 06 - Arrays - 07 - Emptying an Array - 4.14

“Now we know how to remove an element from an array. But what about emptying an array? How can we remove all the elements in this array? Well, there are two different solutions. Here’s the first solution; we can simply reassign this to a new array.”

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

// Solution 1

numbers = [];

“Now, in this case because we have declared this array using const, we cannot do this. So, we have to change this to let…” Log on console. “Now our array is empty.”

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

// Solution 1

numbers = [];

console.log(numbers)

[]

“However, note that this old array is still in the memory, so if there are no other references to this object, eventually it will be removed by the garbage collector. So, this means if we have, let’s say another variable called another, and this is also pointing to the same object, this original array will not be garbage collected.”

“So, on line 6, when we reassign the numbers variable, no numbers is pointing to a new object in memory, but another is still pointing to the old object.”

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

let another = numbers;

// Solution 1

numbers = [];

console.log(numbers);

“Let me show you… So, here, instead of logging, instead of logging numbers, let’s log another. So, we can see another is still pointing to this array, to the original array. But if you look at the numbers… numbers is pointing to a new object, a new empty array. So, this solution works if you do not have any other references to the original array.”

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

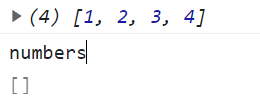
let another = numbers;

// Solution 1

numbers = [];

console.log(another);

(4) [1, 2, 3, 4]



“If you do have multiple references to the original array, then you need to use one of the other solutions.” Solution 2. “We can simply set the length property to zero, and this will truncate the array. It will remove all elements.”

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

let another = numbers;

// Solution 1

// numbers = [];

// Solution 2

numbers.length = 0;

console.log(another);

“Here, we’re logging another. Let’s also log the original array. Save. The original array is truncated so it doesn’t matter how many references you have to that array.

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

let another = numbers;

// Solution 1

// numbers = [];

// Solution 2

numbers.length = 0;

console.log(numbers);

console.log(another);

[]

[]

“The other solution is to use the splice method. So, in the last lecture we learned that with the splice method, we can go to a given position and remove one or more elements. So, we can start with the first element, and remove all elements in this array. How many numbers do we have in this array? Well, that is numbers.length. Save. Again, both arrays are emptied.”

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

let another = numbers;

// Solution 1

// numbers = [];

// Solution 2

// numbers.length = 0;

// Solution 3

numbers.splice(0, numbers.length);

console.log(numbers);

console.log(another);

[]

[]

“And finally the last solution is to use the pop method. So, we learned that the pop method removes the last element in the array. Well, we can put this in the loop and keep calling this method as long as we have an element in this array. While numbers.length is greater than 0, we can keep popping elements.”

“This last solution is not something I recommend you to do, because if you are dealing with a large array; let’s say you have a million objects in this array, there is going to be a performance cost. Your essentially calling this pop method a million times. However, that’s not the situation with most applications out there unless you work for Google.”

“But, in terms of clean coding you can see that this approach is pretty noisy. Compare that to the first solution. It is very simple, elegant. But the only issue at has is it doesn’t work if you have multiple references to this array. Mosh is personal preference is to set the length of the array to zero… But the first solution is also perfectly valid if you have a single reference to your array.”

“And finally, similar to the last solution, we can see that solution three is also a little bit noisy. Overall, mosh would go for either solution two or solution one.”

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

let another = numbers;

// Solution 1

// numbers = [];

// Solution 2

// numbers.length = 0;

// Solution 3

// numbers.splice(0, numbers.length);

// Solution 4

while (numbers.length > 0)

    numbers.pop();

console.log(numbers);

console.log(another);

[]

[]

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

let another = numbers;

// Solution 1

// numbers = [];

// Solution 2

// numbers.length = 0;

// Solution 3

// numbers.splice(0, numbers.length);

// Solution 4

while (numbers.length > 0)

    numbers.pop();

console.log(numbers);

console.log(another);

## 06 - Arrays - 08 - Combining and Slicing Arrays - 3.40

“Now let’s see how we can combine two arrays or slice an array into two parts. So, let’s start by declaring two arrays.”

const first = [1, 2, 3];

const second = [4, 5, 6];

“To combined these two arrays we use the concat method. first.concat… this will basically concatenate the first array, with the second. Now, both arrays are unaffected; so this will return a new array, which is a combination of the first and second arrays”

const first = [1, 2, 3];

const second = [4, 5, 6];

first.concat(second);

“So, let’s store the result in combined and log it on the console. So, we get 123456. Pretty easy.”

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

console.log(combined);

## (6) [1, 2, 3, 4, 5, 6]

“Now, we have another method that is the opposite of concat, and that is slice. And, we use that to slice an array into two parts.”

“So, let’s get the combined array and slice it”

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

combined.slice()

console.log(combined);

“There are multiple ways to use this method. One way is to pass a zero-based index for the start and end. So, we can start at two and finish at four.”

const second = [4, 5, 6];

const combined = first.concat(second);

combined.slice(2, 4)

console.log(combined);

“Let’s see what we get. So, we call that slice and log it on the console. So, this is what we get. We, went to the **combined** array, started after index 2, and finished after index 4. So, we extracted 3 and 4.”

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

const slice = combined.slice(2, 4);

console.log(combined);

console.log(slice);

(6) **[1, 2, 3, 4, 5, 6]**

(2) [3, 4]

“Now, let me show you another way to use the splice method. We can exclude the end index, and with this we get all the elements in the original array starting from the given index.”

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

const slice = combined.slice(2);

console.log(combined);

console.log(slice);

(6) [1, 2, 3, 4, 5, 6]

(4) [3, 4, 5, 6]

“And finally, the third way is to exclude the starting index, and with this we get a copy of the original array.”

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

const slice = combined.slice();

console.log(combined);

console.log(slice);

(6) [1, 2, 3, 4, 5, 6]

(6) [1, 2, 3, 4, 5, 6]

“So, to recap: if you want to combine two arrays, use the concat method. If you want to slice an array, use the slice method.”

“One thing to understand about both of these methods, is that if you are dealing with primitive values, these primitives will be copied into the target array. But, if you have objects in your array, the objects themselves are not copied… their references are copied. And that means the elements in both the input and output arrays will point to the same object. Let me show you what I mean.”

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

const slice = combined.slice();

console.log(combined);

console.log(slice);

“So, for simplicity, let’s remove these three numbers here, and add an object. So, an object with ID set to one. So, now we have a reference type. When we call the concat method, this object itself is not copied to the combined array. Only its references copied.”

const first = [{ id: 1 }];

const second = [4, 5, 6];

const combined = first.concat(second);

const slice = combined.slice();

console.log(combined);

console.log(slice);

“And this means if I go to the first array… get the first element, which is our object, and change its ID to ten. Now when we save the changes, we can see in the combined array… here’s the first element… that’s our object. Note that it’s ID is changed to 10 as well. So, because we are dealing with a reference type, this object is copied by its reference.”

“The same principle applies to the slice method. So, as part of slicing this array, if the elements are primitive, they are copied by value. If they are reference types, they are copied by their reference.”

const first = [{ id: 1 }];

const second = [4, 5, 6];

const combined = first.concat(second);

first[0].id = 10;

const slice = combined.slice();

console.log(combined);

console.log(slice);

(4) [{…}, 4, 5, 6]

0: {id: 10}

1: 42: 5

3: 6length:

4[[Prototype]]: Array(0)

index.js:14 (4) [{…}, 4, 5, 6]

/\*

\*/

const first = [{ id: 1 }];

const second = [4, 5, 6];

const combined = first.concat(second);

first[0].id = 10;

const slice = combined.slice();

console.log(combined);

console.log(slice);

## 06 - Arrays - 09 - The Spread Operator - 2.25

“In the previous lecture we learned how to use the concat method to combined two arrays. There is another way to achieve the same thing by using the spread operator in ES six. Let’s rewrite this code using the spread operator and we’ll see why this approach is cleaner and more flexible.”

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

const slice = combined.slice();

“So, I’m going to define combined as a new array. Now, we want to combine all the elements of the first array and the second array.”

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [];

const slice = combined.slice();

“So, we use the spread operator like this. First, and then second. When we spread an array, all its elements are returned individually. So, basically here we are declaring a new array and in that array we are adding the individual elements of the **first** and **second** arrays…”

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [...**first**, ...**second**];

const slice = combined.slice();

“So, this piece of code here, is exactly like this.”

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [...first, ...second];

[1, 2, 3, 4, 5, 6]

const slice = combined.slice();

“So, we can see with the spread operator we have a better visualization of how these arrays are combined. It’s very clear that we have an array, and in this array, we have all the elements of the first array, followed by all the elements of the second array.”

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [...first, ...second];

const slice = combined.slice();

“Now, with this spread operator we have more flexibility. Let’s say, as part of combining these two arrays we want to add an element in between them. We can do that easily like this. Or, we can add something at the end. Again, we can clearly see what the combined array looks like. In contrast, if we want to do the same thing using the concat method, our code is going to be more complex. So, this is how we can use the spread operator to combine two arrays.”

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [...first, 'a', ...second, 'b'];

const slice = combined.slice();

“Now, in the last lecture we learned that if you call the slice method without any arguments, this will return a copy of the original array here…”

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [...first, 'a', ...second, 'b'];

const slice = combined.slice();

“… So, let’s call that copy.”

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [...first, 'a', ...second, 'b'];

const copy = combined.slice();

“Now, once again we can use the spread operator to copy all the elements of an array into a new array. So, we can rewrite this code like this. Constant, copy… we create a new array. And then spread the combined array. So, this will return all the elements of the combined array and put them into this new array.”

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [...first, 'a', ...second, 'b'];

// const copy = combined.slice();

const copy = [...combined]

const first = [1, 2, 3];

const second = [4, 5, 6];

// const combined = first.concat(second);

const combined = [...first, 'a', ...second, 'b'];

// const copy = combined.slice();

const copy = [...combined]