Building and calling functions

In this reading, you will learn how to build and call a function. The purpose of this reading is to provide you with an example of function declaration (build) and function invocation (call). In the next lesson you will be writing the code.

By the end of this reading you should be able to:

* Code simple functions that can accept an array and iterate through it

Let's start with giving our function declaration a name:

function listArrayItems(arr) {

    // ... code to be added ...

}

So, I've declared a **listArrayItems** function, and I've set it up to accept a single parameter, **arr** - which stands for an array.

Now, I'll need to code a for loop to loop over an array.

As covered in previous lessons in this course, a for loop needs the following information:

1. the starting loop counter value as a temporary variable **i**

2. the exit condition (the maximum value of the loop counter variable **i**, above which the loop no longer runs)

3. how to update the value of **i** after each loop

Here's the information I'll use in this function declaration: 1. The loop's starting counter will be **0**. The reason for setting it to zero is due to the fact that arrays are also counted from zero.

This means that I'll have a one-to-one mapping of the current value of the **i** variable at any given time, corresponding to the same index position of any item in the **arr** array 2.

The for loop's exit condition is when the value of **i** is equal or greater than **arr.length**.

Since the **arr.length** counts the number of items in the array from one, and the array items are indexed from zero, this effectively means that as soon as **i** is equal to **arr.length**, the loop will finish and any other code after it will be run.

This practically means that the exit condition for this for loop will be **i < arr.length** returning **false**.

In other words, as long as **i < arr.length** is true, this for loop will continue to run. 3. To make sure that none of the items in the **arr** array are skipped, I have to increase the value of **i** by **1** after each loop.

Now that I know exactly how my for loop should behave, I can add it to my **listArrayItems()** function:

function listArrayItems(arr) {

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

        // ... code pending here ...

    }

}

Now all that I have left to decide is how I want to output each item from the received **arr** array.

It can be as simple as console logging the array item index of the current value of **i**:

function listArrayItems(arr) {

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

        console.log(arr[i]) //display the array item where the index is euqal to i

    }

}

If I now invoke the **listArrayItems** function, I can, for example, give it the following array of colors:

var colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple', 'pink'];

listArrayItems(colors); //display all items in the array at once

The output will be:

red

orange

yellow

green

blue

purple

pink

I can update the output any way I like. For example, here are my **arr** items with a number in front of each item:

//function that takes an array as input and display all items of this array

function listArrayItems(arr) {

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

        console.log(i, arr[i])

    }

}

var colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple', 'pink'];

listArrayItems(colors);

Now the output of the above code will be as follows:

0 'red'

1 'orange'

2 'yellow'

3 'green'

4 'blue'

5 'purple'

6 'pink'

To start the count from one instead of zero, I can update my function declaration as follows:

function listArrayItems(arr) {

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

        console.log(i+1, arr[i])

    }

}

Invoking the above, updated, function declaration on my **colors** array, will now result in the following output:

1 'red'

2 'orange'

3 'yellow'

4 'green'

5 'blue'

6 'purple'

7 'pink'

I can even add one or more conditions, such as:

function listArrayItems(arr) {

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

        if (arr[i] == 'red') {

            console.log(i\*100, "tomato!")

        } else {

            console.log(i\*100, arr[i])

        }

    }

}

Now I'm adding control flow right inside my function, based on whether a specific array member matches a specific value - in this case the string **"red"**.

Invoking my newest updated version of the **listArrayItems** function on the **colors** array will now result in the following output:

0 'tomato!'

100 'orange'

200 'yellow'

300 'green'

400 'blue'

500 'purple'

600 'pink'

# Object Literals and the Dot Notation

By the end of this reading, you'll be able to:

* Explain one of the three common ways to build objects ( using the object literal notation)
* Outline the common way to add new properties to objects (or update existing properties) using the dot notation

## Object literals and the dot notation

One of the most common ways of building an object in JavaScript is using the object literal syntax: **{}**.

To be able to access this object literal, it is very common to assign it to a variable, such as:

1

var user = {}; //create an object





Now an object literal is assigned to the variable **user**, which means that the object it is bound to can be extended and manipulated in a myriad of ways.

Sometimes, an entire object can be immediately built, using the object literal syntax, by specifying the object's properties, delimited as key-value pairs, using syntax that was already covered in an earlier lesson item in this lesson.

Here's one such previously built object:

1

2

3

4

5

6

7

8

9

//creating an object with properties and their values

var assistantManager = {

    rangeTilesPerTurn: 3,

    socialSkills: 30,

    streetSmarts: 30,

    health: 40,

    specialAbility: "young and ambitious",

    greeting: "Let's make some money"

}





The beauty of this syntax is that it's so easily readable.

It essentially consists of two steps:

1. Declaring a new variable and assigning an object literal to it - in other words, this: **var assistantManager = {}**

2. Assigning the values to each of the object's keys, using the assignment operator, **=**

Notice that it's very easy to build any kind of an object in JavaScript using this example syntax.

For example, here's a **table** object:

1

2

3

4

5

var table = {

    legs: 3,

    color: "brown",

    priceUSD: 100,

}





To access the **table** object, I can simply console log the entire object:

1

console.log(table);//display the object in the developer console





The returned value is the entire **table** object:

1

{legs: 3, color: 'brown', priceUSD: 100}





Additionally, I can console log any individual property, like this:

1

console.log(table.color); // 'brown'





Now that I have this "syntax recipe", I can build any other object in a similar way:

1

2

3

4

5

var house = {

    rooms: 3,

    color: "brown",

    priceUSD: 10000,

}





An alternative approach of building objects is to first save an empty object literal to a variable, then use the dot notation to declare new properties on the fly, and use the assignment operator to add values to those properties; for example:

1

2

3

4

var house2 = {};

house2.rooms = 4;

house2.color = "pink";

house2.priceUSD = 12345;





Additionally, nothing is preventing me from combining the two approaches. For example:

1

2

3

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000}

house.windows = 10;

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000, windows: 10}





This flexbility additionally means that I can update already existing properties, not just add new ones:

1

2

house.windows = 11;

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000, windows: 11}

# Object Literals and the Brackets Notation

By the end of this reading, you'll be able to:

* Explain how to build objects using the brackets notation
* Explain that with the brackets notation you can use the space character inside keys, since property keys are strings
* Explain that the keys inside the brackets notation are evaluated

## Object literals and the brackets notation

There is an alternative syntax to the dot notation I used up until this point.

This alternative syntax is known as the brackets notation.

To understand how it works, it's best to use an example, so I'll go through the process of coding the **house2** object again, in the same way that I did with the dot notation, only this time, I'll use the brackets notation.

1

2

3

4

5

var house2 = {};

house2["rooms"] = 4;

house2['color']= "pink";

house2["priceUSD"] = 12345;

console.log(house2); // {rooms: 4, color: 'pink', priceUSD: 12345}





Note that using the brackets notation, I essentially just wrap each property's key **as a string**, inside either the single or double quotes - just like with regular strings.

Then I wrap the entire property key into an opening and a closing square bracket.

That's essentially all there is to it.

I can both access and update properties on objects using either the dot notation, or the brackets notation, or a combination of both, like in the following example:

1

2

3

4

5

6

var car = {};

car.color = "red";

car["color"] = "green";

car["speed"] = 200;

car.speed = 100;

console.log(car); // {color: "green", speed: 100}





For the time being, this is probably enough information on object creation.

Before I discuss the topic of arrays and objects, let me just give you another important piece of information about the brackets notation.

With the brackets notation, I can add space characters inside property names, like this:

1

2

car["number of doors"] = 4;

console.log(car); // {color: 'green', speed: 100, number of doors: 5}





Additionally, I can add numbers (as the string data type) as property keys:

1

2

car["2022"] = 1901;

console.log(car); // {2022: 1901, color: 'green', speed: 100, number of doors: 5}





However, doing this is discouraged, due to obvious reasons of having a property key as a number string not really conveying a lot of useful information.

Finally, there's one really useful thing that bracket notation has but is not available in the dot notation: It can evaluate expressions.

To understand what that means, consider the following example:

1

2

3

4

5

6

7

8

9

var arrOfKeys = ['speed', 'altitude', 'color'];

var drone = {

    speed: 100,

    altitude: 200,

    color: "red"

}

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

    console.log(drone[arrOfKeys[i]])

}





The above code will result in the following output:

1

2

3

100

200

red





Using the fact that brackets notation can evaluate expressions, I accessed the **arrOfKeys[i]** property on the **drone** object.

This value changed on each loop while the for loop was running.

Specifically, the first time it ran, it was evaluated like this:

* The value of **i** was **0**
* The value of **arrOfKeys[i]** was **arrOfKeys[0]**, which was **"speed"**
* Thus, **drone[arrOfKeys[i]]** was evaluated to **drone["speed"]** which is equal to **100**

This allowed me to loop over each of the values stored inside the **drone** object, based on each of its properties' keys.

# Arrays are Objects

By the end of this reading, you'll be able to:

* Explain that arrays are objects, with their own built-in properties and methods
* Outline the common way to extend arrays using the push() method
* and explain how to trim the last member of an array using the pop() method

## Arrays are objects

In JavaScript, arrays are objects. That means that arrays also have some built-in properties and methods.

One of the most commonly used built-in methods on arrays are the **push()** and the **pop()** methods.

To add new items to an array, I can use the **push()** method:

1

2

3

var fruits = [];

fruits.push("apple"); // ['apple']

fruits.push('pear'); // ['apple', 'pear']





To remove the last item from an array, I can use the **pop()** method:

1

2

fruits.pop();

console.log(fruits); // ['apple']





Tying into some earlier lessons in this course, I can now build a function that takes all its arguments and pushes them into an array, like this:

1

2

3

4

5

6

7

function arrayBuilder(one, two, three) {

    var arr = [];

    arr.push(one);

    arr.push(two);

    arr.push(three);

    console.log(arr);

}





I can now call the **arrayBuilder()** function, for example, like this:

1

arrayBuilder('apple', 'pear', 'plum'); // ['apple', 'pear', 'plum']





Even better, I don't have to console log the newly built array.

Instead, I can return it:

1

2

3

4

5

6

7

function arrayBuilder(one, two, three) {

    var arr = [];

    arr.push(one);

    arr.push(two);

    arr.push(three);

    return arr;

}





Additionally, I can save this function call to a variable.

I can name it anything, but this time I'll use the name: **simpleArr**.

1

var simpleArr = arrayBuilder('apple', 'pear', 'plum');





And now I can console log the values stored in **simpleArr**:

1

console.log(simpleArr); // ['apple','pear','plum']

# Math object cheat sheet

JavaScript has handy built-in objects. One of these popular built-in objects is the Math object.

By the end of this reading, you'll be able to:

* Outline the built-in properties and methods of the Math object

## Number constants

Here are some of the built-in number constants that exist on the Math object:

* The PI number: **Math.PI** which is approximately 3.14159
* The Euler's constant: **Math.E** which is approximately 2.718
* The natural logarithm of 2: **Math.LN2** which is approximately 0.693

## Rounding methods

These include:

* **Math.ceil()** - rounds up to the closest integer
* **Math.floor()** - rounds down to the closest integer
* **Math.round()** - rounds up to the closest integer if the decimal is **.5** or above; otherwise, rounds down to the closest integer
* **Math.trunc()** - trims the decimal, leaving only the integer

## Arithmetic and calculus methods

Here is a non-conclusive list of some common arithmetic and calculus methods that exist on the **Math** object:

* **Math.pow(2,3)** - calculates the number **2** to the power of **3**, the result is **8**
* **Math.sqrt(16)** - calculates the square root of **16**, the result is **4**
* **Math.cbrt(8)** - finds the cube root of **8**, the result is **2**
* **Math.abs(-10)** - returns the absolute value, the result is **10**
* Logarithmic methods: **Math.log()**, **Math.log2()**, **Math.log10()**
* Return the minimum and maximum values of all the inputs: **Math.min(9,8,7)** returns **7**, **Math.max(9,8,7)** returns **9**.
* Trigonometric methods: **Math.sin()**, **Math.cos()**, **Math.tan()**, etc.

String cheat sheet

By the end of this reading, you'll be able to:

* Identify examples of String functions and explain how to call them

In this cheat sheet, I'll list some of the most common and most useful properties and methods available on strings.

For all the examples, I'll be using either one or both of the following variables:

var greet = "Hello, ";

var place = "World";

Note that whatever string properties and methods I demo in the following examples, I could have ran it on those strings directly, without saving them to a variable such as the ones I named **greet** and **place**.

In some of the examples that follow, for the sake of clarity, instead of using a variable name, I'll use the string itself.

All strings have at their disposal several built-in properties, but there's a single property that is really useful: the **length** property, which is used like this:

1

greet.length; // 7





To read each individual character at a specific index in a string, starting from zero, I can use the **charAt()** method:

1

greet.charAt(0); // 'H'





The **concat()** method joins two strings:

1

"Wo".concat("rl").concat("d"); // 'World'





The **indexOf** returns the location of the first position that matches a character:

1

2

3

"ho-ho-ho".indexOf('h'); // 0

"ho-ho-ho".indexOf('o'); // 1

"ho-ho-ho".indexOf('-'); // 2





The **lastIndexOf** finds the last match, otherwise it works the same as **indexOf**.

The **split** method chops up the string into an array of sub-strings:

1

"ho-ho-ho".split("-"); // ['ho', 'ho', 'ho']





There are also some methods to change the casing of strings. For example:

1

2

greet.toUpperCase(); // "HELLO, "

greet.toLowerCase(); // "hello, "





Here's a list of all the methods covered in this cheat sheet:

* **charAt()**
* **concat()**
* **indexOf()**
* **lastIndexOf()**
* **split()**
* **toUpperCase()**
* **toLowerCase()**

Object Methods

You might already be familiar with objects in JavaScript.

In this video, you will learn how to design objects as combinations of data and functionality.

As you might already know, an object consists of key-value pairs, known as properties.

We can add new key-value pairs to existing objects using the dot notation and the assignment operator.

var car = {};

car.color = "red"; //update the value of a property of the car object

These are known as properties, and can take many data types, including functions.

var car = {};

car.color = "red";

//add a method to the car object so that it can be called as car.turnkey()

car.turnKey = function() {

  console.log('engine running');

}

If the function is a property of an object, it is then referred to as a method.

This is a function that can be accessed only through the JavaScript object that it is a member of. For example, the log method, which belongs to the console object, can only be accessed through the console object.

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

Let's explore this further now. I will create an object using something known as the constructor function.

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

RunReset

First, I'll build a new object literally named **car**. I type **var**, space, **car**, space, equal sign, space, followed by a set of curly braces, and finally a semicolon.

Now, I'll extend the **car** object by assigning it a property named **mileage**.

When I inspect the object, I can confirm that it contains a **mileage** property set to **98765**.

I want to add another property to the **car** object. This time, I will add a property named **color** and set it to the value of **"red"**.

I can inspect the object again by typing its name into the browser console. So now, when I type **console.log(car)**, I get an object with two properties: the **mileage** property, which is set to **98765**, and the **color** property, set to **"red"**.

Great, now I've added two properties to my object.

Next, I want to add a method to my **car** object. And this method, when called, will output some text to the console.

So, once again, I add another property to my **car** object. After all, a method is just another property of the **car** object. It's just another key-value par that the car object holds.

What's unique is that the value I'm assigning to it is a function.

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

car.turnTheKey = function() {

    console.log("The engine is running")

}

console.log(car);

RunReset

So, I begin by typing **car** dot **turnTheKey**, equals, and then I type the code for my function. So **function**, open-close parentheses. Then the two curly braces where I will place my code. Finally, inside the curly braces, I type the console dot log followed by the message **"The engine is running"**.

Now I can inspect my **car** object again by typing its name into the console log method. This time, it displays that the **car** object contains three properties; the **color** property, the **mileage** property, and the **turnTheKey** property.

Remember that all the key-value pairs in an object are referred to simply as properties. However, if I want to differentiate between the properties that can be executed, then I refer to such properties as methods.

So, now I want to add another method to the **car** object. I'll name this one **lightsOn**.

Once again, I type **car.lightsOn**, and then I add an equals sign, and again since it's a method, I'm assigning it to a function. This function will also have a console log in its body, and I'm just logging the string with the text **"The lights are on"**.

//example of adding properties and methods to an object

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

car.turnTheKey = function() {

    console.log("The engine is running")

}

car.lightsOn = function() {

    console.log("The lights are on.")

}

console.log(car);

car.turnTheKey();

car.lightsOn();

RunReset

Ok, so now I have added four properties to my object. And two of those are methods.

I've already ensured that I'm getting the correct **mileage** and **color** from my **car** object. Now, I'll try executing the **turnTheKey** and the **lightsOn** methods.

First, I'll invoke the **turnTheKey** method.

Remember that this method can be accessed only through the **car** object, so I first need to type the name of the object that holds the **turnTheKey** method. In other words, I need to type the word **car**, followed by a dot, and then the name of my method, which is **turnTheKey**.

Remember that this property is a method. So, to run it, I need to append an opening and a closing parenthesis so that the JavaScript engine can process my JavaScript code.

Notice that this results in the **"The engine is running"** string logged to the console.

Now I'll test the other method. Once again, I need to access it through the **car** object, so I type **car.lightsOn**, and again, I need to add those parentheses to invoke the **lightsOn** method. I press the ENTER key and notice the text displays in the console.

Success! It's important to remember that when the JavaScript engine runs this line of code, it locates the **car** object in its memory. Then, it finds the **lightsOn** method on the **car** object. It reads the function declaration that's saved on this property and runs it, line by line.

Since there's only a single line of code, the JavaScript engine logs the string **"The lights are on"** to the console.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

[JavaScript Functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions)

[JavaScript Object Basics](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics)

[typeof operator in JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof)

[Arrays are "list-like objects"](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Arrays)

# Syntax, logical and runtime errors

By the end of this reading, you'll be able to:

* Recognize common types of errors in JavaScript

Here are some of the most common errors in JavaScript:

* ReferenceError
* SyntaxError
* TypeError
* RangeError

There are some other errors in JavaScript. These other errors include:

* AggregateError
* Error
* InternalError
* URIError

However, in this reading I'll focus on the Reference, Syntax, Type, and Range errors.

## ReferenceError

A ReferenceError gets thrown when, for example, one tries to use variables that haven't been declared anywhere.

An example can be, say, attempting to console log a variable that doesn't exist:

1

console.log(username);





If the variable named **username** hasn't been declared, the above line of code will result in the following output:

1

Uncaught ReferenceError: username is not defined





## SyntaxError

Any kind of invalid JavaScript code will cause a SyntaxError.

For example:

1

var a "there's no assignment operator here";





The above line of code will throw the following error:

1

Uncaught SyntaxError: Unexpected string





There's an interesting caveat regarding the SyntaxError in JavaScript: it cannot be caught using the **try-catch** block.

## TypeError

A TypeError is thrown when, for example, trying to run a method on a non-supported data type.

A simple example is attempting to run the **pop()** method on a string:

1

"hello".pop() // Uncaught TypeError: "hello".pop is not a function





The array-like behavior of strings was already covered in an earlier lesson in this course.

However, as can be confirmed by running the above line of code, strings do not have all the array methods readily available to them, and trying to use some of those methods will result in a TypeError being thrown.

## RangeError

A RangeError is thrown when we're giving a value to a function, but that value is out of the allowed range of acceptable input values.

Here's a simple example of converting an everyday Base 10 number (a number of the common decimal system) to a Base 2 number (i.e binary number).

For example:

1

(10).toString(2); // '1010'





The value of **2** when passed to the **toString()** method, is like saying to JavaScript: "convert the value of **10** of the Base 10 number system, to its counter-part in the Base 2 number system".

JavaScript obliges and "translates" the "regular" number 10 to its binary counter-part.

Besides using Base 2 number system, I can also use the Base 8, like this:

1

(10).toString(8); // 12





I get back the value **12**, which is the plain number 10, writen in Base 8 number system.

However, if I try to use a non-existing number system, such as an imaginary Base 100, since this number system effectively doesn't exist in JavaScript, I will get the RangeError, because a non-existing Base 100 system is **out of range** of the number systems that are available to the **toString()** method:

1

(10).toString(100); // Uncaught RangeError: toString() radix argument must be between 2 and 36





In this reading, you've covered some of the most common errors in JavaScript.

# Additional resources

Here is a list of resources that may be helpful as you continue your learning journey.

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

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

[Iteration protocols](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols)

[The Math object on MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)

[The String object on MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

[MDN JavaScript error reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors)

[The null value on MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null)

[The undefined property on MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined)