

# Functions

This lesson covers the fundamentals of Functions in JavaScript.

## WE'LL COVER THE FOLLOWING

- Understanding function scope and the `this` keyword
- The `this` keyword

Functions are a very important tool we use to perform tasks and calculations of all kinds. In `JavaScript`, we can declare a function in different ways.

Let's have a look at a **function definition**:

```
function greet(name){  
  console.log("hello " + name);  
}  
greet("Alberto")  
// hello Alberto
```



This is a simple function that when called will log a string. The variable inside the parenthesis at line 1, is called a **parameter** while the code inside of the curly brackets is a **statement**, in this case a very simple `console.log()`.

A very important thing to remember is that **primitives** are passed to a function **by value**, meaning that the changes done to those values are not reflected globally. On the other hand, if the value is not a primitive, such as an `Object` or an `Array`, it is then passed **by reference**, meaning that any modification done to it will be reflected in the original `Object`.

```
let myInt = 1;  
  
function increase(value){  
  return value + 1;  
}
```



```
return value +=1;
}

console.log(myInt);
// 1
console.log(increase(myInt));
// 2
console.log(myInt);
// 1
```



As you can see, we increased the value of the integer, but that didn't affect the original variable. Let's look at an example with an **Object**.

```
let myCar = {
  make: "bmw",
  color: "red"
}

console.log(myCar)
// {make: "bmw", color: "red"}

function changeColor(car){
  car.color = "blue"
}

changeColor(myCar)
console.log(myCar)
// {make: "bmw", color: "blue"}
```



As you can see, since the parameter **car** was just a reference to the **Object myCar**, modifying it resulted in a change in the **myCar Object**.

Another way of declaring a function is by using a **function expression**.

```
const greeter = function greet(name){
  console.log("hello " + name);
}
greeter("Alberto")
// hello Alberto
```



Here we assigned our **function greet** to a **const**, called **greeter**. We got the same result as in the first example, but with a **function expression** we can

same result as in the first example, but with a **function expression** we can also create **anonymous functions**.

```
const greeter = function(name){  
  console.log("hello " + name);  
}  
greeter("Alberto")  
// hello Alberto
```



We changed **function greet** to **function** and we got an **anonymous function**.

We could also tweak it a little bit further by using an **arrow function**, introduced by ES2015.

```
const greeter = (name) => {  
  console.log("hello " + name);  
}  
greeter("Alberto")  
// hello Alberto
```



The **function** keyword goes away and we get a nice fat arrow (**=>**) after the parameters.

We'll look at arrow functions in detail in the lesson [Arrow Functions](#).

## Understanding function scope and the **this** keyword #

One of the most important concepts to understand in **JavaScript** is the scope.

### What is Scope?

The **scope** of a variable controls where that variable can be accessed from. We can have a **global scope**, meaning that the variable can be accessed from anywhere in our code or we can have a **block scope**, meaning that the variable can be accessed only from inside of the block where it has been declared.

A **block** can be a function, a loop or anything delimited by *curly brackets*.

Let's have a look at two examples, first using the keyword **var**.

```
var myInt = 1;

if(myInt === 1){
  var mySecondInt = 2
  console.log(mySecondInt);
  // 2
}
console.log(mySecondInt);
// 2
```

As you can see, we were able to access the value of **mySecondInt** even outside of the **block** scope, as variables declared with the keyword **var** are not bound to it.

Now let's use the keyword **let**.

```
var myInt = 1;

if(myInt === 1){
  let mySecondInt = 2
  console.log(mySecondInt);
  // 2
}
console.log(mySecondInt);
// Uncaught ReferenceError: mySecondInt is not defined
```

This time we couldn't access the variable from outside of the **block** scope and we got an error **mySecondInt is not defined**.

Variable declared with the keywords **let** or **const** are bound to the **block** scope where they have been declared. You will learn more about them in Chapter 1.

## The `this` keyword #

The second important concept I want to discuss is the `this` keyword.

Let's first start with a simple example:

```
const myCar = {  
  color: 'red',  
  logColor: function(){  
    console.log(this.color)  
  }  
}  
myCar.logColor();  
// red
```

As you can see in this example, it's self-explanatory that the `this` keyword refers to the `myCar` Object.

The value of `this` depends on how a function is called. In the example above, the function was called as a method of our `Object`.

Look at this other example:

```
function logThis(){  
  console.log(this);  
}  
logThis();  
// Window {...}
```

To see the expected output of the code above, please run it in the Console of your Browser.

We called this function in the global context, therefore the value of `this` referred to the `Window Object`.

We can avoid accidentally referring to the `Window Object` by turning on `strict` mode.

You can do that by writing `'use strict';` at the beginning of your `JavaScript`

file.

By doing so you will enable a stricter set of rules for `JavaScript`, among which there's one that sets the value of the `Global Object` to `undefined` instead of to the `Window Object`. This causes our `this` keyword to also become `undefined`.

If we want to manually set the value of `this` to something we can use `.bind`.

```
const myCar = {
  color: 'red',
  logColor: function(){
    console.log(this.color)
  }
}

const unboundGetColor = myCar.logColor;
console.log(unboundGetColor())
// undefined
const boundGetColor = unboundGetColor.bind(myCar)
console.log(boundGetColor())
// red
```

Let's go through what we just did:

- First we created an `Object` similarly to the previous example
- We set `unboundGetColor` equal to the method of `myCar`
- When we try to call `unboundGetColor`, it tries to look for `this.color`, but since it gets invoked in the global context, the value of `this` is the `Window Object`. There's no `color` on it, therefore we get `undefined`
- Lastly, we use `.bind` to specifically tell `boundGetColor` that the `this` keyword will refer to the `Object` inside of the parenthesis, in this case `myCar`
- When we call `boundGetColor` you can see that this time we get the result that we were looking for

There are two other methods we can use to set the value of the `this` keyword: `.call()` and `.apply()`.

They are both similar in that both methods call a function with a given `this` value. The arguments they accept are a bit different.

`.call()` accepts a *list of arguments* while `.apply()` accepts a *single array of arguments*.

Look at this example using `.call()`

```
function Car(make,color){
  this.carMake = make;
  this.carColor = color;
}

function MyCar(make,color){
  Car.call(this,make,color);
  this.age = 5;
  console.log(this);
  // MyCar { carMake: 'bmw', carColor: 'red', age: 5 }
}

const myNewCar = new MyCar('bmw','red')
console.log(myNewCar.carMake)
// bmw
console.log(myNewCar.carColor)
// red
```

We are passing our `MyCar` Object inside of the `.call()` so that `this.carMake` will get set to the *make* that we passed as an argument of `MyCar`. Same for the *color*.

Look at this example with `.apply()` to see the differences between the two.

```
function Car(make,color){
  this.carMake = make;
  this.carColor = color;
}

function MyCar(make,color){
  Car.apply(this,[make,color]);
  this.age = 5;
  console.log(this);
  // MyCar { carMake: 'bmw', carColor: 'red', age: 5 }
}

const myNewCar = new MyCar('bmw','red')
console.log(myNewCar.carMake)
// bmw
console.log(myNewCar.carColor)
// red
```

As you can see, the result was the same, but in this case `.apply()` accepts an

array with a list of arguments.

The major difference between the two comes into play when you are writing a function that does not need to know, or doesn't know the number of arguments required. In that case, since `.call()` requires you to pass the arguments individually, it becomes problematic to do. The solution is to use `.apply()`, because you can just pass the array and it will get unpacked inside of the function, no matter how many arguments it contains.

```
const ourFunction = function(item, method, args){  
  method.apply(args);  
}  
ourFunction(item,method, ['argument1', 'argument2'])  
ourFunction(item,method, ['argument1', 'argument2', 'argument3'])
```



No matter how many arguments we pass, they will get *applied* individually when `.apply()` is called.

We have now covered the basics of `JavaScript`. In the next chapter, we will be looking at different features of `JavaScript` in detail.