Back to Week 2

# Bonus: JS | Context & Function invocation

SELF GUIDED

## Learning Goals

After this learning unit you will be able to:

- Understand what is context and how to access it using `this`
- Understand the Global Context, and how to use it
- Understand what are Function Internals in JavaScript, and how to use them:
  - Method invocation
  - Function invocation
  - Constructor invocation
  - `Apply` and `Call` invocations
- Understand which are the differences between `apply` and `call` invocations
- Understand what `bind` is and how to use it correctly

## Introduction

Context refers to the value of `this` for the code that is running:

> CONTEXT === THIS

## Global Context

When we are in the global scope, `this` is always a reference to the `window` object:

```
var a = 1;

console.log(this);           // Window object
console.log(this === window); // true

console.log(window.a);       // 1
console.log(this.a);         // 1
console.log(a);              // 1
```

Explain this code

If we declare a function, it will create a new `scope`, but the `context` in which that function runs will still be the same. By default, a function always runs in the scope of the object it belongs to:

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

foo();
```

Explain this code

# JavaScript Function Internals

JavaScript has the ability to modularize logic in functions which can be invoked at any point within the execution.

Invoking a function is pretty easy, but what does exactly happens when we call a function? Javascript follows these steps:

- Suspends execution of the current function
- Passes controls to the invoked function
- Passes (secretly) two parameters to the invoked function:
  - An array named `arguments`
  - A parameter named `this`

```
function doStuff (a, b) {
  console.log(arguments); // ['hi', 2, 8]
}
doStuff('hi', 2, 3+5);
```

Explain this code

Even though there is only one invocation operator `()`, there are **four invocation patterns**, 4 different ways of calling functions. Each pattern differs in how the `this` parameter is initialized. Invoking a function with a different pattern can produce a vastly different result.

## Method Invocation

When a function is part of an object, it is called a method. Method invocation is the pattern of invoking a function that is part of an object. For example:

```
var obj = {
  value: 0,
  increment: function() { this.value++; }   // this == obj
};

obj.increment();
```

Explain this code

**Method invocation** can be easily identified when a function is preceded by `object.`, where `object` is the name of some object. **JavaScript will set the `this` parameter to the object where the method was invoked on**. JavaScript binds this at **runtime** (also known as **late binding**).

## Function Invocation

When we call a function *normally*, we are using the **function invocation pattern**, and JavaScript will bind the value of `this` to the `global object`.

What would the following code do?

```
var value = 500;
var obj = {
  value: 0,
  increment: function() {
    this.value++;

    var innerFunction = function() {
      console.log(this.value);
    }

    innerFunction(); // Function invocation pattern
  }
}
obj.increment(); // Method invocation pattern
```

Explain this code

The real answer is `500`, not `1`. Note that `innerFunction` is called using the **function invocation pattern**, therefore this is set to the `global object`. The result is that innerFunction will not have `this` set to the `current object`. Instead, it is set to the `global object`, where `value` is defined as `500`.

If we really want to have `innerFunction` access the context of it's parent, we may want to keep state in the closure by storing `that` as the previous `this`.

```
var value = 500;
var obj = {
  value: 0,
  increment: function() {
    var that = this;
    that.value++;

    var innerFunction = function() {
      console.log(that.value);
    }

    innerFunction(); // Function invocation pattern
  }
}
obj.increment();
```

Explain this code

## Constructor Invocation

In classical **object oriented programming**, an object is an instantiation of a class. In `C++` and `Java`, this instantiation is performed by using the `new` operator.

The **constructor invocation pattern** involves putting the `new` operator just before the function is invoked. For example:

```
var Cheese = function(typeOfCheese) {
  var cheeseType = typeOfCheese;
  return cheeseType;
}

cheddar = new Cheese("cheddar"); // new object returned, not the type.
```

Explain this code

Even though `Cheese` is a function object (and intuitively, one thinks of functions as running modularized pieces of code), we have created a new object by invoking the function with `new` in front of it.

**The `this` parameter will be set to the newly created object** and the `return` operator of the function will have its behavior altered.

## `Apply` And `Call` Invocation

Because JavaScript is a functional object-oriented language, functions can also have methods.

### `Apply` Method

The **apply method** allows manual invocation of a function.

Apply is a hidden method of every function, so we call it by adding `.apply()` to the function itself and it takes two parameters:

- An object to bind the `this` parameter to
- An array which is mapped to the parameters

```
var obj = {
  foo: function(a, b, c) {
    console.log( arguments );
    console.log( this );
  }
};

obj.foo(1,2,3);
// ==> [1,2.3]
// ==> obj {}

obj.foo.apply(window, [1,2,3]);
// ==> [1,2.3]
// ==> window {}
```

Explain this code

> **Note:** `arguments` is an *array like* object, which has the `length` property and no other array methods we can use. This probably came as a decision to make the language faster as EVERY function call will implicitly have `arguments` available.

### `Call` Method

JavaScript also has another invoker called **call**, that is identical to `apply` except that **instead of taking an array of parameters, it takes an argument list**.

```
var fer   = {name: 'Fer',   coder: true};
var harry = {name: 'Harry', coder: true};

var hi = function(){
  console.log('Whatsup, ' + this.name);
};

var bye = function(){
  console.log('Laters, ' + this.name);
};

hi();   // Error

hi.call(fer);
bye.call(harry);
```

Explain this code

All four of these lines do exactly the same thing. The run `hi` or `bye` in the scope of either `fer` or `harry`.

```
var fer   = {name: 'Fer',   coder: true, nationality: 'Mexican'};
var harry = {name: 'Harry', coder: true, nationality: 'Taiwanese'};

var update = function(name, coder, nationality){
  this.name = name;
  this.coder = coder;
  this.nationality = nationality;
};

update.call(fer, 'Fer', true, 'Spanish');
update.apply(harry, ['Harry', true, 'Canadian']);
```

Explain this code

## `Apply` Vs `Call`

```
Function.call(this, param1, param2, param 3,... )
Function.apply(this, [param1, param2, param 3, ...])
```

Explain this code

The limitations of `call` quickly become apparent when you want to write code that doesn’t know the number of arguments that the functions need.

```
var addGrades_CALL = function(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10) {
    console.log(arguments); // ==> [1,2,3,4,5,6,7,8,9,10]
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

addGrades_CALL.call(null, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Explain this code

In the case of `apply`, passing an array as a parameter will allow us to iterate through all the parameters easily:

```
var grades1 = [1,2,3,4,5];
var grades2 = [1,2,3,4,5,6,7,8,9,10];

var addGrades_APPLY = function() {
    console.log(arguments);
    var sum = 0;
    for (var i=0; i < arguments[1].length; i++) {
        sum += arguments[1][i];
    }
    return sum;
}

addGrades_APPLY(null, grades1);
addGrades_APPLY(null, grades2);
```

Explain this code

## Bind

Bind creates a new function that, when called, has its `this` keyword set to the first provided parameter, with a given sequence of arguments preceding any provided when the new function is called.

```
var obj = {
  foo: function() {
    console.log( this );
  }
};

var bindFoo = obj.foo.bind(window);

obj.foo(); // ==> obj
bindFoo(); // ==> window
```

Explain this code

## Summary

In this lesson, you have learned what is the `this` context, and the differences between context and scope. We have seen that JavaScript has the ability to modularize logic in functions.

We have seen the four different ways to invoke a function in JavaScript, and which are the main differences between them. Finally, we have learnt what is the `bind` method and how it can helps us on our developments.

# Extra Resources

- **Understanding Scope and Context in JS**
- **Bind and this - Object creation in JavaScript**


**Mark as completed**

PREVIOUS
← Bonus: LAB | JavaScript Koans

NEXT
DOM | Introduction & Selectors →