

the Master Course

{C0DENATION}

Scope and Higher Order Functions

{CODENATION}

Learning Objectives

To understand what **variable scope** is in Javascript.

To understand what happens with our variables in the **memory**.

To be able to work with **Higher Order Functions**.

As we have noticed, sometimes we have access to the variables we create... but sometimes we don't!

What's going on here?

Variable Scope

There are **three types** of variable scope we will work with:

1. Global Scope
2. Function Scope
3. Block Scope

Global Scope

The global scope refers to the global namespace of our code. Any variable created inside the global scope can be accessed anywhere in it.

Global Scope



```
let a = 1  
let b = 2  
let c = 3
```

```
console.log(a)
```

```
const myFunction = () => {  
  console.log(b)  
  console.log(c)  
}
```

```
myFunction()
```


Function Scope

The function scope refers to the space within the curly braces of our function. Any variables declared in our function, can only be accessed from within this function.

```
let a = 1  
let b = 2  
let c = 3
```

```
console.log(a)
```

```
const myFunction = () => {  
  let d = 4  
  console.log(b)  
  console.log(d)  
}
```

```
myFunction()  
console.log(d) // doesn't work
```

**Global
Scope**



**Function
Scope**



Scoping works outwards.

When JS looks for a variable, it first searches the scope it is in, if it can't find it, it will move **outwards** and check the next scope, and keep moving **outwards** until it reaches the global scope – finally, if it can't find it in the global scope it will return a reference error.

**This is called the
scope chain.
Let's have a look.**

```
let a = 1
let b = 2
let c = 3
```

```
console.log(a)
console.log(c)
```

```
const myFunction = () => {
  console.log(b)
```

```
  const myFunction2 = () => {
    let d = 4
    console.log(c)
    console.log(d)
  }
```

```
  myFunction2()
  console.log(c)
  console.log(d)
}
```

```
myFunction()
```

← **Global Scope**

← **myFunction
Scope**

← **myFunction2
Scope**

Block Scope

Block scope refers to a code block (between a pair of curly braces).

Variables declared inside a block, can only be accessed from within the block.

```
let a = 1  
let b = 2  
let c = 3
```

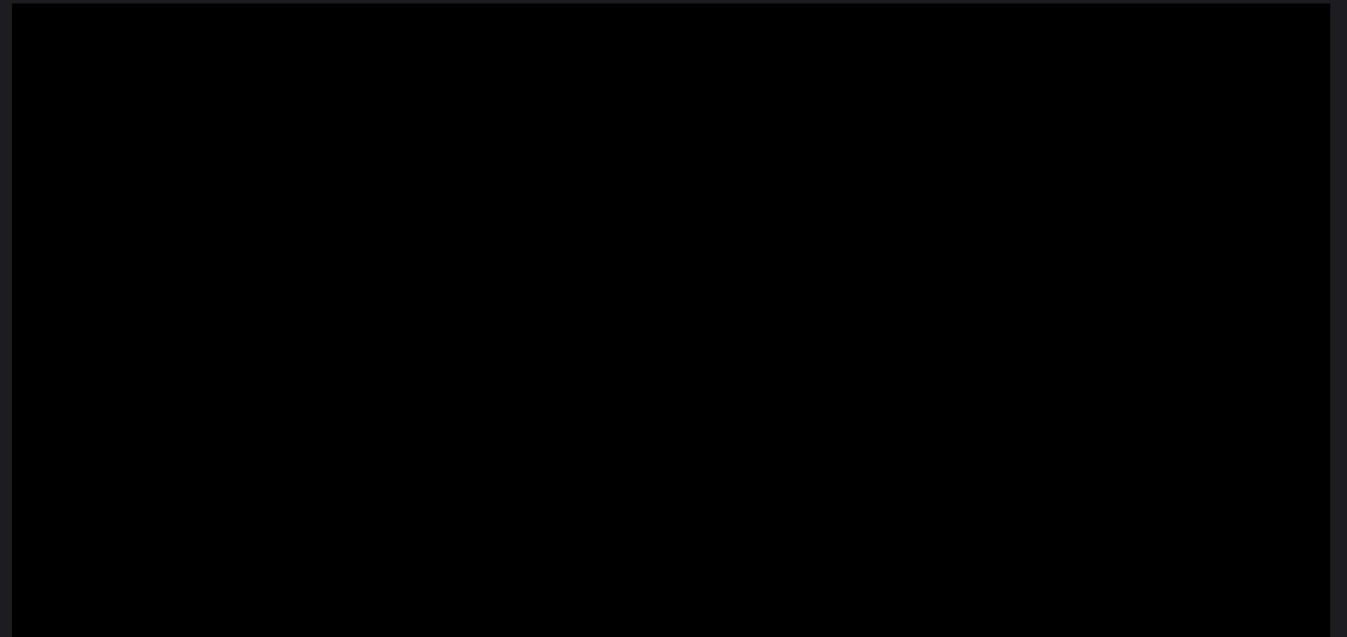
```
console.log(a)  
console.log(c)
```

```
const myFunction = () => {  
  console.log(b)
```

```
    if (true){  
      let d = 4  
      console.log(d)  
    }
```

```
  console.log(d)  
}
```

```
myFunction()
```



Activity

Create some variables in the global scope, some variables in a function (local) scope. Create a function inside your function and declare a variable in there too.

Then try and access them from different places in your code using console.log

**Why does this
happen
though?**

It's to do with how the **memory
heap stores our variable data.**

When you store a variable inside a function, it's only created (stored in memory) after you **call the function.**

And then **when the function has finished running, the variable is destroyed! (removed from memory)**

The **memory heap** is part of the javascript engine where variable data gets stored. It's where **memory allocation** takes place when we create stuff.

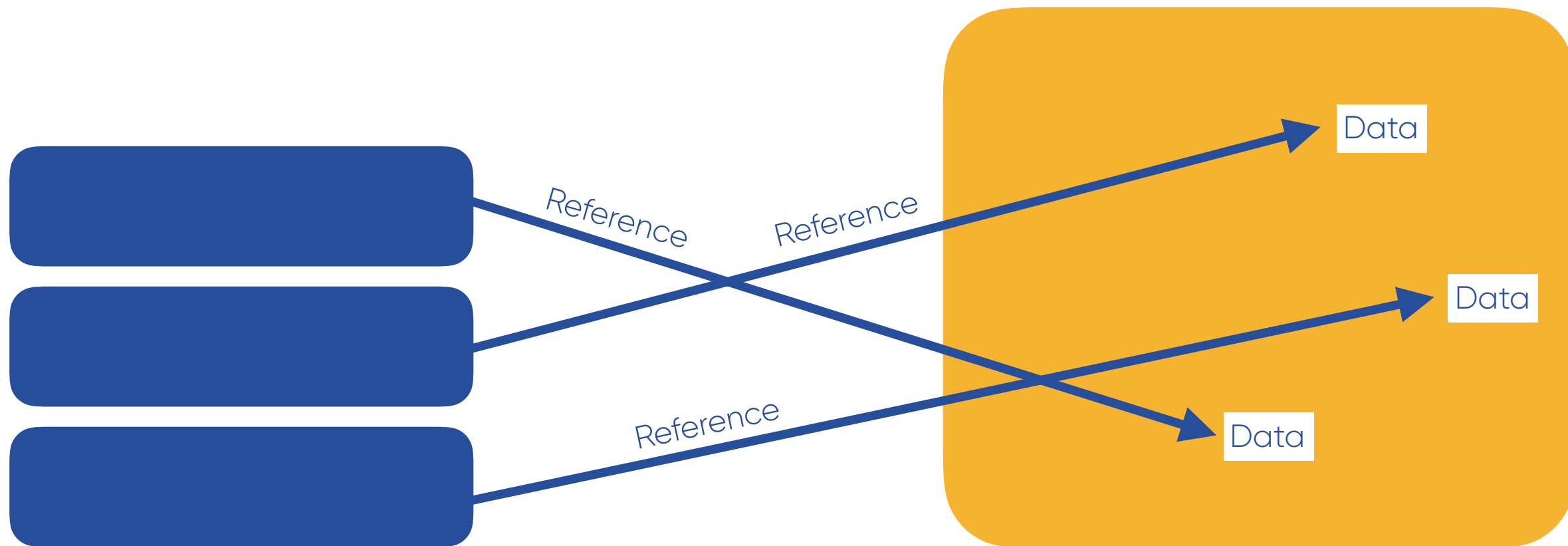
The `call stack` keeps track of our code as it executes. We will come back to the call stack in more detail when we look at asynchronous javascript.

Call stack

Functions and their local variables get added to the stack as our code runs.

Memory Heap

The data and objects our functions and variables point to.

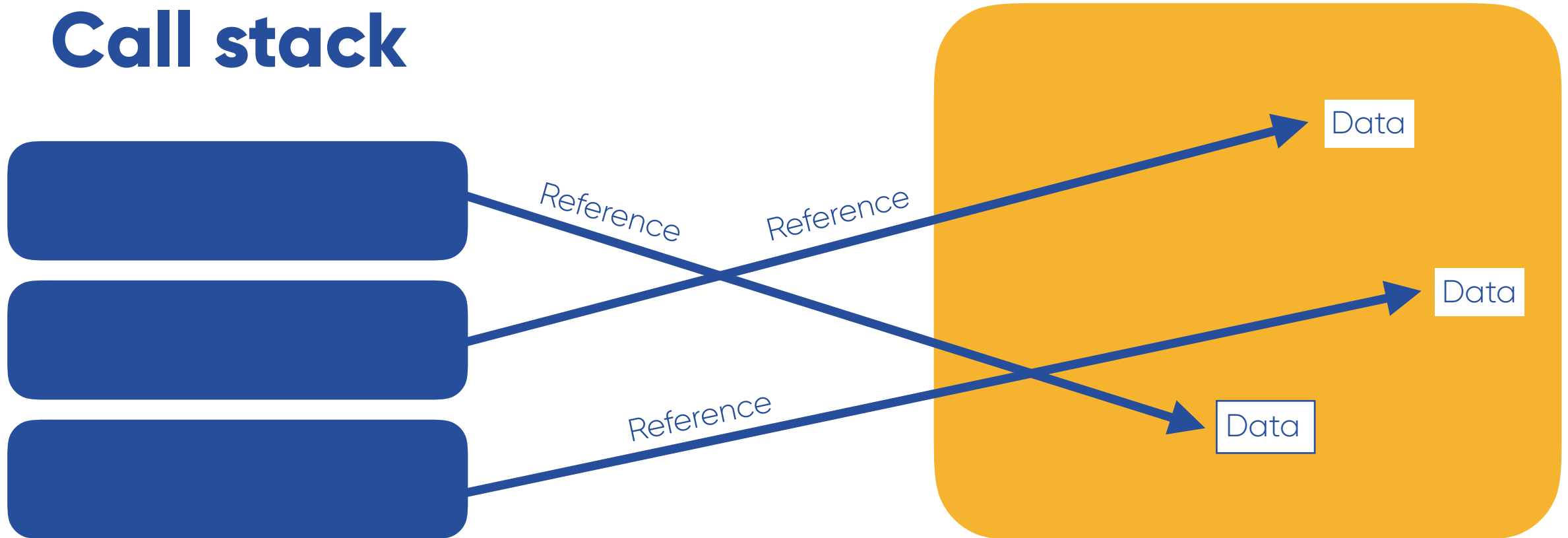


When a function has finished running, it is removed from the call stack, along with its reference to the memory.

Memory Heap

The data and objects our functions and variables point to.

Call stack

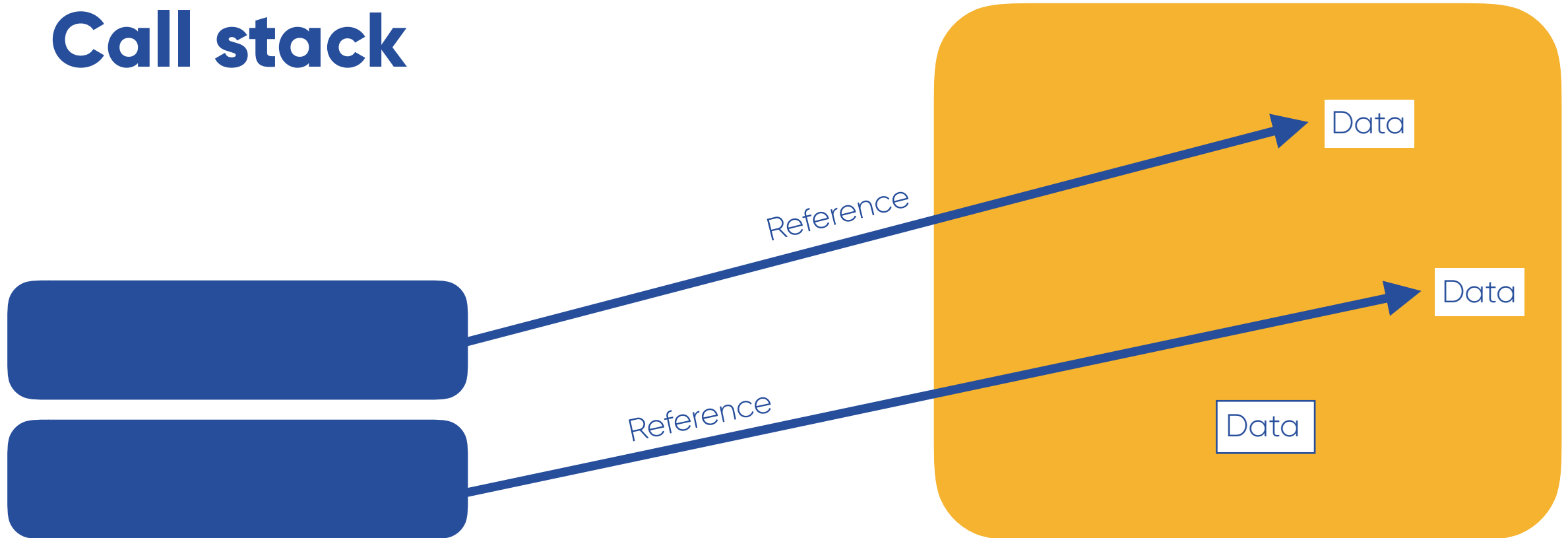


Once the reference has been removed,
the data is swept from the memory
heap, because it's not longer needed.

Memory Heap

The data and objects our
functions and variables point to.

Call stack



**Why are we
even talking
about this?**

A **local variable only exists in the memory when the function gets called, but is removed from the memory when the function has finished.**

This is why we can't access variables outside the function's scope. They don't exist!

**As we move down
our code – if the
variable exists, we
can access it.**

```
let a = 1
let b = 2
let c = 3
```

```
console.log(a)
console.log(c)
```

```
const myFunction = () => {
  console.log(b)
```

```
  const myFunction2 = () => {
    let d = 4
    console.log(c)
    console.log(d)
  }
```

```
myFunction2()
console.log(c)
console.log(d)
}
```

```
myFunction()
```

Let's follow it down.

Global variables are not removed from the memory in this way. That's why you may hear that global variables are dangerous! You run the risk of using up all the memory.

SO. Recap.

When a function runs and creates a local variable, it is stored in memory with a reference. When the function ends, the reference is removed – and the memory allocation is freed.

...However.

There is a pretty awesome way in javascript to retain a reference to local variable, even after a function has ended! It's called Closure, and we'll look at it in the next session.

But first...

**Higher Order
Functions!!**

Higher Order Functions are...

Functions which accept a function as a parameter

OR

Functions which return a function.



Let's look at
functions which take
a function **as a**
parameter.



Let's take this in.

```
let whichGreeting = (timeOfDay) => {  
  console.log(`Good ${timeOfDay}`)  
}  
  
let greet = (time, fn) => {  
  if (time < 12) {  
    fn("Morning")  
  } else if (time > 12 && time < 18){  
    fn("Afternoon")  
  } else {  
    fn("Evening")  
  }  
}  
  
greet(14, whichGreeting)  
// returns Good Afternoon
```



The difference between invoking a function and referencing a function

```
let add = () => {  
  return 2 + 3  
}
```

```
add() // returns 5
```

```
add // without the brackets just holds a reference the function
```

We can use console.log to see the difference

```
let add = () => {  
  return 2 + 3  
}
```

```
console.log(add())
```

```
console.log(add_
```

5

```
▼ () => {  
  return 2 + 3;  
}
```

Prove it to yourself:

Create a simple function which returns some data.

**Console.log the function call,
Then console.log the function reference.**



```
let whichGreeting = (timeOfDay) => {  
  console.log(`Good ${timeOfDay}`)  
}
```

```
let greet = (time, fn) => {  
  if (time < 12) {  
    fn("Morning")  
  } else if (time > 12 && time < 18){  
    fn("Afternoon")  
  } else {  
    fn("Evening")  
  }  
}
```

```
greet(14, whichGreeting)  
// returns Good Afternoon
```

Reference to the whichGreeting function is passed to the greet function.

The whichGreeting function is then invoked inside the greet function.



Jeez, I know right.

Take a breath.



Activity:

Create a higher order function, which takes a function as a parameter. Let us know when you think you've got it working and we'll come and check it out.

I'll leave the example up for you to look at.



```
let whichGreeting = (timeOfDay) => {  
  console.log(`Good ${timeOfDay}`)  
}  
  
let greet = (time, fn) => {  
  if (time < 12) {  
    fn("Morning")  
  } else if (time > 12 && time < 18){  
    fn("Afternoon")  
  } else {  
    fn("Evening")  
  }  
}  
  
greet(14, whichGreeting)  
// returns Good Afternoon
```

Reference to the whichGreeting function is passed to the greet function.

The whichGreeting function is then invoked inside the greet function.



We've basically levelled up our functions.

Now, not only can we tell them **what data to use, but we can tell them **exactly what to do** - by making use of other functions.**



**What about
functions which
return functions?
Strap in.**



What is the return value?

```
let myFunction = () => {  
  return () => {  
    console.log("hello")  
  }  
}
```

```
console.log(myFunction())
```

**Write this
function and tell
me what is
logged in the
console.**




What is the return value?

```
let multiply = (num1) => {
```

```
  return (num2) => {  
    return num1 * num2
```

```
  }  
}
```

```
multiply(2)
```



```
(num2) => {  
  return 2 * num2  
}
```

```
let multiply = (num1) => {
```

```
  return (num2) => {  
    return num1 * num2
```

```
  }
```

```
}
```

```
multiply(2)
```



```
multiply(2) = (num2) => {  
  return 2 * num2  
}
```

So basically, the function call becomes the name for the returned function.


Remind me. How do we invoke/call a function?

NOT real code, its just here so you can visualise it!



```
let multiply = (num1) => {  
  return (num2) => {  
    return num1 * num2  
  }  
}
```

`multiply(2)(4)`



num1 num2

Now **multiply(2)** is equal to the returned function.

So to call this returned function we can run **multiply(2)(4)**




```
let multiply = (num1) => {  
  return (num2) => {  
    return num1 * num2  
  }  
}
```

What we would usually do however, is **store multiply(2) inside a variable.**

Then we can use that variable name to call the function

```
let myReturnedFunction = multiply(2)  
myReturnedFunction(4)
```



We have looked at Higher Order Functions.

These are functions which take a function as parameter or return a function (or both)



Challenge 1:

Write a simple function which logs "Hello Code Nation" to the console.

Then write a higher order function which will run our simple function 5 times, even though you only call it one time.

Hint: pass the simple function as a parameter, and this will involve a for loop.



Challenge 2:

Write a simple function which has two parameters and returns their sum (basically takes in two numbers, adds them together and returns the result)

Then create a higher order function, which takes three parameters (a function and two numbers). Use this higher order function to run your simple function 5 times, each time adding the result to a **total** variable. Then return the total variable.

Show the result in the console.

Hint: similar-ish to challenge 1



Challenge 3:

The array method **map** is an example of a higher order function.

Declare an array with five numbers, then use **map** to iterate through the array and multiply each array item by 3.



Challenge 4 (conceptually trickier):

Test this function to make sure it works by passing a number to the doMath function, then passing a number and one of the four maths functions, to the returned function. You'll need to write this code out.

```
const multiply = (a,b) => {  
  return a*b  
}
```

```
const add = (a,b) => {  
  return a+b  
}
```

```
const divide = (a,b) => {  
  return a/b  
}
```

```
const subtract = (a,b) => {  
  return a-b  
}
```

```
const doMath = (num1) => {  
  return (num2, fn) => {  
    return fn(num1, num2)  
  }  
}
```

Revisiting Learning Objectives

To understand what **variable scope** is in Javascript.

To understand what happens with our variables in the **memory**.

To be able to work with **Higher Order Functions**.