TABLE OF CONTENTS

1. GETTING STARTED
2. Introduction
3. How to Execute JavaScript (Where does JS run?)
4. JavaScript Versions (ES5 vs ES6
5. Course Format
6. LANGUAGE BASICS
7. Introduction
8. Using JavaScript - Inline HTML
9. Using JavaScript - HTML Import
10. Variables - Basics
11. Variables - null, undefined and NaN
12. Variables - Objects
13. Strict Mode
14. Dynamic Typing
15. Hoisting
16. Functions - Creating and Calling Them
17. Functions - Arguments & Returning Data
18. Control Structures - If Statement
19. Control Structures - If Statement Advanced
20. Control Structures - Switch
21. Control Structures - Switch & Break
22. Control Structures - For Loop
23. Control Structures - Nested Loops
24. Control Structures - Controlling Loops with break & continue
25. Control Structures - For Loop Variations
26. Control Structures - Looping through Arrays
27. Control Structures - While Loop
28. Operators - Addition
29. Operators - Subtraction
30. Operators - Multiplication & Floating Point Problems
31. Operators - Division & Modulus
32. Operators - Comparing Values
33. Operators - Important Rules
34. Operators - Boolean
35. Operators - Ternary Operator
36. Operators - Precedence
37. TYPES & SCOPE
38. Primitive vs Reference Types
39. Primitive vs Reference Types - Exercise

40. Global vs Local Scope
41. Global vs Local Scope - Exercise

42. ARRAYS

43. Introduction
44. Basics & Managing Elements
45. The forEach() Method
46. Working with Elements: Push, Pop, Unshift and Shift
47. Working with Parts of an Array: IndexOf, Splice and Slice
48. Filtering, Mapping and Reversing Array Elements
49. The Difference between Concat and Join
50. Reducing Arrays

51. OBJECTS

52. Object Basics & Literal Notation
53. Properties & "this"
54. Alternative Way of Creating Objects: Using the Object Constructor
55. Objects are Reference Types (Refresher)
56. Object.create()
57. Prototypes Introduction
58. Prototypes in Action
59. Prototypes Summary
60. Constructor Functions
61. Constructor Functions & Arguments
62. Object Creation & Prototypes Summary
63. The "this" Keyword and why it might behave strangely
64. Handling "this" with bind(), call() & apply()
65. Creating Properties with defineProperty()
66. Important built-in Methods and Properties

67. ERRORS & DEBUGGING

68. Debugging with the Browser Developer Tools
69. Logging Data to the Console
70. Handling Errors with try and catch

71. FUNCTIONS

72. Introduction
73. Closures
74. Immediately Invoked Function Executions (IIFEs)
75. Built-in Methods & Properties

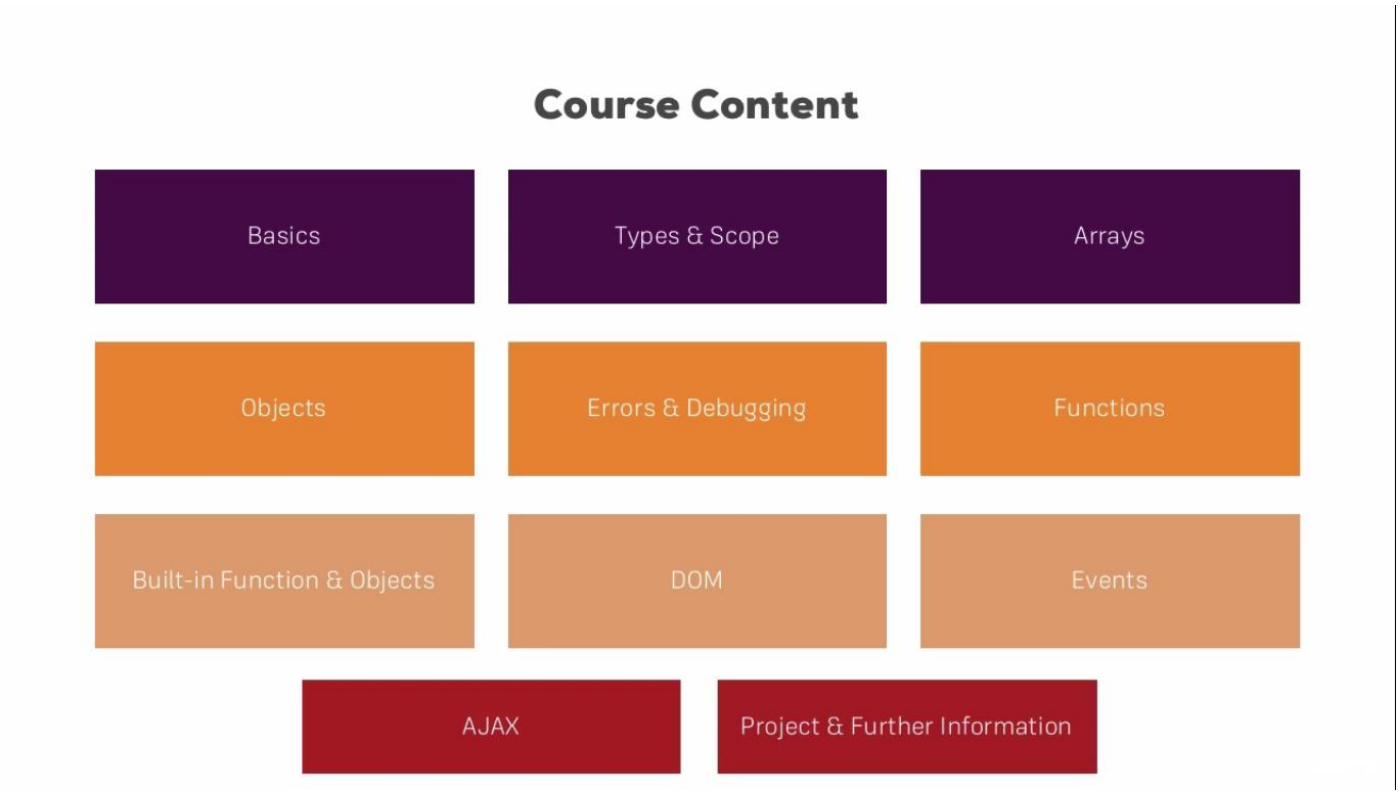
76. BUILT-IN OBJECTS & FUNCTIONS

77. Introduction
78. Timers & Intervals
79. Transforming Formats & Values
80. String Functions

81. The Math Object
82. The Date Object
83. Regular Expressions
84. WORKING WITH THE WINDOW AND DOCUMENT OBJECT MODEL (DOM)
85. Introduction
86. The Window Object
87. The Location Object
88. The Document Object and How to Interact with It
89. Traversing the DOM
90. Selecting Elements
91. Selecting Elements with the Query Selector
92. Selecting Elements - Exercises
93. Creating and Inserting Elements
94. Deleting Elements
95. Elements and Nodes
96. DOM Interaction Summary
97. Dialogs
98. EVENTS
99. Introduction
100. The Event Object
101. Event Handlers
102. Event Listeners
103. Event Behavior
104. Event Object Properties
105. Changing Propagation Order
106. JAVASCRIPT AND HTTP REQUESTS (AJAX)
107. Introduction
108. Setup and Sending a GET Request
109. POST Request
110. LIBRARIES, FRAMEWORKS AND MODULES
111. Introduction
112. Libraries & jQuery
113. Frameworks
114. Writing Modular Code
115. COURSE PROJECT
116. Introduction
117. Setup
118. Fetching User Input
119. Http Request Setup
120. Creating a Weather Object
121. Fetching & Handling Weather Data

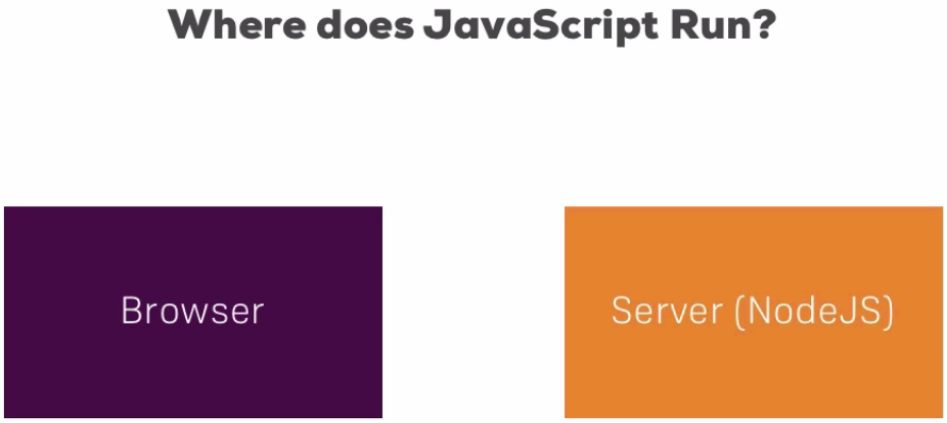
GETTING STARTED

INTRODUCTION: -



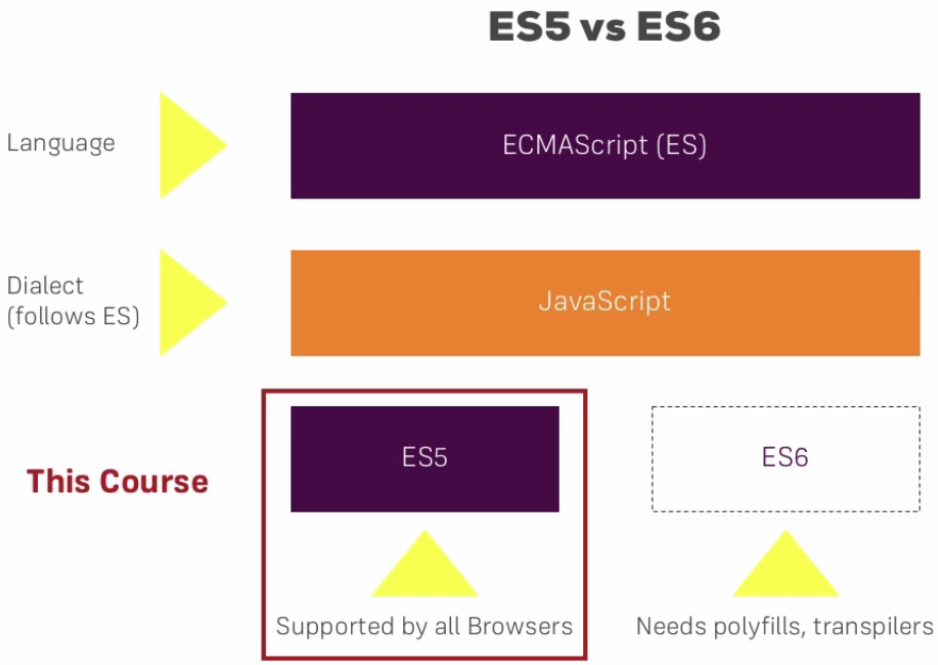
HOW TO EXECUTE JAVASCRIPT WHERE DOES JS RUN?

Javascript runs in the browser and this makes it widely popular because it doesn't need to wait for server to run. Javascript can also run through the server with node js but we will be focusing on the browser part in this course.

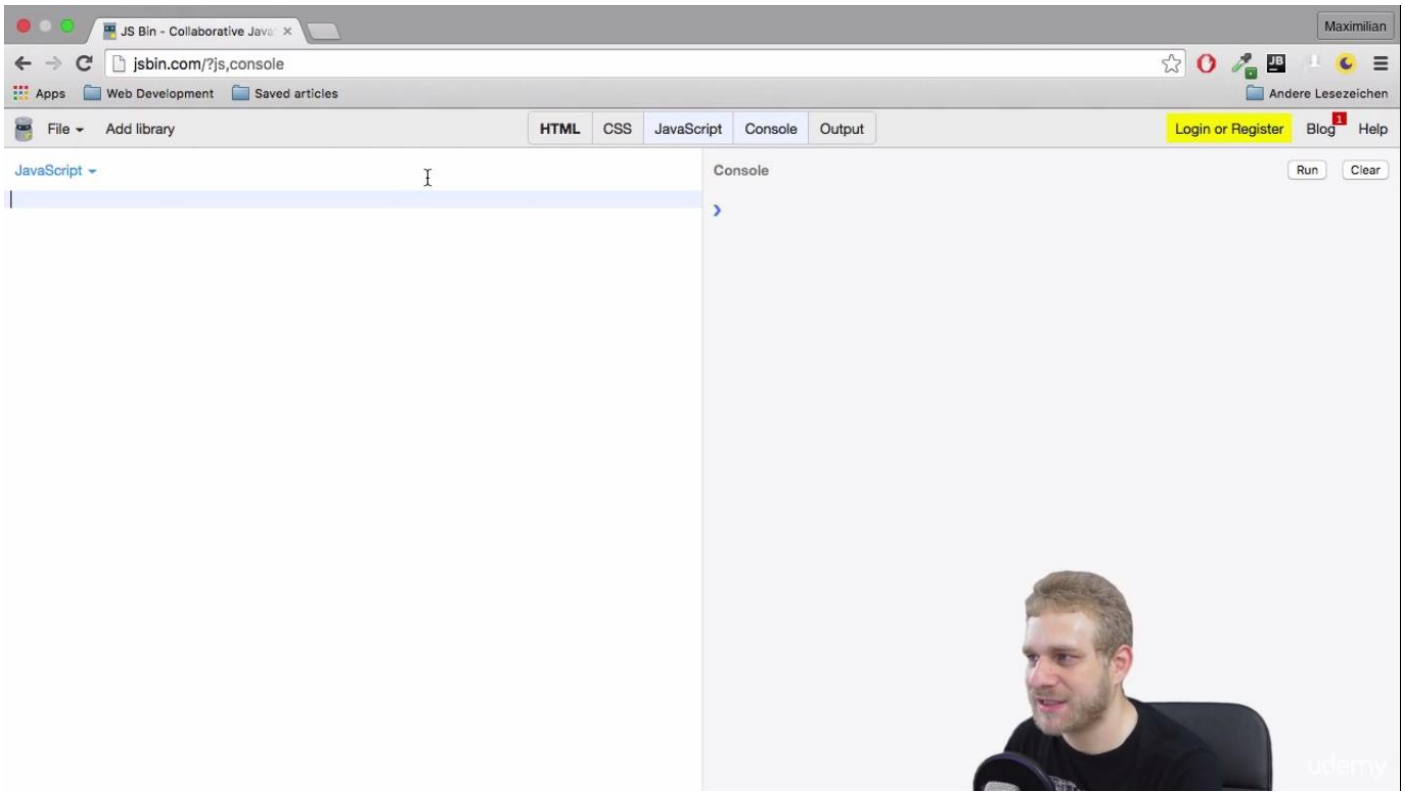


JAVASCRIPT VERSIONS ES5 VS ES6: -

ES stands for ECMAScript from where Javascript was derived. Javascript is a dialect of ES. ES5 is supported by all browsers and it's the latest version while ES6 is the next big thing with some additional cool feature but needs polyfills or transpilers to run. However we will be focusing on ES5 in this course.



COURSE FORMAT: - We will be using www.jsbin.com for this course.



- **Ensure you only click the JavaScript and Console tab so that they are the only one you have opened in the browser.**
- **Later we will be talking about setting our workspace because you can't use jsbin for a real project but for now we will be using jsbin because its good for learning.**

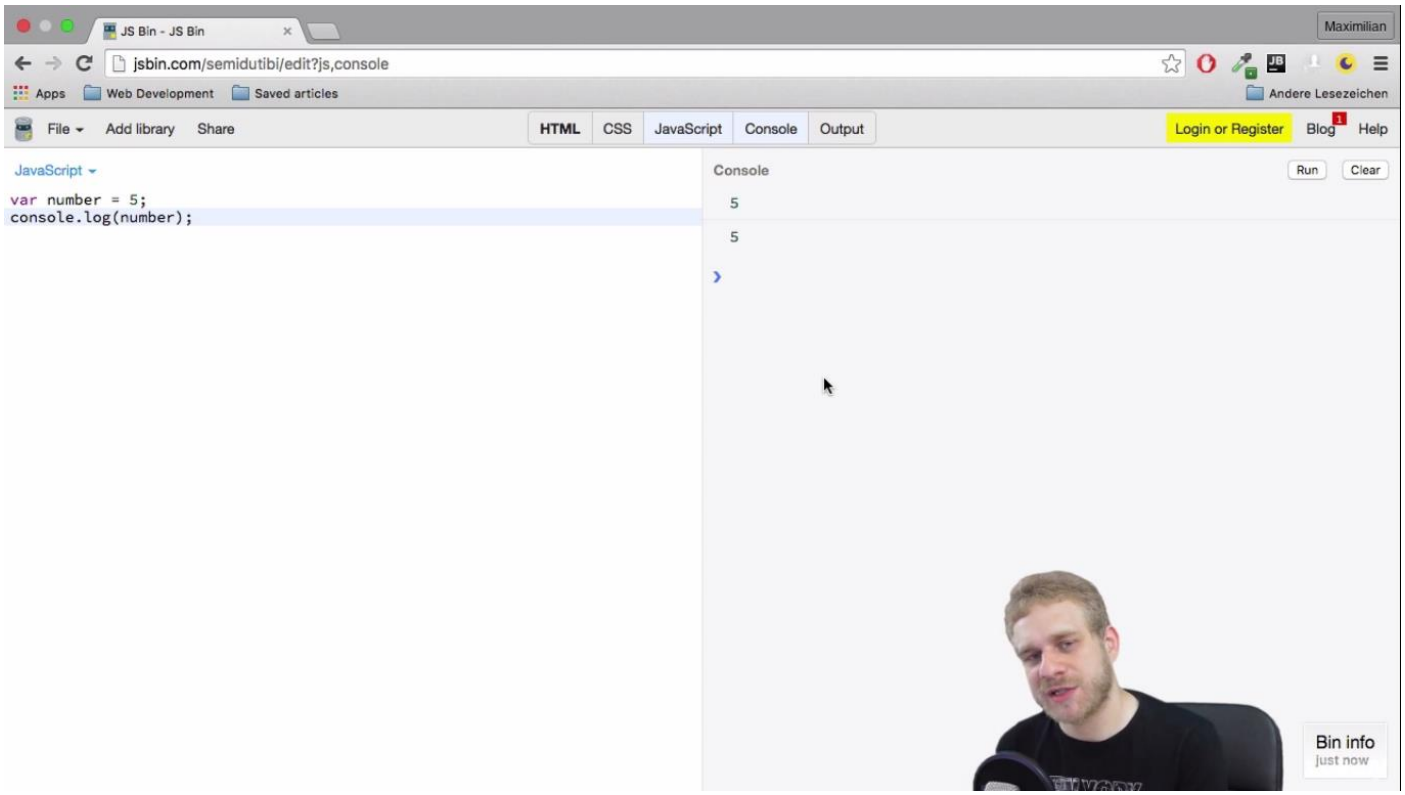


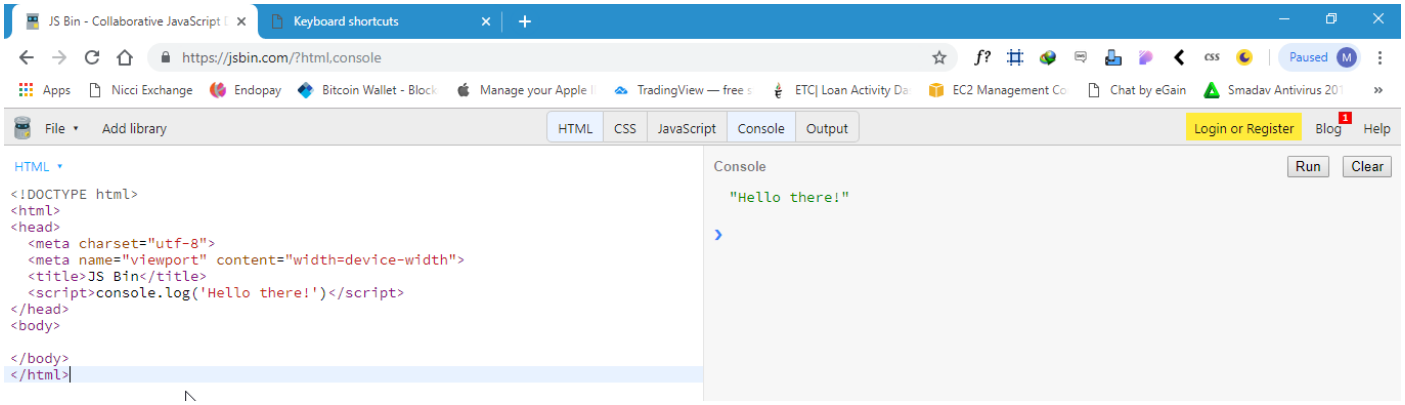
Image 1: Running jsscript and logging to the console with jsbin

LANGUAGE BASICS

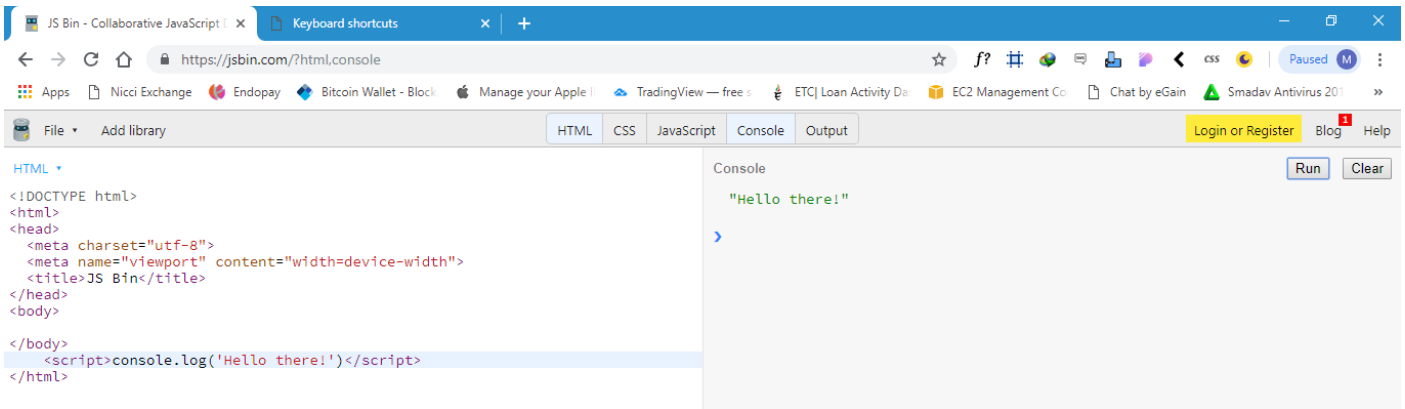
INTRODUCTION: - We will look at the basic of javascript that allows you to write awesome apps.

USING JAVASCRIPT – INLINE HTML: - We will look at how we can implement javascript in our html.

You can place your javascript inside the head tag:



You can also place your jsscript before the closing body tag



NOTE: Best practice is placing your js here before the closing body tag because when the code is compiled, it starts running from top to bottom and if you have chunks of js code inside the head tag, the rest of your html code might not be able to run.

DISPLAYING MESSAGE: - If your web page rely heavily on javascript, it might happen that the user might turn off javascript in the browser. In a situation like this, you will have to display a message to the user to turn javascript on for good experience.

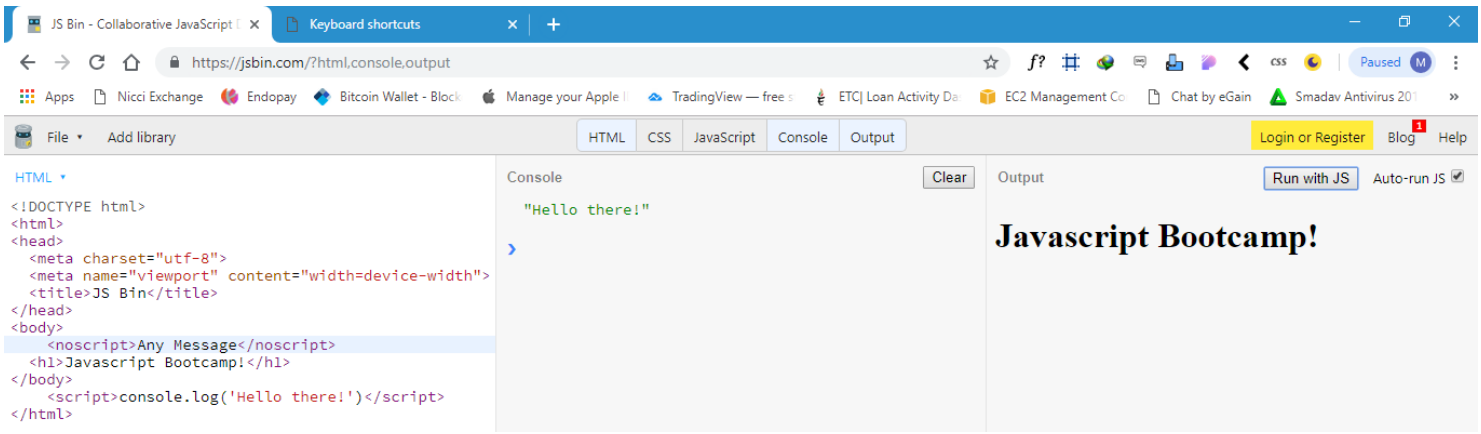
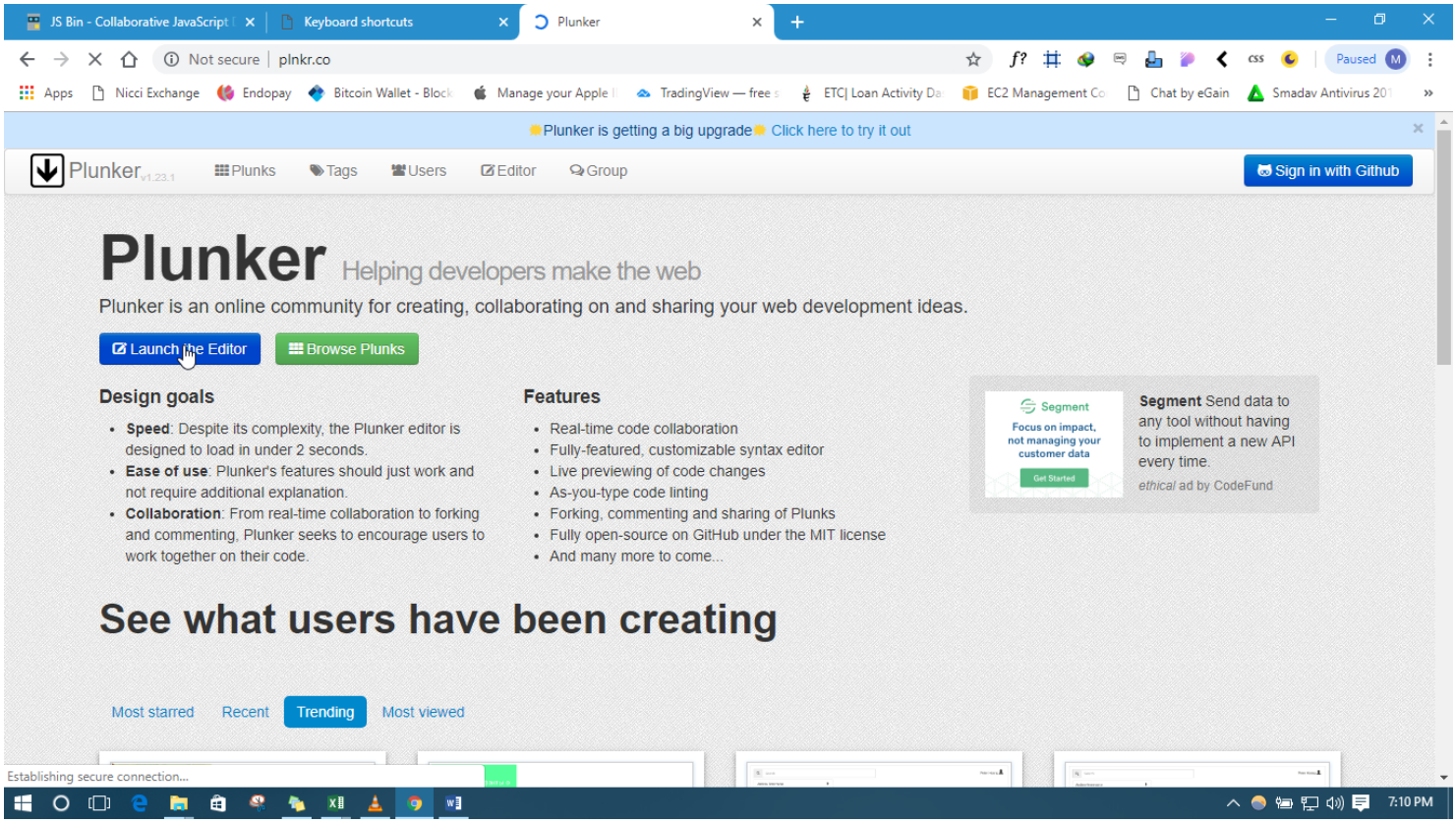


Image 2: Using <noscript>Any Message</noscript> to display msg to user when js is turned off in the browser

USING JAVASCRIPT – HTML IMPORT: - We've learnt how to put our js inside our html document but incase of a chunk of js code, it will be a bad practice if we put all the codes inside our html document because it might be hard to deal with such file as a developer.

In this case, we can outsource our js code and import it into our html document. For this, we cannot use jsbin.com instead we will use www.plnkr.co



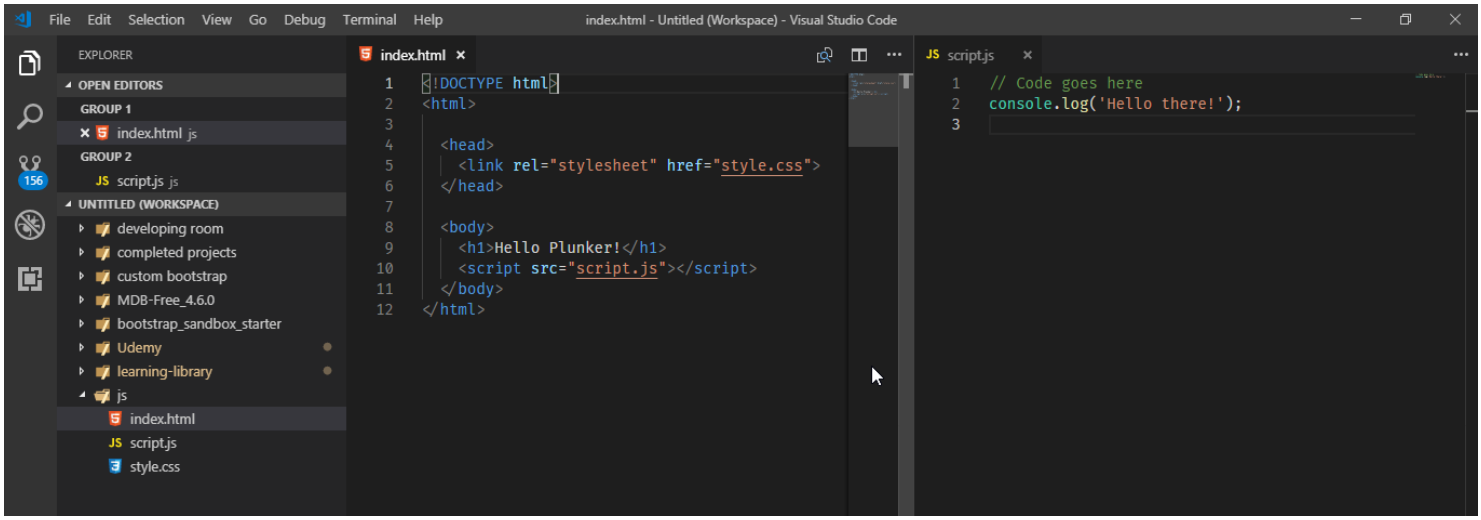
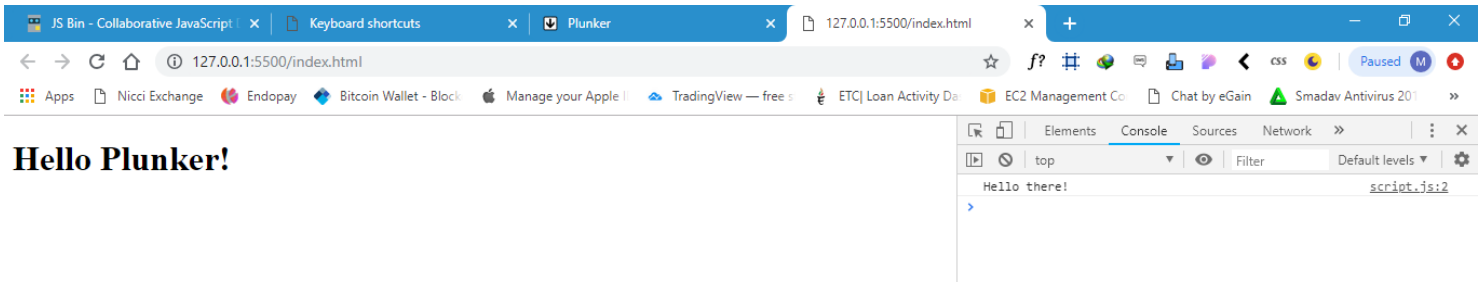
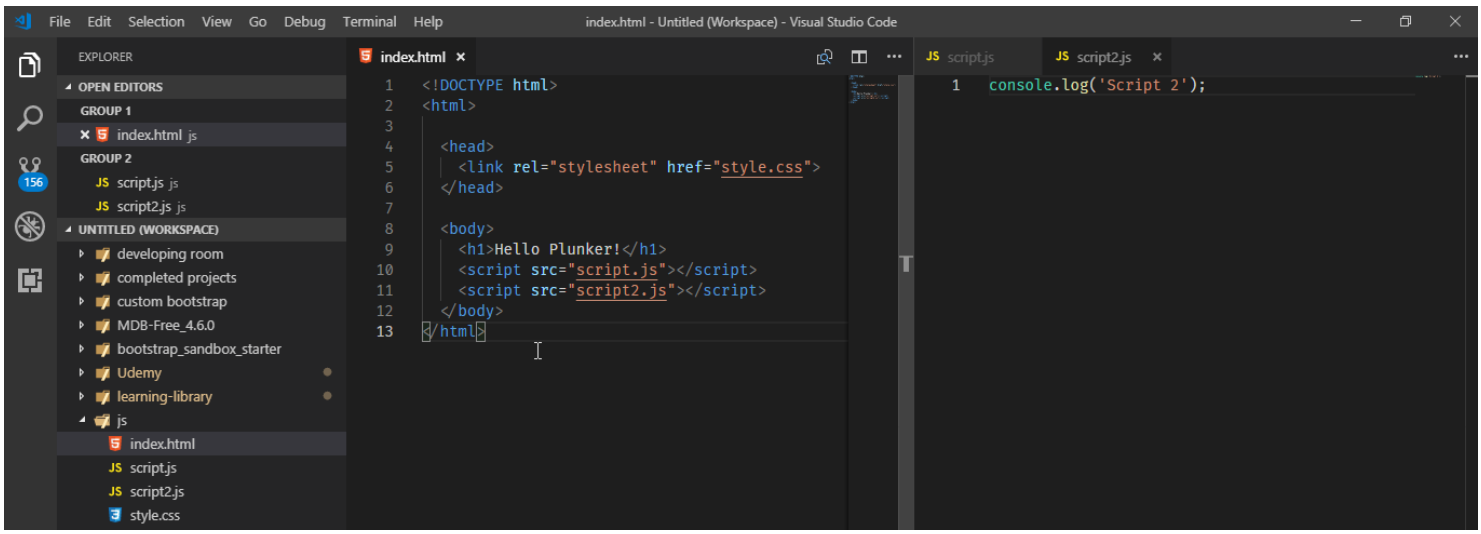
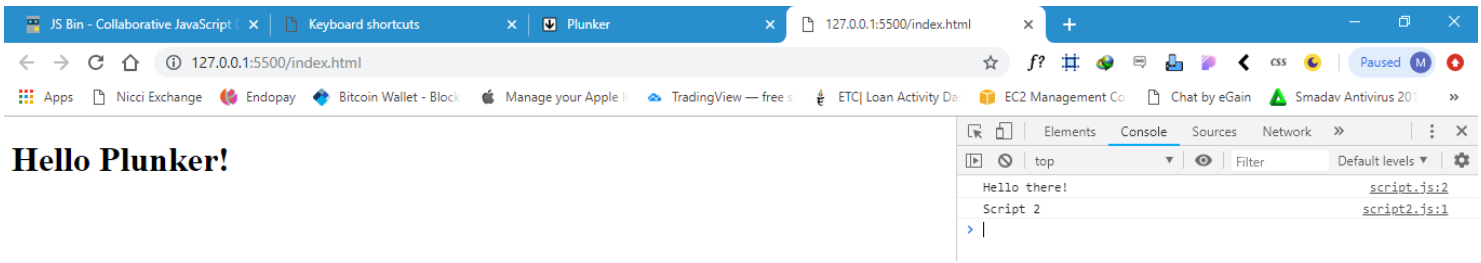


Image 3: JS html import



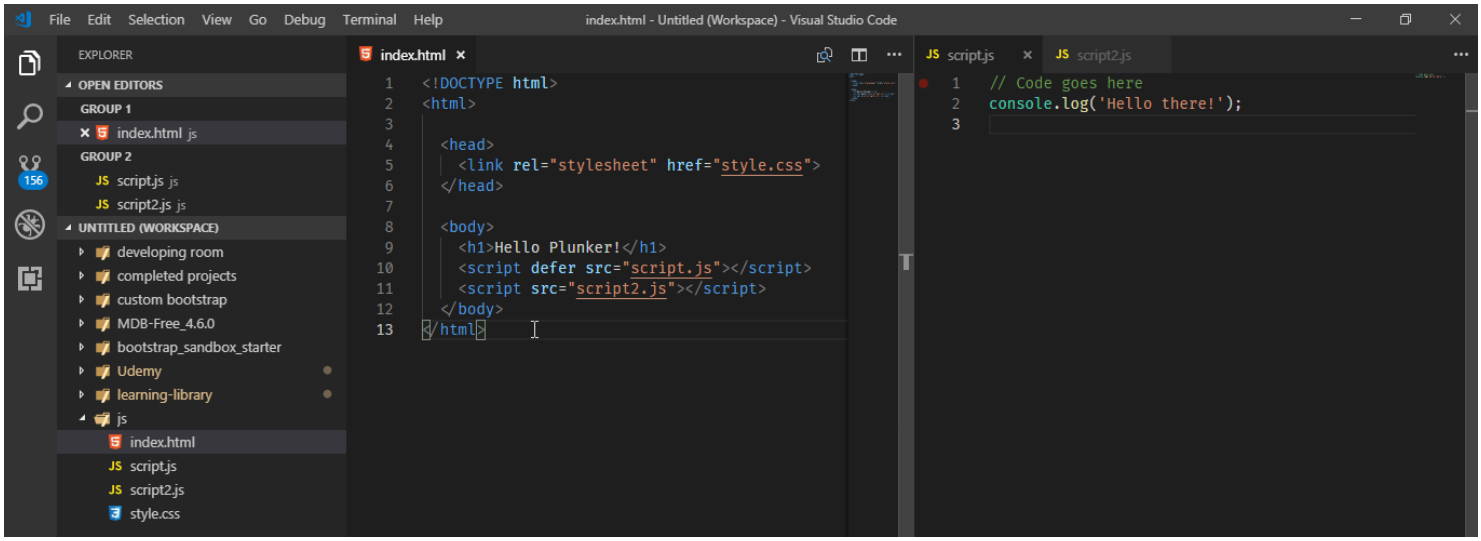
You can also import multiple js files:

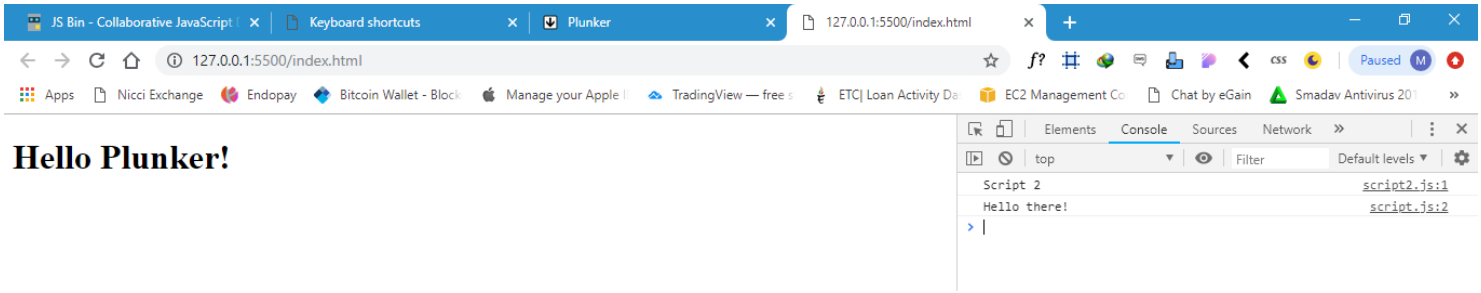




The order of these imports is very important, if you have a file depending on the other then the file on which the second file depends on must come first. Also if you use jquery in your html then the jquery import must also come first.

You can change the ordering of the js import by adding the tag "defer" to the first import in the list.





Script 2 now gets executed first even though it's the second on the list. THIS IS A BAD PRACTICE ANYWAY BECAUSE THE ORDERING SHOULD NOT BE TAMPERED WITH BECAUSE IT CAN CAUSE CONFUSION ESPECIALLY WHEN WORKING WITH OTHER DEVELOPERS.

VARIABLES – BASICS: - Variables are the very basic thing in each language that without them no code will work. Variables are data structures or storages which allows you to store values or data. Variables are defined in js with "var" and after which you assign a name to the variable and then a value, and then the assigned name can be used in any place in the file.

NUMBER

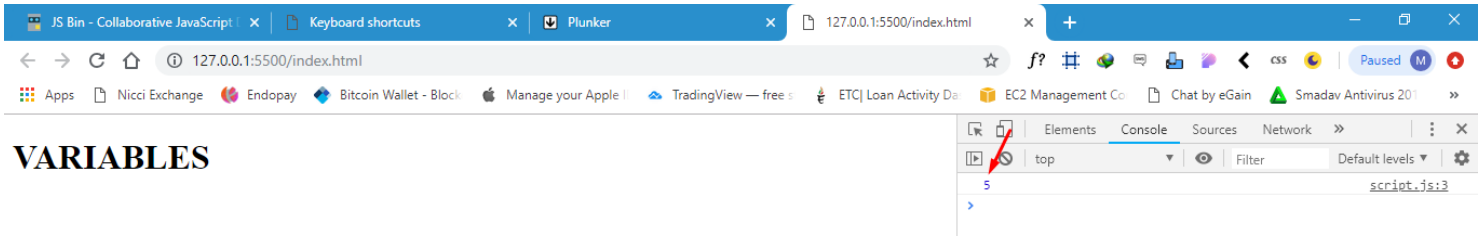
Script.js

```
// Code goes here
var variable = 5;
console.log(variable);
```

index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
```

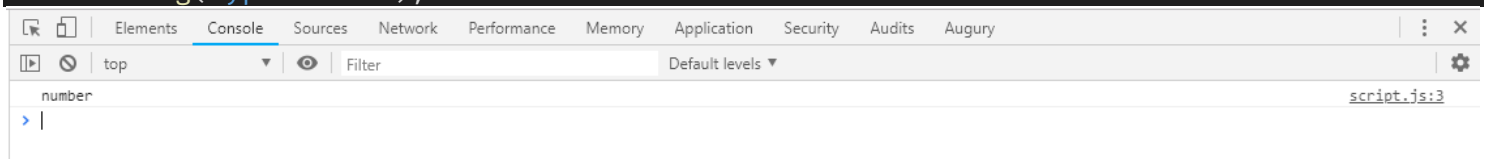
```
  </head>

  <body>
    <h1>VARIABLES</h1>
    <script src="script.js"></script>
  </body>
</html>
```



Number can also be in decimal or floating i.e. 6.5

Script.js

```
// Code goes here
var var1 = 6.5;
console.log(typeof var1);
```



WHICH DATA CAN WE STORE? Since variable stores data, we need to know which data we can store. In the previous example, we stored a number which we typed just as 5, assuming we want to store a string of character e.g. text.

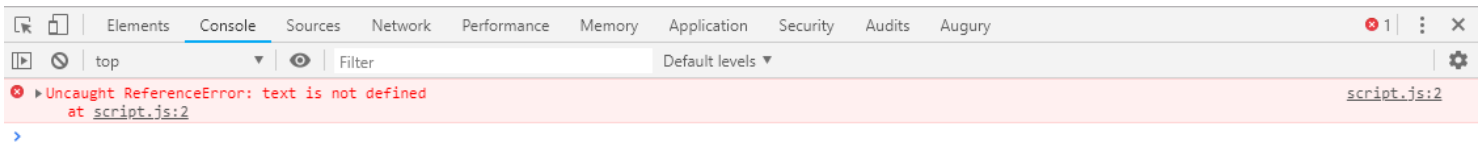
VARIABLES:

- **NUMBER**
- **STRING**
- **BOOLEAN (true/false)**

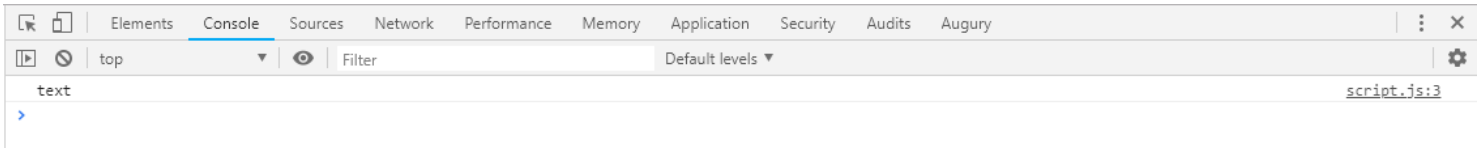
STRING

Script.js

```
// Code goes here
var variable = text;
console.log(variable);
```



Above we can see that we get an error that text is not defined. This is because to use string like this in js, we need to place them inside quotation marks, else js will see them as a new variable which we are yet to define.

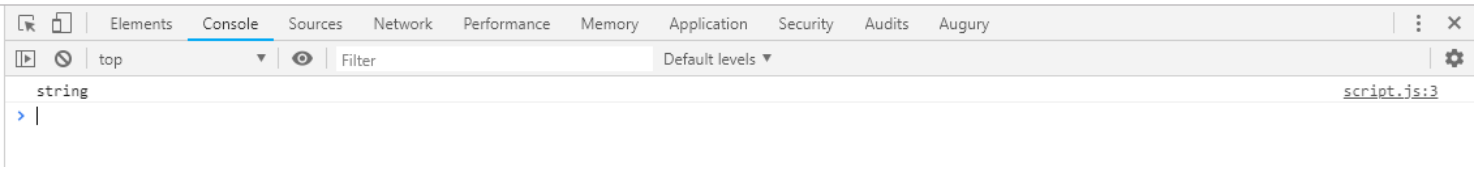


Script.js

```
// Code goes here
var variable = 'text';
console.log(variable);
```

Script.js (Variable type)

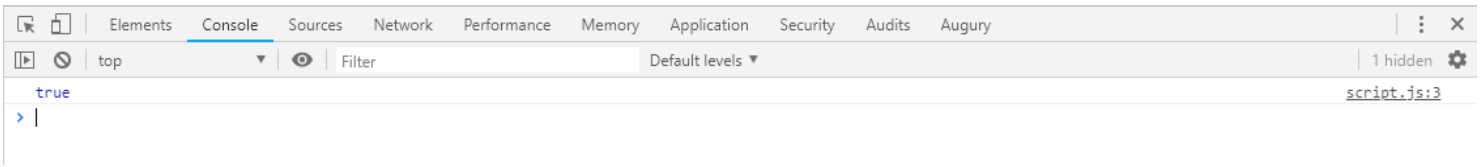
```
// Code goes here
var variable = 'text';
console.log(typeof variable);
```



BOOLEAN

Script.js

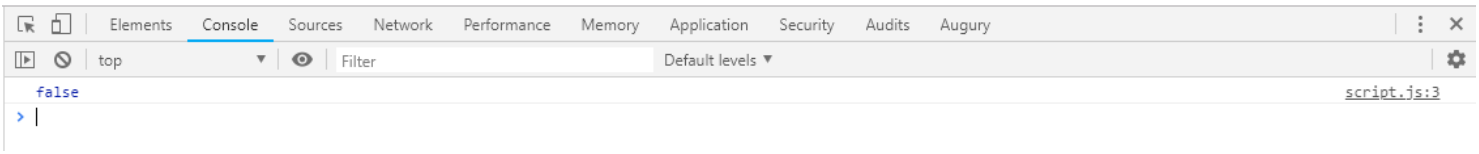
```
// Code goes here
var variable = true;
console.log(variable);
```



We see boolean (true/false) that it's written just like that without quotation marks and js didn't see it as variable and returned an error even though it did when we typed text without quotations. WHY? This is because Boolean are reserved js string that can't be used as variables but they will always return Boolean values.

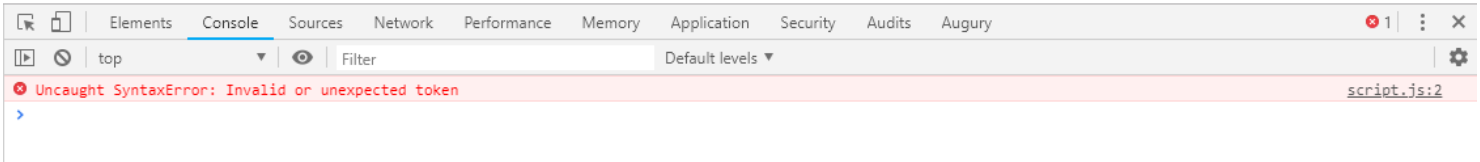
Script.js

```
// Code goes here
var variable = false;
console.log(variable);
```



FORMAT FOR WRITING VARIABLE NAMES: - You can include number in your variable name but NUMBERS CANNOT COME FIRST.

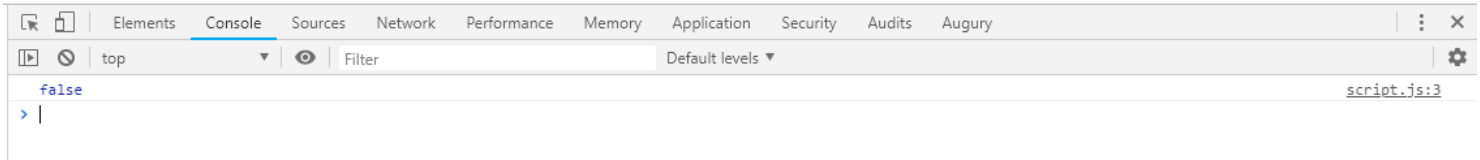
```
// Code goes here
var 1var = false;
console.log(1var);
```



We see here that we get an error because our variable name "1var" is invalid because it starts with number and also "var" is a reserved keyword on its own, but if we move the number to the back as "var1", now our variable should work as expected.

Script.js

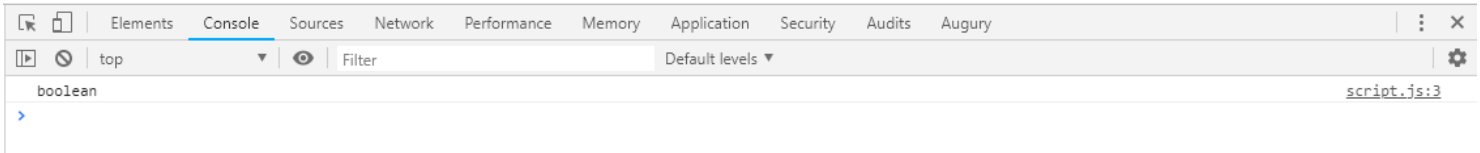
```
// Code goes here
var var1 = false;
console.log(var1);
```



GETTING THE VARIABLE TYPE IN THE CONSOLE: You can know the type of variable it is you are using by including the syntax "typeof"

Script.js

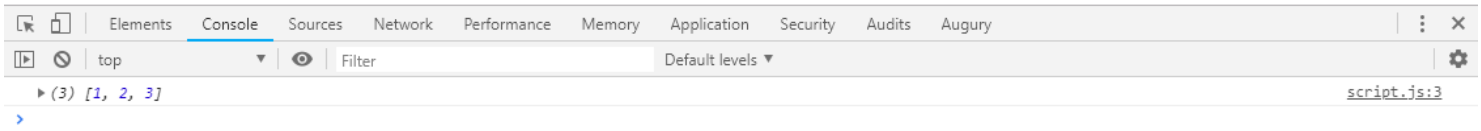
```
// Code goes here
var var1 = false;
console.log(typeof var1);
```



VARIABLE – null Undefined and NaN:

Script.js (Array)

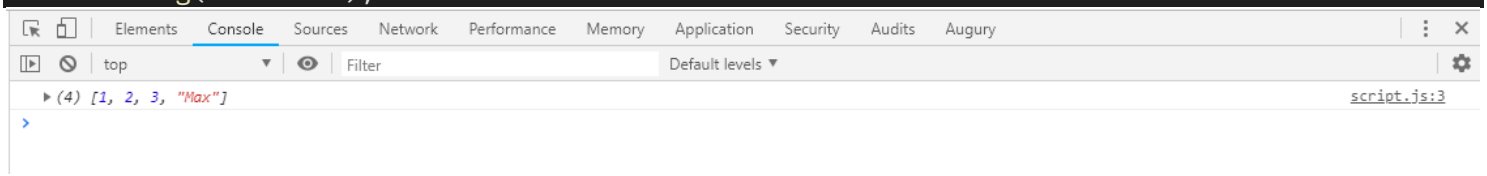
```
// Code goes here
var variable = [1, 2, 3]; //This is an array
console.log(variable);
```



Above we have an array type of variable, an array containing 3 numbers. The array can also include a string.

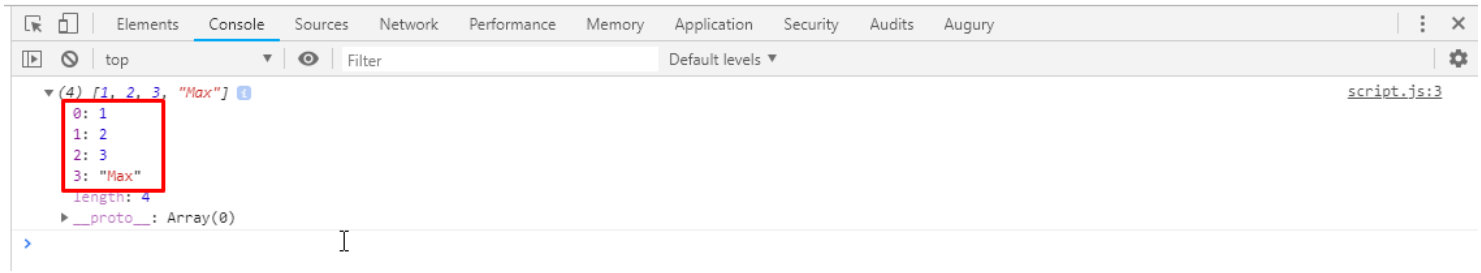
Script.js

```
// Code goes here
var variable = [1, 2, 3, 'Max']; //This is an array
console.log(variable);
```



Now we have an array having the length of 4 which contains 3 numbers and a string.

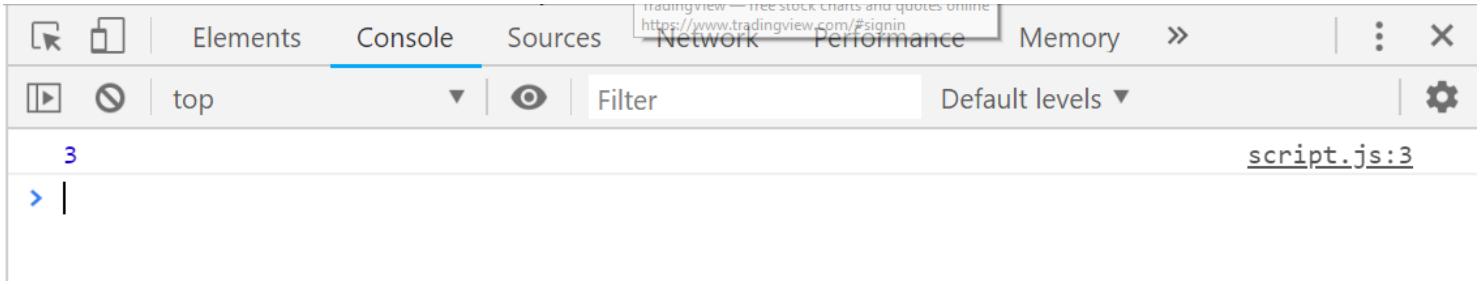
ARRAY INDEX: Each of the element in the array have its own specific index which we can use to reference it or to log only a specific number from the array into the console. Array index starts counting from 0 so in this case where we have 4 elements, the index will be 0,1,2 and 3. This can also be seen in the chrome console.



USING THE ARRAY INDEX TO REF AN ELEMENT FROM THE ARRAY: Using the index value, we can reference a specific element into the console.

Script.js

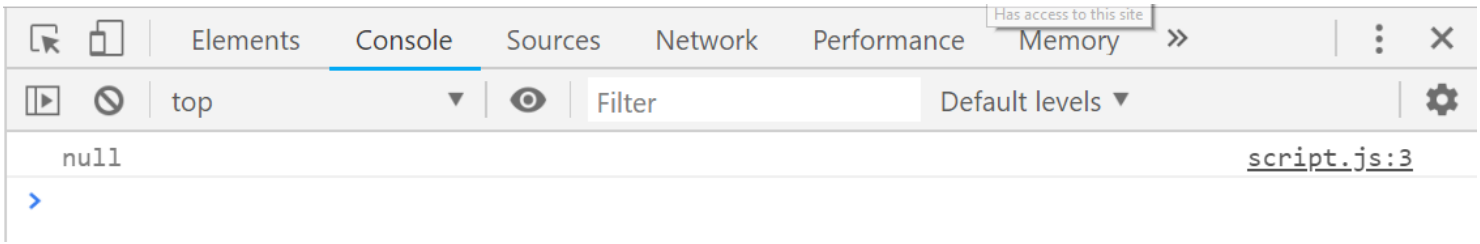
```
// Code goes here
var variable = [1, 2, 3, 'Max']; //This is an array
console.log(variable[2]); //refrencing the element with the index of 2 to the console.
cou
nting from 0, this element is 3
```



SETTING EMPTY VARIABLE: Cases might arise where we want to set an empty variable. This might be maybe we want to resolve or clear it later.

Script.js

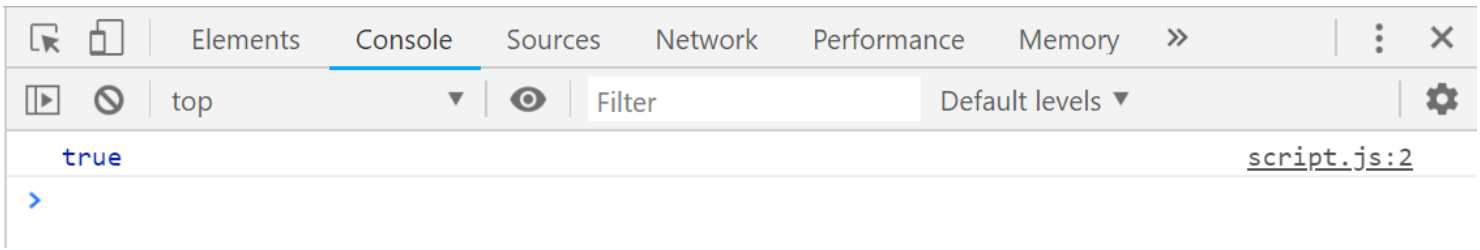
```
// Empty Variable
var variable = null;
console.log(variable);
```



This variable above now is not undefined but empty. In js, null and undefined has the same value but they are of different types. We can compare if null & undefined are of the same value by using 2 equal signs ==

Script.js

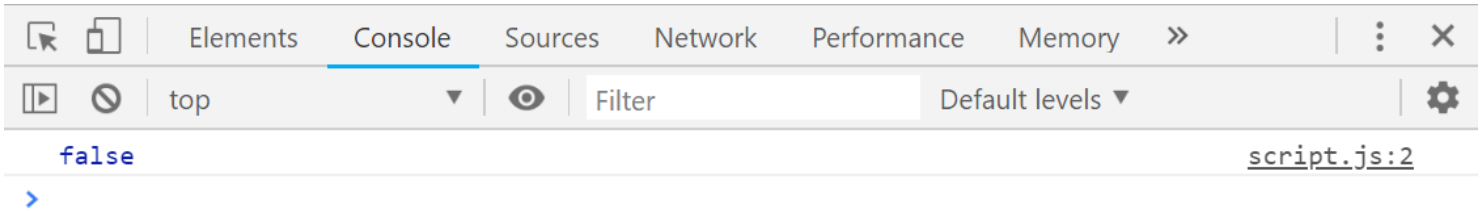
```
var variable = null;
console.log(null == undefined); //checking if null & undefined are of tyhe same value
```



We get true in the console which means they are of the same value. We can now check if they are of the same type by using 3 equal signs ====

Script.js

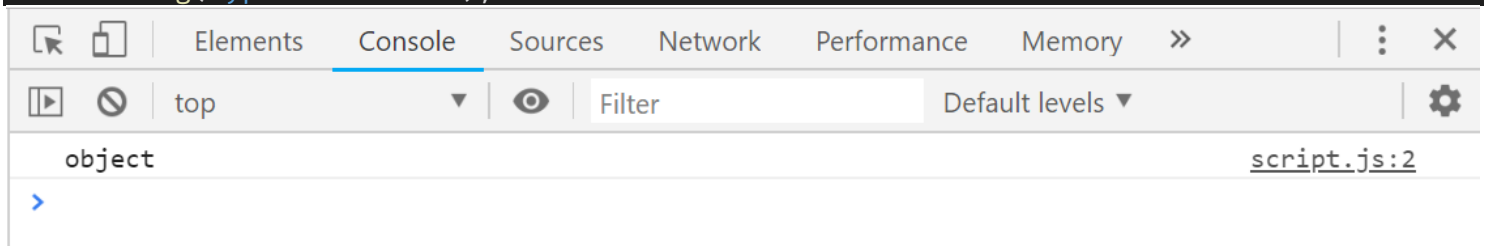
```
var variable = null;
console.log(null === undefined); //checking if null & undefined are of tyhe same value
```



We get false, which means they are of different type of variable but of the same value. WHY IS THIS SO?

Script.js

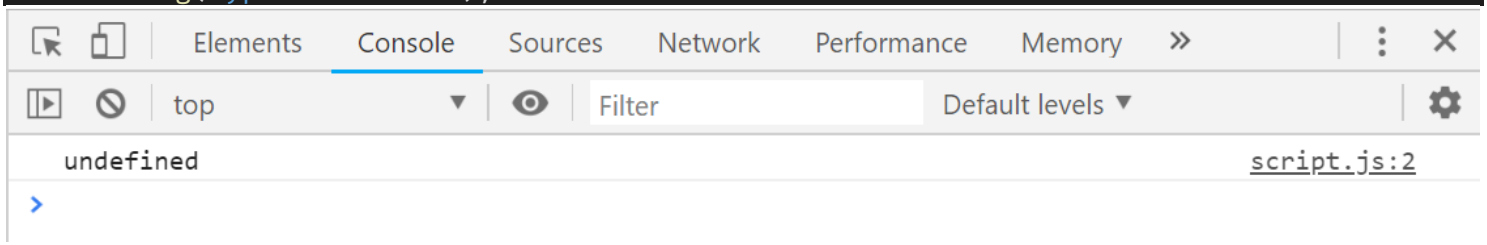
```
var variable = null;
console.log(typeof variable);
```



Here, null is of type "object". Now let's check for undefined.

Script.js

```
var variable = undefined;
console.log(typeof variable);
```

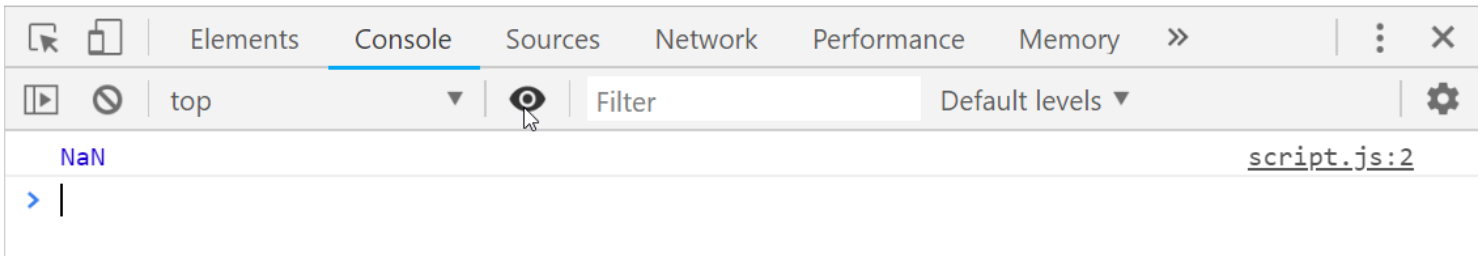


Now we see that undefined is of type undefined why null is of type object. This is while we get false when we compare them to be of the same type. They are of the same value but different type.

NaN: this is another type of variable which means "Not a Number"

Script.js

```
var variable = NaN;
console.log(variable);
```

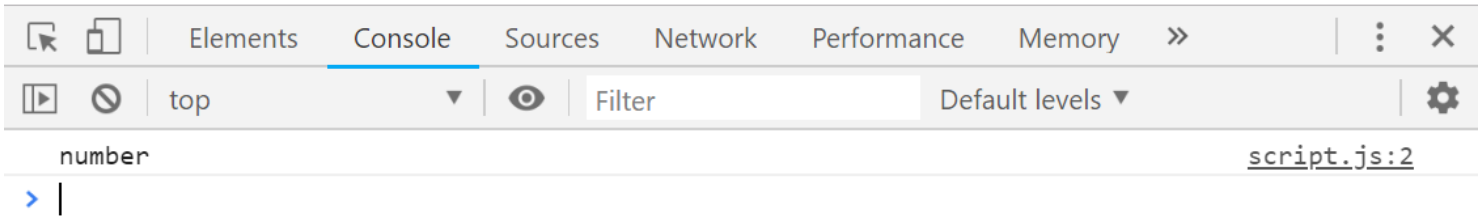


Though you cannot set a variable as NaN but it is used to output mathematical error messages for example, when you try to multiply a number with a string, the result will be NaN because it is Not a Number (NaN).

What type of variable will NaN be?

Script.js

```
var variable = NaN;
console.log(typeof variable);
```



This might look confusing that NaN (Not a Number) is of variable type "number". This is because, it is a result of a mathematical calculation so it is still a number but an undefined number. An error message from a mathematical calculation.

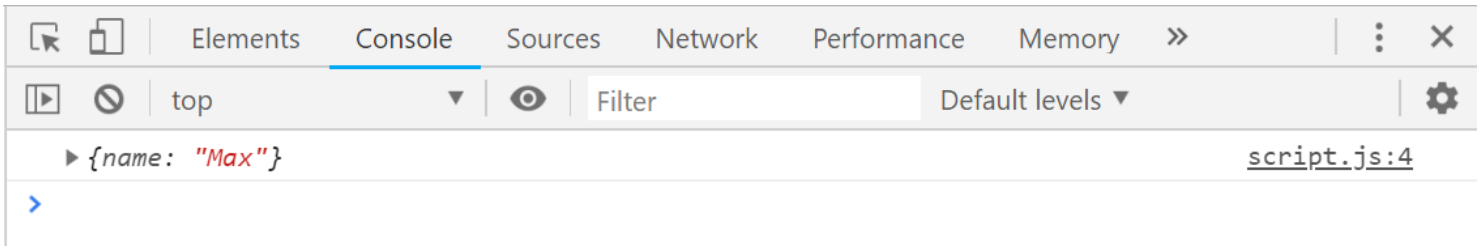
VARIABLES – OBJECTS: This is another value which can be assigned to variables just like number, Boolean, string, null & NaN but it's own syntax is quite different. It comes with the {}. Inside this {}, we have "fields/properties" and "methods".

Fields – this are variables inside an object also known as properties.

Methods – this are functions inside variables.

Scripts.js {Objects}

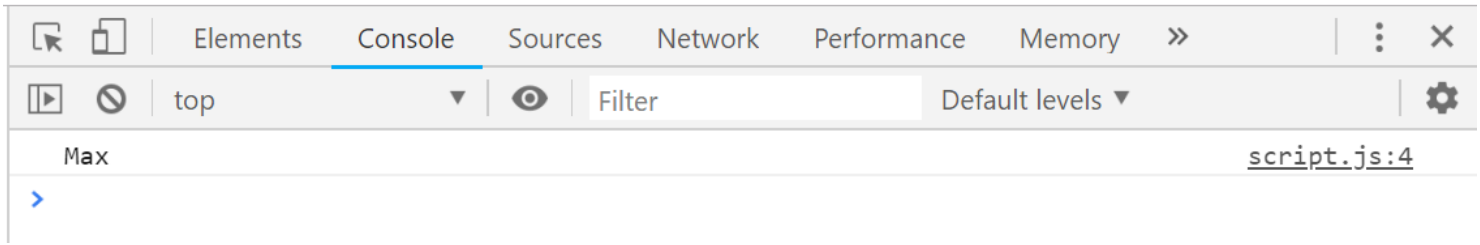
```
var variable = {
    name: 'Max'
};
console.log(variable);
```



Now we can reference the field "name" in the console using the syntax below.

Script.js

```
var variable = {
    name: 'Max'
};
console.log(variable.name);
```

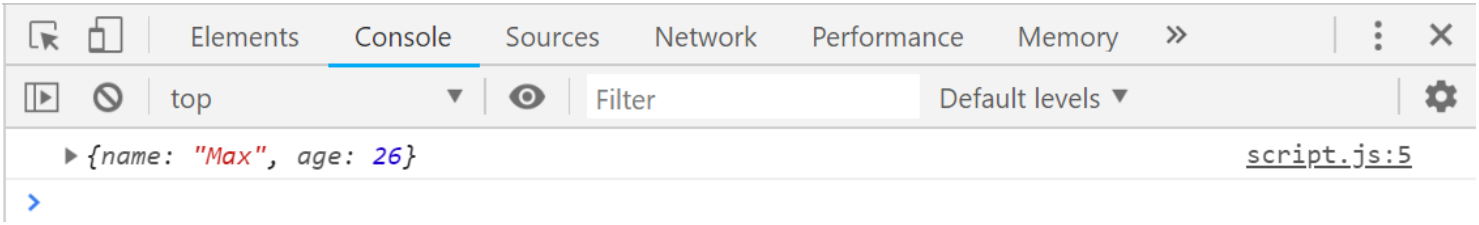


Now we get the value of the field 'name' in the console.

You can have multiple fields in your objects like below.

Script.js

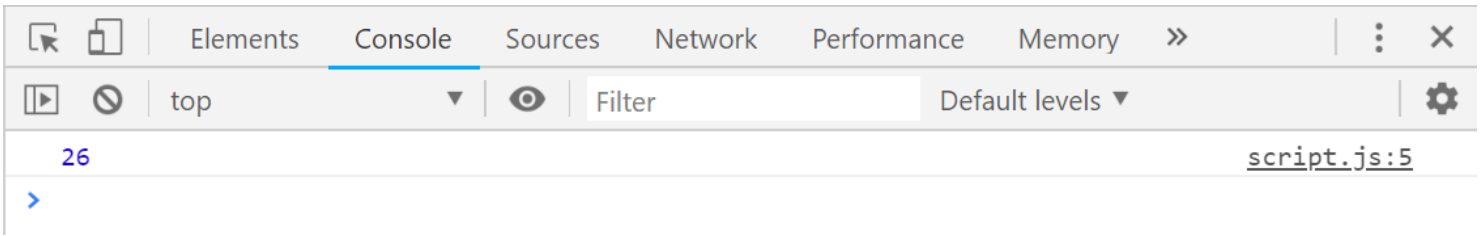
```
var variable = {
    name: 'Max',
    age: 26
};
console.log(variable);
```



You can also refrence any of the two fields e.g. age

Script.js

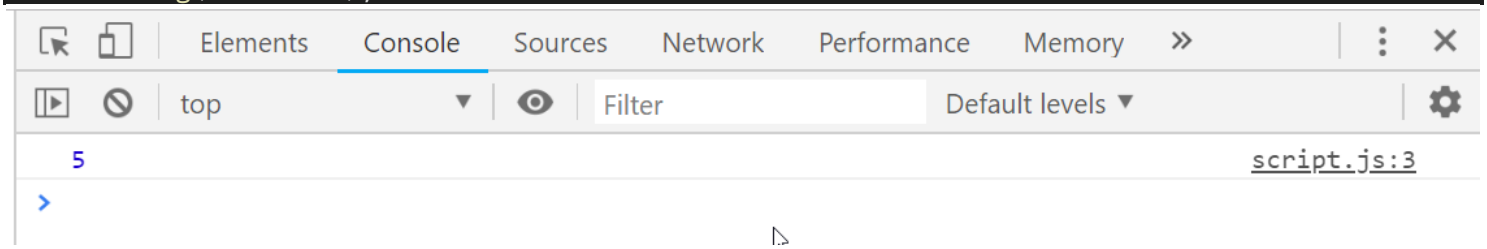
```
var variable = {
    name: 'Max',
    age: 26
};
console.log(variable.age);
```



STRICT MODE: Before we dive deeper, let's check some tactical things about js i.e. somethings we can change about the code we have been writing so far and check if they still run or not.

- **Removing the semicolon ;**
  Script.js

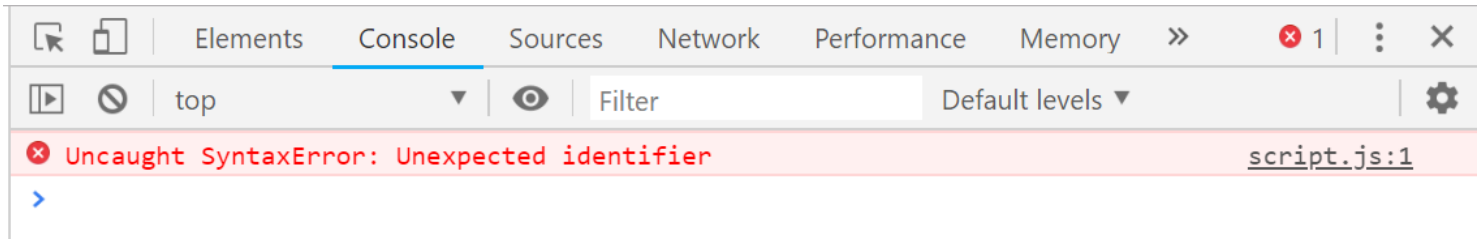
```
var variable = 5
//we are running this code without including ; at the end of the value
console.log(variable);
```



Our code still run successfully. This is because js is able to determine when our lines ends but if we write all the code on the same line like below, we get an error because the line never ends.

Script.js

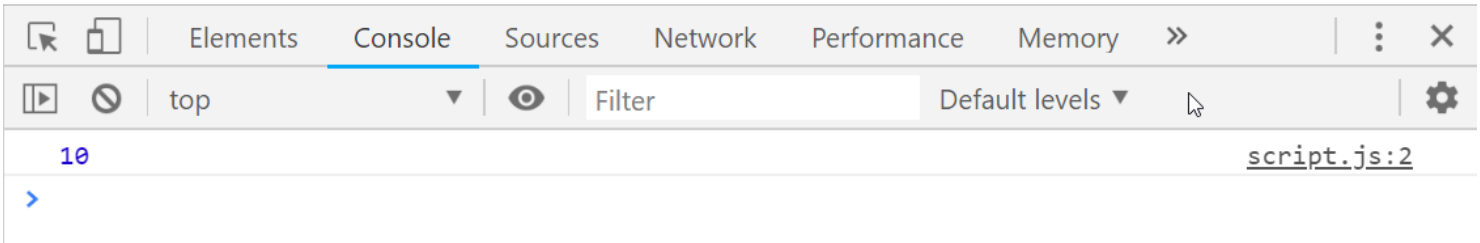
```
var variable = 5 console.log(variable)
```



Removing the 'Var' keyword: we can also try to remove the var keyword we adds before writing the variable name.

Script.js

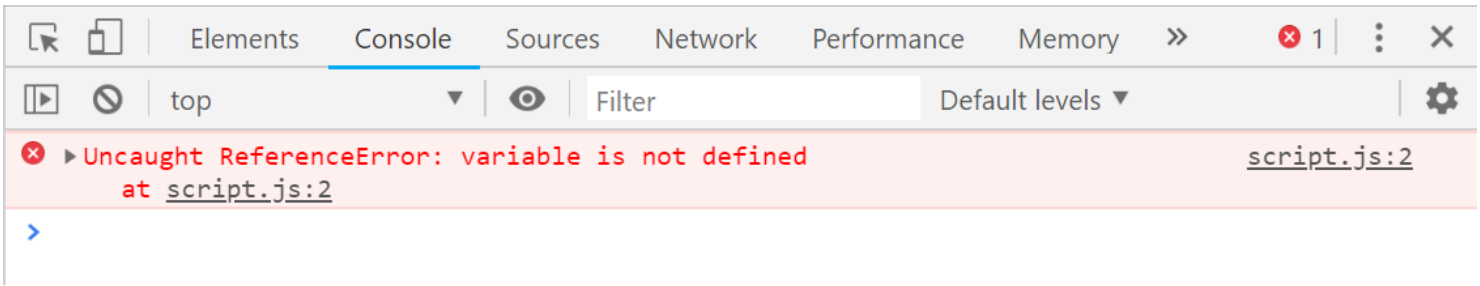
```
variable = 10
console.log(variable)
```



Our code still works even after removing the 'var' but this is a bad practice because even though it works without defining the variable with the keyword "var", js will have to figure out the variable itself and this will result to more work for js. This is where STRICTMODE comes into play, when strict mode is enabled, our code won't work unless our variable is defined and we also add semicolon to define where our line ends (the semi colon is browser dependent). So strict mode is there to ensure we are writing better, safer and more optimized codes. We can add strict mode easily as we can see below.

Script.js

```
"use strict"
variable = 10
console.log(variable)
```

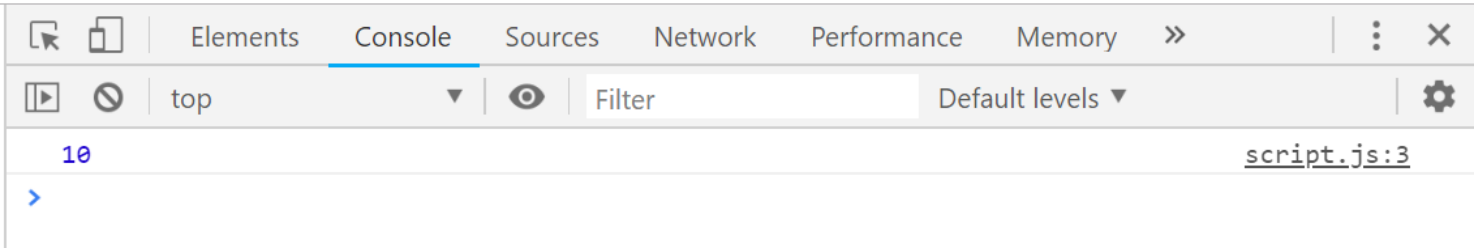


Now we can see that our code is not working unlike before because strictmode is enabled and so we must define our variable before we can get it to work.

Script.js

```
"use strict";
```

```
var variable = 10;
console.log(variable);
```

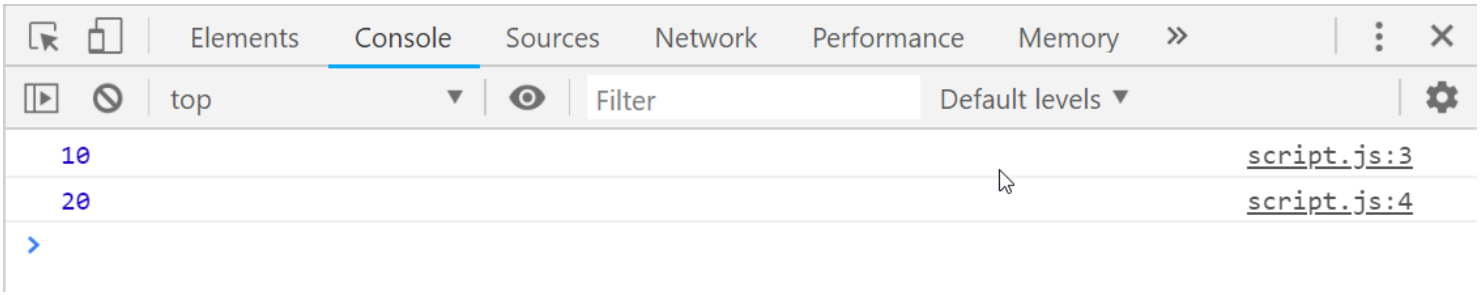


Our code now works!

DYNAMIC TYPING: We need to know that js is case sensitive and so two words that appears the same might function differently for example:

Script.js

```
var variable = 10;
var Variable = 20;
console.log(variable);
console.log(Variable);
```

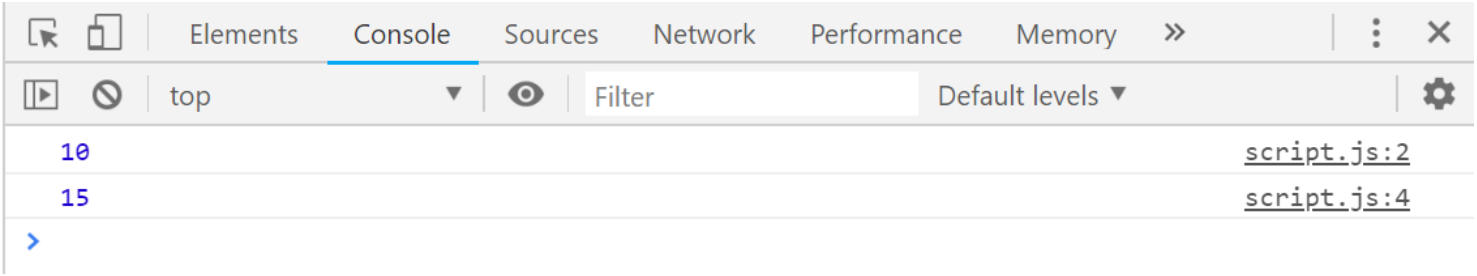


We can see two different variable values printed in the console even though the variable name is the same, just because the second one starts with an upper case, this makes js reads it different from the first.

Re-assigning variable values: we can re-assign variables value easily

Script.js

```
var variable = 10;
console.log(variable);
variable = 15; //We dont need to add the 'var' keyword here because we are not
declaring a new variable but re-assigning a new value for the exisiting variable
console.log(variable);
```

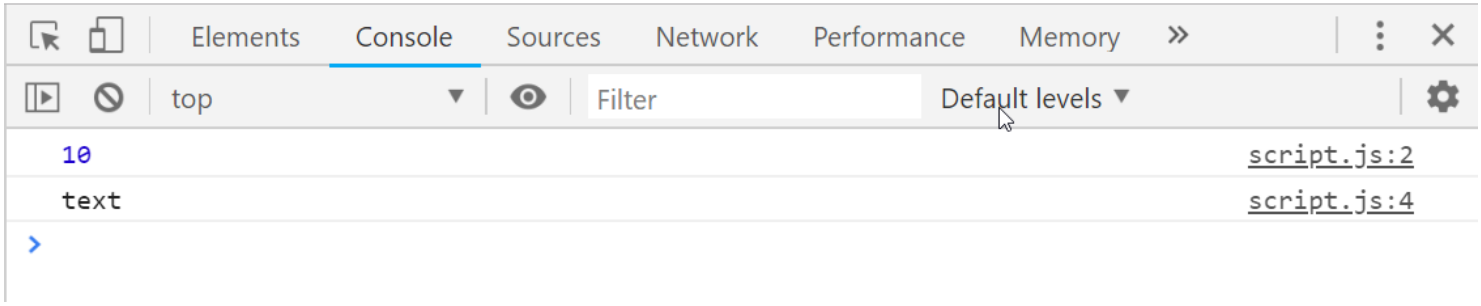


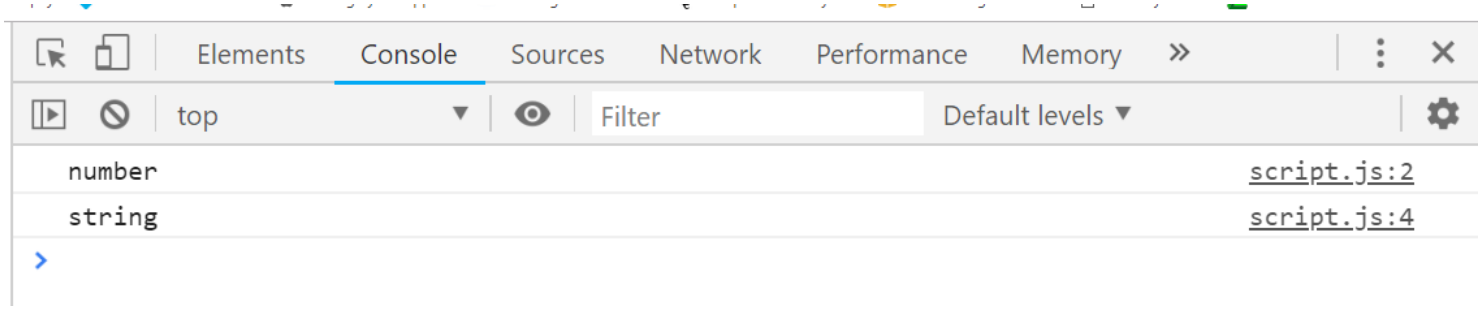
Here we simply re-assigned the number 15 to the existing variable which was having 10 as the previous value and this is why we didn't need to add 'var' again to the 3rd line of our code because we are not declaring a new variable but only changing the value. 'var' is only needed when we are declaring a new variable.

We can also re-assign another variable type to the variable like re-assigning a string, Boolean or null even though the first value is a number

Script.js

```
var variable = 10;
console.log(variable);
variable = 'text';
console.log(variable);
```





Now we see that the first value is number while the second is a string. So we can see that the type of variable was dynamically changed which means JS uses DYNAMIC TYPING which allows variables to change their type.

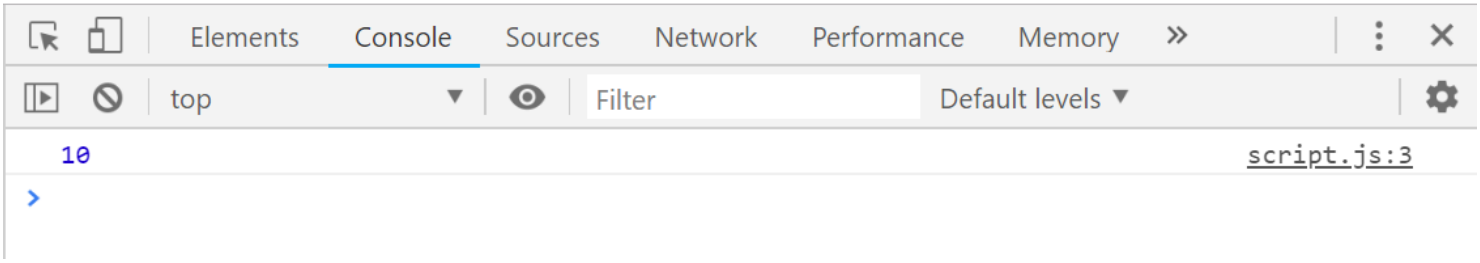
This might be useful in some situations but it's a good practice to not switch type constantly but stick to the type you begin with.

HOISTING: This is another interesting feature JS ships with.

Script.js

```
var1 = 10;
console.log(var1);
var var1;
```

```
"use strict";
var1 = 10;
console.log(var1);
var var1;
```



Here we can see an interesting feature, we already assigned a value to the variable before declaring it in the last line, this works even in strict mode and this is called HOISTING. Using this feature, JS will run over the code twice, the first time it runs, it searches for all our declarations and the second time it re-arrange the code to proper way of writing JS code and assign the values. So HOISTING means declaring a variable after using them or after assigning a value to them.

Another way of writing our JS code is

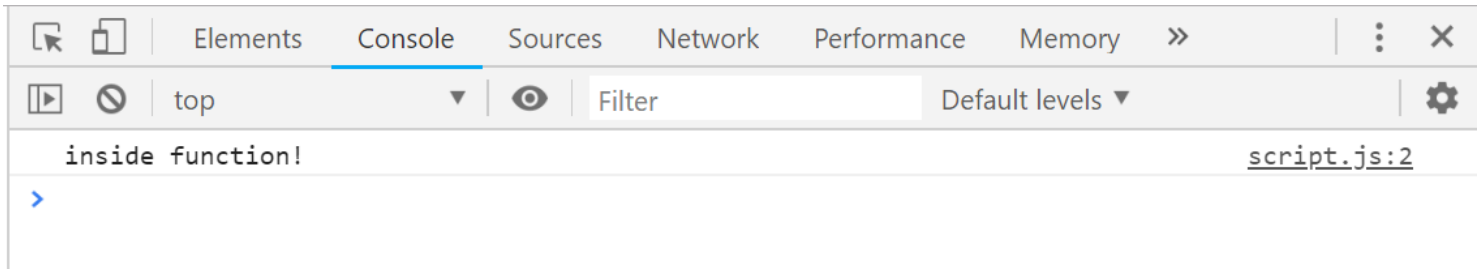
Script.js

```
var var1;
var1 = 10;
console.log(var1);
```

FUNCTIONS – CREATING AND CALLING THEM: A function is a kind of enclosed code with which you can execute whatever you want even multiple times. Let's examine how we can write functions.

Script.js

```
function calc() {
    console.log('inside function!');
}
calc();
```

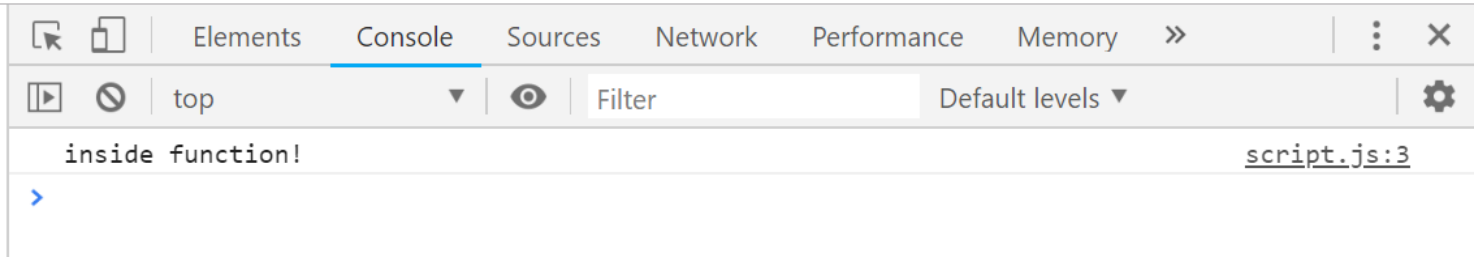


We start by writing the string 'function' to tell JS that this is a function just as we use 'var' for variable, then we write the function name with the () in front, then we write the logic inside {}and then we output the function name again.

HOISTING FOR FUNCTION:

Script.js

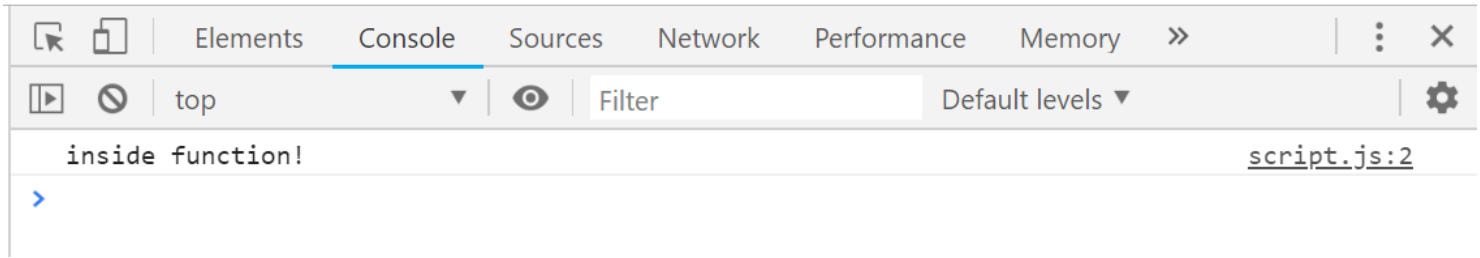
```
calc();
function calc() {
    console.log('inside function!');
}
```



It is not uncommon to have our functions written this way using hoisting. Another way of writing functions is adding a variable first and then assigning function to it.

Script.js

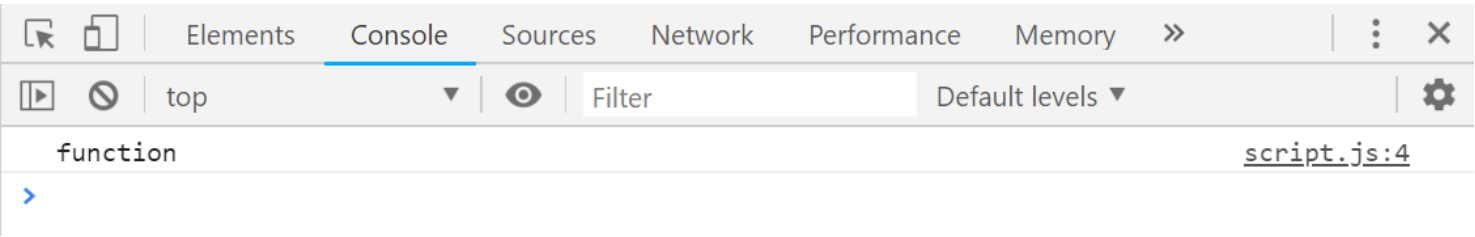
```
var calc = function () {
    console.log('inside function!');
}
calc();
```



We can check the variable type

Script.js

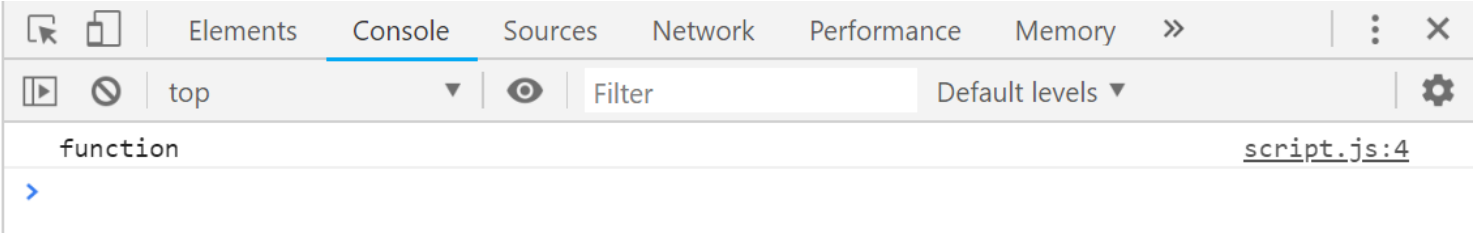
```
var calc = function () {
    console.log('inside function!');
}
console.log(typeof calc);
```



OR

Script.js

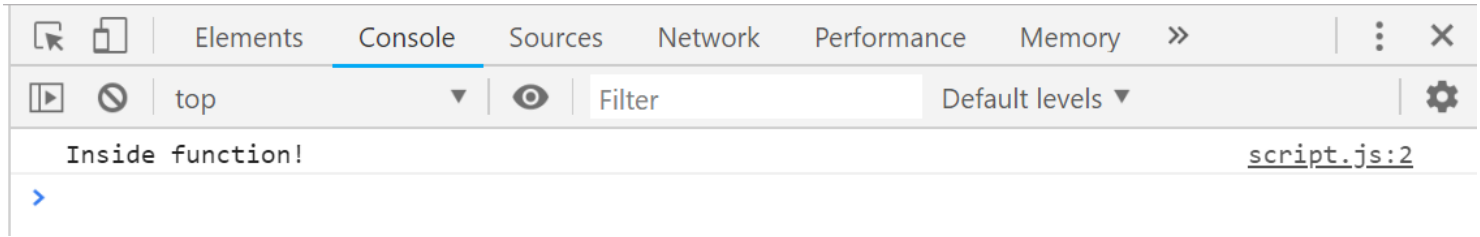
```
function calc() {
    console.log('inside function!');
}
console.log(typeof calc);
```



We can re-assigned the function to another variable.

Script.js

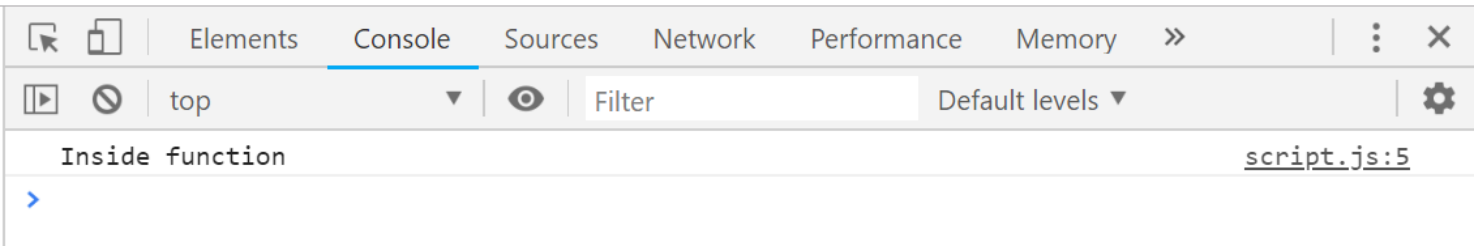
```
function calc() {
    console.log('Inside function!');
}
var anotherFn = calc;
anotherFn();
```



FUNCTIONS – ARGUMENTS RETURNING DATA:

Script.js

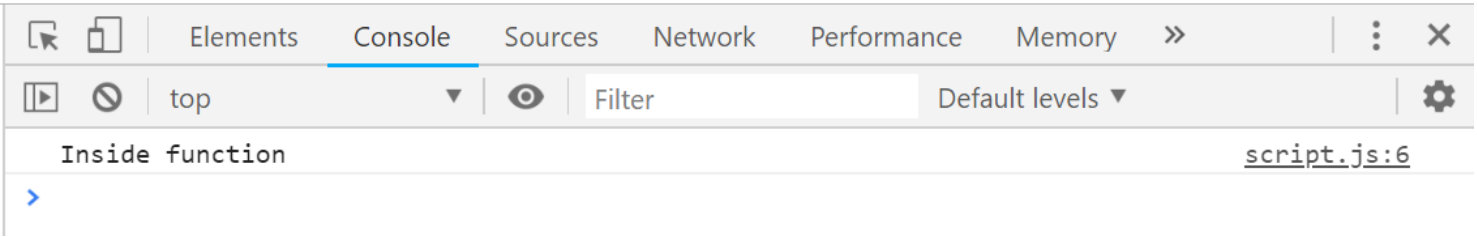
```
function calc() {
    return 'Inside function';
};
var returned = calc();
console.log(returned);
```



OR

Script.js

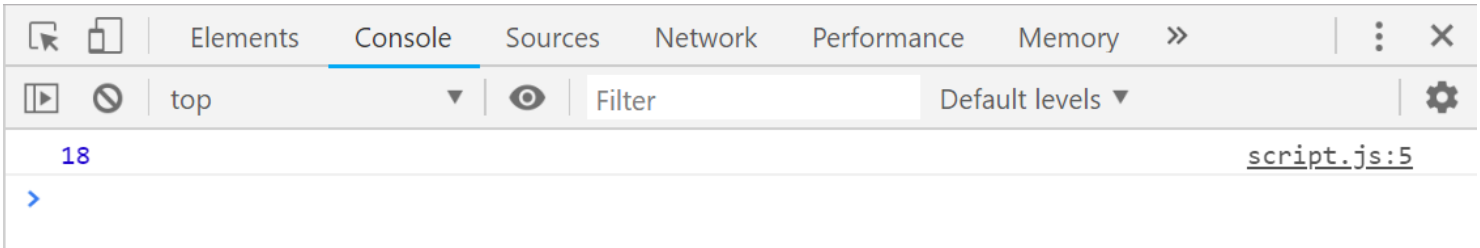
```
function calc() {
    var stringVar ='Inside function';
    return stringVar
}
var returned = calc();
console.log(returned);
```



PASSING ARGUMENTS INSIDE THE FUNCTIONS: We can pass arguments in between the bracket after the function name.

Script.js

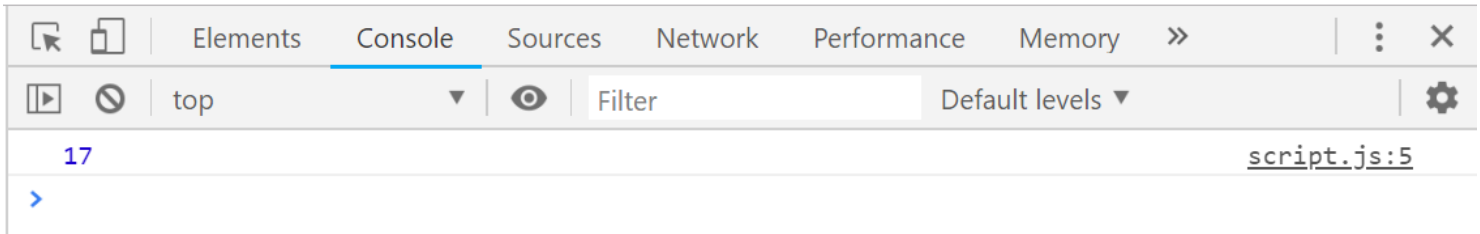
```
function calc(number1, number2) {
    return number1 + number2;
}
var returned = calc(10, 8);
console.log(returned);
```



Another way is:

Script.js

```
function calc(number1, number2) {
    return number1 + number2;
}
var calculator = calc;
console.log(calculator(15, 2));
```



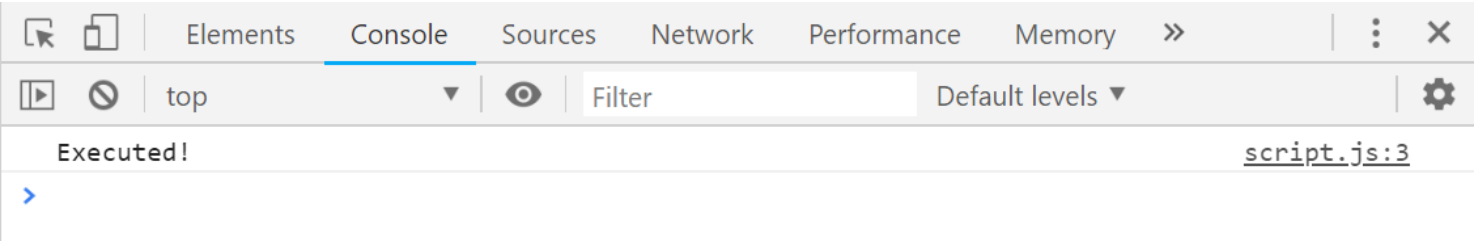
CONTROL STRUCTURES – IF STATEMENTS: Control structures help to tell your browser when to execute certain code or how often should certain code be executed so it is an awesome feature to have for building awesome apps. The popular control structures is the if statement which allows you to run code conditionally.

Use the if statement to specify a block of code to be executed if a condition is true.

Script.js

```
var condition = true;
```

```
if (condition) {
    console.log('Executed!')
}
```

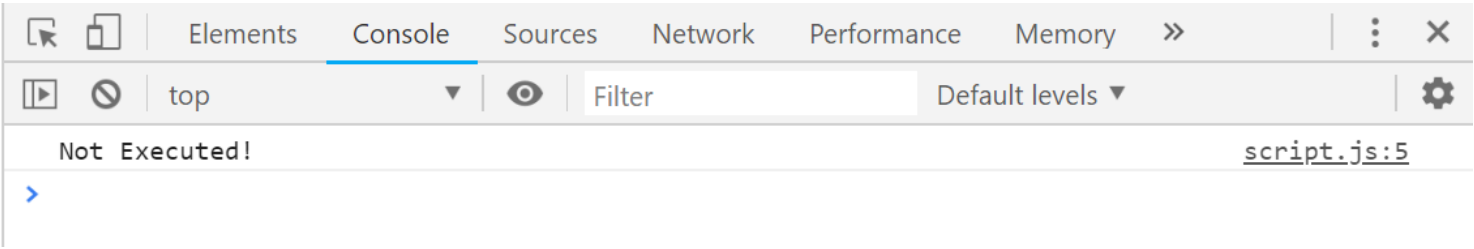


The code above checks if the variable 'condition' is true and then execute the code inside the { } which is to print the message "Executed!" in the console.

The else Statement: Use the else statement to specify a block of code to be executed if the condition is false.

Script.js

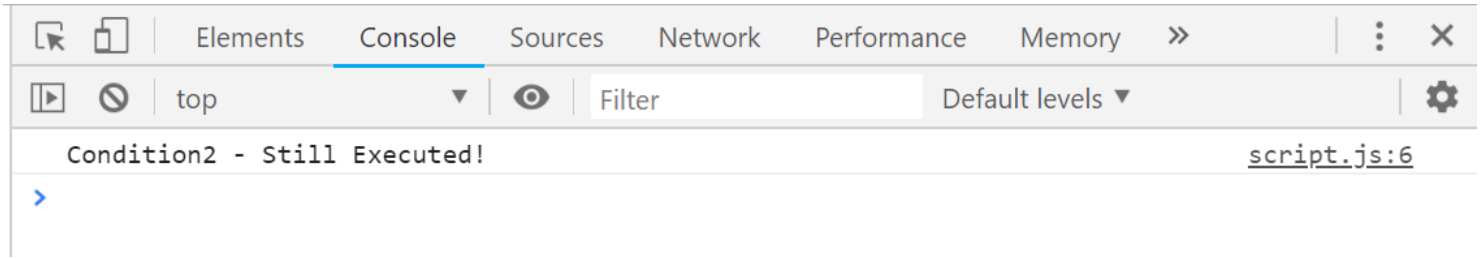
```
var condition = false;
if (condition) {
    console.log('Executed!')
} else {
    console.log('Not Executed!')
}
```



The else if statement: Use the else if statement to specify a new condition if the first condition is false.

Script.js

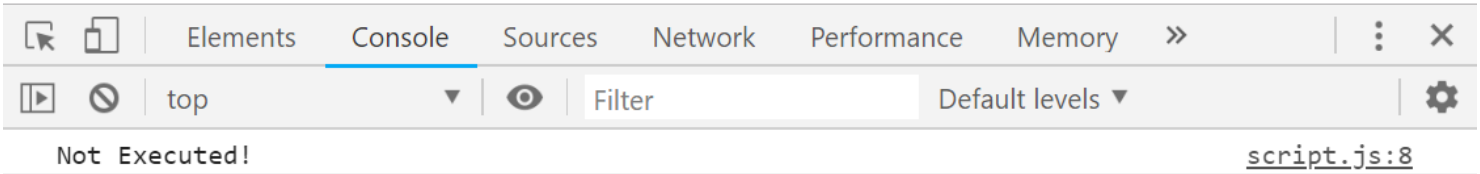
```
var condition1 = false;
var condition2 = true;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
```



In the code above, we get 'Condition2 – still Executed!' because we are not meeting the first condition since it's false but we met the second condition since it's true. The 3rd condition "else" will be executed if all the two conditions are false which we can see below.

Script.js

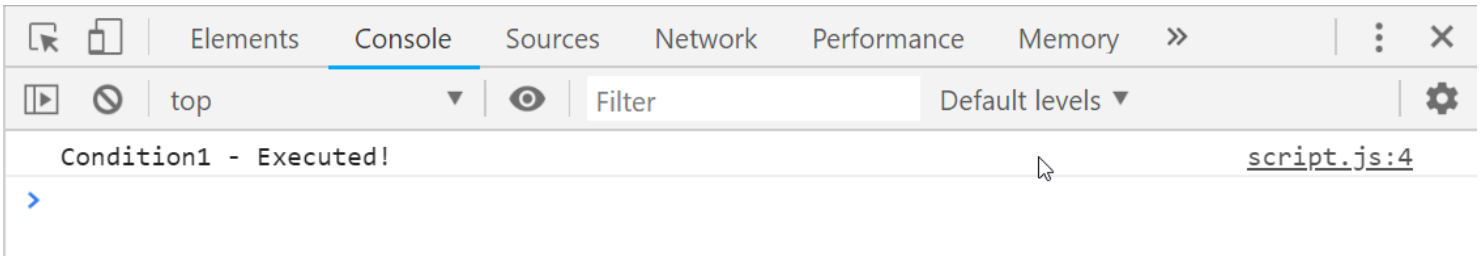
```
var condition1 = false;
var condition2 = false;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
```



CONTROL STRUCTURES – IF STATEMENT ADVANCED: - instead of using Boolean values for our conditional statements, JS allows us to also use 0 and 1, where 0 = false and 1 = true.

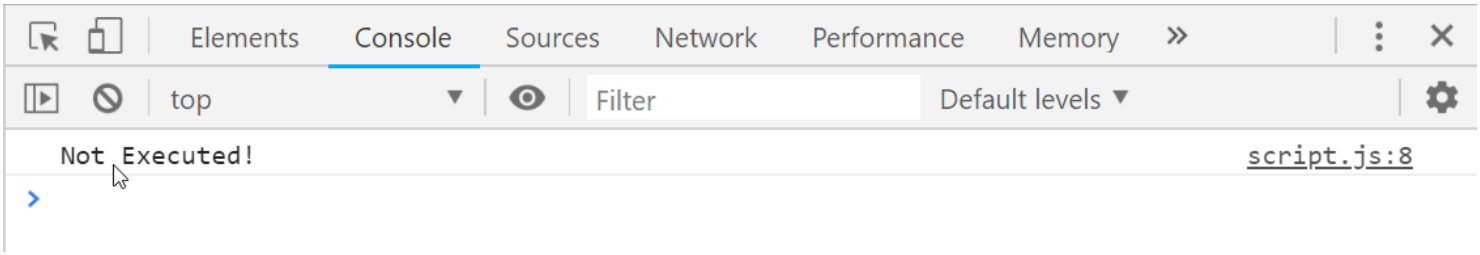
Script.js

```
var condition1 = 1;
var condition2 = false;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
```



Script.js

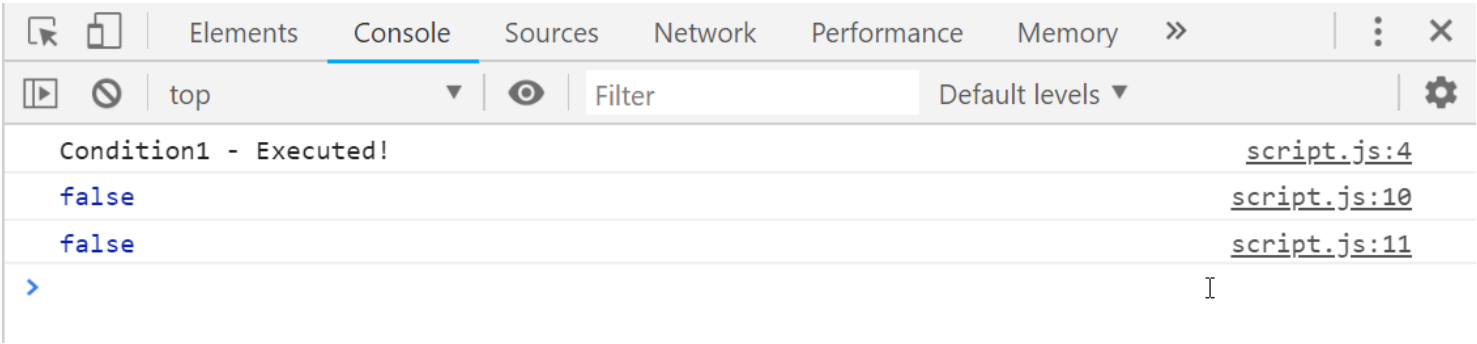
```
var condition1 = 0;
var condition2 = false;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
```



What if we set the condition to 2? Technically, only 1 is equals to true i.e 1 == true but JS will still treats all other numbers which are not 0 as to be true and only 0 will be treated as false. This doesn't mean that 2 is now of the same value or equals to true which we can confirm below.

Script.js

```
var condition1 = 2;
var condition2 = false;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
console.log(2 == true); //checking if 2 is the same value as true, we will get false
bcos it's not
console.log(2 === true) //checking if 2 is of the same var type as true, we will get
false bcos 2 is number and true is boolean
```

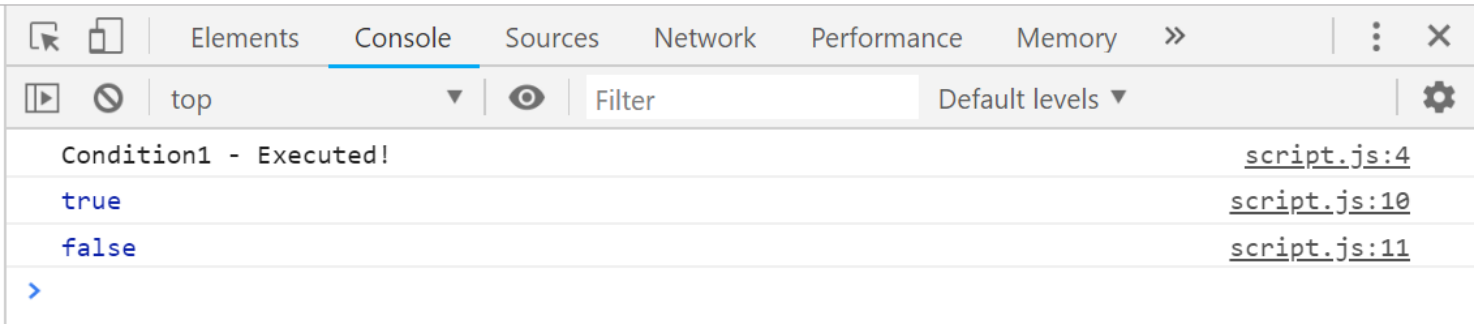


We can test for 1 also

Script.js

```
var condition1 = 1;
```

```
var condition2 = false;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
console.log(1 == true); //checking if 2 is the same value as true, we will get true
bcos it is
console.log(1 === true) //checking if 2 is of the same var type as true, we will get
false bcos 2 is number and true is boolean
```

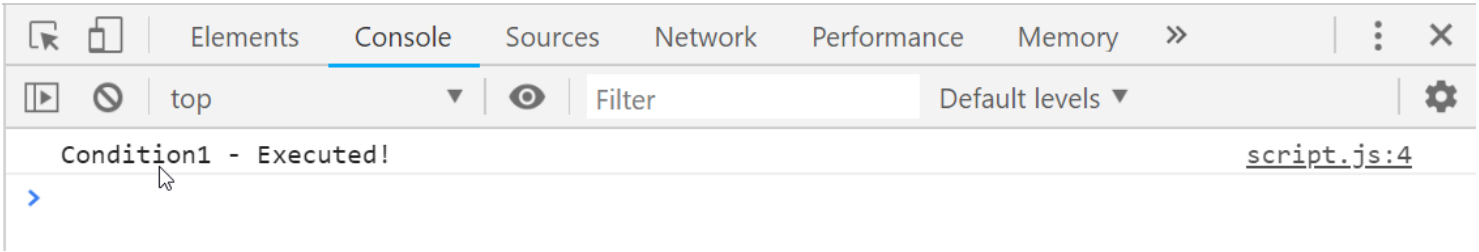


So in a nutshell, only zero and one can be called into Boolean, the rest are only treated as true.

What about negative values? E.g. -1, -2

Script.js

```
var condition1 = -1;
var condition2 = false;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
```



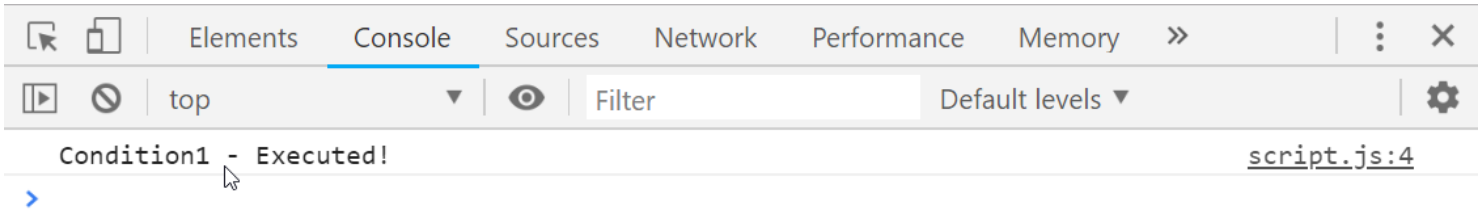
As we can see above, negative values are also treated to be true.

Testing for String: what about assigning a string instead of Boolean?

Script.js

```
var condition1 = 'text';
var condition2 = false;
if (condition1) {
```

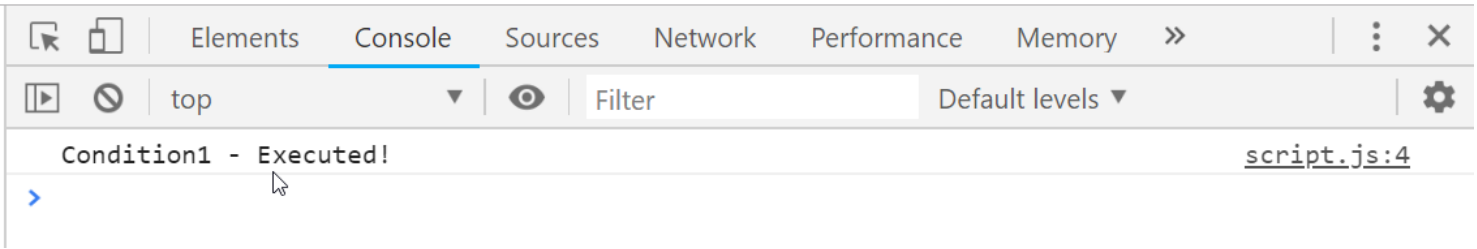
```
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
```



As we can also see above, string is also interpreted to be true. Even that string is written as 'false'

Script.js

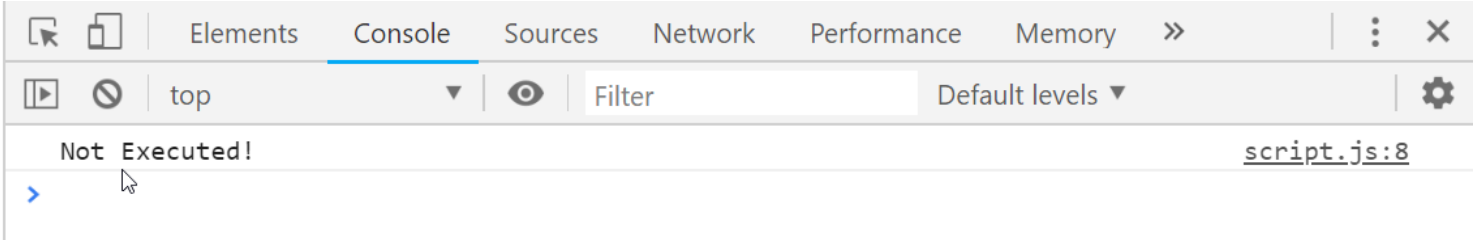
```
var condition1 = 'false';
var condition2 = false;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
```



Testing for null: What happens if we set the variable to null? How will it be interpreted?

Script.js

```
var condition1 = null;
var condition2 = false;
if (condition1) {
    console.log('Condition1 - Executed!')
} else if (condition2) {
    console.log('Condition2 - Still Executed!')
} else {
    console.log('Not Executed!')
}
```



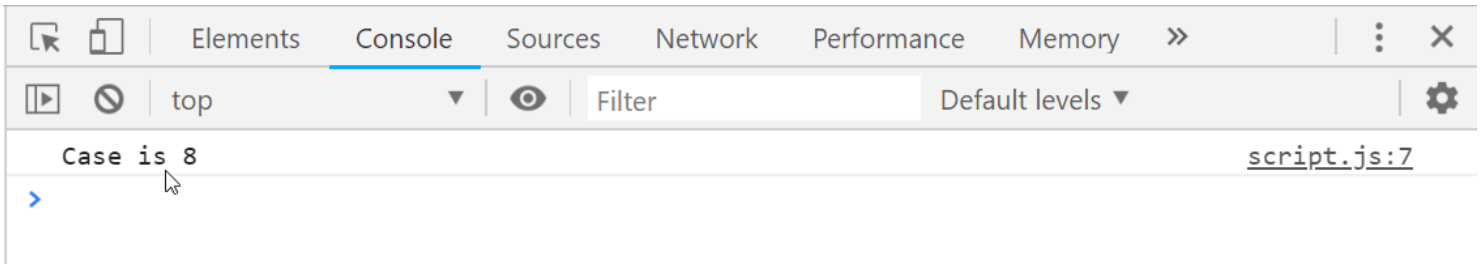
We get "Not Executed" when we set to null which means JS treats null as false.

CONTROL STRUCTURES –SWITCH:

Script.js

```
var luckyNumber = 8;
switch (luckyNumber) {
    case 1:
    console.log('Case is 1');
    break;
    case 8:
    console.log('Case is 8');
    break;
    default:
    console.log('Default');
    break;
}
```

This will check for which of the case is having a value that is equal to the variable 'luckyNumber' value and when it finds the case, it executes the block of code after it. In a case where it finds no matching case, it prints out the default. The syntax must be followed accordingly.



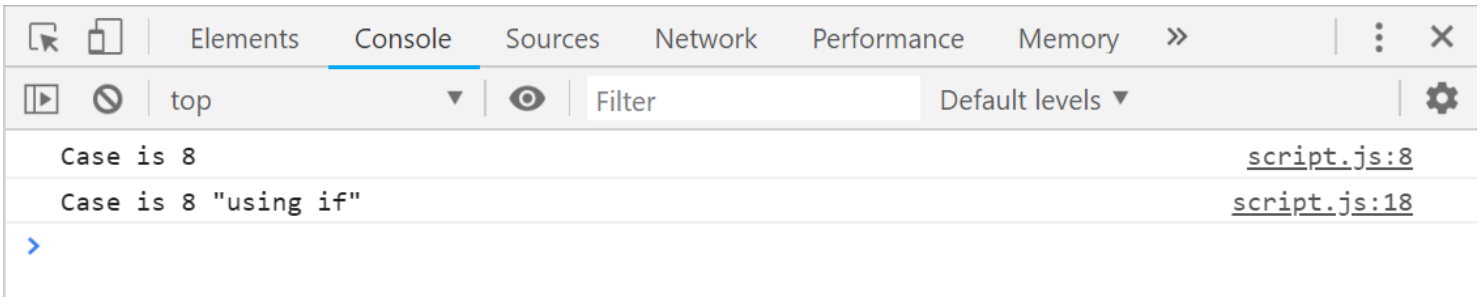
So the switch statement allows us execute conditionally code depending on the value of something i.e. depending on the variable value.

BUT WHY DO WE NEED SWITCH STATEMENT IN THE FIRST PLACE? We can write the code we've written now using if and else statement which we can see below and after we will discuss why we need the switch statement when we can use the if conditional statement.

Script.js

```
var luckyNumber = 8;
//Using switch case
switch (luckyNumber) {
    case 1:
    console.log('Case is 1');
    break;
    case 8:
    console.log('Case is 8');
    break;
```

```
    default:
    console.log('Default');
    break;
}
// writing the same code using if statement
if (luckyNumber == 1) {
console.log('Case is 1 "using if"');
} else if (luckyNumber == 8) {
    console.log('Case is 8 "using if"');
} else {
    console.log('Default "using if"')
}
```

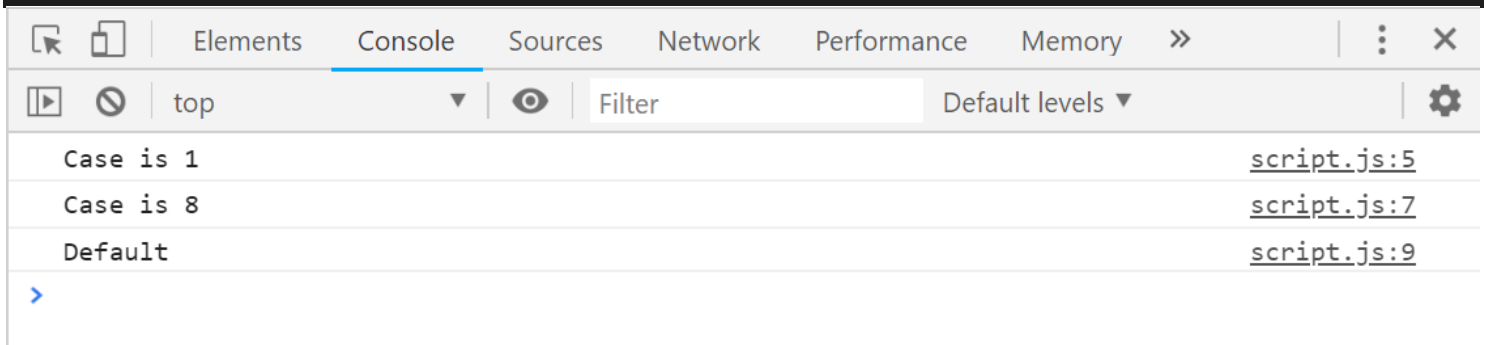


Now we can see that we achieved the same result using if statement, so why bother using switch case? Let's assume what we are checking for is longer than this, using if statement, we will continue to type if and elseif on plenty lines which quickly becomes very inconvenient where with switch case, we have a neater pattern of structuring our code and it checks across all cases and output the one that matches the variable value.

CONTROL STRUCTURES – SWITCH BREAK: What sup with the 'break;' keyword we have in the switch statement? We can remove the keyword and see what happens.

Script.js

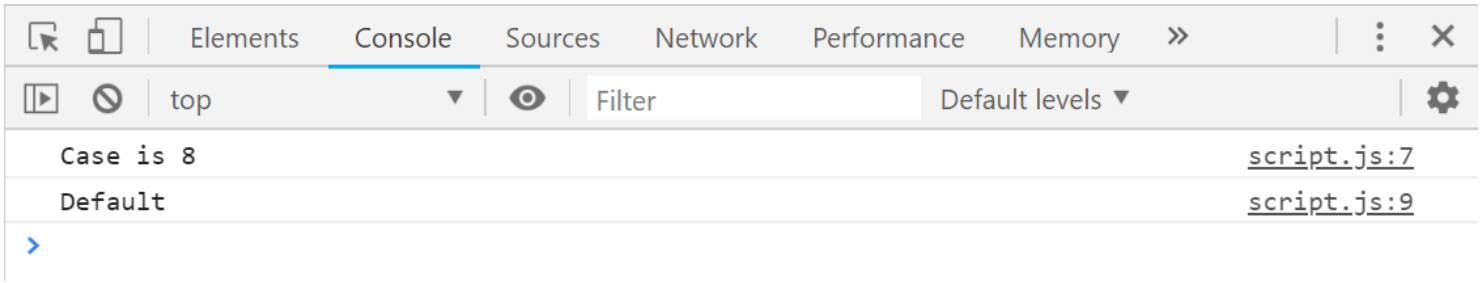
```
var luckyNumber = 1;
//Removing the keyword 'break'
switch (luckyNumber) {
    case 1:
    console.log('Case is 1');
    case 8:
    console.log('Case is 8');
    default:
    console.log('Default');
}
```



OR

Script.js

```
var luckyNumber = 8;
//Removing the keyword 'break'
switch (luckyNumber) {
    case 1:
    console.log('Case is 1');
    case 8:
    console.log('Case is 8');
    default:
    console.log('Default');
}
```

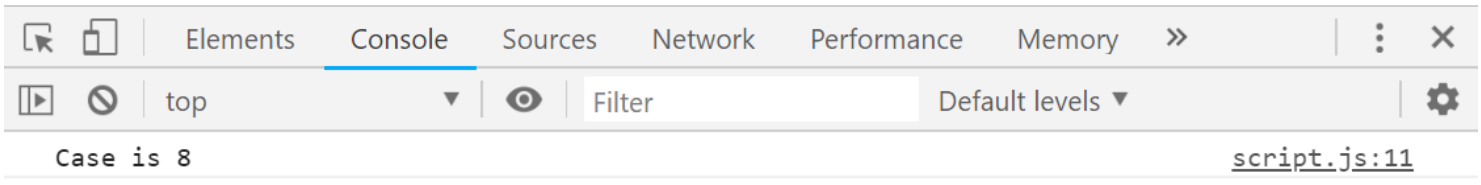


As we can by checking the two codes and the results, when the break keyword is removed, the switch case still checks for the right case first and after which it continues to also execute other case below it instead of breaking out of it. So when JS reaches the break keyword, it breaks out of the switch block. This will stop the execution of inside the block. It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

CONTROL STRUCTURES – THE DEFAULT KEYWORD: The default keyword specifies the code to run if there is no case match. The default case does not have to be the last case in a switch block.

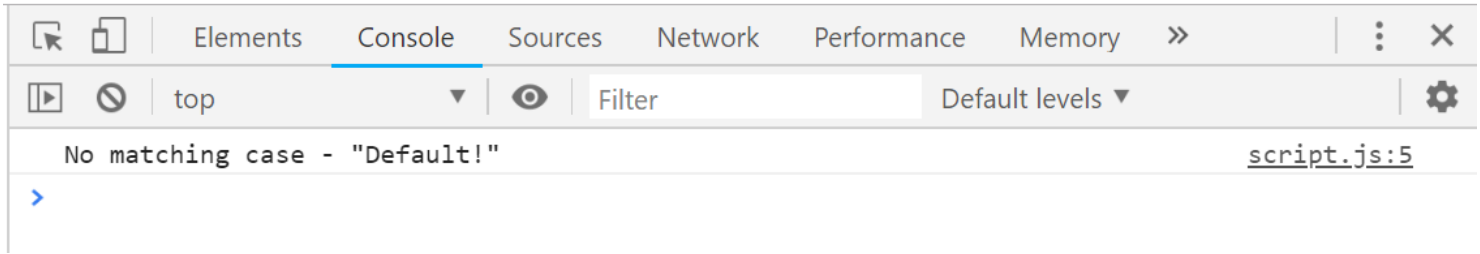
Script.js

```
var luckyNumber = 8;
//Removing the keyword 'break'
switch (luckyNumber) {
    default:
    console.log('Default'); //default keyword coming first
    break;
    case 1:
    console.log('Case is 1');
    break;
    case 8:
    console.log('Case is 8'); //no need to break the last case
}
```



Script.js

```
var luckyNumber = 8;
//Removing the keyword 'break'
switch (luckyNumber) {
    default:
    console.log('No matching case - "Default!"'); //default keyword coming first
    break;
    case 1:
    console.log('Case is 1');
    break;
    case 3:
    console.log('Case is 8'); //no need to break the last case
}
```

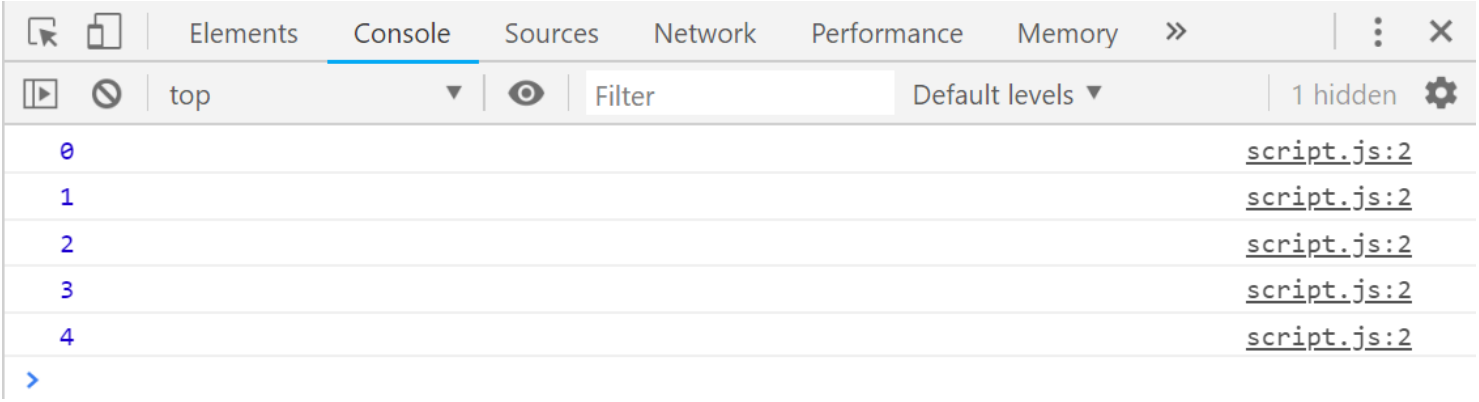


So we can see that the code still works even when we start with the default case and also even though we didn't break the last case on the list.

CONTROL STRUCTURES – FOR LOOP: This another type of control structure which allows us to execute code multiple times.

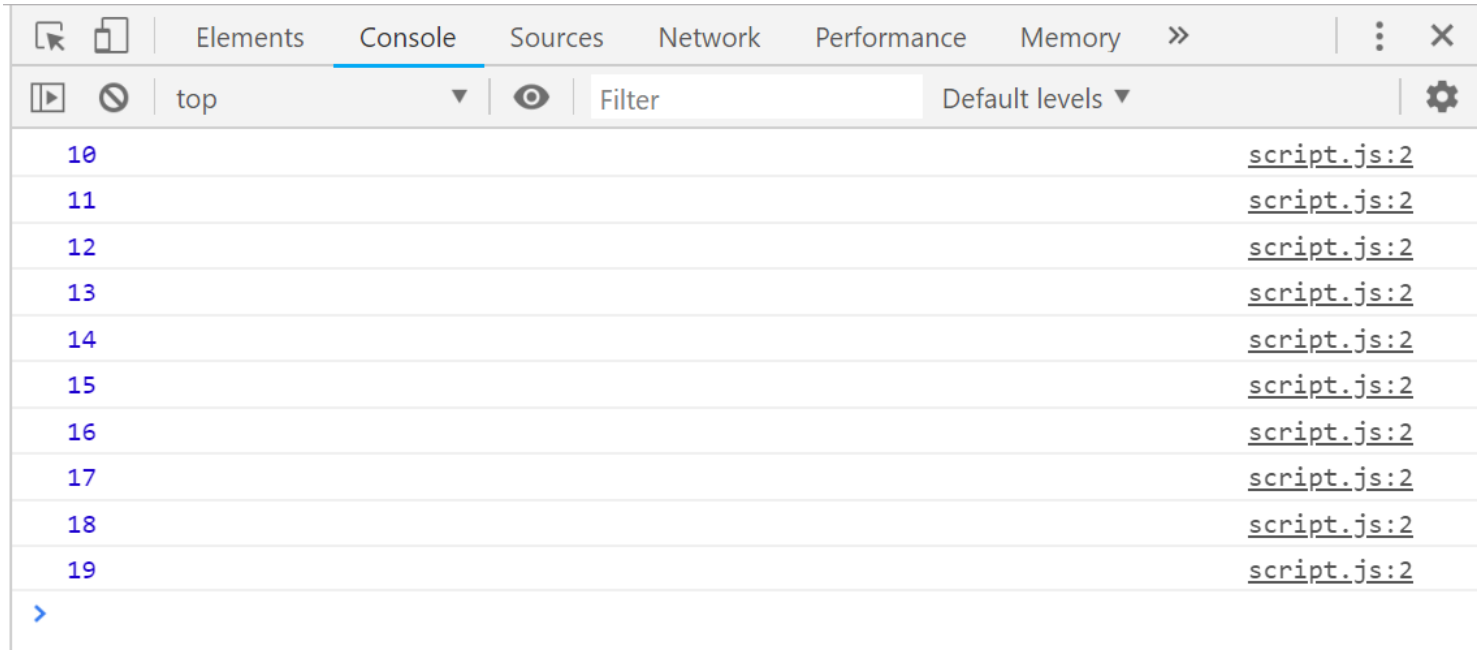
Script.js

```
for (var i = 0; i < 5; i++) {
    console.log(i)
}

//The code reads: The var i has an initial value of 0, now as long as i is less than 5,
continue to loop through increment for i i.e. add 1 to the value of i for as long as
the new result is less than 5 and print the new i value in the console
```



Script.js

```
for (var mayowa = 10; mayowa < 20; mayowa++) {
    console.log(mayowa)
}
```



CONTROL STRUCTURES – NESTED LOOPS: We can also nest loop inside loops that is to add another loop inside loop.

Script.js

```
for (var i = 0; i < 5; i++) {
    for (var j = 0; j < 2; j++) { //nested loop
    console.log(i * j);
}
}
```

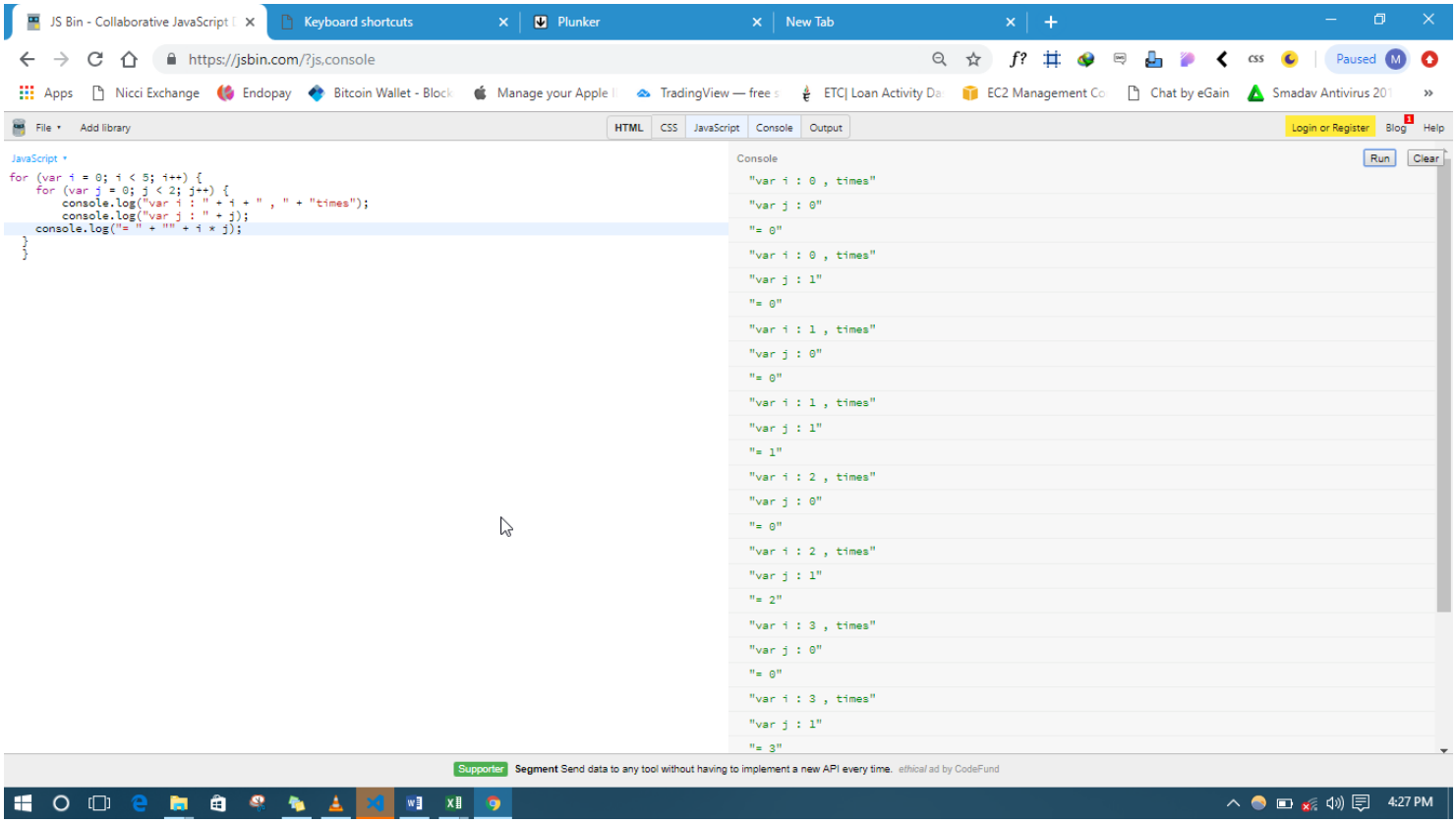
```
3 0                                   script.js:3
  1                                   script.js:3
  0                                   script.js:3
  2                                   script.js:3
  0                                   script.js:3
  3                                   script.js:3
  0                                   script.js:3
  4                                   script.js:3
>
```

How we come about the figures through the multiplication might look confusing but we can further break down the code in the console to see what was happening behind the code.

Script.js

```
for (var i = 0; i < 5; i++) {
  for (var j = 0; j < 2; j++) {
      console.log("var i : " + i + " , " + "times");
      console.log("var j : " + j);
  console.log("= " + "" + i * j);
}
}
```

```
var i : 0 , times                     script.js:3
var j : 0                             script.js:4
= 0                                   script.js:5
var i : 0 , times                     script.js:3
var j : 1                             script.js:4
= 0                                   script.js:5
var i : 1 , times                     script.js:3
var j : 0                             script.js:4
= 0                                   script.js:5
var i : 1 , times                     script.js:3
var j : 1                             script.js:4
= 1                                   script.js:5
var i : 2 , times                     script.js:3
var j : 0                             script.js:4
= 0                                   script.js:5
var i : 2 , times                     script.js:3
var j : 1                             script.js:4
= 2                                   script.js:5
var i : 3 , times                     script.js:3
var j : 0                             script.js:4
= 0                                   script.js:5
var i : 3 , times                     script.js:3
var j : 1                             script.js:4
= 3                                   script.js:5
var i : 4 , times                     script.js:3
var j : 0                             script.js:4
= 0                                   script.js:5
var i : 4 , times                     script.js:3
var j : 1                             script.js:4
= 4                                   script.js:5
>
```

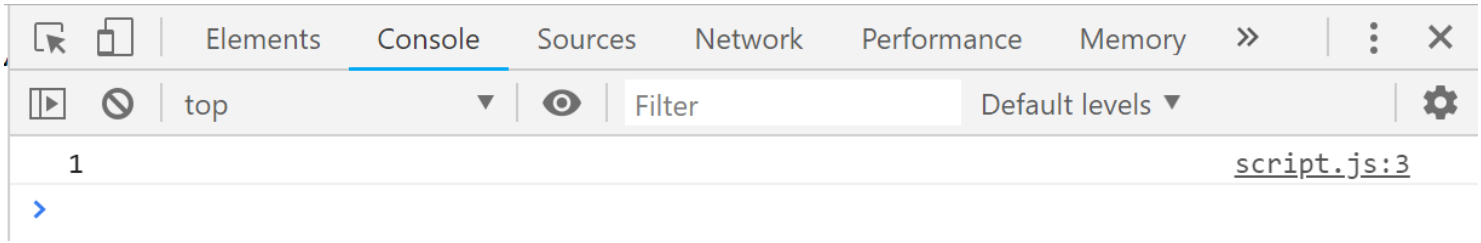


Now we can understand how JS Loop is done and how the multiplication is carried out.

CONTROL STRUCTURES – Controlling loop with Break Continue: **We can also use if statement inside our loop.**

Script.js

```
for (var i = 0; i < 5; i++) {
  if (i == 1) {
  console.log("1");
}
}
```



We also have some keywords we can use in our loops as well. The keyword break; can also be use in our loop. This will tell JS to exit from the loop if a condition is true.

Script.js

```
for (var i = 0; i < 5; i++) {
  if (i == 1) {
  break;
}
console.log(i);
}
//The loop checks for the first value which is o and print it, if i equals 1 which is
true for the 2nd value then it breaks i.e. it exit the loop
```

```
0                                                    script.js:5
>
```

Keyword Continue;

Script.js

```
for (var i = 0; i < 5; i++) {
  if (i == 1) {
  continue;
}
console.log(i);
}
//The loop checks for the first value which is o and print it, if i equals 1 which is
true for the 2nd value then it continues with the next value ignoring the current value
for which the condition was check i.e. 1 is jumped
```

```
0                                                    script.js:5
2                                                    script.js:5
3                                                    script.js:5
4                                                    script.js:5
>
```

Script.js

```
for (var i = 0; i < 5; i++) {
  for (var j = 0; j < 2;j++) {
    if (i == 1) {
      continue;
    }
    console.log('inner loop, j = ' + j)
  }
console.log(i);
}
```

```
inner loop, j = 0                    script.js:6
inner loop, j = 1                    script.js:6
0                                    script.js:8
1                                    script.js:8
inner loop, j = 0                    script.js:6
inner loop, j = 1                    script.js:6
2                                    script.js:8
inner loop, j = 0                    script.js:6
inner loop, j = 1                    script.js:6
3                                    script.js:8
inner loop, j = 0                    script.js:6
inner loop, j = 1                    script.js:6
4                                    script.js:8
>
```

Here JS checks the nested loop first and prints its value which is 0 and 1 and then it prints the value of i.

When the value of I equals to 1, it skips and won't print the nested loop value in this case, it only prints the value of I which is 1.

For other values whose values is not 1, we first print the string 'inner loop, j =' before printing the value of j.

Script.js

```
for (var i = 0; i < 5; i++) {
  for (var j = 0; j < 2;j++) {
    if (i == 1) {
      break;
    }
    console.log('inner loop, j = ' + j)
  }
console.log(i);
}
```

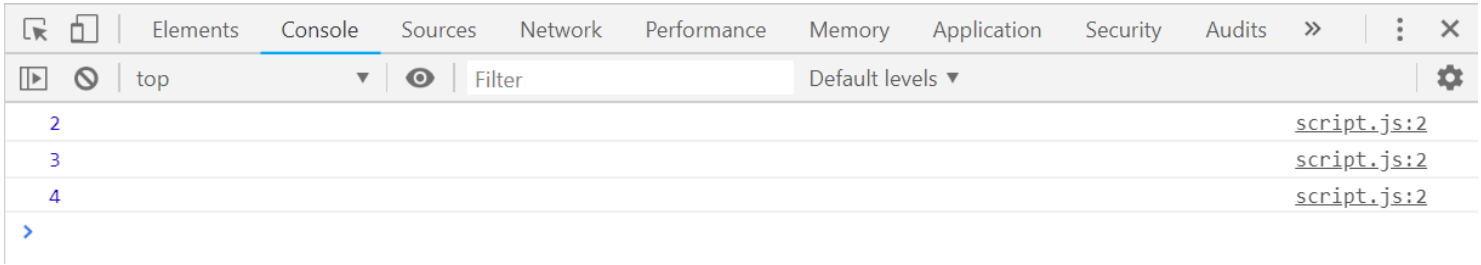
```
inner loop, j = 0                    script.js:6
inner loop, j = 1                    script.js:6
0                                    script.js:8
1                                    script.js:8
inner loop, j = 0                    script.js:6
inner loop, j = 1                    script.js:6
2                                    script.js:8
inner loop, j = 0                    script.js:6
inner loop, j = 1                    script.js:6
3                                    script.js:8
inner loop, j = 0                    script.js:6
inner loop, j = 1                    script.js:6
4                                    script.js:8
>
```

We still get the same behavior using break keyword because we are breaking out of the inner loop and not of the overall loop.

CONTROL STRUCTURES – For Loop Variations: You are not limited to starting your loop from 0, you can start from other numbers as well e.g. 2.

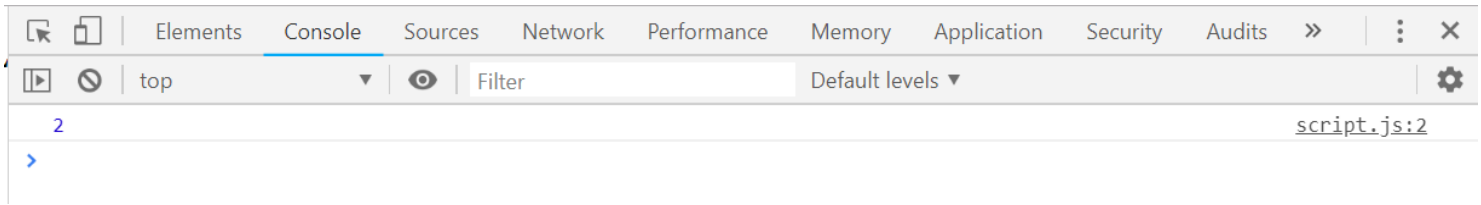
Script.js

```
for (var i = 2; i < 5; i++) {
console.log(i);
}
```



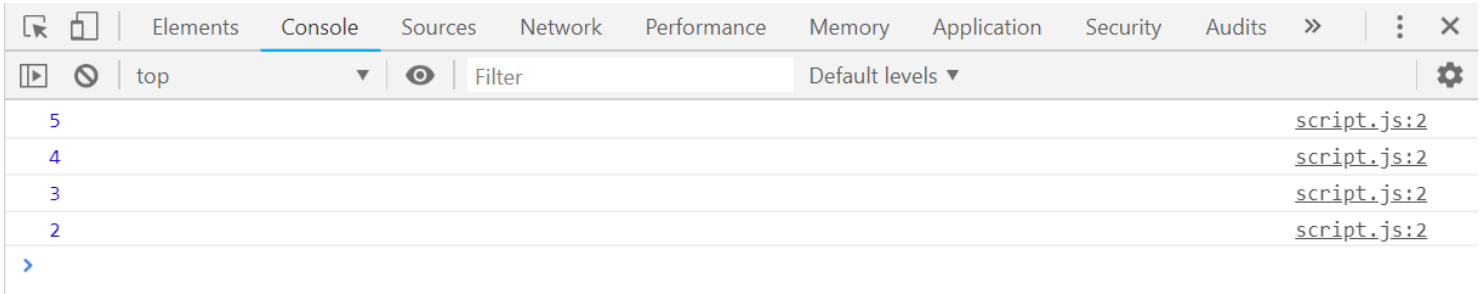
Script.js

```
for (var i = 2; i < 5; i = i + 3) { //we pick the first value of 2, we add 3 to it so
we get 5 and we print i which is 2 and we stop since the next value will be 5.
console.log(i);
}
```



Script.js

```
for (var i = 5; i > 1; i--) {
console.log(i);
}
```

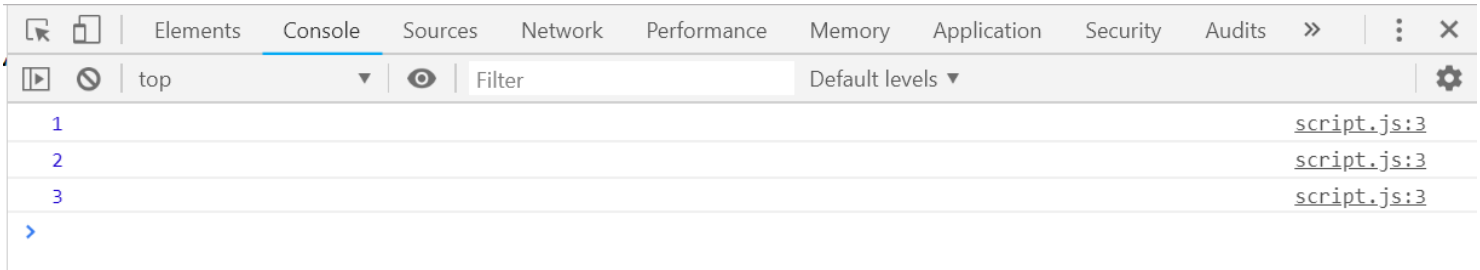


We pick the first value of 5 and print it, we minus 1 from 5 and it gives us 4 and since it's also greater than 1, we print it, we continue this loop till we get to 1 which is not greater than 1 and we terminate the loop. This is how you can play around with loop and set it for your program to work correctly.

CONTROL STRUCTURES – Loop through Arrays: The most common thing to use such loop for is to use them in conjunction with array.

Scripts.js

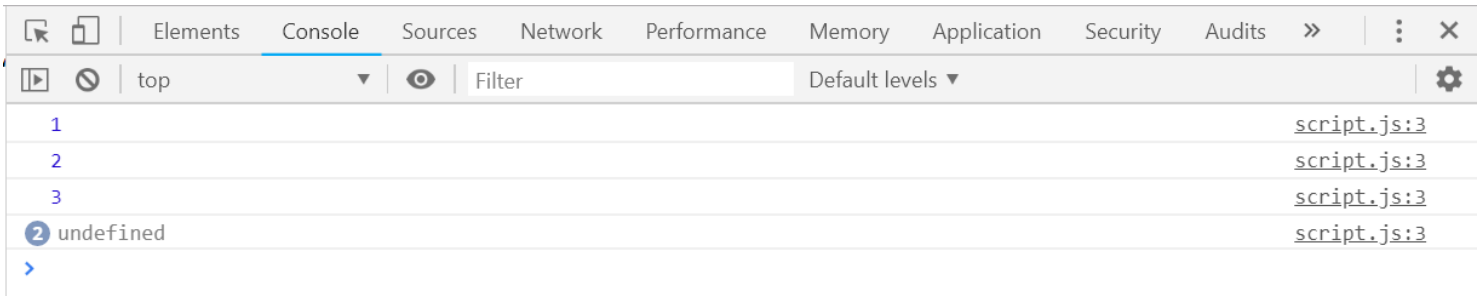
```
var array = [1, 2, 3];
for (var i = 0; i < 3; i++) {
console.log(array[i]);
}
```



JS will print the array values of 3 elements. I < 3 means prints up to 3 values.

Script.js

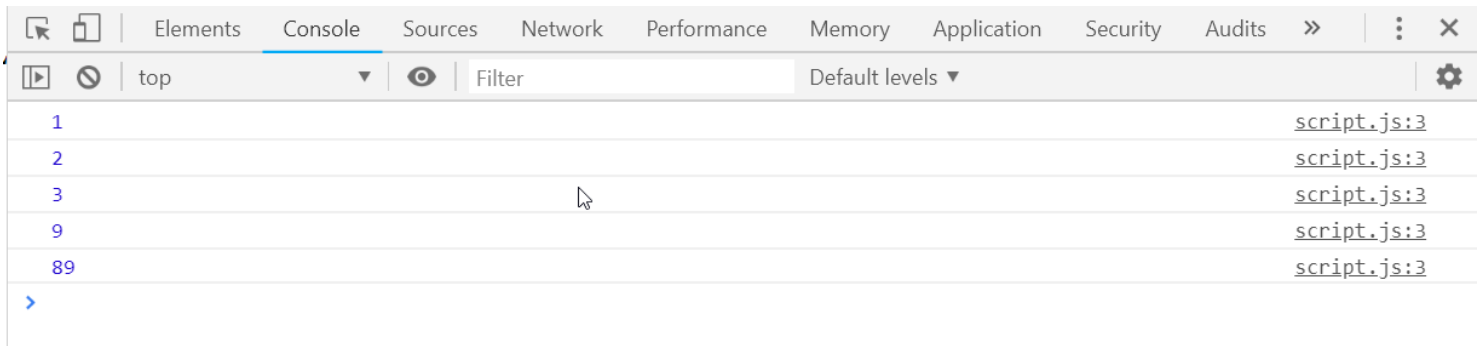
```
var array = [1, 2, 3];
for (var i = 0; i < 5; i++) {
console.log(array[i]);
}
```



Here we print 3 values and 2 undefined because we asked JS to print 5 values whereas we only have 3 elements in our array.

Script.js

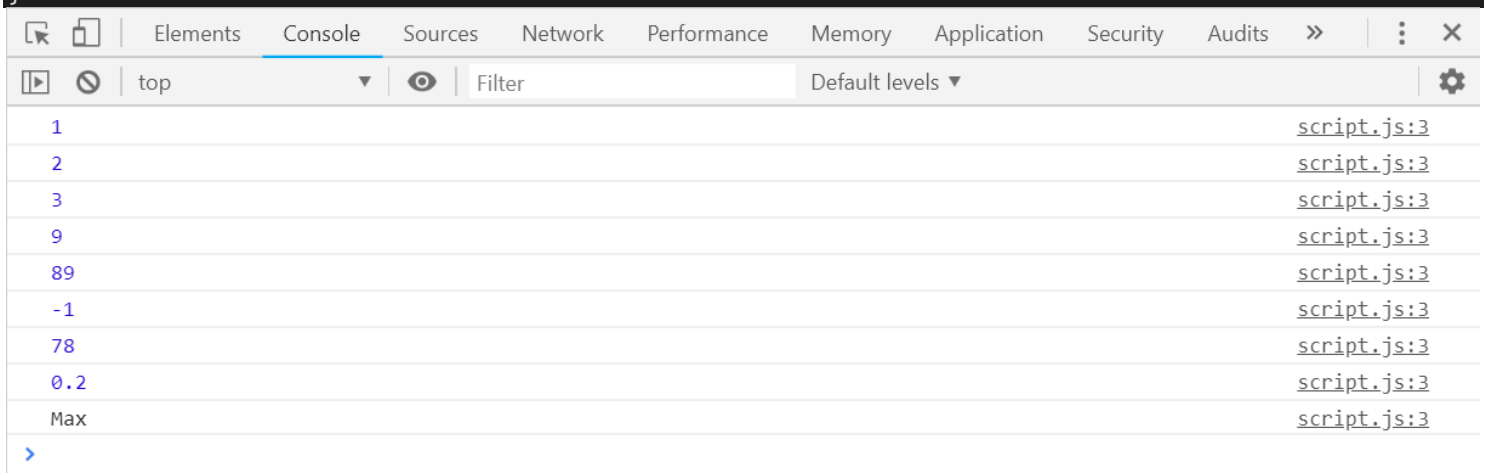
```
var array = [1, 2, 3, 9, 89];
for (var i = 0; i < 5; i++) {
console.log(array[i]);
}
```



The undefined(s) are not removed. This is not the right way to loop through array because we always need to change the number we used to specified the length anytime our array changes elements. The better way to do this is below.

Script.js

```
var array = [1, 2, 3, 9, 89, -1, 78, 0.2, 'Max'];
for (var i = 0; i < array.length; i++) {
console.log(array[i]);
}
```

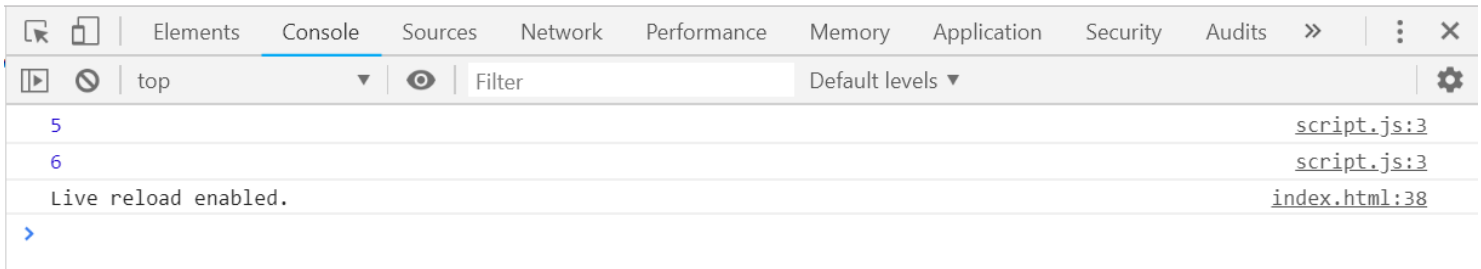


Now we successfully looped through all the array without hardcoding the length as a number.

CONTROL STRUCTURES – While Loop: This is another type of loop.

Script.js

```
var number = 5;
while (number < 7) {
  console.log(number);
  number++;
}
```

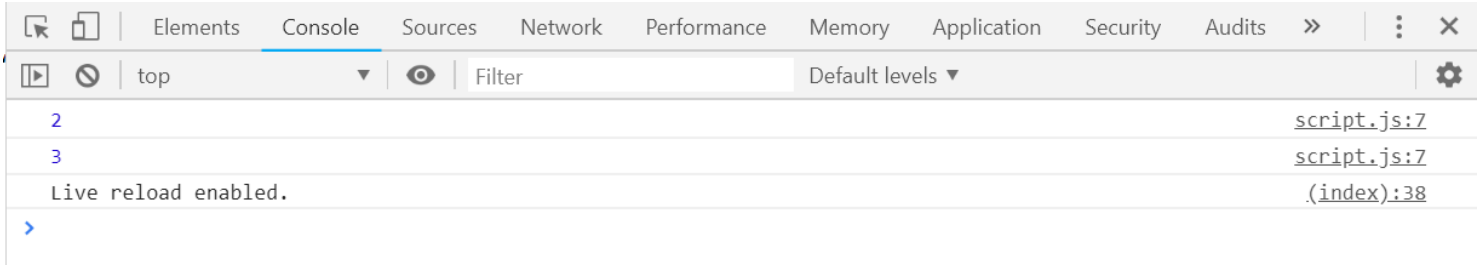


This works this way: The first number is 5 and it's printed, it checks for the next number because we are dealing with increment and if it's lower than 7 which is true, we print it. The next number is 7 and it's not lower than 7 so we exit. So while the condition (number < 7) is true, we print the number.

Script.js

```
var condition = true;
var i = 2;
while (condition) {
  if (i == 3) {
    condition = false;
  }
  console.log(i);
```

```
  i++;
}
```

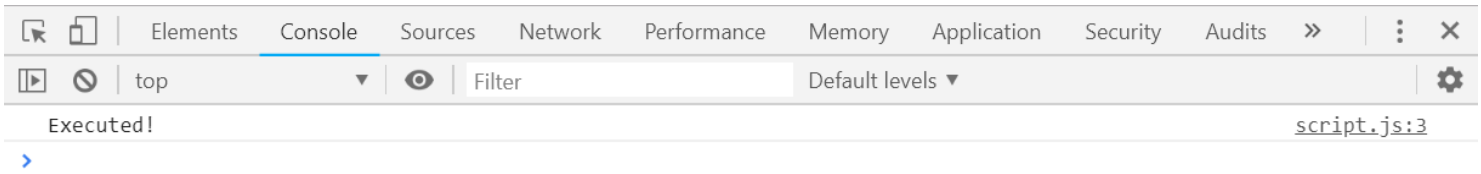


This means: by default, the variable 'condition' is set to true and we print I and loop by increment. As soon as I equal to 3, we set our condition to false and we stop printing because we are checking for while condition is true.

DO WHILE LOOP:

Script.js

```
var condition = false;
do{
  console.log('Executed!');
}
while (condition);
```

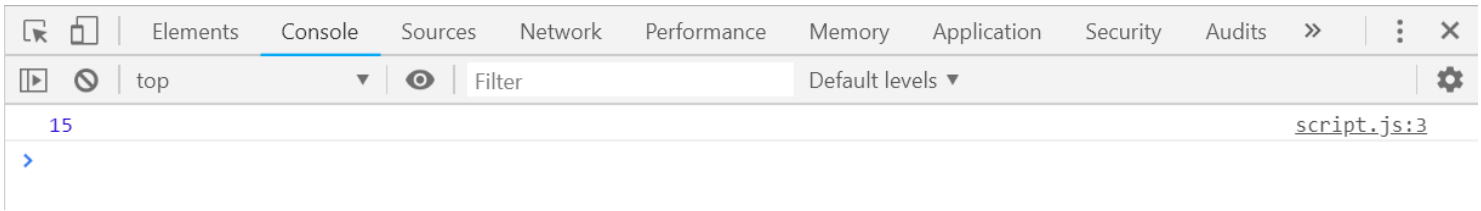


The Do while loop allows you to execute a block of code only once even if the variable value and while are against each other i.e. in this case, the code is only supposed to be executed while the variable 'condition' is but now it's false but still we can execute the code once using the do.

OPERATORS – ADDITION: JS offers us some interesting operator among which we have addition.

Script.js

```
var a = 5;
var b = 10;
console.log(a + b);
```

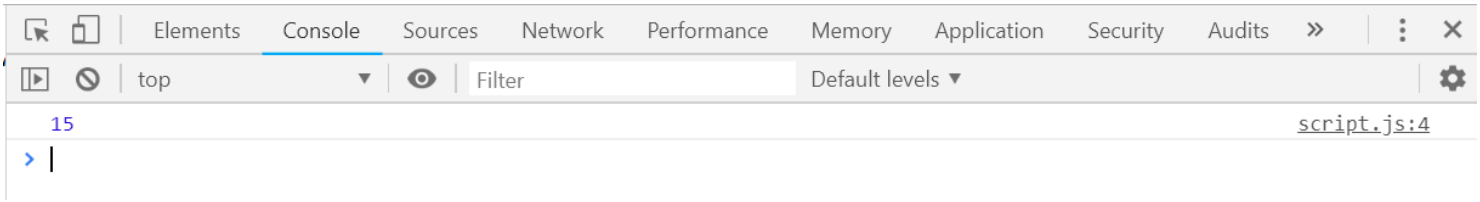


+=

script.js

```
var a = 5;
var b = 10;
```

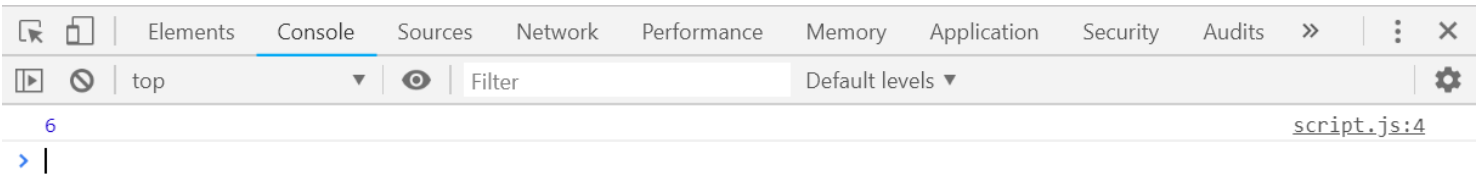
```
a += b // a += b means a = a + b
console.log(a);
```



++

Script.js

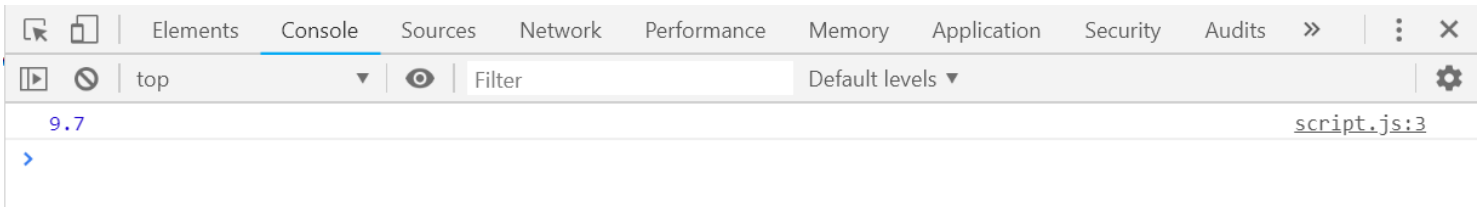
```
var a = 5;
var b = 10;
a ++ // a ++ means a = a+1
console.log(a);
```



Adding floating (decimal) numbers

Script.js

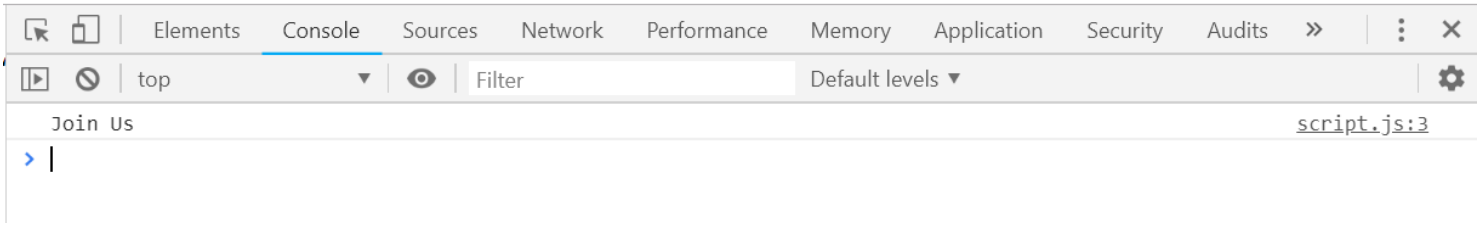
```
var a = 5.2;
var b = 4.5;
console.log(a + b);
```



Floating/decimal numbers works the same way.

Script.js

```
var a = 'Join ';
var b = 'Us';
console.log(a + b);
```

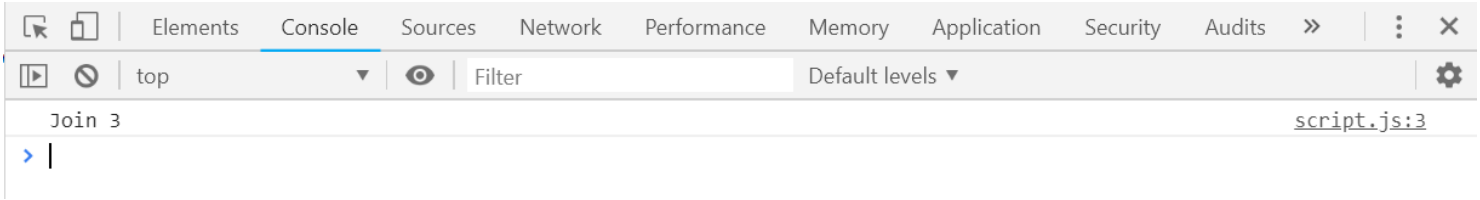


As we can see above, we can add strings together also and they will result to one sentence/phrase.

String + number = string

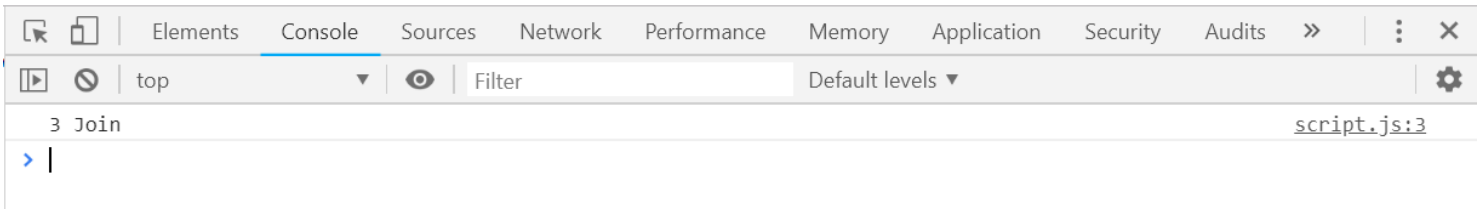
Script.js

```
var a = 'Join ';
var b = 3;
console.log(a + b);
```



Script.js

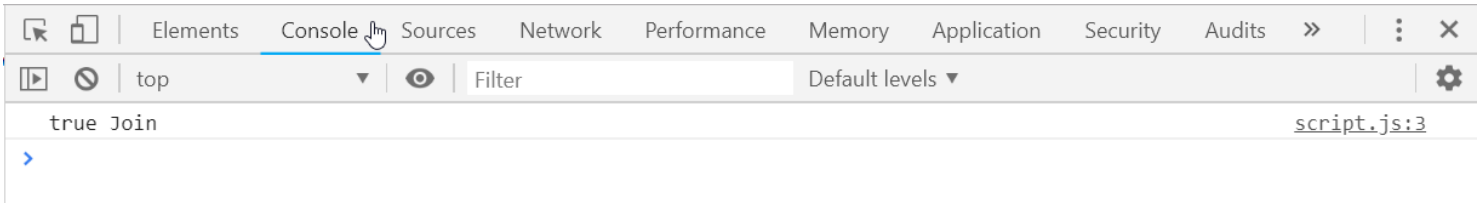
```
var a = 3;
var b = ' Join';
console.log(a + b);
```



Boolean + String = string

Script.js

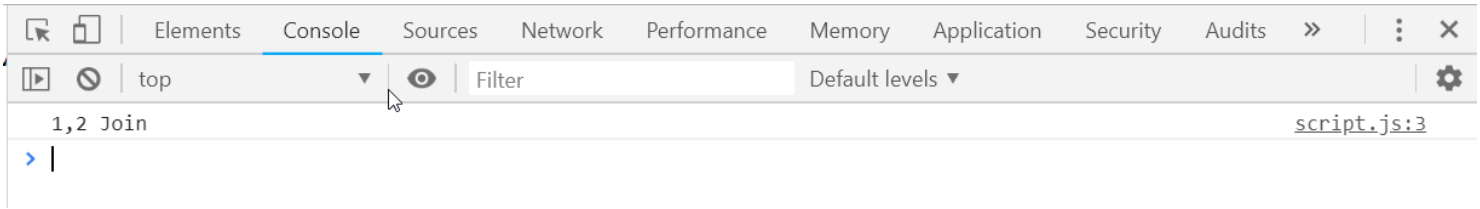
```
var a = true;
var b = ' Join';
console.log(a + b);
```



Array + string = string

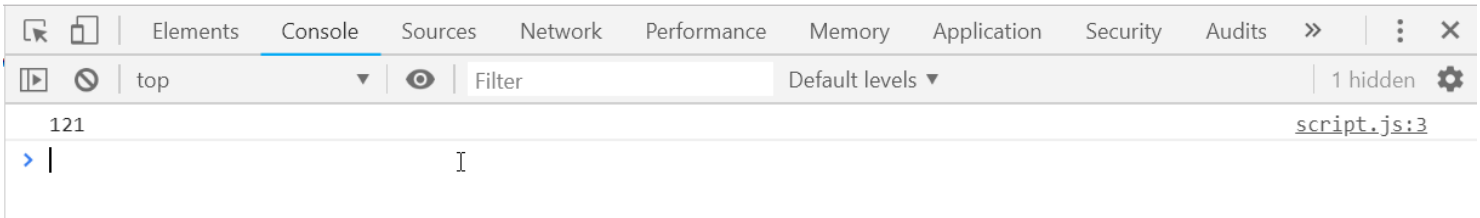
Script.js

```
var a = [1, 2];
var b = ' Join';
console.log(a + b);
```



Script.js

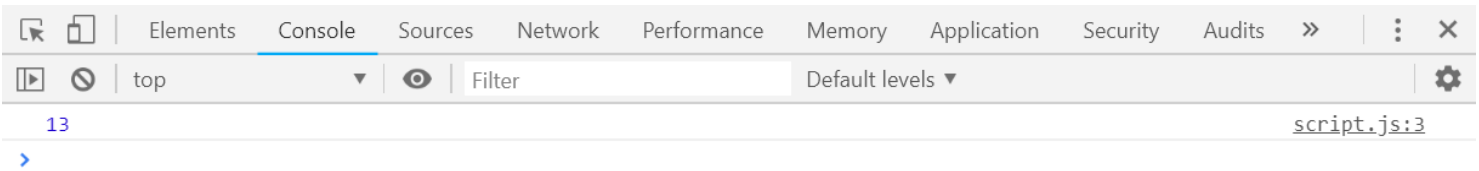
```
var a = 12;
var b = '1';
console.log(a + b);
```



Number + Boolean = number

Script.js

```
var a = 12;
var b = true;
console.log(a + b);
```

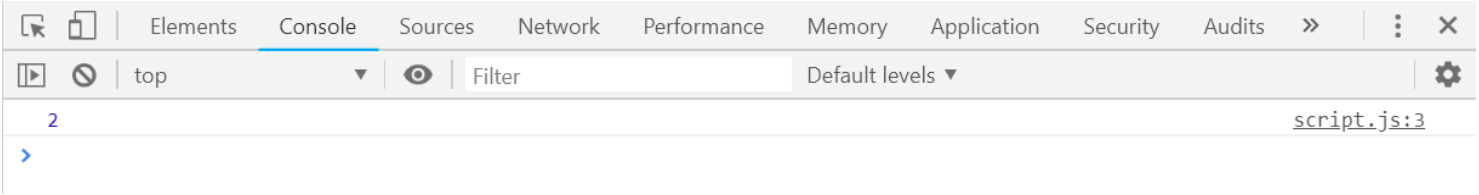


12 + true = 13 because as we've learnt, true also equals 1.

True + true = 2

Script.js

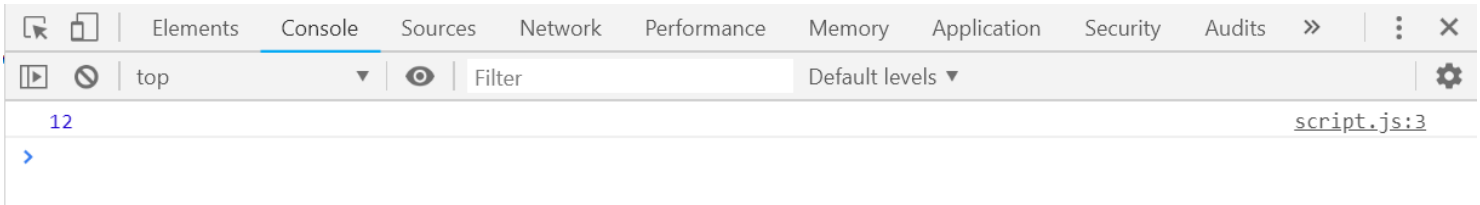
```
var a = true;
var b = true;
console.log(a + b);
```



Numbers + null = the number. Null will translate to zero.

Script.js

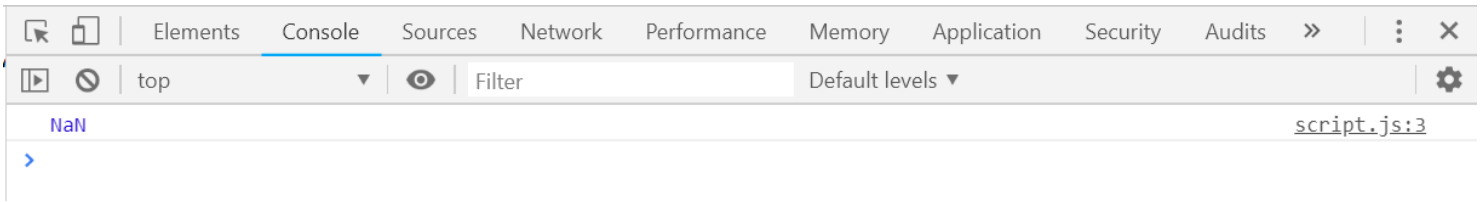
```
var a = 12;
var b = null;
console.log(a + b);
```



Number + undefined = NaN

Script.js

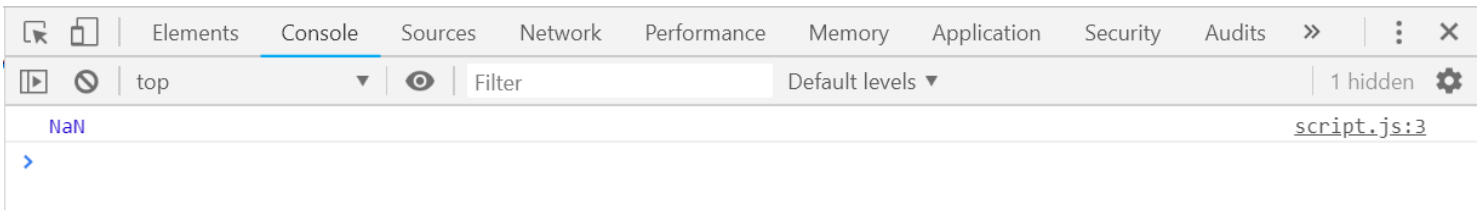
```
var a = 12;
var b = undefined;
console.log(a + b);
```



Number + NaN = NaN

Script.js

```
var a = 12;
var b = NaN;
console.log(a + b);
```

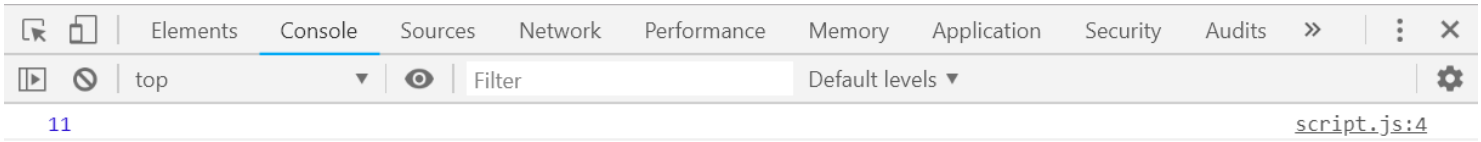


OPERATORS – SUBTRACTION:

-=

Script.js

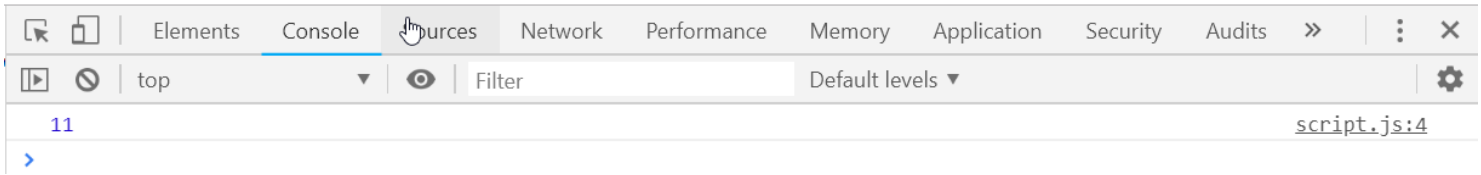
```
var a = 12;
var b = 1;
a -= b; //a-= b means a = a – b
console.log(a);
```



--

Script.js

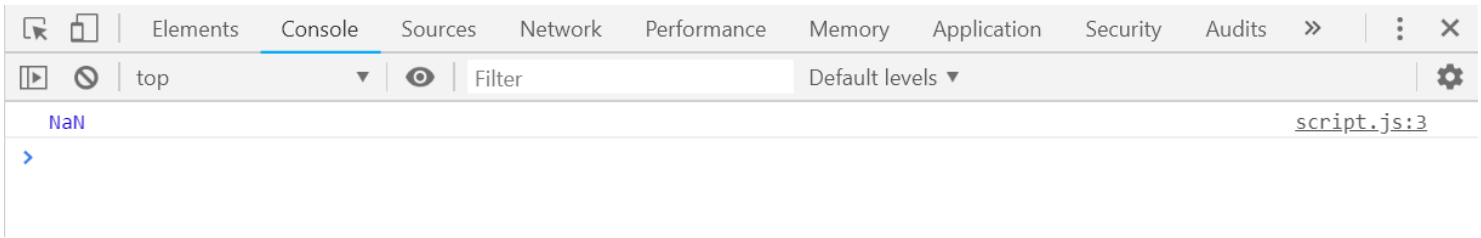
```
var a = 12;
var b = 1;
a -- ; //a-- means a - 1
console.log(a);
```



String – string = NaN

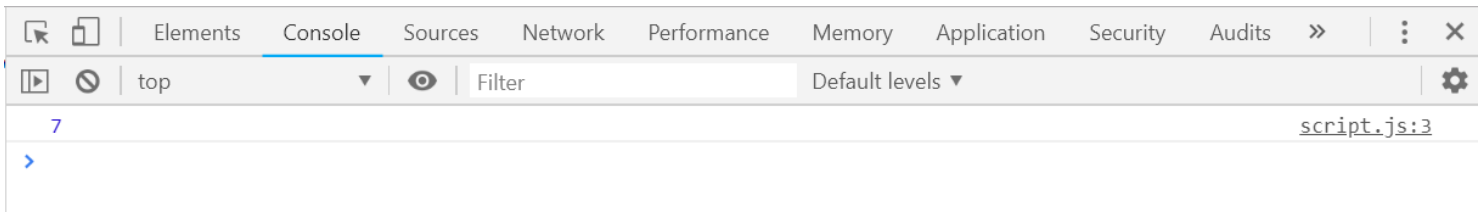
Script.js

```
var a = 'split us';
var b = 'us';
console.log(a-b);
```



Script.js

```
var a = 12;
var b = "5";
console.log(a - b);
```



OPERATORS – MULTIPLICATION FLOATING POINT PROBLEMS:

***=**

Script.js

```
var a = 12;
var b = 2;
a *= b // means a = a * b
console.log(a);
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels

```
24                                                    script.js:4
>
```

Script.js

```
var a = 1.3;
var b = 2;
console.log(a * b);
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels

```
2.6                                                   script.js:3
>
```

Script.js

```
var a = 1.3;
var b = 2.2;
console.log(a * b);
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels

```
2.8600000000000003                                    script.js:3
>
```

The answer should be 2.86 but we are getting this answer as a bug in JS when multiplying two decimal number.

SOLUTION

Script.js

```
var a = 1.3;
var b = 2.2;
var c = a * b
console.log(c.toFixed(2)); //Using .toFixed to solve the bug. .toFixed(2) write the
result into 2 decimal places i.e. only two figures after the decimal
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels 1 hidden

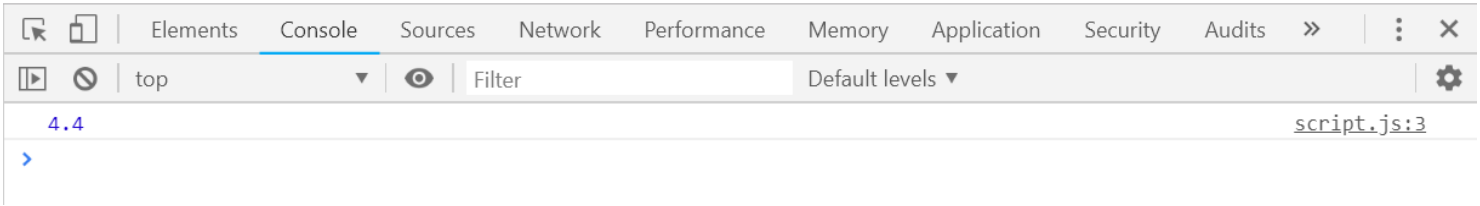
```
2.86                                                  script.js:4
>
```

Script.js

```
var a = 1.3;
var b = 2.2;
var c = a * b
console.log(c.toFixed(2)); //Using .toFixed to solve the bug. .toFixed(2) write the
result into 2 decimal places i.e. only two figures after the decimal
if ( c.toFixed(2) == 2.86){
  console.log('true');
} else {
  console.log('false');
} //Using if statement to check if the value of c equals 2.86
```

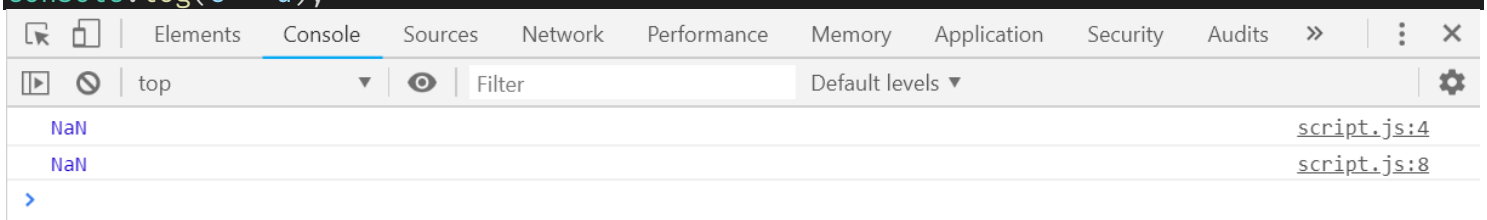
Script.js

```
var a = 2;
var b = "2.2";
console.log(a * b);
```



Script.js

```
//first operation
var a = 2;
var b = "text";
console.log(a * b);
//Second operation
var c = 'join';
var d = 'us';
console.log(c * d);
```



Script.js

```
var a = 2;
var b = null;
```

```
console.log(a * b); //we will get zero because null is interpreted as 0
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels 1 hidden

```
0                                                    script.js:3
>
```

Script.js

```
var a = 2;
var b = Infinity;
console.log(a * b);
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels

```
Infinity                                             script.js:3
>
```

2 * Infinity simply means multiplying 2 by an infinity value. We will definitely get an infinity value in return.

OPERATORS – DIVISION MODULUS:

Script.js

```
var a = 10;
var b = 2;
a /= b // means a = a / b
console.log(a);
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels

```
5                                                    script.js:4
>
```

Script.js

```
var a = 10;
var b = "2";
console.log(a / b);
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels

```
5                                                    script.js:3
>
```

Script.js

```
var a = 3.3;
var b = 2.2;
console.log((a / b).toFixed(2));
```

Elements Console Sources Network Performance Memory Application Security Audits

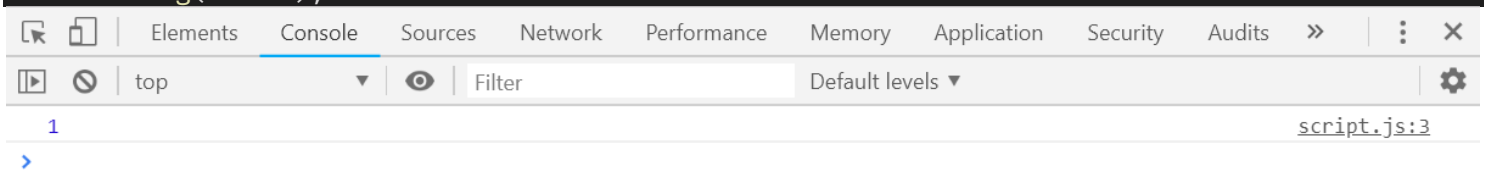
top Filter Default levels

```
1.50                                                 script.js:3
>
```

Modulus (%) : This gives us the remainder number in a division. 10 % 3 will give us 1 which means remainder 1.

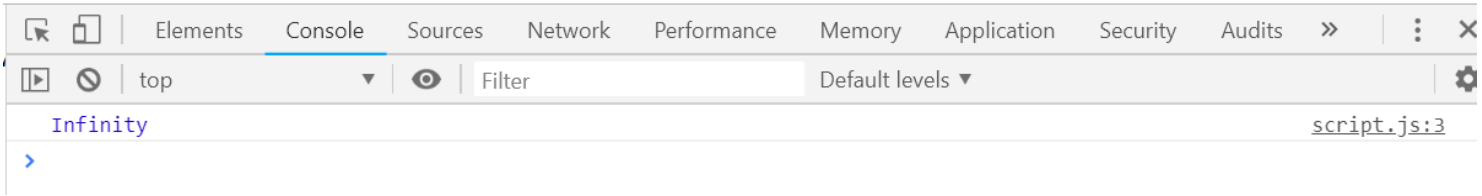
Script.js

```
var a = 10;
var b = 3;
console.log(a % b);
```



Script.js

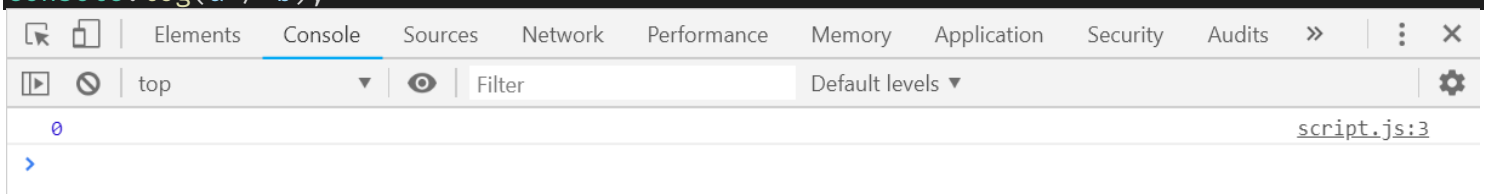
```
var a = 10;
var b = 0;
console.log(a / b);
```



JS takes this that dividing 10 with a very small number 0 should give a very big result i.e. infinity.

Script.js

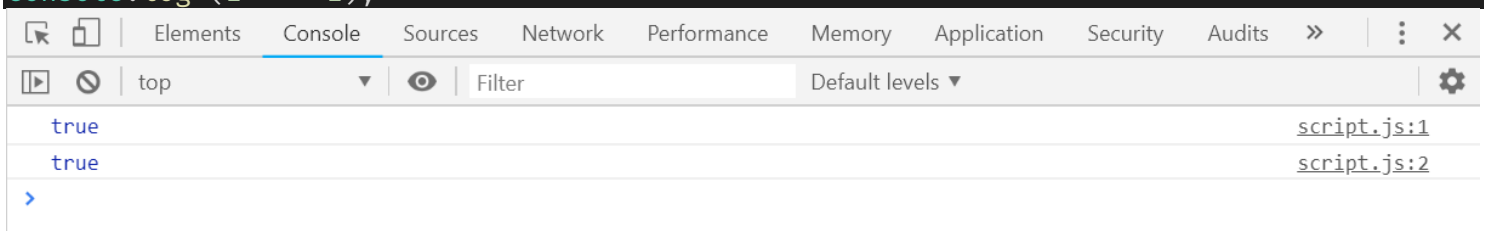
```
var a = 10;
var b = Infinity;
console.log(a / b);
```



OPERATORS – COMPARING VALUES:

Script.js

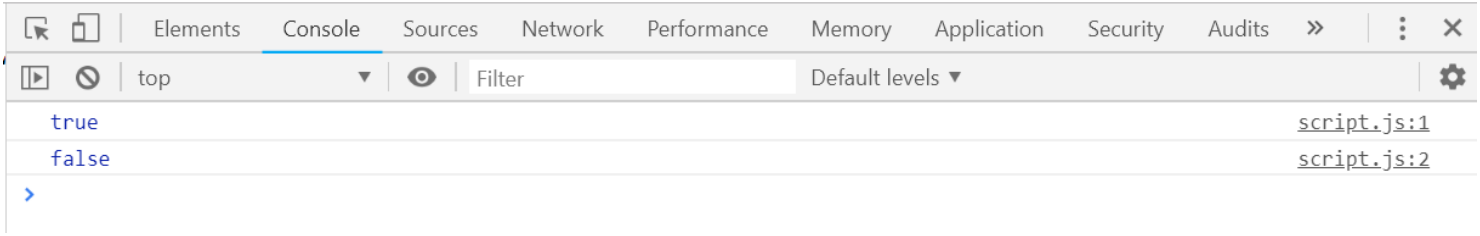
```
console.log (1 == 1);
console.log (1 === 1);
```



2 equal signs check if the value is the same while 3 equal signs checks if the type is the same.

Script.js

```
console.log (1 == "1"); //we get true for this because JS can detect that we have 1 inside the ""
console.log (1 === "1"); //we get false because they are of different types : number & string
```

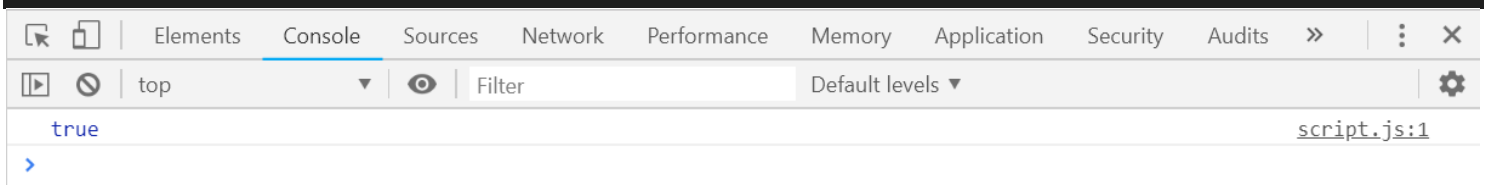


!=

This means “not equal”

Script.js

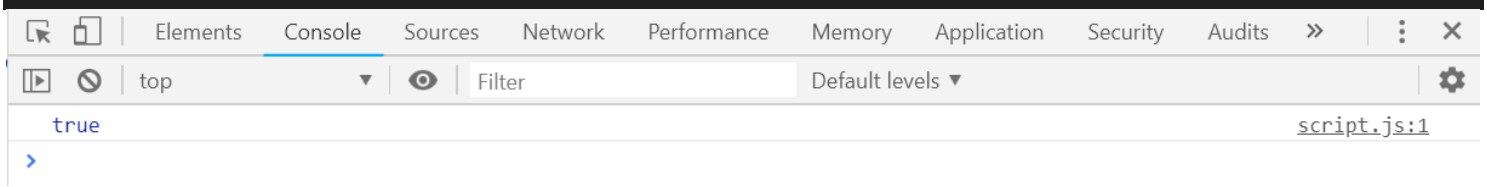
```
console.log (1 != 2); //checks if 1 is not equal to 2
```



!==

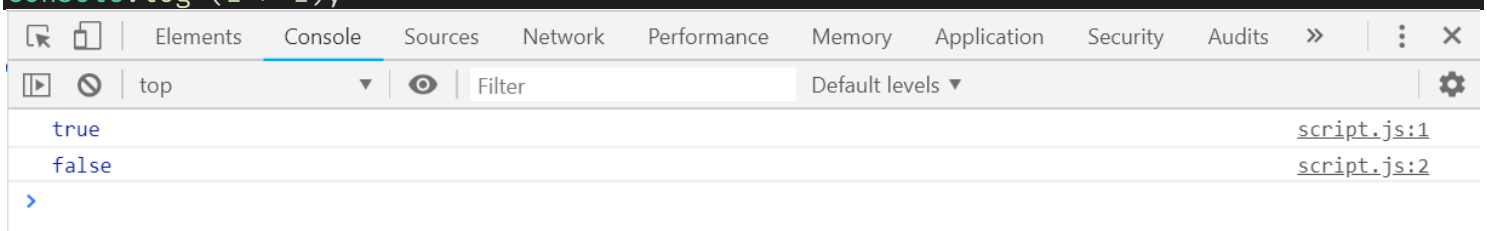
Script.js

```
console.log (1 !== '1'); //checks if the value and type of 1 is not the same as '1'
```



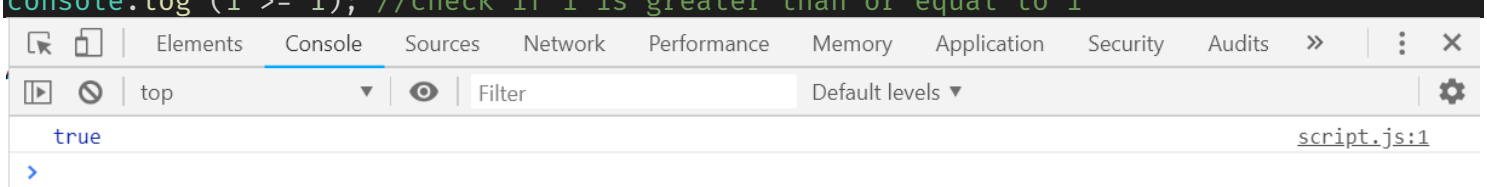
script.js

```
console.log (1 > 0);
console.log (1 > 1);
```



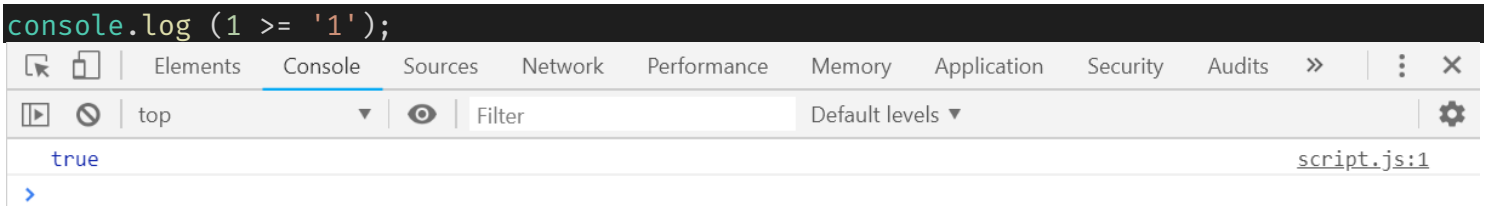
Script.js

```
console.log (1 >= 1); //check if 1 is greater than or equal to 1
```



script.js

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

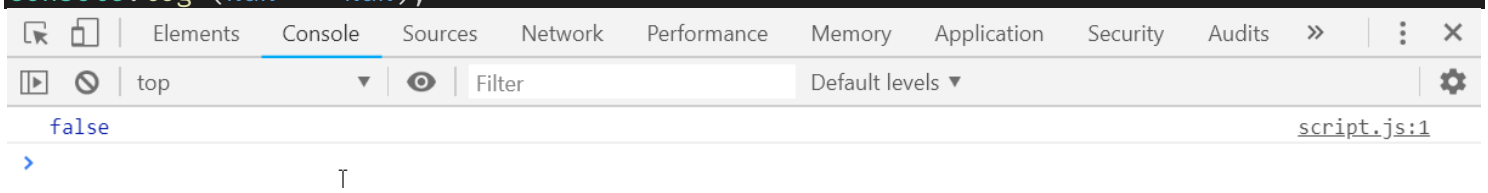


OPERATORS – IMPORTANT RULES:

1. **NaN != NaN**

Script.js

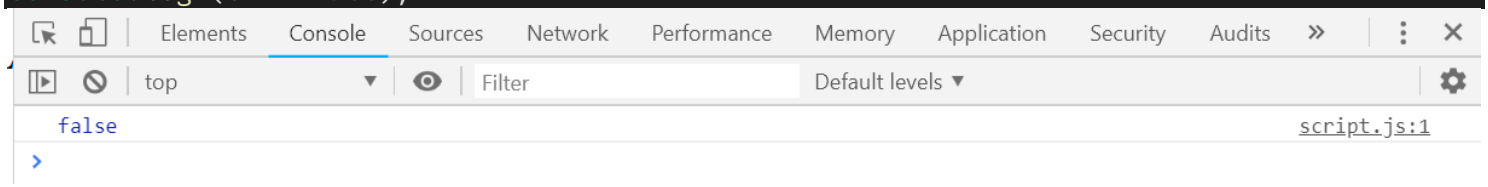
```
console.log (NaN == NaN);
```



2. **0 != null**

Script.js

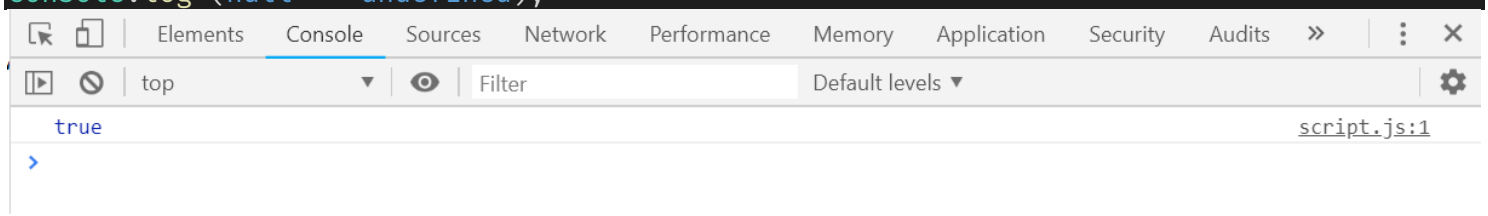
```
console.log (0 == null);
```



3. **Null == undefined**

Script.js

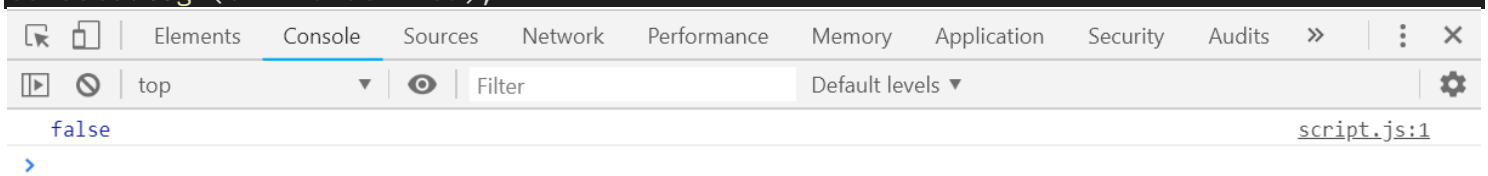
```
console.log (null == undefined);
```



0 != undefined

Script.js

```
console.log (0 == undefined);
```

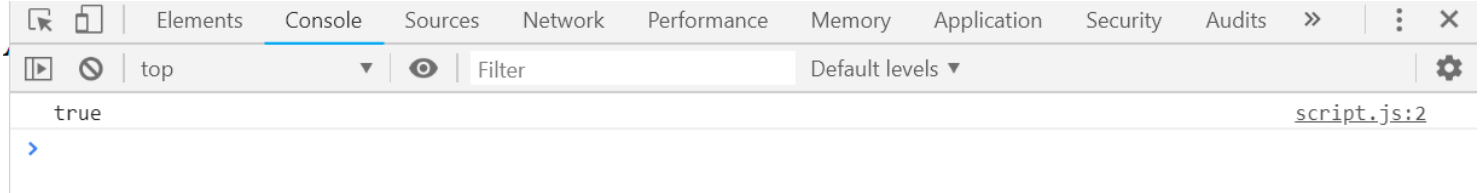


Summary: null can't be compared except with undefined.

OPERATORS – BOOLEAN &&: The && operator will check if both condition on the left and right side are true. If one of the condition is not then we get false. It joins two conditions together as a single condition. So both sides must be valid.

Script.js

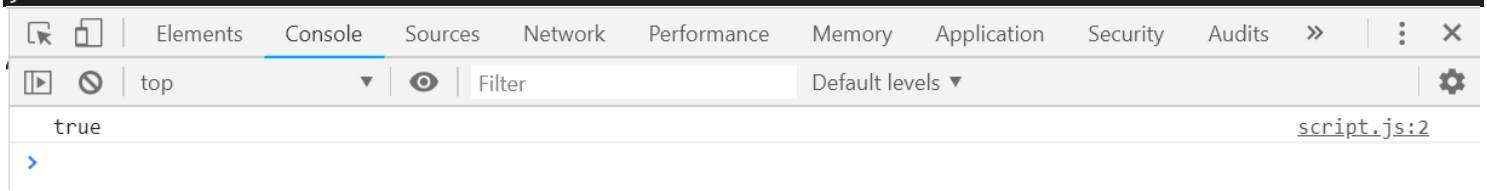
```
if (1 == 1 && 2 == 2){
  console.log('true')
} else {
  console.log('false')
}
```



The code can also be re-written has this:

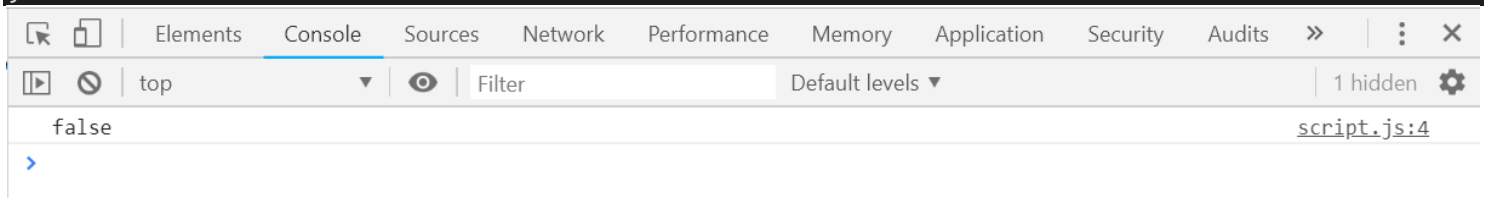
Script.js

```
if ((1 == 1) && (2 == 2)) {
  console.log('true')
} else {
  console.log('false')
}
```



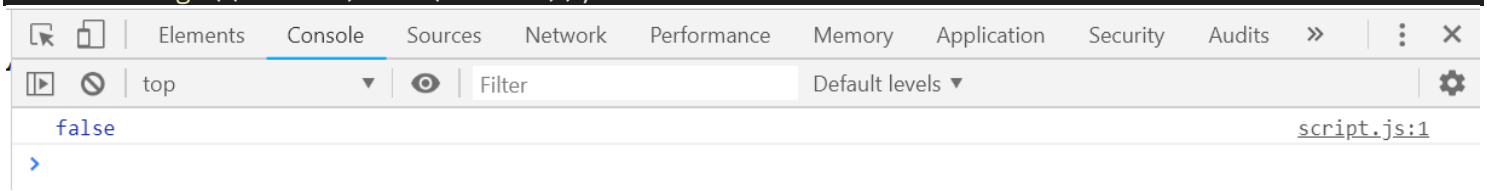
script.js

```
if ((1 == 1) && (2 == 3)) {
  console.log('true')
} else {
  console.log('false')
}
```



Script.js

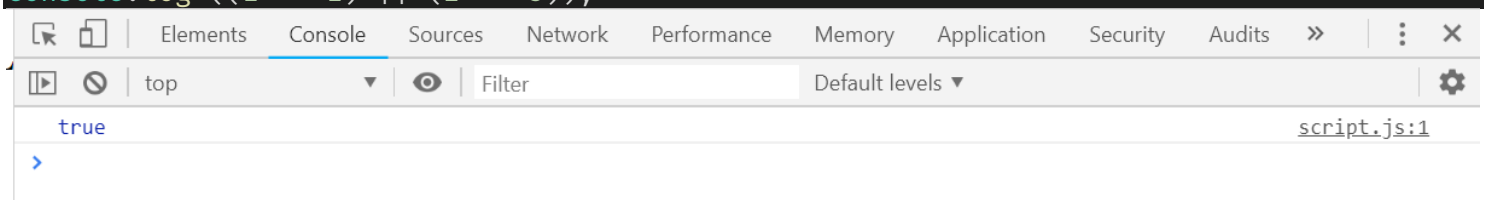
```
console.log ((1 == 1) && (2 == 3));
```



The || operator: This can be read as OR. It checks if any of the condition on both sides is true and return true until no condition is.

Script.js

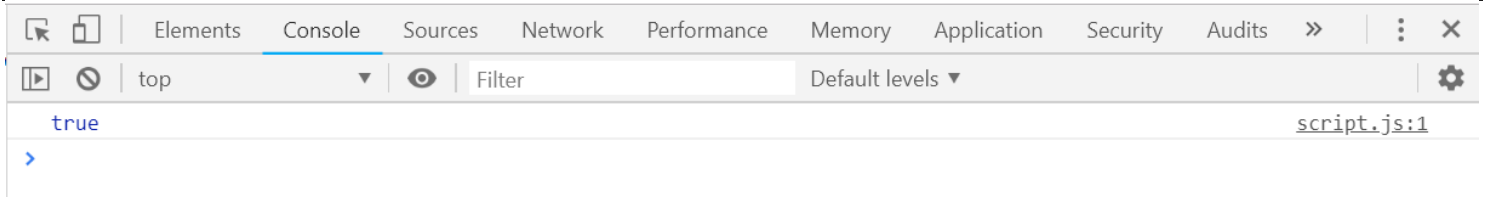
```
console.log ((1 == 1) || (2 == 3));
```



You can run multiples

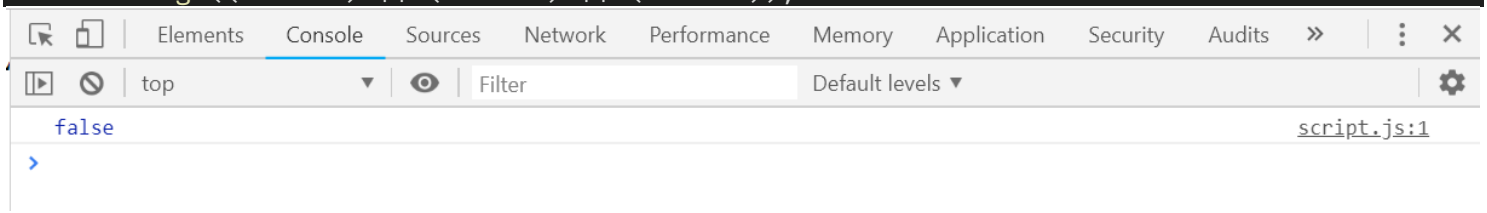
Script.js

```
console.log ((1 == 1) || (2 == 3) || (4 == 5));
```



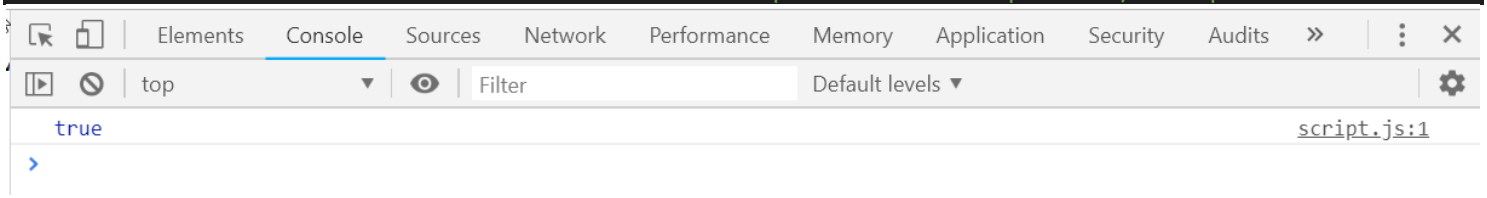
Script.js

```
console.log ((1 == 2) || (2 == 3) || (4 == 5));
```



Script.js

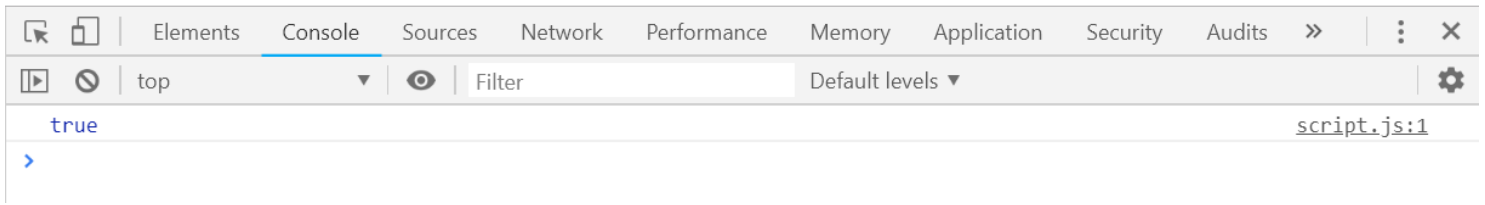
```
console.log ((1 == 1) || (2 == 3) && (4 == 5)); // the && joins two conditions 2geda as one so the statement now reads : check if 1 equals 1 or 2 equals 3, 4 equals 5
```



Script.js

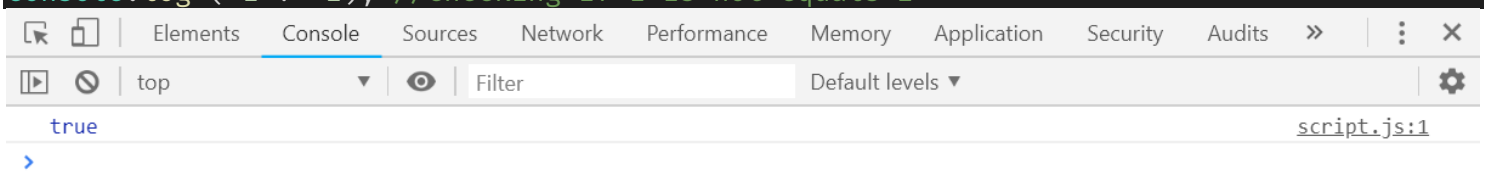
```
console.log ((3 == 1) && (2 == 3) || (4 == 4));
```

This translates as check if 3 equals 1 and also 2 must be equal to 3 OR 4 equals 4.



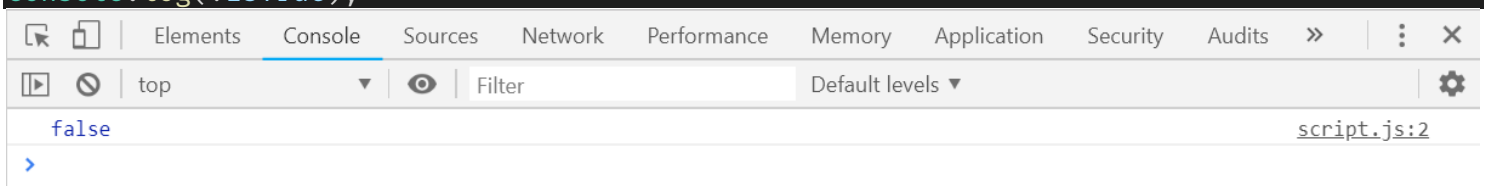
Script.js

```
console.log ( 1 != 2); //checking if 1 is not equals 2
```



Script.js

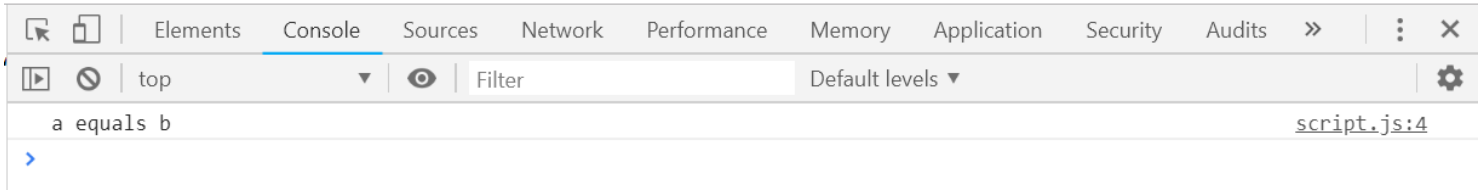
```
var isTrue = true;
console.log(!isTrue);
```



OPERATORS – TENARY OPERATORS: **This a shorter way of writing some common if statements. Like examine an if statement below.**

Script.js

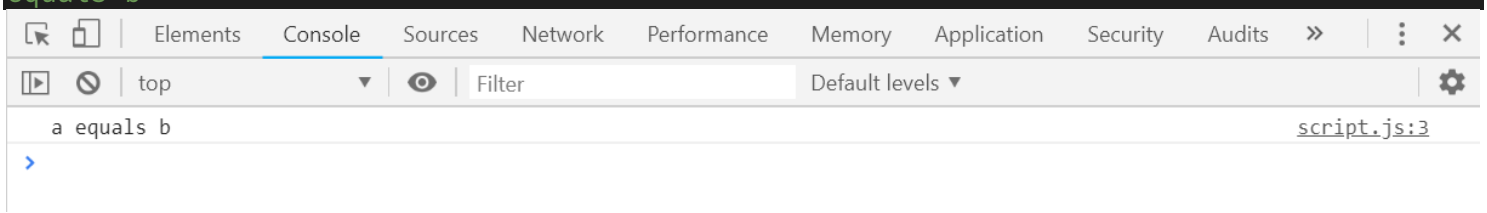
```
var a = 5;
var b = 5;
if ( a == b) {
  console.log ('a equals b');
} else {
  console.log ('a is not equal b');
}
```



There is a shorter way of writing the if statement above using ternary operator.

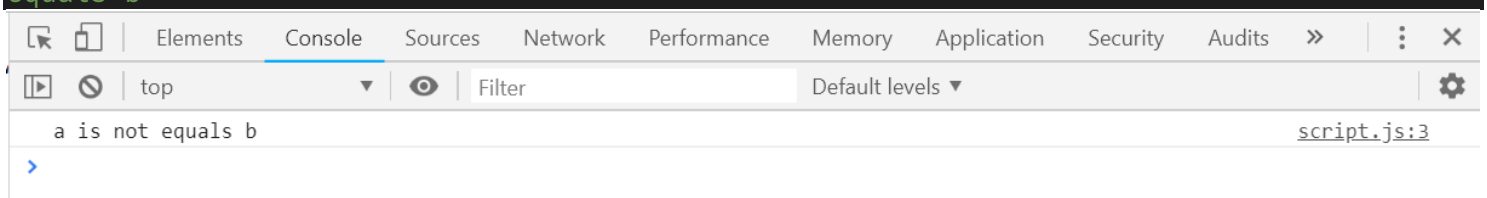
Script.js

```
var a = 5;
var b = 5;
console.log ( a == b ? 'a equals b' : 'a is not equals b');
//The ? sign means "then", The : sign means "else"
//Now we can read it as: while a equals b "then" print a equals b "else" print a is not
equals b
```



script.js

```
var a = 5;
var b = 9;
console.log ( a == b ? 'a equals b' : 'a is not equals b');
//The ? sign means "then", The : sign means "else"
//Now we can read it as: while a equals b "then" print a equals b "else" print a is not
equals b
```



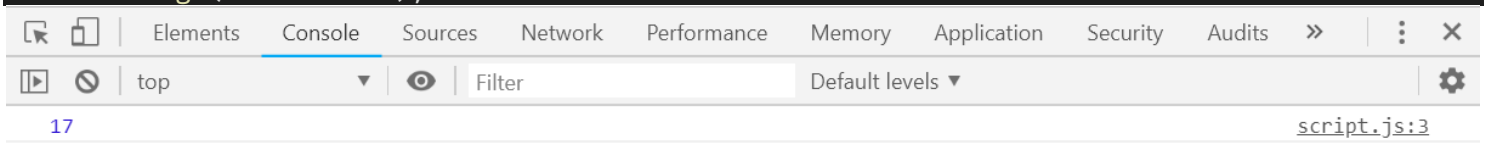
So that is the ternary operator for you but NOTE: You should only use it for conditional statements that are pretty simple and straightforward else you will write codes that are not easy to read.

OPERATORS – PRECEDENCE: **We use this to show which operator takes precedence/priority over the other. To break this down, precedence determines which operator gets solved first just like BODMAS determines in mathematics.**

Script.js

```
var a = 5;
```

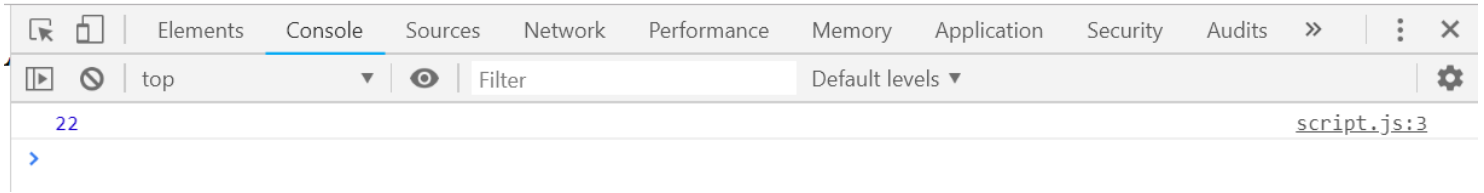
```
var b = 6;
console.log ( a + b * 2);
```



We can see that we get 17 because JS first multiplied 6 by 2 and the add it to the additional operator 5.

Script.js

```
var a = 5;
var b = 6;
console.log ((a + b) * 2);
```



Now we get 22 because we've enclosed a + b in a bracket and according to JS precedence bracket get solved first. For more about precedence, check this link –

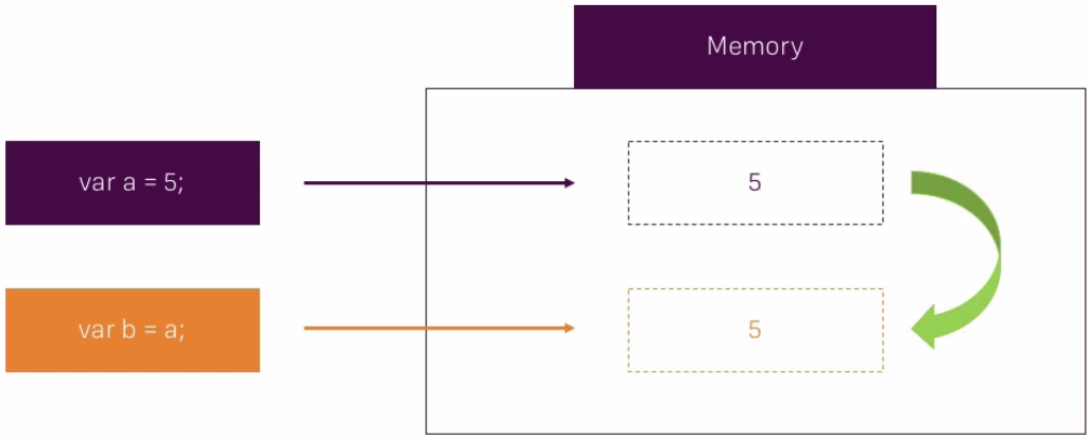
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

WRAP UP: Now we've covered a lot about language basics and this are the basic destiny of JS. Now we can move forward.

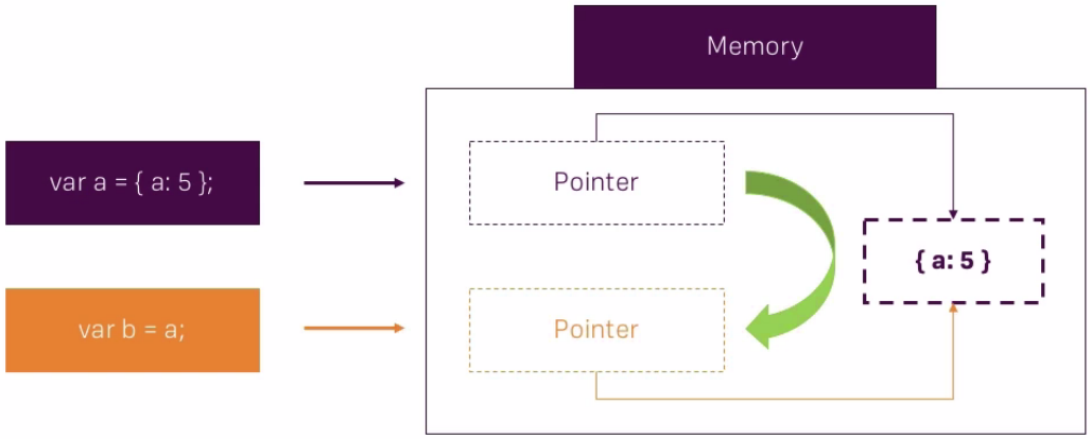
TYPES AND SCOPE

PRIMITIVE & REFERENCE TYPES: We've learnt about types that variables are of different types e.g. Boolean, number, string etc. This types are of two categories:

1. **Primitive type – this is the first category which can be Boolean, string or number.**



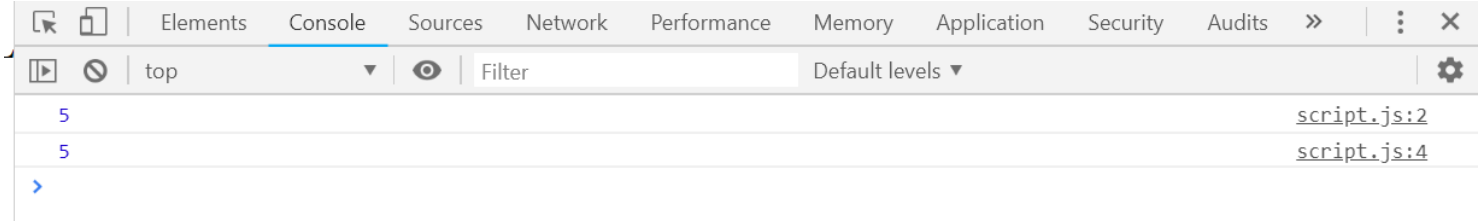
2. **REFERENCE TYPE – This are arrays or you call them objects.**



PRIMITIVE VS REFERENCE TYPES – EXERCISE:

Script.js

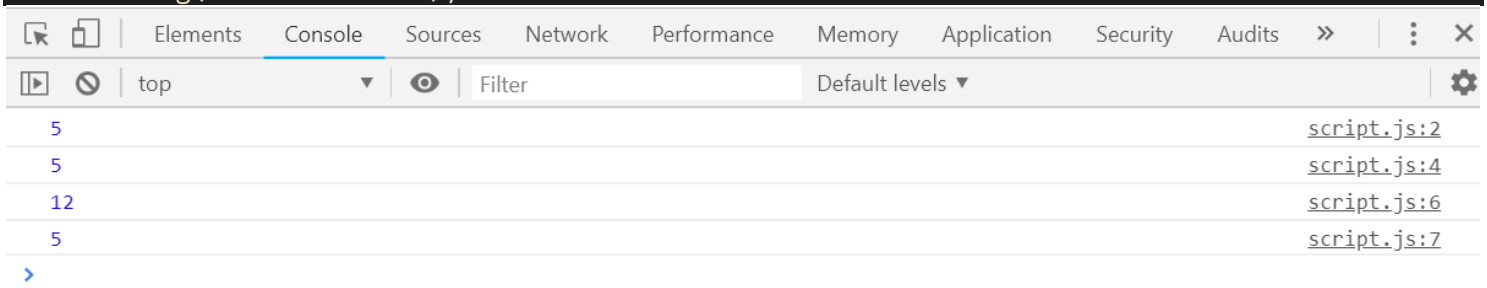
```
var aNumber = 5;
console.log(aNumber);
var anotherNumber = aNumber;
console.log(anotherNumber);
```



Now lets examine this

Script.js

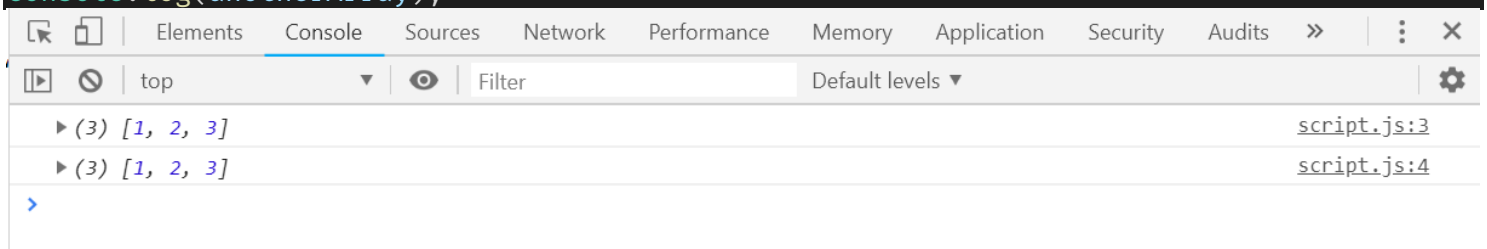
```
var aNumber = 5;
console.log(aNumber);
var anotherNumber = aNumber;
console.log(anotherNumber);
var aNumber = 12;
console.log(aNumber);
console.log(anotherNumber);
```



surprisingly we can see that even after changing aNumber to 12, another number still retains the first value of 5. This is how primitive types works, they retain the first value.

Script.js

```
var array = [1, 2, 3];
var anotherArray = array;
console.log(array);
console.log(anotherArray);
```



Script.js

```
var array = [1, 2, 3];
var anotherArray = array;
console.log(array);
console.log(anotherArray);

array.push(4); //this push a new element '4' into the array and increase the array length
console.log(array);
console.log(anotherArray);
```

```
▸ (3) [1, 2, 3]                    script.js:3
▸ (3) [1, 2, 3]                    script.js:4
▸ (4) [1, 2, 3, 4]                 script.js:7
▸ (4) [1, 2, 3, 4]                 script.js:8
>
```

Here we can see that when we increase the length of array, anotherArray also changed. This is because it's a referenced type.

Script.js

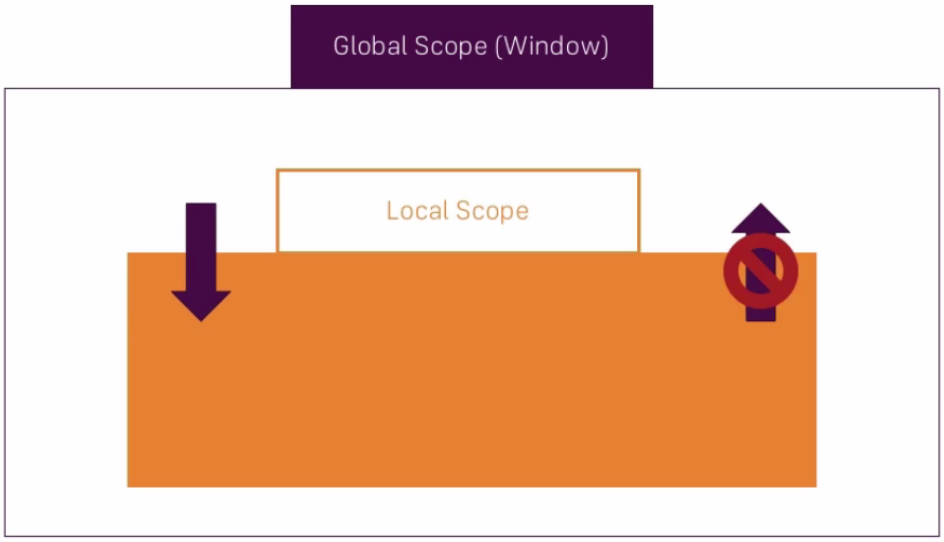
```
var array = [1, 2, 3];
var anotherArray = array;

array = ['a', 'b'];
console.log(array);
console.log(anotherArray);
```

```
▸ (2) ["a", "b"]                   script.js:5
▸ (3) [1, 2, 3]                    script.js:6
>
```

Funny enough we can see what happens now when we assigned another value to array, anotherArray still retains the first value even though it's a ref type. Why? This is because unlike before when anotherArray took the value we changed to, the value then was only modified by pushing a new value but here now the value was changed completely.

GLOBAL VS LOCAL SCOPE: Scope is the registry on which your variables are registered. We have two types of scope: The Global scope – which is the top most level where our variables are registered by default. We also have local scope – This is nested inside the global scope. Local scope is created when we create a function.

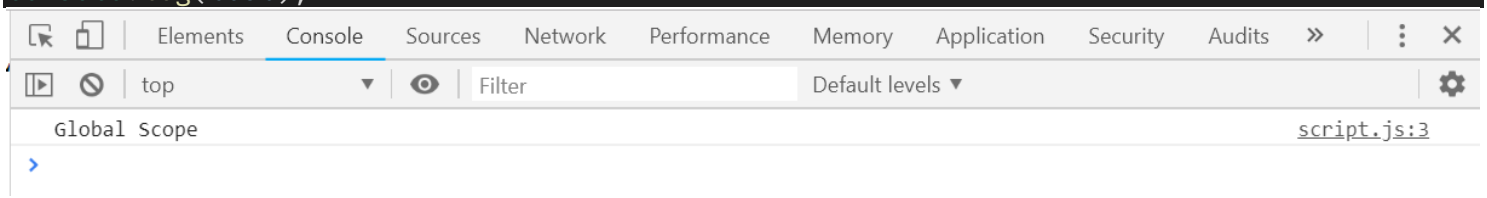
Our variables are registered in the global scope by default.



GLOBAL SCOPE VS LOCAL SCOPE – EXERCISE:

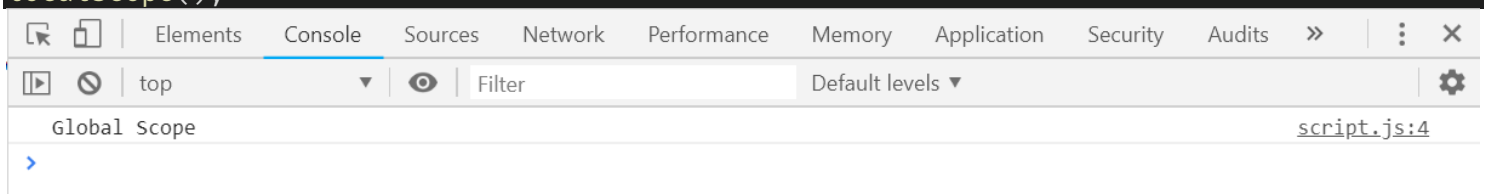
Script.js

```
//Global scope registry
var test = 'Global Scope';
console.log(test);
```



script.js

```
//Calling a global scope from a function
var test = 'Global Scope';
function localScope() {
  console.log(test);
}
localScope();
```



Creating our local scope

Script.js

```
//Creating our local scope inside our function
var test = 'Global Scope';
function localScope() {
  var test = 'local scope'
```

```
  console.log(test);
}
localScope();
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  »
top  Filter  Default levels
local scope                                                    script.js:5
>
```

Script.js

```
var test = 'Global Scope';
function localScope() {
  var test = 'Local scope'
  console.log(test); // @ ds junction, our local scope is called which we have inside
the function
}
localScope();
console.log(test); // here we are outside the function and so global scope will be
called because local scope cannot be refrenced outside its function
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  »
top  Filter  Default levels
Local scope                                                    script.js:4
Global Scope                                                   script.js:7
>
```

Script.js

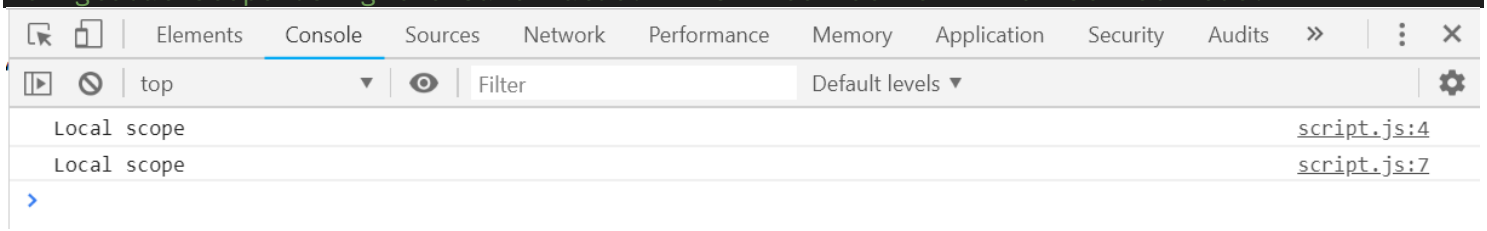
```
function localScope() {
  var test = 'Local scope'
  console.log(test); // @ ds junction, our local scope is called which we have inside
the function
}
localScope();
console.log(test); // here we will get an error because though we have variable 'test'
defined inside local scope but it stays inside it, it can be refrenced from outside of
the function
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  »  1
top  Filter  Default levels
Local scope                                                    script.js:4
▸Uncaught ReferenceError: test is not defined                  script.js:7
    at script.js:7
>
```

script.js

```
function localScope() {
  test = 'Local scope'
  console.log(test);
}
localScope();
```

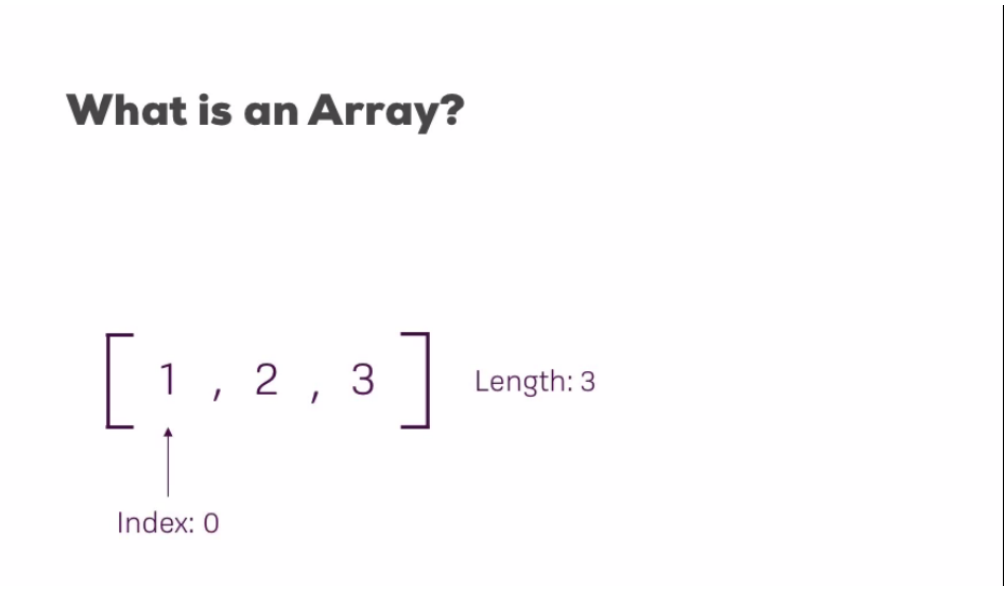
```
console.log(test); // though we dont have a global scope set but because we have removed 'var' keyword from the local scope definition, JS will automatically generate a new global scope using the same value. This will not work with strict mode.
```



Script.js

ARRAYS

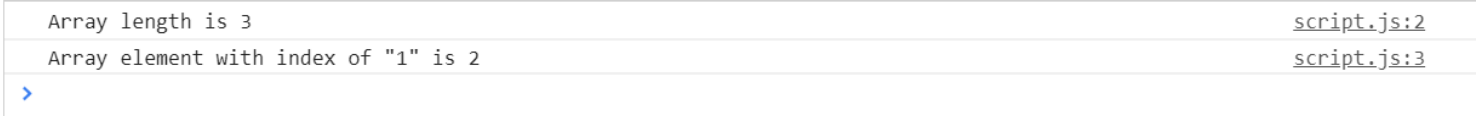
INTRODUCTION: An array is a collection of items which we use square brackets and comma to create – [1, 2, 3]



BASICS – MANAGING ELEMENTS:

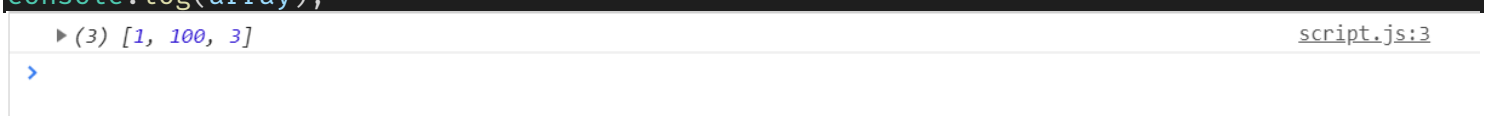
Script.js

```
var array = [1, 2, 3];
console.log ('Array length is ' + (array.length)); //getting the length of the array
console.log ('Array element with index of "1" is ' + (array[1])); //calling each element with there index
```



Script.js

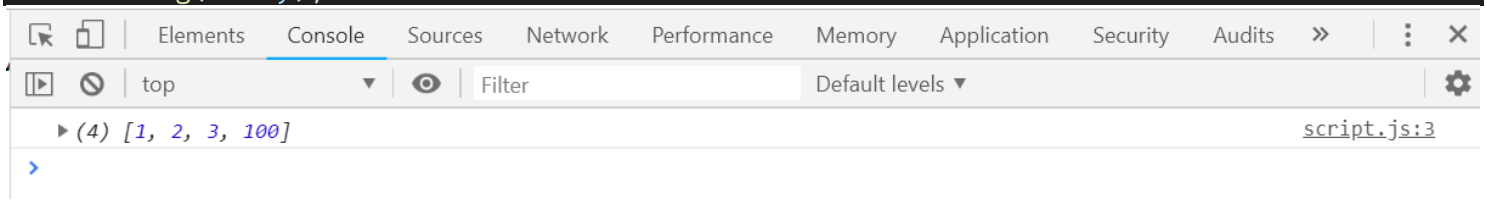
```
var array = [1, 2, 3];
array[1] = 100; //you can also use the index number to change the element
console.log(array);
```



Script.js

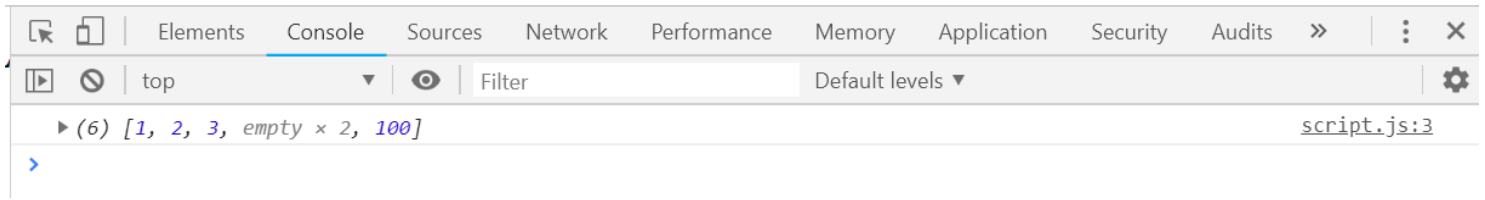
```
var array = [1, 2, 3];
```

```
array[3] = 100; //the maximum index we have in the array is 2 but since we are giving a value to the 3rd index which is currently undefined, we will get a 4th element in the array using the value.
console.log(array);
```



Script.js

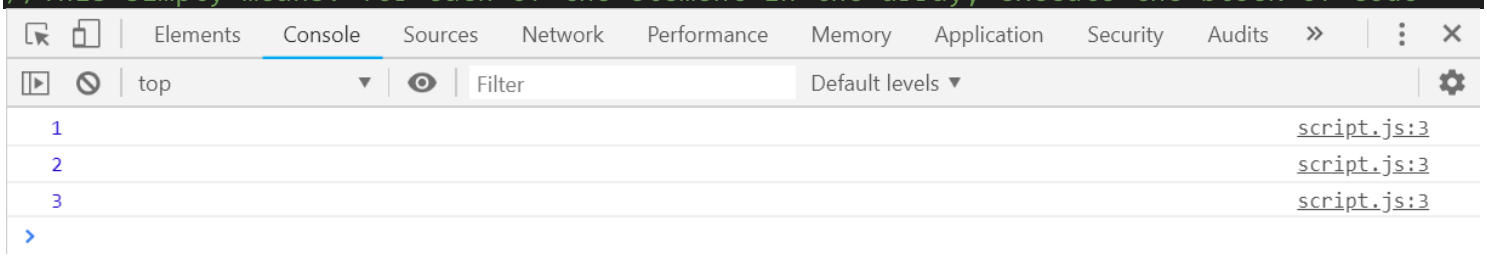
```
var array = [1, 2, 3];
array[5] = 100;
console.log(array);
```



THE forEach METHOD:

Script.js

```
var array = [1, 2, 3];
array.forEach(function(element){
  console.log(element)
});
//This simply means: for each of the element in the array, execute the block of code
```



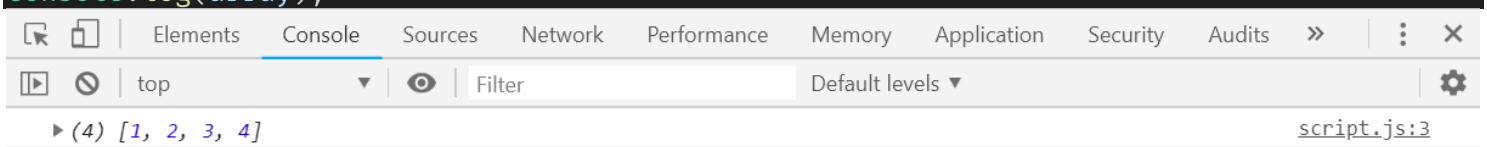
Not only can you use the forEach method to print each element in the array, you can do anything else that is possible with JS with each of the array element.

WORKING WITH ELEMENTS – PUSH POP SHIFT & UNSHIFT:

PUSH – This basically adds an element to the end of the array.

Script.js

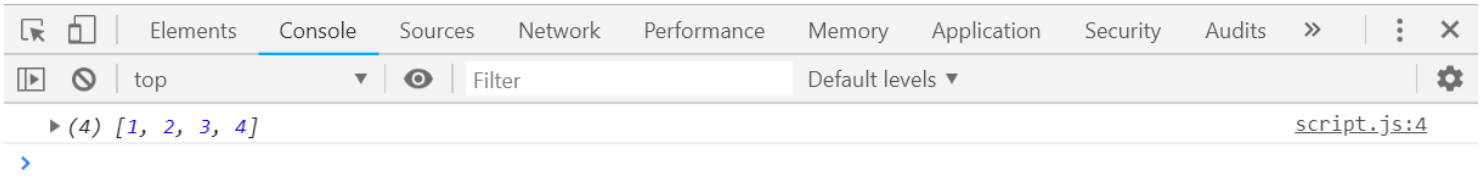
```
var array = [1, 2, 3];
array.push(4);
console.log(array);
```



Script.js

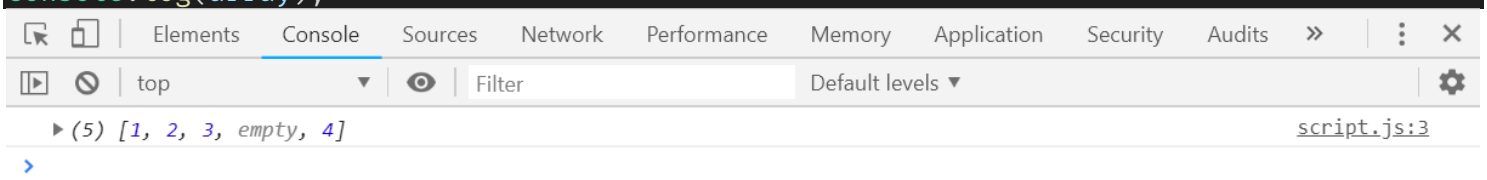
```
var array = [1, 2, 3];
```

```
array[3] = 4; //you can also use this method instead but push is better because for a long array, you will need to know the index of the last number to add a new one at the back
// array.push(4);
console.log(array);
```



Script.js

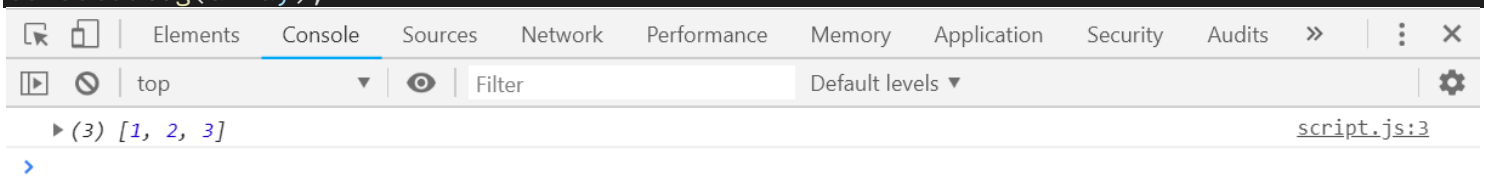
```
var array = [1, 2, 3,,]; //we are creating an empyt/undefined space
array.push(4);
console.log(array);
```



POP – This removes the last element from an array and return the array

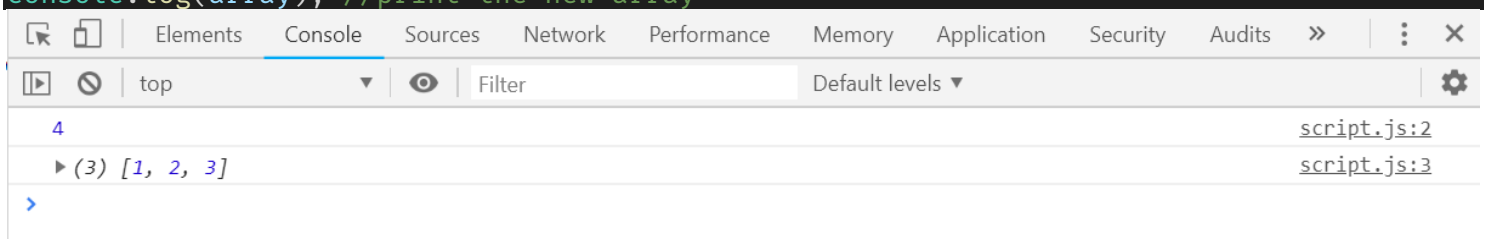
Script.js

```
var array = [1, 2, 3, 4];
array.pop()
console.log(array);
```



Script.js

```
var array = [1, 2, 3, 4];
console.log(array.pop()); //print the element that was poped (taken away)
console.log(array); //print the new array
```



SHIFT – This takes away the first element in the array (opposite of POP)

Script.js

```
var array = [1, 2, 3, 4];
console.log(array.shift()); //print the first element that was shifted (taken away)
console.log(array); //print the new array
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
1                                                    script.js:2
▶ (3) [2, 3, 4]                                      script.js:3
>
```

UNSHIFT – This inserts new elements to the start of an array.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new'); //Add a new elements at the begining of the array.
console.log(array);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▶ (5) ["new", 1, 2, 3, 4]                            script.js:3
>
```

WORKING WITH PARTS OF AN ARRAY – INDEXOF, SPLICE & SLICE:

INDEX OF - Returns the index of the first occurrence of a value in an array.

Script.js

```
var array = [1, 2, 3, 4];
console.log(array.indexOf(3)); //we are asking for the index of the element 3
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
2                                                    script.js:2
>
```

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
console.log(array);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▶ (5) ["old", 1, 2, 3, 4]                            script.js:4
>
```

Here in the code above, we called the index of the element we added 'new' and we replace it with a new element 'old'.

SPLICE - Removes elements from an array and, if necessary, inserts new elements in their place, returning the deleted elements.

Script.js

```
var array = [1, 2, 3, 4];
console.log(array.splice(2));
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (2) [3, 4]                                                    script.js:2
>
```

Here the program spliced the elements starting from index of 2 i.e. they are removed from the array and logged to the console.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
var newArray = array.splice(3);
console.log(newArray);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (2) [3, 4]                                                    script.js:5
>
```

Here the program first add a new element 'new' to the array, the newly added element is then replaced with 'old' and now we spliced from index of 3 to the last element in the array and logged their values to the console.

Telling JS the number of elements in the array to splice.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
var newArray = array.splice(2, 2); //The first number '2' is the index the splice
should start with. the second number tells JS how many elements to remove
console.log(newArray);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (2) [2, 3]                                                    script.js:5
>
```

We are splicing two elements starting from index of 2.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
var newArray = array.splice(2, 2); //The first number '2' is the index the splice
should start with. the second number tells JS how many elements to remove
console.log(newArray);
console.log(array); // rememeber we already spliced 2 & 3 so this will return only the
unspliced elements
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (2) [2, 3]                                  script.js:5
▸ (3) ["old", 1, 4]                           script.js:6
>
```

SLICE - Returns a section of an array. Slice will not remove the elements from the array, it will only return a part.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
var newArray = array.slice(2, 4); // slice(2, 4) means return the elements from index 2
to index 4
console.log(newArray);
console.log(array); // since slice is not removing our elements but only returning a
part of it, we still get our array untouched
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (2) [2, 3]                                  script.js:5
▸ (5) ["old", 1, 2, 3, 4]                     script.js:6
>
```

FILTERING, MAPPING & REVERSING ARRAY ELEMENTS:

FILTER - Returns the elements of an array that meet the condition specified in a callback function.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
console.log(array.filter(function(value){
  return value > 2
})); // This will filter for array elements greater than 2 and log them in the console
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels  1 hidden
▸ (2) [3, 4]                                  script.js:4
>
```

MAP - Calls a defined callback function on each element of an array, and returns an array that contains the results. i.e. map will do something with each of the array element and return the result as a new array

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
console.log(array.map(function(value){
  return value * 2
})); // This will multiply all the array element by 2 and return the result in a new
array
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (5) [NaN, 2, 4, 6, 8]                                    script.js:4
>
```

Bear in mind that with filter and map, we are not changing our original array, we are only using them for functions. We can confirm our original array after mapping or filtering.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
console.log(array.map(function(value){
  return value * 2
}));
console.log(array); //confirming our original array
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (5) [NaN, 2, 4, 6, 8]                                    script.js:4
▸ (5) ["old", 1, 2, 3, 4]                                  script.js:7
>
```

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
console.log(array.filter(function(value){
  return value > 2
}));
console.log(array); //confirming our original array after filtering
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (2) [3, 4]                                               script.js:4
▸ (5) ["old", 1, 2, 3, 4]                                  script.js:7
>
```

REVERSE - Reverses the elements in an Array.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
console.log(array.reverse());
console.log(array); //confirming our original array after filtering
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (5) [4, 3, 2, 1, "old"]                    script.js:4
▸ (5) [4, 3, 2, 1, "old"]                    script.js:5
>
```

As we can see, even our original array got reversed, so reverse affects our original array unlike map and filter.

THE DIFFERENCE BETWEEN CONCAT & JOIN:

CONCAT – This join two arrays together as one. It doesn't change our original array but creates a new one.

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
var newArray = ['Join', 'Me'];
console.log(array.concat(newArray));
console.log(array);
console.log(newArray);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ (7) ["old", 1, 2, 3, 4, "Join", "Me"]      script.js:5
▸ (5) ["old", 1, 2, 3, 4]                    script.js:6
▸ (2) ["Join", "Me"]                         script.js:7
>
```

JOIN – This doesn't join two arrays as one but mix up the array elements to form a string

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
var newArray = ['Join', 'Me'];
console.log(array.join(newArray));
console.log(array);
console.log(newArray);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
oldJoin,Me1Join,Me2Join,Me3Join,Me4          script.js:5
▸ (5) ["old", 1, 2, 3, 4]                    script.js:6
▸ (2) ["Join", "Me"]                         script.js:7
>
```

script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
var newArray = ['Join', 'Me'];
console.log(array.join(' lol, '));
console.log(array);
console.log(newArray);
```

```
old lol, 1 lol, 2 lol, 3 lol, 4                    script.js:5
▶(5) ["old", 1, 2, 3, 4]                           script.js:6
▶(2) ["Join", "Me"]                                script.js:7
>
```

Script.js

```
var array = [1, 2, 3, 4];
array.unshift('new');
array[array.indexOf('new')] = 'old';
var newArray = ['Join', 'Me'];
console.log(array.join(', '));
console.log(array);
console.log(newArray);
```

```
old, 1, 2, 3, 4                                    script.js:5
▶(5) ["old", 1, 2, 3, 4]                           script.js:6
▶(2) ["Join", "Me"]                                script.js:7
>
```

REDUCING ARRAY: This reduce the whole array into a single value using a function where you determine how you want this reduction to happen maybe using addition, multiplication etc.

Script.js

```
var array = [1, 2, 3, 4];
console.log(array.reduce(function(total, value){
  return total + value;
}));
console.log(array);
```

```
10                                                 script.js:2
▶(4) [1, 2, 3, 4]                                  script.js:5
>
```

script.js

```
var array = [1, 2, 3, 4];
console.log(array.reduce(function(total, value){
  console.log('Total: ' + total + ', Value: ' + value);
  return total + value;
}));
console.log(array);
```

```
Total: 1, Value: 2                                  script.js:3
Total: 3, Value: 3                                  script.js:3
Total: 6, Value: 4                                  script.js:3
10                                                  script.js:2
▶ (4) [1, 2, 3, 4]                                  script.js:6
>
```

OBJECTS

OBJECT BASICS LITERAL NOTATIONS: -

In this module, we will be focusing on objects. How to create them and their characteristics.

CREATING AN OBJECT – When creating an object, you first declare a variable followed by { }, inside this {} we have the field name or properties, colon : and values e.g. name : 'Max'.

Script.js

```
var person =  {
  name : 'Max',
  age : 27
};
console.log(person);
/* When creating an object, you use colon : and not equal = sign i.e. name : 'Max' and
not name = 'Max'.
name & age are the field names or properties. 'Max" & 27 are the field values */
```

```
▼ {name: "Max", age: 27}                            script.js:5
    age: 27
    name: "Max"
  ▶ __proto__: Object
>
```

We can access just a property inside an object.

Script.js

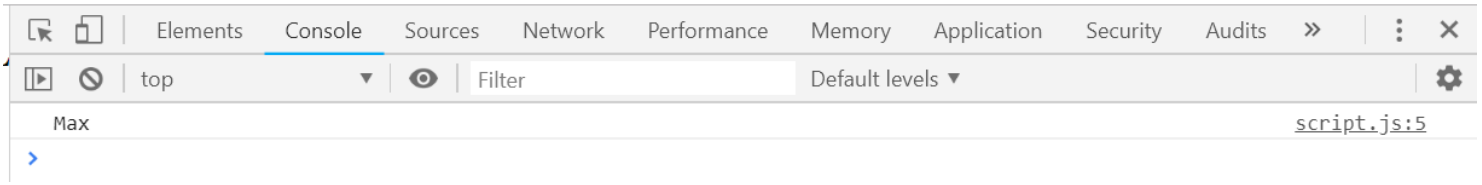
```
var person =  {
  name : 'Max',
  age : 27
};
console.log(person.name); // here we are accessing just the name property.
```

```
Max                                                 script.js:5
>
```

Another way of accessing the object properties is by passing them as a string inside square brackets.

Script.js

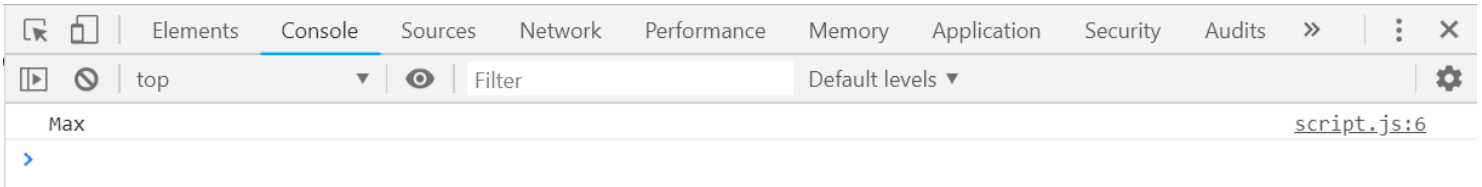
```
var person =  {
  name : 'Max',
  age : 27
};
console.log(person['name']); // here we are accessing just the name property using the
string & square brackter [] method.
```



But this second method should not be used this way by default. There are certain scenario where we should access the property using the square bracket. This we have below.

Script.js

```
var person =  {
  name : 'Max',
  age : 27
};
var field = 'name';
console.log(person[field]); // This is the only scenario where the 2nd method sud be
use. here we've assigned the value of the property 'nanme' to a new property 'field'
and we are not calling the 'field'property
```



WHAT KINDS OF PROPERTIES CAN WE HAVE? Here we already have name and age but we can also have object as another property inside the same object. This will be a nested object which will be a property.

Script.js

```
var person =  {
  name : 'Max',
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'], //an array of values assgined to the
property 'hobbies'
    location : 'Nigeria'
  }
};
var field = 'name';
console.log(person.details);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▼{hobbies: Array(3), location: "Nigeria"}                                  script.js:10
  ▶hobbies: (3) ["singing", "browsing", "fucking"]
   location: "Nigeria"
  ▶__proto__: Object
>
```

We can also access each property inside 'details' object

Script.js

```
var person =  {
  name : 'Max',
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'], //an array of values assgined to the
property 'hobbies'
    location : 'Nigeria'
  }
};
var field = 'name';
console.log(person.details.hobbies); // accessing 'hobbies' inside 'details' object
nested inside 'person' object
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels  1 hidden
▶(3) ["singing", "browsing", "fucking"]                                    script.js:10
>
```

Our object can also contain functions

Script.js

```
var person =  {
  name : 'Max',
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
  greet : function () {
    console.log('Hello!')
  } //function inside object
};
person.greet(); // remember functions must always end with this
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
Hello!                                                                     script.js:9
>
```

if we are creating the function outside and object then it should appear this way:

Script.js

```
var person =  {
  name : 'Max',
```

```
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
  greet : function () {
    console.log('Hello!')
  } //function inside object
};
person.greet();

// The function we have inside the object will look like this if we are creating it
outside an object
/*
var greet = function() {

}
*/
```

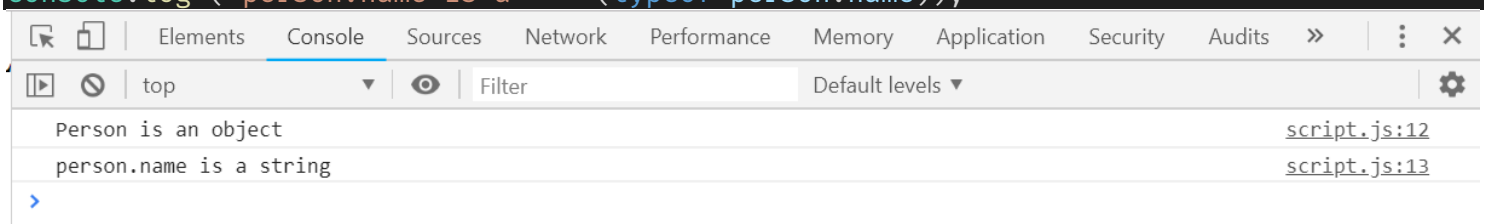
What is the advantage of using an object?

1. **With what we've learnt about object, we should have seen that we can do this without object, we can get data by simply using different variables for name, age, hobbies, location etc. but the benefit object offers us is that we can group all this data inside a single object and use it throughout our app. For example, if we are dealing with details of a person, we don't need declaring different variable for data over 20 when we can group them all in a single object and call them once.**
2. **With variable, we cannot call all the details at once because they are of different variables so we will have to be calling each variable separately but because they are all grouped inside an object, we can call them all at once.**

We can get the type that 'person', 'person.name' is

Script.js

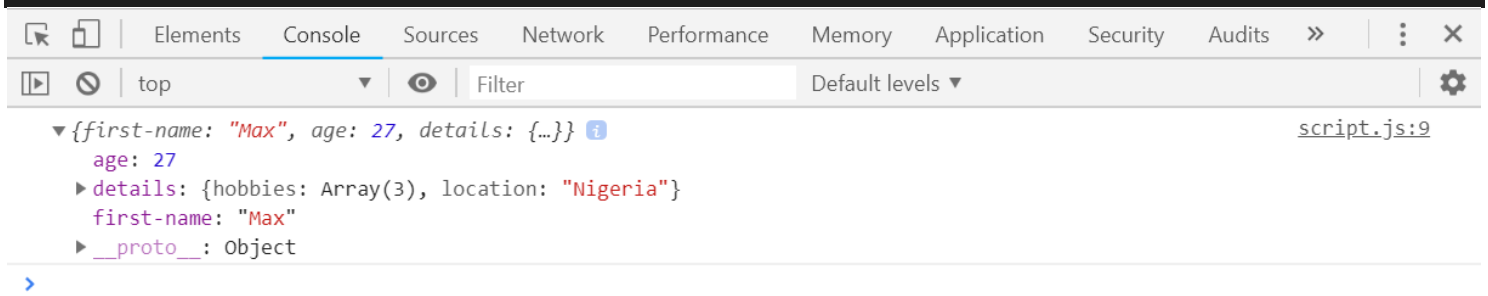
```
var person =  {
  name : 'Max',
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
  greet : function () {
    console.log('Hello!')
  } //function inside object
};
console.log ('Person is an ' + (typeof person));
console.log ('person.name is a ' + (typeof person.name));
```



Your object properties can also be written as strings, this will allow you to use other characters you can't use by default.

Script.js

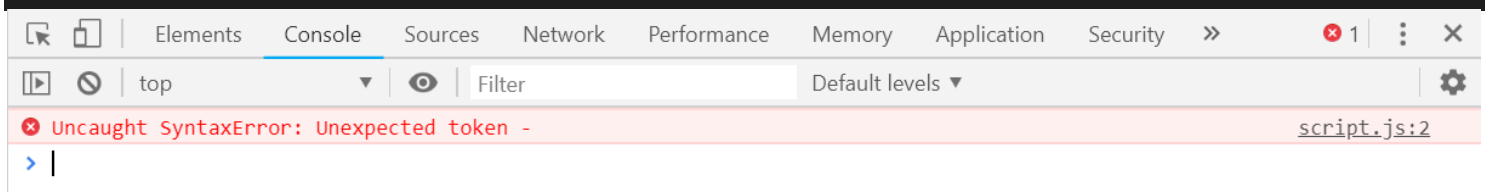
```
var person =  {
  'first-name' : 'Max',
  'age' : 27,
  'details' : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
};
console.log(person);
```



let's try to use the same dash – character without turning the property to string

Script.js

```
var person =  {
  first-name : 'Max', //using dash without turning the property to string
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
};
console.log(person);
```



So you can write your property without a string (as a variable) or use a string.

Since our properties is now a string, how can we access them because we cannot access them using 'person.name' or 'person.'first-name''

Script.js

```
var person =  {
  'first-name' : 'Max', //using dash without turning the property to string
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
};
console.log(person['first-name']);
```

```
Max                                                    script.js:9
Live reload enabled.                                   index.html:38
>
```

PROPERTIES & "this": - You can change your property values after setting up your object.

Script.js

```
var person =  {
  name : 'Max',
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
};
person.name = 'Anna';
console.log(person);
```

```
▼{name: "Anna", age: 27, details: {…}}                 script.js:10
    age: 27
  ▶details: {hobbies: Array(3), location: "Nigeria"}
    name: "Anna"
  ▶__proto__: Object
>
```

This is useful so you can edit your data later on.

The "this" keyword

Script.js

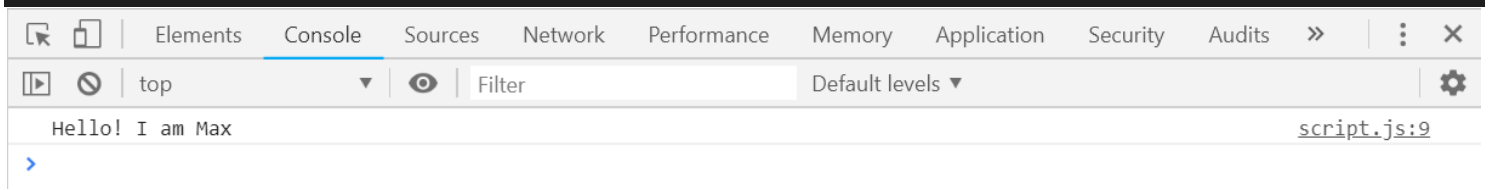
```
var person =  {
  name : 'Max',
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
  greet : function () {
    console.log('Hello! I am ' + name);
  }
};
person.greet();
```

```
Hello! I am                                            script.js:9
>
```

Here we are trying to access the 'name' property which we have inside our object 'person' but as we can see we are not getting it accessed to be logged to the console. Why is this? The properties inside the objects are not variables which we can access directly, they are inside our object 'person' so they are not a variable, the only variable we have in the code is 'person'. This is where 'this' comes into play. We need to include the 'this' keyword in front of the properties to be able to access them.

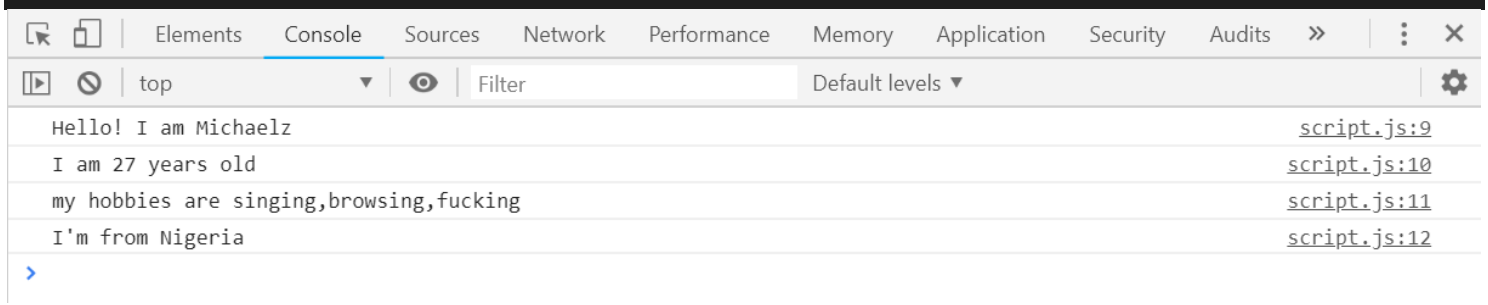
Script.js

```
var person =  {
  name : 'Max',
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
  greet : function () {
    console.log('Hello! I am ' + this.name);
  }
};
person.greet();
```



script.js

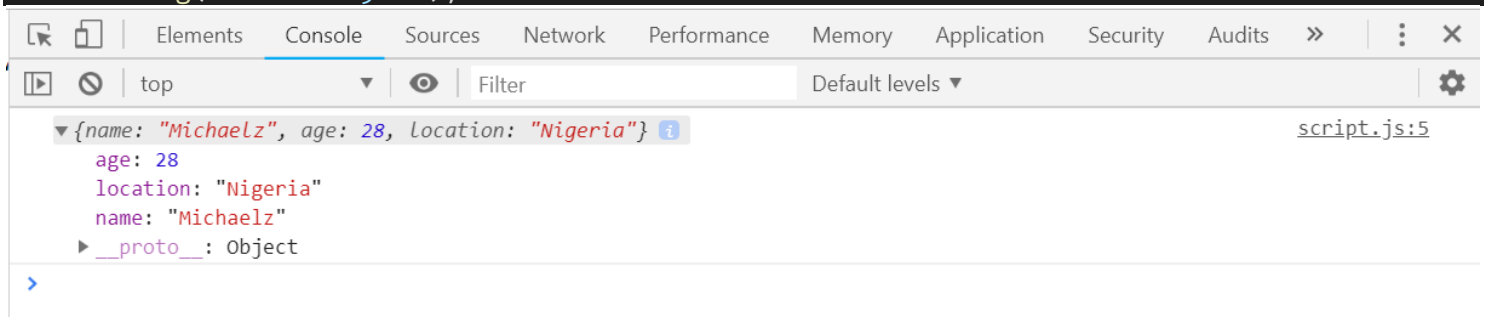
```
var person =  {
  name : 'Michaelz',
  age : 27,
  details : {
    hobbies : ['singing', 'browsing', 'fucking'],
    location : 'Nigeria'
  },
  greet : function () {
    console.log('Hello! I am ' + this.name);
    console.log('I am ' + this.age + ' years old');
    console.log('my hobbies are ' + this.details.hobbies);
    console.log("I'm from " + this.details.location);
  }
};
person.greet();
```



ALTERNATIVE WAY OF CREATING OBJECT USING THE OBJECT CONSTRUCTOR: - We another way of creating our object.

Script.js

```
var anotherObject = new Object();
anotherObject.name = 'Michaelz';
anotherObject.age = 28;
anotherObject.location = 'Nigeria';
console.log(anotherObject);
```

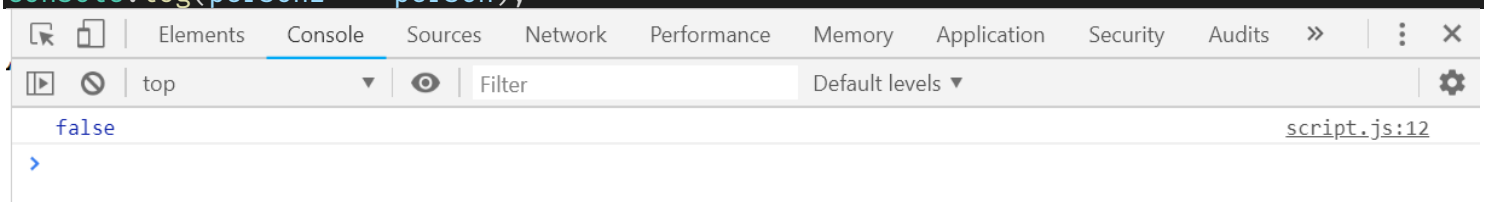


It's advisable to use the other method because this one takes longer.

OBJECTS ARE REFERENCE TYPES (Refresher): - Even if we have two different objects with the same properties and values, comparing them will return false because objects uses pointer and in that case we are dealing with two different pointers pointing towards two different directions.

Script.js

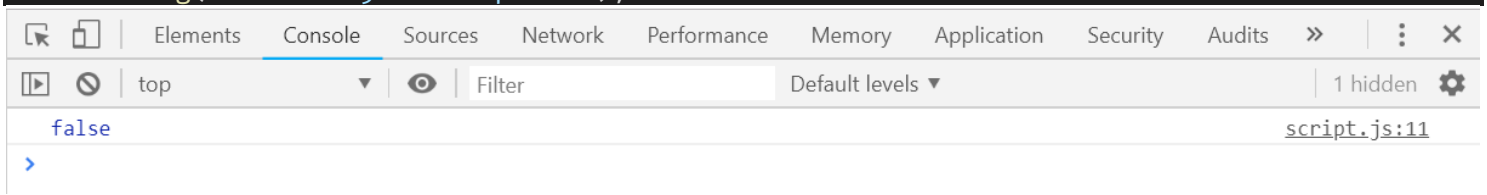
```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};

var person2 = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
console.log(person2 == person);
```



Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};

var anotherObject = new Object();
anotherObject.name = 'Michaelz';
anotherObject.age = 28;
```

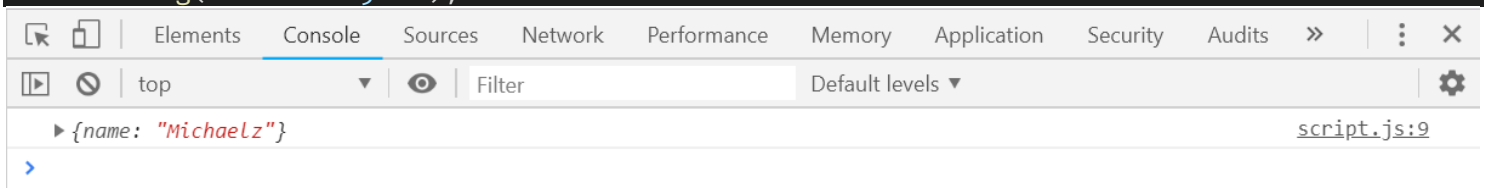
```
anotherObject.location = 'Nigeria';
console.log(anotherObject == person);
```



OBJECT.CREATE: - This is another way of creating an object.

Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
//Using the Object.create method.
var anotherObject = Object.create(null);
anotherObject.name = 'Michaelz';
console.log(anotherObject);
```



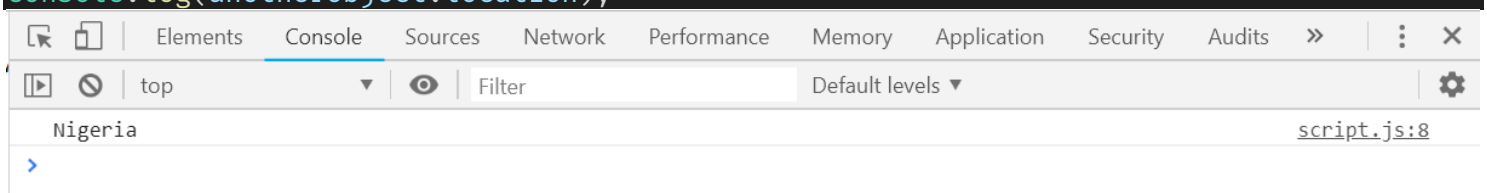
Why would we need the object.create method? Even though this method is also longer like the 2nd method we talked about which we said it's not advisable to use. However, this object.create has its own advantage because it allows us to use PROTOTYPE. What is a prototype? Prototypes are forms of JS form of inheritance so prototype allows you to have a prototype of an object and then create objects of that prototype and if that prototype has a function, all the objects created base on that prototype will have that function too.

Each object you create in JS has a prototype by default and this is "Object.prototype". this is the root prototype.

USING PROTOTYPES IN OBJECTS

Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
var anotherObject = Object.create(person); //passing 'person' as our prototype.
anotherObject.name = 'Michaelz';
console.log(anotherObject.location);
```

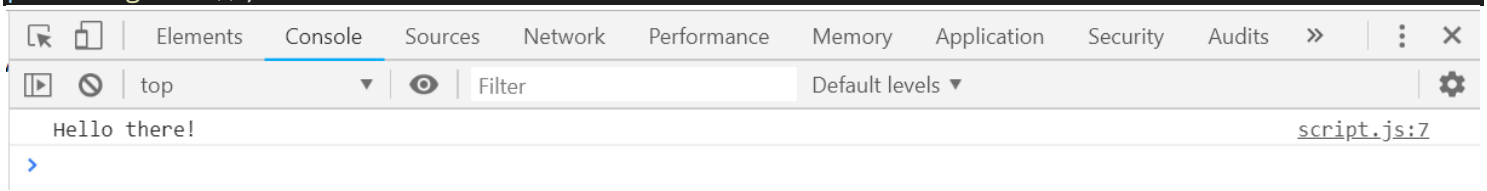


Here we successfully passed our variable 'person' as a prototype and because of that we were able to log it's data to the console using 'anotherObject' even though we don't have 'location' as a property of 'anotherObject'.

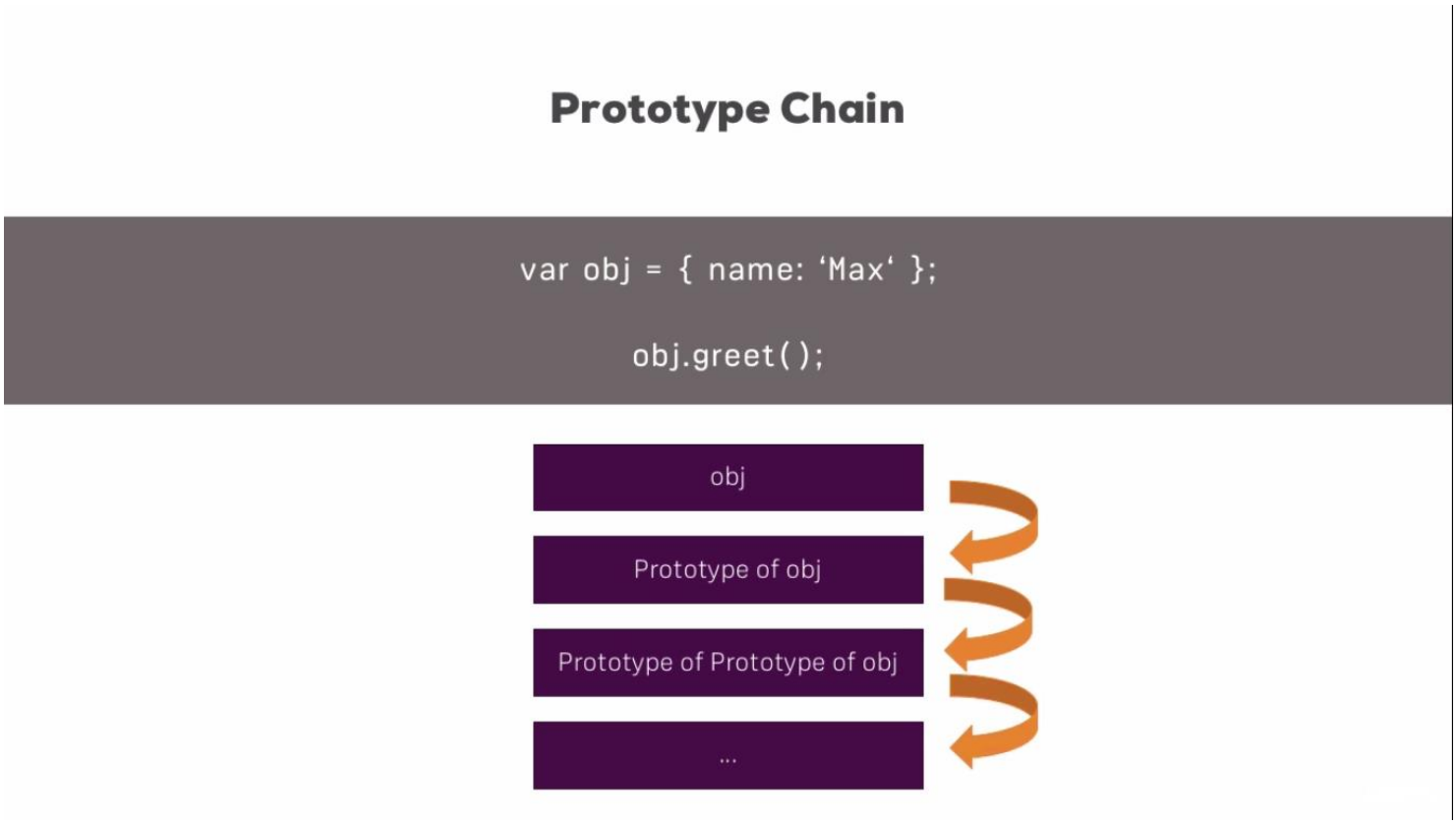
PROTOTYPES INTRODUCTION: - Now that we've learnt about the basics of prototypes, we can now dive deeper. Remember we've mentioned that all objects in JS have a default prototype which is "Object.protoype".

Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there!')
};
person.greet();
```



In this case above, we can see a typical used case of prototype. We assigned a function 'greet' to the prototype which happens to be the default prototype of JS objects and this allows us to now pass the function 'greet' to our object and log it to the console even though we don't have 'greet' as part of our property.



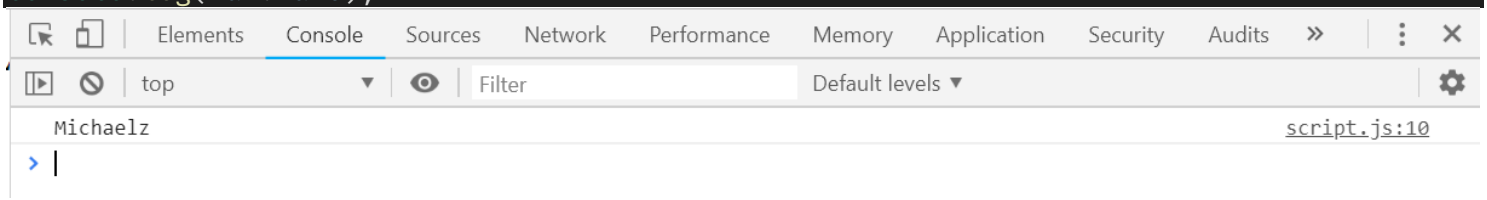
Prototype chain diagram explanation: When we declare the method 'obj.greet', JS checks if it is in our object and because it is not there, it then check for it inside our prototype, if JS can't find it, it checks inside "Prototype of Prototype of obj" etc.

In our case, JS found the greet method inside "Prototype of obj" which is the default object prototype but we can also set custom prototype.

PROTOTYPES IN ACTION: - Let's see some prototype usage in action.

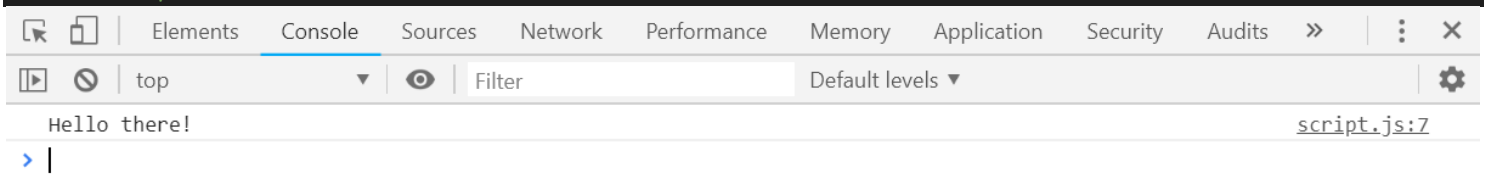
Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there!')
};
var max = Object.create(person);
console.log(max.name);
```



script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there!')
};
var max = Object.create(person);
max.greet();
/* Here JS will check for the method 'greet' inside the variable max (not there), it checks inside var person (not there), it checks inside the greet function and it finds it there*/
```



Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there!')
};
var max = Object.create(person);
var anna = Object.create(person);
anna.name = 'Anna';
anna.greet();
```

```
Hello there!                                    script.js:7
>
```

As you can see we created another object 'Anna' and they all still have access to the prototype.

Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there, I am ' + this.name + '!')
};
var max = Object.create(person);
var anna = Object.create(person);
anna.name = 'Anna';
max.greet();
anna.greet();
```

```
Hello there, I am Michaelz!                     script.js:7
Hello there, I am Anna!                         script.js:7
>
```

Since we called the 'greet' function inside max and anna where they both passed 'person' as their prototypes but anna also has a name on its own. The 'this' keyword now search for the name property not from the function but from max and anna.

Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there, I am ' + this.name + '!')
};
var max = Object.create(person);
var anna = Object.create(person);
max.greet();
anna.greet();
```

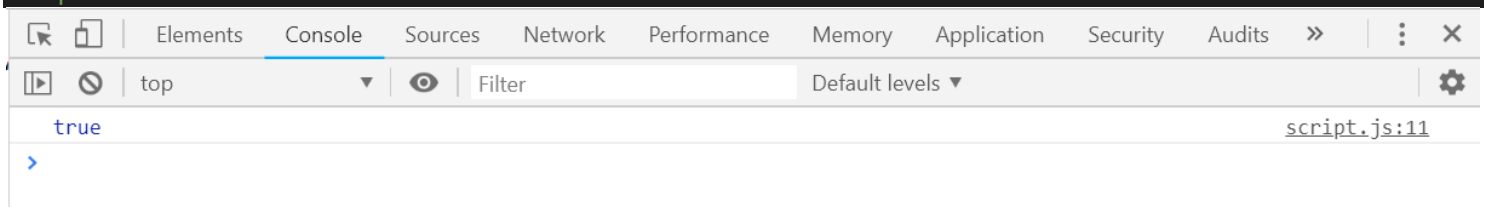
```
Hello there, I am Michaelz!                     script.js:7
Hello there, I am Michaelz!                     script.js:7
>
```

Now Anna doesn't have a name anymore and so JS search for the name property inside anna's prototype which is 'person' and that is why we have 'Michaelz' printed twice.

PROTOTYPES SUMMARY: - Now that we've seen prototypes in action, let's further prove that by comparing some things.

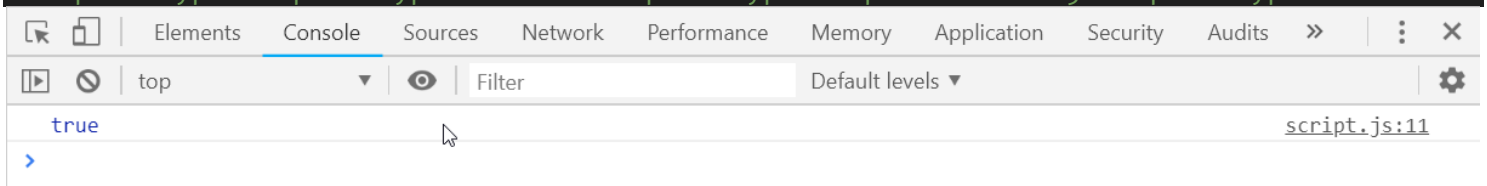
Script.js (Let's compare if truly the prototype of anna is person)

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there, I am ' + this.name + '!')
};
var max = Object.create(person);
var anna = Object.create(person);
console.log(anna.__proto__ == person); //Here we are comparing if the prototype of anna is person
```



script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there, I am ' + this.name + '!')
};
var max = Object.create(person);
var anna = Object.create(person);
console.log(anna.__proto__.__proto__ == Object.prototype); //Here we are comparing if the prototype of prototype of anna or prototype of person is Object.prototype
```



OR

Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there, I am ' + this.name + '!')
};
var max = Object.create(person);
var anna = Object.create(person);
```

```
console.log(person.__proto__ == Object.prototype); //Here we are comparing if the prototype of prototype of anna or prototype of person is Object.prototype
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
true                                                                    script.js:11
>
```

As we've learnt that using '.__proto__' is a bad way of getting prototypes. Lets check out the preferred way of getting or comparing the prototype of anna.

Script.js

```
var person = {
  name : 'Michaelz',
  age : 28,
  location : 'Nigeria',
};
Object.prototype.greet = function () {
  console.log('Hello there, I am ' + this.name + '!')
};
var max = Object.create(person);
var anna = Object.create(person);
console.log(Object.getPrototypeOf(anna) == person); // this is the preferred method of getting the prototype of anna
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
true                                                                    script.js:11
>
```

CONSTRUCTOR FUNCTIONS: - We've learnt a lot about objects, we've learnt 3 different ways of creating them and the important role of prototypes, how to nest object inside object, how to have methods and functions inside object. We also learnt that with the 'this' keyword we can access the property of an object from inside of that object.

Now we want to learnt another way of creating an object and you may be thinking why another way of creating objects when we already have 3 different ways already? This way really makes sense and very powerful.

We are talking about *constructor functions*. Let's see it in action.

Script.js

```
function Person() {

}
var person = new Person();
person.name = 'Max'
console.log(person);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▶ Person {name: "Max"}                                                  script.js:6
>
```

PROTOTYPE CHECK

Script.js

```
function Person() {

}
var person = new Person();
person.name = 'Max'
console.log(person.__proto__ == Person);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
false                                                    script.js:6
>
```

we checked if the prototype of person is Person and we get false

Script.js

```
function Person() {

}
var person = new Person();
person.name = 'Max'
console.log(person.__proto__ == Object.prototype);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
false                                                    script.js:6
>
```

Here we checked again if the prototype is Object.prototype and we got false again. So what then is the prototype?

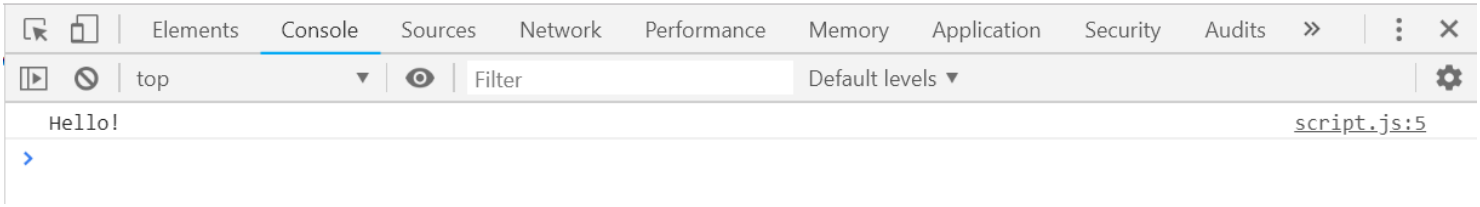
Script.js

```
function Person() {

}
var person = new Person();
person.name = 'Max'
console.log(person.__proto__ == Person.prototype);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
true                                                     script.js:6
>
```

script.js

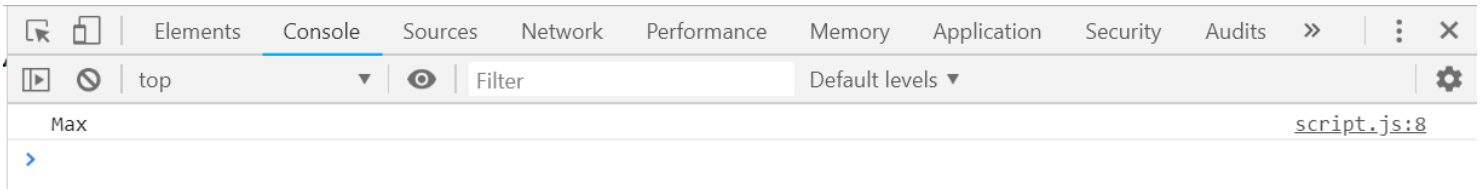
```
function Person() {

}
Person.prototype.greet = function () {
  console.log('Hello!')
};
var person = new Person();
person.name = 'Max'
person.greet();
```



Script.js

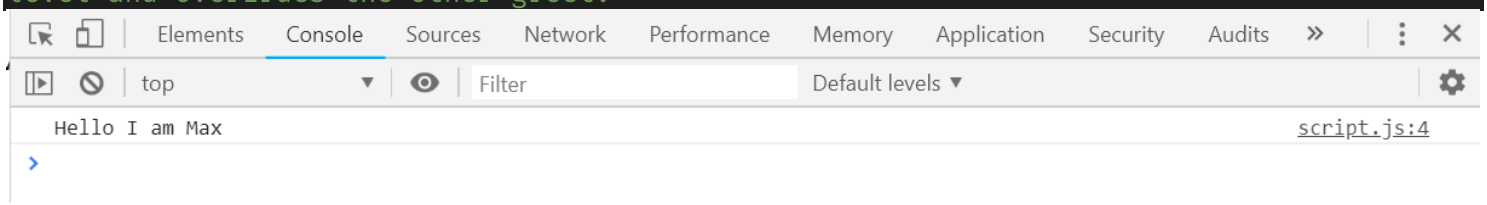
```
function Person() {
  this.name = 'Max'
}
Person.prototype.greet = function () {
  console.log('Hello!')
};
var person = new Person();
console.log(person.name);
```

Here as we can see we are also using 'this' keyword here inside the constructor which is not the way we've been using it before. The keyword for this circumstance is referring to the constructor 'Person'



Script.js

```
function Person() {
  this.name = 'Max',
  this.greet = function () {
    console.log('Hello I am ' + this.name)
  }
}
Person.prototype.greet = function () {
  console.log('Hello!')
};
var person = new Person();
person.greet(); //The greet method here will use the 'this.greet' which we have inside the constructor because the 'this' keyword we attached to it makes it to be on a root level and overirdes the other greet.
```



Script.js

```
function Person() {
  this.name = 'Max',
  this.greet = function () {
    console.log('Hello I am ' + this.name)
  }
}
Person.prototype.greet = function () {
  console.log('Hello!')
```

```
};
var person = new Person();
person.name = 'Anna';
person.greet(); //Here the name that will be used here is 'Anna'
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
Hello I am Anna                                                    script.js:4
>
```

Script.js

```
function Person() {
  this.name = 'Max',
  this.greet = function () {
    console.log('Hello I am ' + this.name)
  }
}
Person.prototype.greet = function () {
  console.log('Hello!')
};
var person = new Person();
person.name = 'Anna';
var anotherPerson = new Person();
person.greet();
anotherPerson.greet();
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
Hello I am Anna                                                    script.js:4
Hello I am Max                                                     script.js:4
>
```

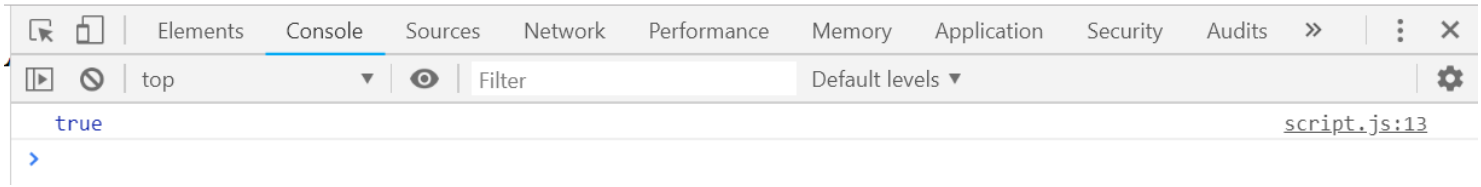
Script.js

```
function Person() {
  this.name = 'Max',
  this.greet = function () {
    console.log('Hello I am ' + this.name)
  }
}
Person.prototype.greetGeneral = function () {
  console.log('Hello!')
};
var person = new Person();
person.name = 'Anna';
var anotherPerson = new Person();
person.greet();
anotherPerson.greetGeneral();
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
Hello I am Anna                                                    script.js:4
Hello!                                                             script.js:8
>
```

Script.js

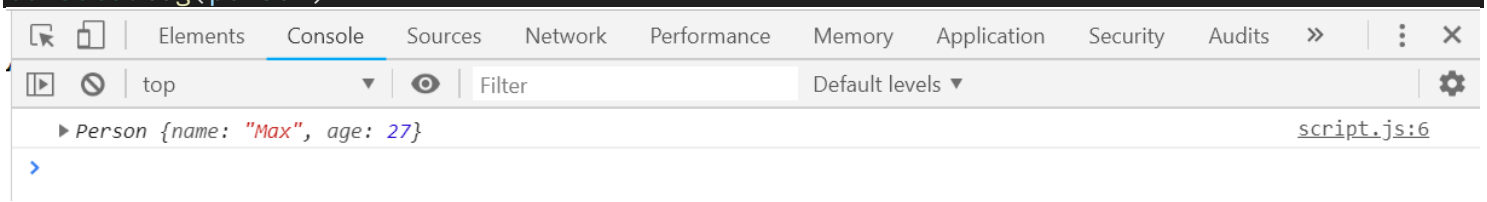
```
function Person() {
  this.name = 'Max',
  this.greet = function () {
    console.log('Hello I am ' + this.name)
  }
}
Person.prototype.greetGeneral = function () {
  console.log('Hello!')
};
var person = new Person();
person.name = 'Anna';
var anotherPerson = new Person();
console.log ( person instanceof Person);
```



CONSCTRUCTOR FUNCTION ARGUMENTS: -

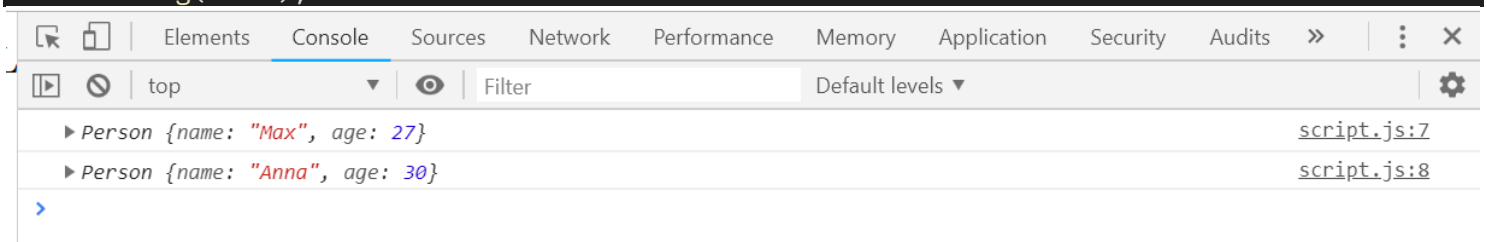
Script.js

```
function Person(name, age) {
  this.name = name;
  this.age = age;
};
var person = new Person ('Max', 27);
console.log(person)
```



Script.js

```
function Person(name, age) {
  this.name = name;
  this.age = age;
};
var max = new Person ('Max', 27);
var anna = new Person ('Anna', 30);
console.log(max);
console.log(anna);
```



OBJECT CREATION & PROTOTYPES SUMMARY: - We have learnt the 4 different ways by which we can create object in JS

Script.js

```
/* 4 WAYS OF CREATING OBJECTS */
// Literal notation
var person = {
  name: 'Max',
  age: 27
}
//
var person = new Object();
person.name = "Max";
person.age = 27;
// using object.create
var person = Object.create(null);
person.name = 'Max';
person.age = 27;
// Using constructor functions
function Person() {
  this.name = 'Max';
  this.age = 27;
}
var person = new Person();
```

THE 'this' KEYWORD AND WHY IT MIGHT BEHAVE STRANGELY: - We learnt that the 'this' keyword has its own heart in JS and it's not always referring to the same thing.

- **On the top level, it was referring to the global/window object in the browser**
- **In an object it was referring to this object.**

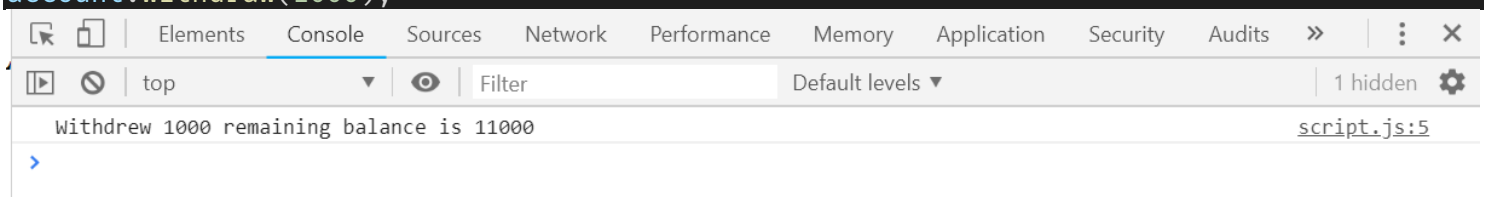
General rule of thumb: 'this' keyword always refers to the left part of the dot which execute something where 'this' is included.

HANDLING THIS WITH bind(), call() & apply(): - Research on this, explanation not well explained.

CREATING PROPERTIES WITH defineProperty: - We want to look at how we can create and delete property of an object which can be done in a more powerful way than we already know.

Script.js

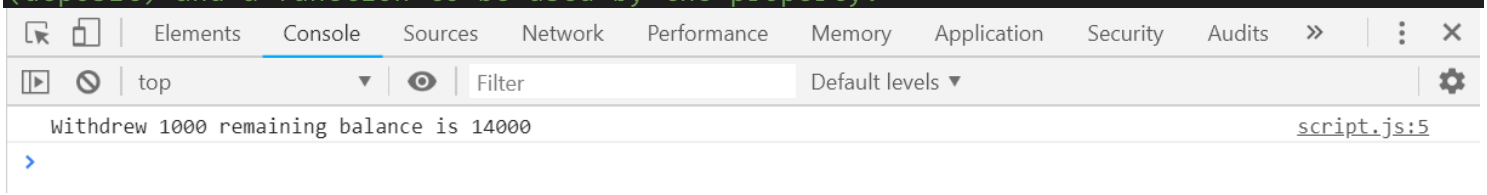
```
var account = {
  cash : 12000,
  withdraw : function (amount) {
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
};
account.withdraw(1000);
```



We can create a new property 'deposit'.

Script.js

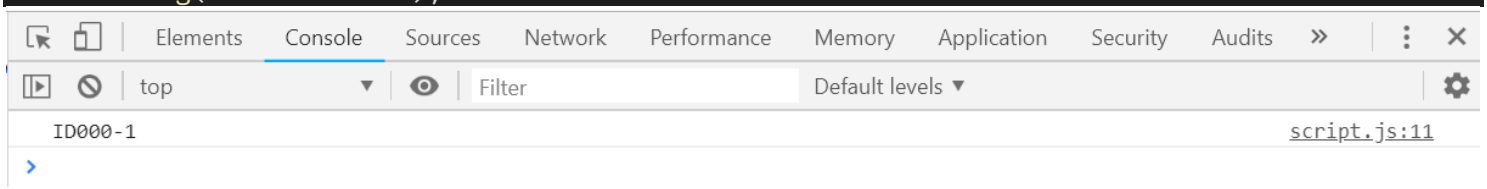
```
var account = {
  cash : 12000,
  withdraw : function (amount) {
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
};
Object.defineProperty(account, 'deposit', {
  value : function (amount){
    this.cash += amount;
  }
})
account.deposit(3000);
account.withdraw(1000);
//In line 8, we passed 3 arguments : the object name (account), the property name
(deposit) and a function to be used by the property.
```



This method looks complicated but why should we used this? It allows us to use some keys like the 'value' we used here. We can also try out some other keys.

Script.js

```
var account = {
  cash : 12000,
  withdraw : function (amount) {
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
};
Object.defineProperty(account, 'name', {
  value : 'ID000-1'
})
console.log(account.name);
```

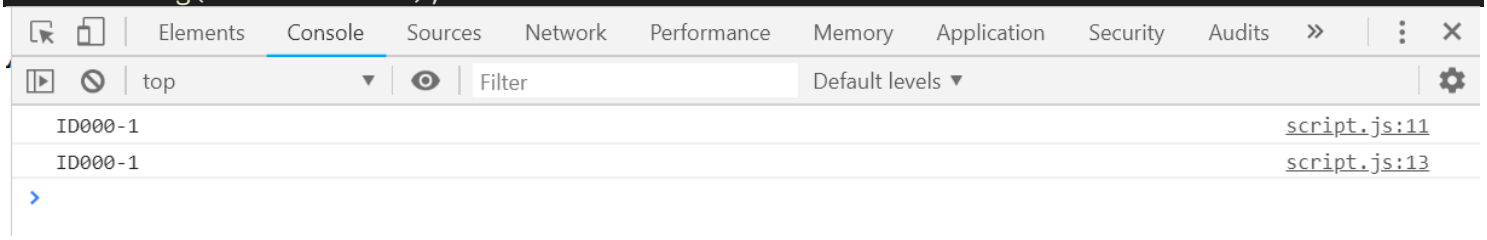


Here we change the property to name, gave it the defineProperty a new value and logged the value to the console. Let's try to change the value and log it again to the console.

Script.js

```
var account = {
  cash : 12000,
  withdraw : function (amount) {
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
```

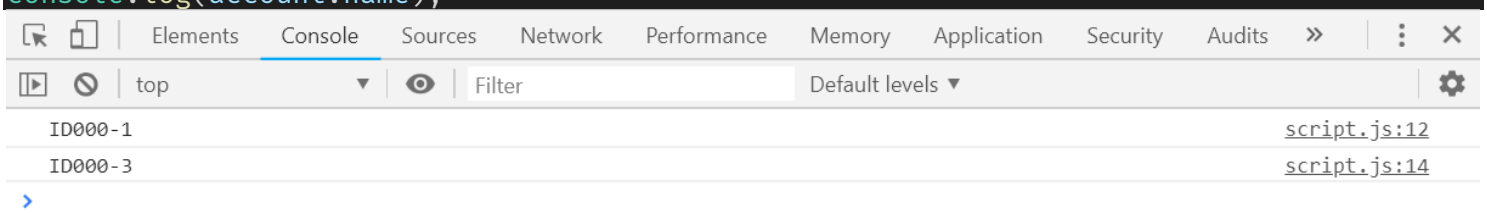
```
};
Object.defineProperty(account, 'name', {
  value : 'ID000-1'
})
console.log(account.name);
account.name = 'ID000-3'; //changing the name value
console.log(account.name);
```



funny enough the name value didn't get updated. This is because by default the defineObject property value is read-only, to get it changed, we need to define a new key.

Script.js

```
var account = {
  cash : 12000,
  withdraw : function (amount) {
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
};
Object.defineProperty(account, 'name', {
  value : 'ID000-1',
  writable : true  //we changing the value from read only to writable
});
console.log(account.name);
account.name = 'ID000-3'; //changing the name value
console.log(account.name);
```



Now our name value gets updated.

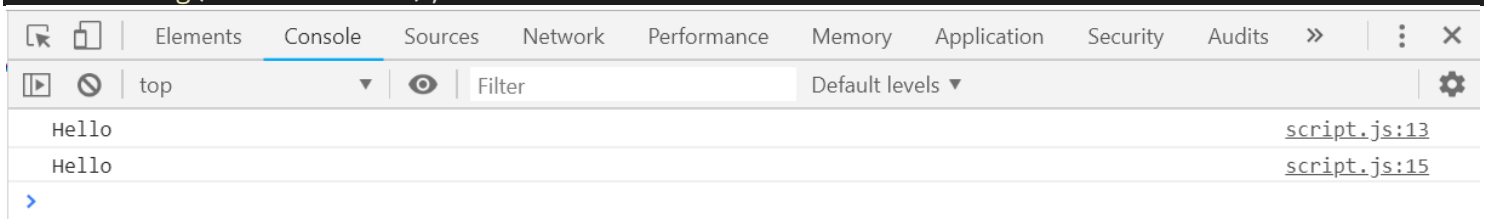
get: function () {

}

Script.js

```
var account = {
  cash : 12000,
  withdraw : function (amount) {
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
};
Object.defineProperty(account, 'name', {
  get: function(){
    return 'Hello';
```

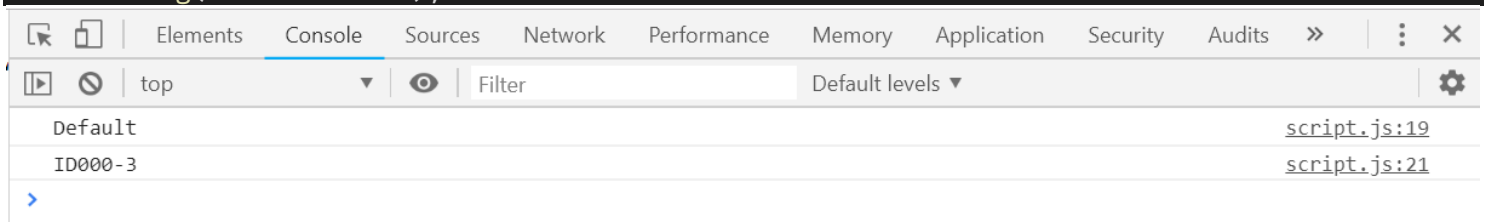
```
  } //Whenever we call the name property, we get the function returning 'Hello'
});
console.log(account.name);
account.name = 'ID000-3';
console.log(account.name);
```



Using the set:

Script.js

```
var account = {
  cash : 12000,
  name2 : 'Default',
  withdraw : function (amount) {
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
};
Object.defineProperty(account, 'name', {
  //using the get:
  get: function(){
    return this.name2;
  },
  //using the set:
  set: function(name){
    this.name2 = name
  }
});
console.log(account.name);
account.name = 'ID000-3';
console.log(account.name);
```

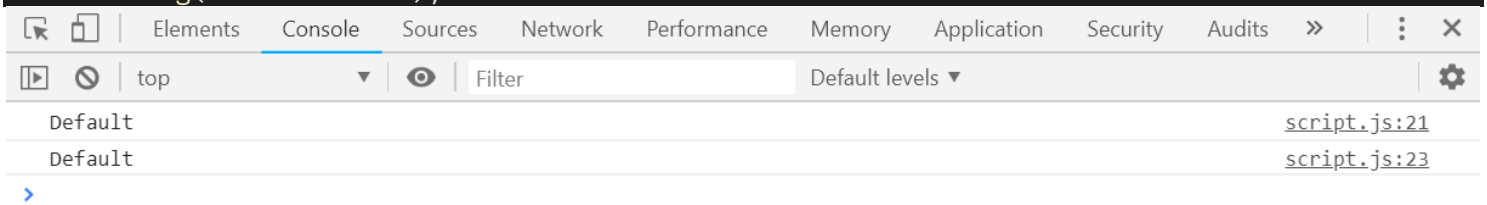


Now we are using the *get function* and *set function* together, the *get* will fetch our value by returning *name2* value then the *set function* passing our property *name* as argument will assign the value of our *name* property to *name2* and then the first console.log will log this *name* to the console which is the *name2* property value we've assigned to, the next line will assign *'ID000-3'* as our *name* and re-print the updated details to the console.

Script.js

```
var account = {
  cash : 12000,
  name2 : 'Default',
  withdraw : function (amount) {
```

```
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
};
Object.defineProperty(account, 'name', {
  //using the get:
  get: function(){
    return this.name2;
  },
  //using the set:
  set: function(name){
   if (name == 'Max') {
     this.name2 = name;
   } else {
     name = this.name2
   }
  }
});
console.log(account.name);
account.name = 'ID000-3'; // the if statement comes here to check for the condition.
console.log(account.name);
```



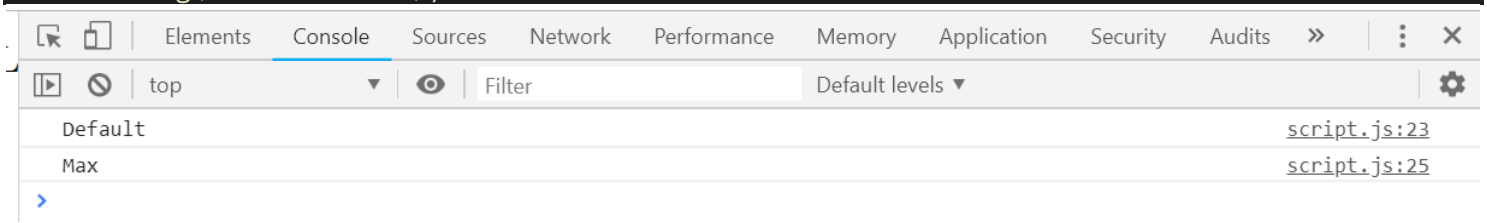
Here according to our code, we are using a conditional statement to print our name property value to the console. The first *get function* assign *name2* property value to our *name* property, the *set function* includes a conditional statement to check if the value of our property *name* is *'Max'* and then assign the value of *name2* to the value of *name* else the value of *name* will remain set to *name2* which is *'Default.'*

Now on the first console.log, we haven't set any value for our *name* property yet which can be checked if it's equals to *'Max'* or not and so we print the value of *name2* to the console, the next line we set a value for our *name* property and JS checked and discovered it is not *'Max'* but *'ID000-3'* and because the if statement is not true *(Max)* the value of name *(ID000-3)* is ignored and we print the value of *name2* to the console instead.

Script.js

```
var account = {
  cash : 12000,
  name2 : 'Default',
  withdraw : function (amount) {
    this.cash -= amount;
    console.log ('Withdrew ' + amount + ' remaining balance is ' + this.cash);
  }
};
Object.defineProperty(account, 'name', {
  //using the get:
  get: function(){
    return this.name2;
  },
  //using the set:
  set: function(name){
```

```
    if (name == 'Max') {
      this.name2 = name;
    } else {
      name = this.name2
    }
  }
});
console.log(account.name);
account.name = 'Max'; // the if statement comes here to check for the condition.
console.log(account.name);
```



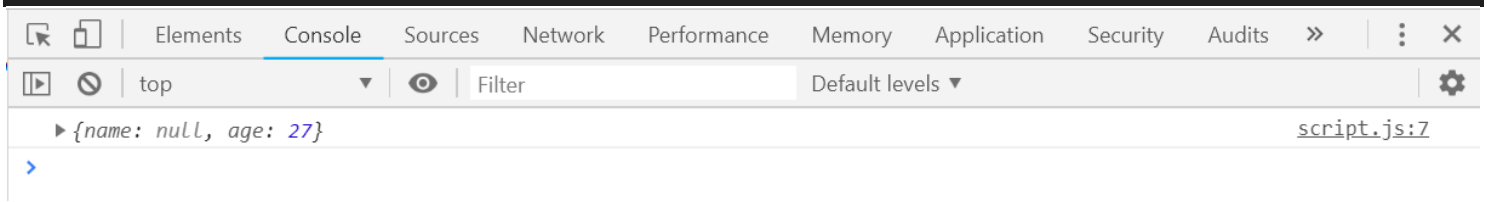
Now that our if statement is true, the second log used the value in the *account.name*.

IMPORTANT BUILT-IN METHODS & PROPERTIES: - Here we want to talk about some generic functions and properties you can always use on your objects.

Let's consider a situation where we create an object *person* and set it to have two properties *name* & *age* now how do we delete a property? We can do this in two ways:

Script.js

```
var person = {
  name : 'Michaelz',
  age : 27,
}
//Let's say we want to delete a property
person.name = null;
console.log(person);
```



But this doesn't really delete it does it? No we still have the field there. Let's check for a better means of getting the deletion done.

Script.js

```
var person = {
  name : 'Michaelz',
  age : 27,
}
//Let's say we want to delete a property
delete person.name;
console.log(person);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▸ {age: 27}                                                    script.js:7
>
```

Now we can see that our *name* property was cleanly removed leaving only the age field.

How do we check if a certain field/property exist in an object?

Script.js

```
var person = {
  name : 'Michaelz',
  age : 27,
}
console.log('name' in person); //this means check if the property/field 'name' exist in the object 'person'
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
true                                                           script.js:5
>
```

Remember though that the check is case sensitive: confirm this below

Script.js

```
var person = {
  name : 'Michaelz',
  age : 27,
}
console.log('Name' in person); //this means check if the property/field 'name' exist in the object 'person'
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
false                                                          script.js:5
>
```

We got false because of case sensitivity.

How do we loop through all the properties that our object *'person'* has?

Script.js

```
var person = {
  name : 'Michaelz',
  age : 27,
  greet : function(){
    console.log('Hello!');
  }
}
//lopping through all the fields/properties of object 'person'
for (var fields in person) {
  console.log(fields);
}
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
name                                    script.js:10
age                                     script.js:10
greet                                   script.js:10
>
```

Now that we know how to loop through the properties, how do we loop through their values?

Script.js

```
var person = {
  name : 'Michaelz',
  age : 27,
  greet : function(){
    console.log('Hello!');
  }
}
//lopping through all the values of the fields/properties of object 'person'
for (var fields in person) {
  console.log(person[fields]);
}
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
Michaelz                                script.js:10
27                                      script.js:10
ƒ (){                                   script.js:10
    console.log('Hello!');
  }
>
```

ERRORS & DEBUGGING

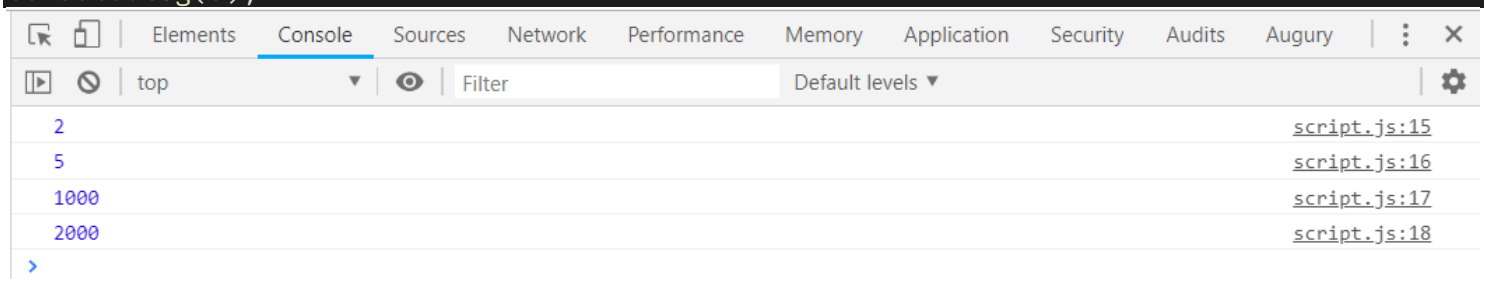
DEBUGGING WITH THE BROWSER DEVELOPER TOOLS: - Now we want to touch a very important topic which is the key to developing successful JS applications. You cannot always write prefect code all the time, you will write code with some nasty bugs and so you will need to debug.

DEBUGGING: - the debugging tutorial will be based on chrome which is the best browser for development and debugging.

Script.js

```
var a = 2;
var b = 5;
var c;
var d = 1000;

for (var i = 0; i < 5; i++) {
    if (i > a && i < b) {
        c = 1000;
    }
    if (i === 3) {
        d *= 2;
    }
}
```

```
console.log(a);
console.log(b);
console.log(c);
console.log(d);
```



/

CODE EXPLANATION:

a is 2, b is 5, c is undefined, d = 1000.

(line 6): i starts from 0 and get increments to less than 5 which is 0,1,2,3,4.

(line 7): we pick the first i value which is 0, if 0 is less than 'a' which is 2 and also less than b which is 5 - this is false so we go to the next value of 2, this is also false bcos 2 != greater than a (2), we pick the next value which is 3 and the condition is now true bcos 3 is greater than a(2) and also less than b(5) so we can proceed to next line

(line 8): c which was undefined is now 1000

(line 10): if i which is now 3 is equals to 3 and the same type (number) as 3 'this is true' then we move to next line.

(line 11): d*=2 = d = 2 * d which is 1000 * 2.

(line 15): we log a - we print 2

(line 16): we log b - we print 5

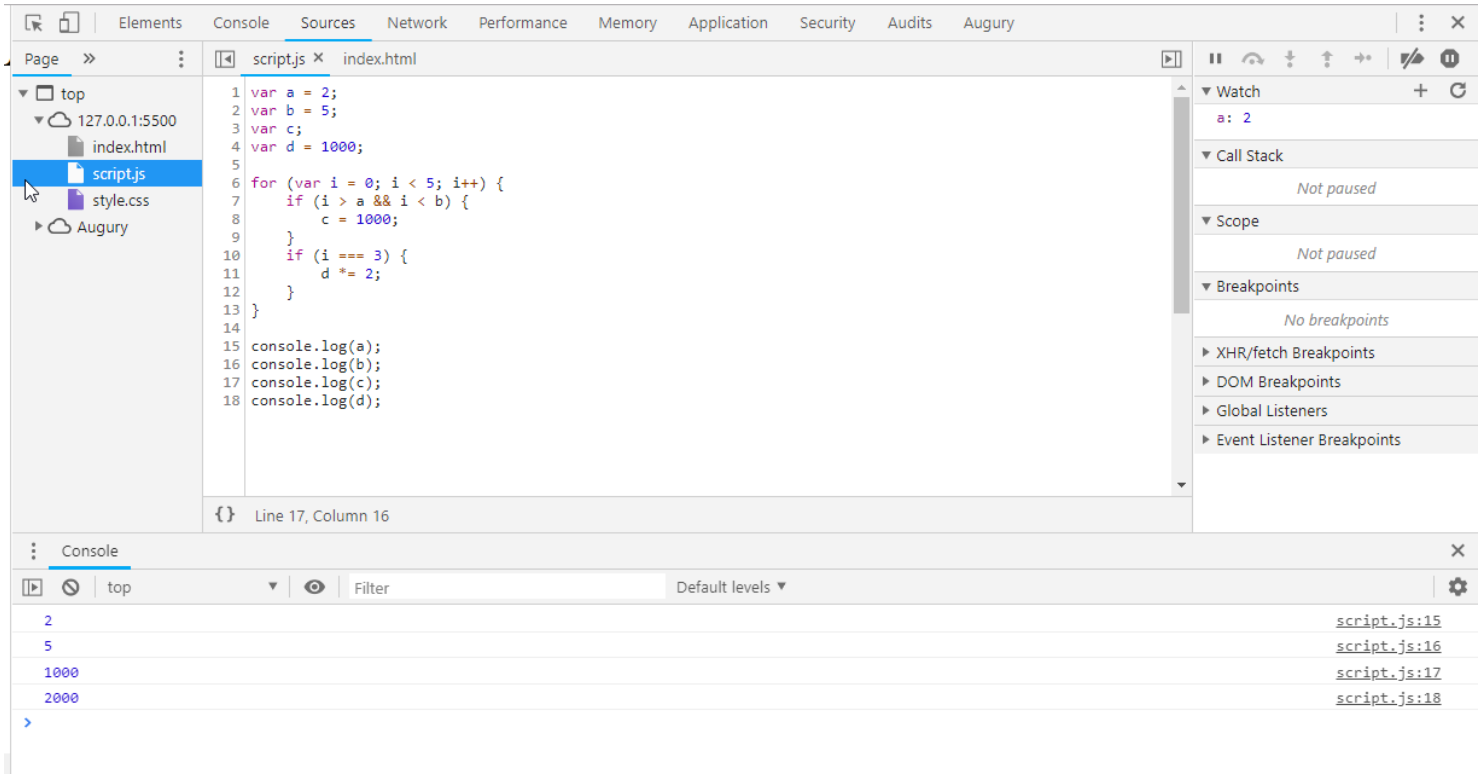
(line 17): we log c - we print 1000

(line 18): we log d - we print 2000

**/*

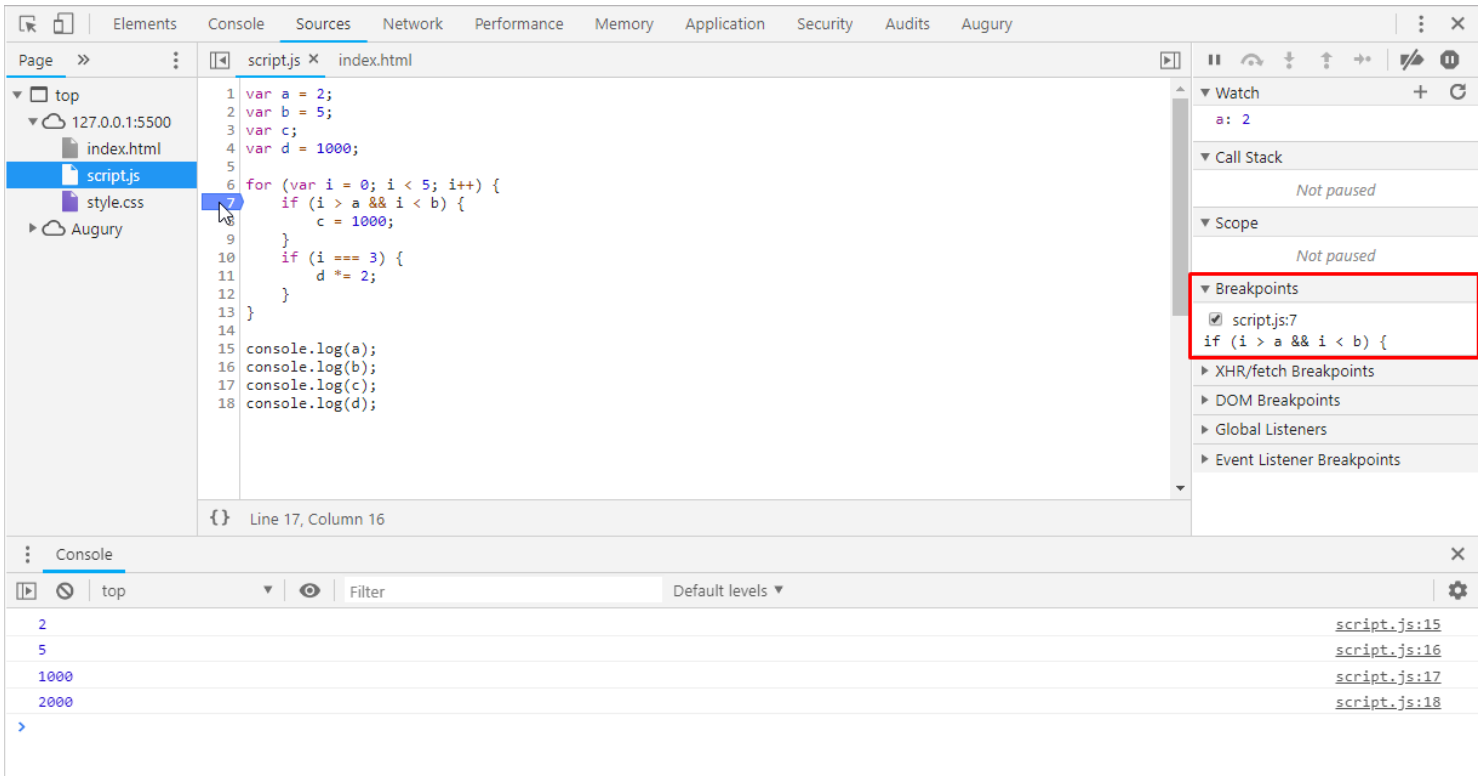
Now let's start by debugging this code: debug allows us not only to correct errors alone but to see as our code runs at runtime.

- **Open your file in chrome with liveserver**
- **Ctrl+shift+I to enter developer mode.**
- **Click on console tab to see our JS results**
- **Click on 'sources' tab to start debugging**
- **Click on the file name of your js file to see your code in the sources window**

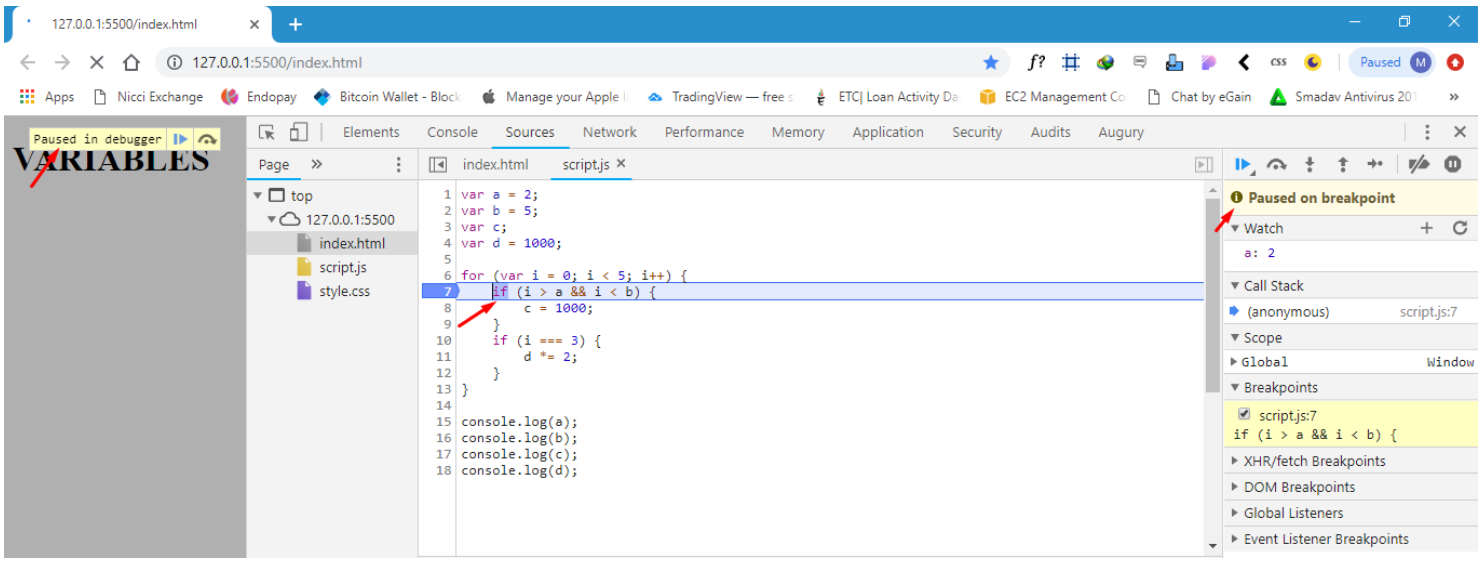


Breakpoints: In the debugger window, you can set breakpoints in your javascript code. At each breakpoint, JS will stop executing, and let you examine JS values like your variable values. In this case JS will let you see the value of I which we used for the if statement. After examine values, you can resume the execution of code (typically with a play button).

To set a breakpoint, click on the line number where you want to set the breakpoint. For our case we will click on line 7 where we have our if statement.



- **Now reload the page (F5) and you should see this below**



Now you can hover over your variables i, a, & b to see their current values.

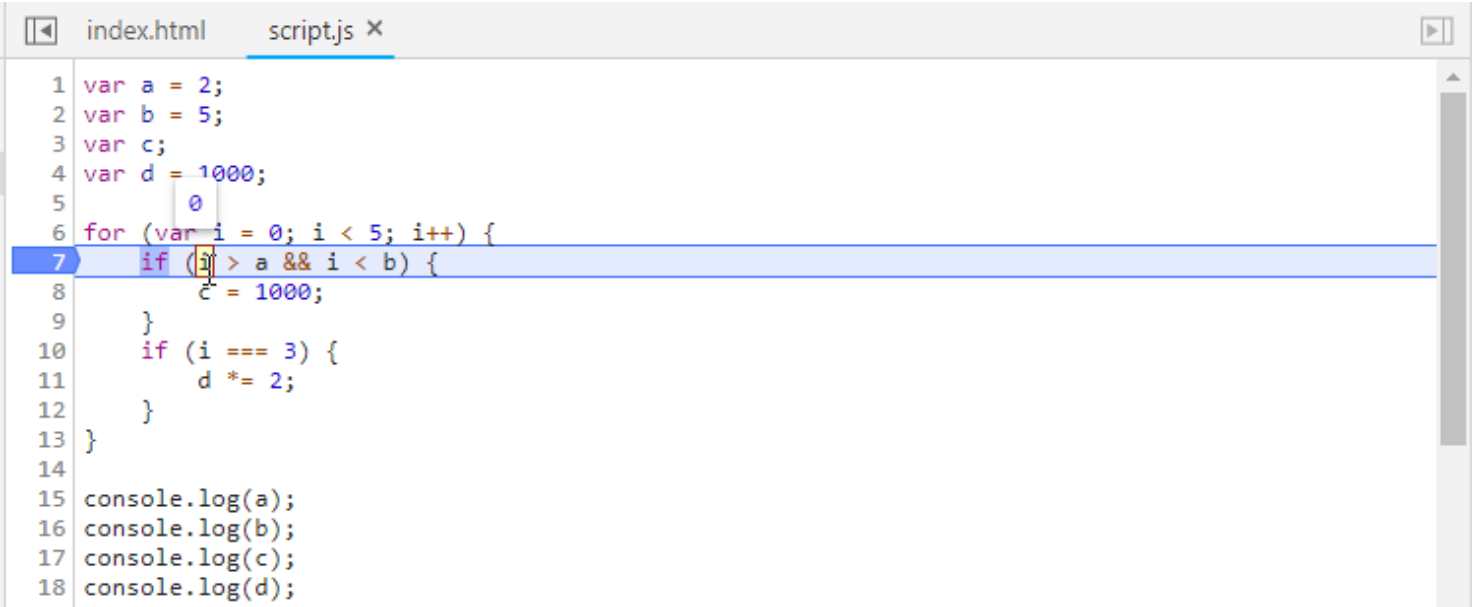


Image 4: 'i' is currently zero because we are the start of our loop

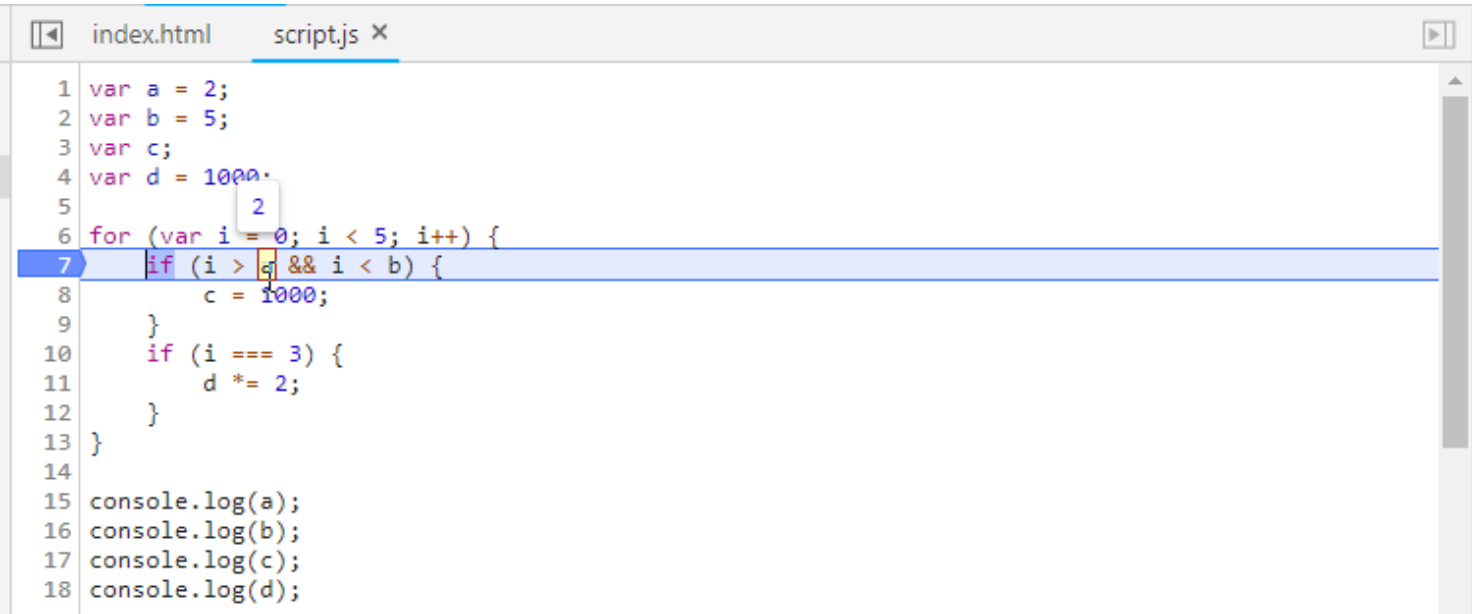


Image 5: 'a' is 2, we have it declared in line 1

index.html script.js

```
1  var a = 2;
2  var b = 5;
3  var c;
4  var d = 1000;
5
6  for (var i = 0; i < 5; i++) {
7      if (i > a && i < b) {
8          c = 1000;
9      }
10     if (i === 3) {
11         d *= 2;
12     }
13 }
14
15 console.log(a);
16 console.log(b);
17 console.log(c);
18 console.log(d);
```

Image 6: 'b' is 5. we have it declared in line 2

index.html script.js

```
1  var a = 2;
2  var b = 5;
3  var c;
4  var d = 1000;
5
6  for (var i = 0; i < 5; i++) {
7      if (i > a && i < b) {
8          c = 1000;
9      }
10     if (i === 3) {
11         d *= 2;
12     }
13 }
14
15 console.log(a);
16 console.log(b);
17 console.log(c);
18 console.log(d);
```

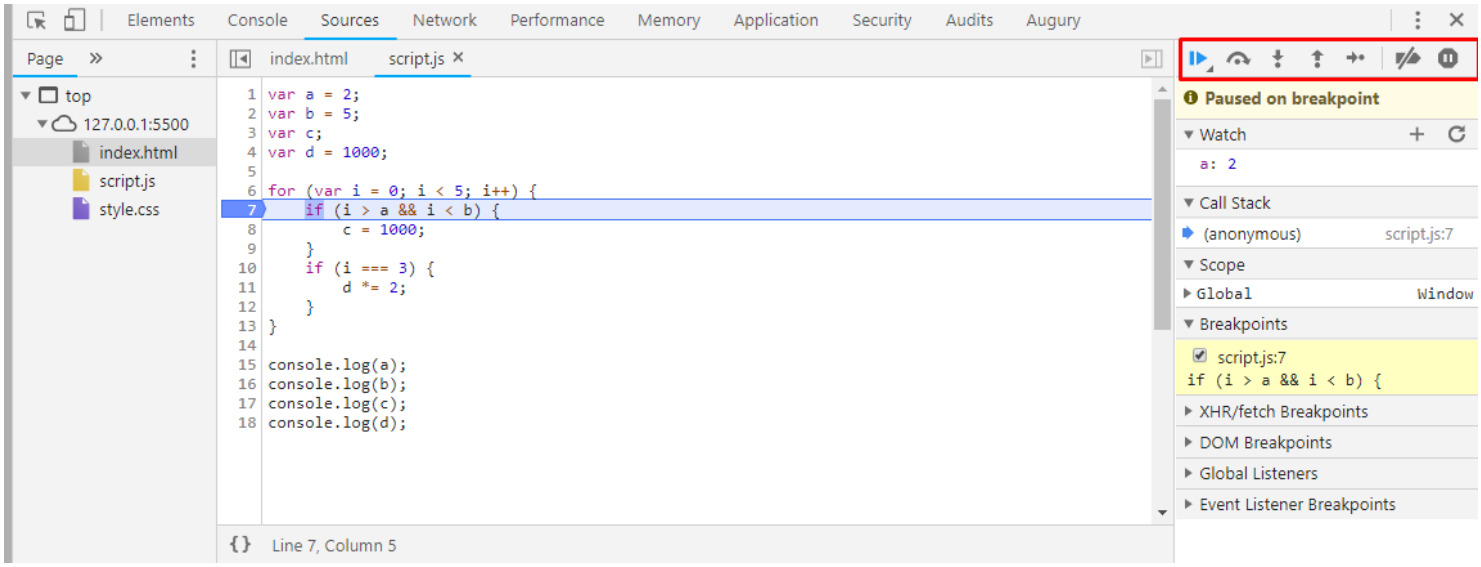
Image 7: 'c' is undefined bcos the if conditional staement is currently false (if i > a) where i = 0, a = 2

index.html script.js

```
1  var a = 2;
2  var b = 5;
3  var c;
4  var d = 1000;
5
6  for (var i = 0; i < 5; i++) {
7      if (i > a && i < b) {
8          c = 1000;
9      }
10     if (i === 3) {
11         d *= 2;
12     }
13 }
14
15 console.log(a);
16 console.log(b);
17 console.log(c);
18 console.log(d);
```

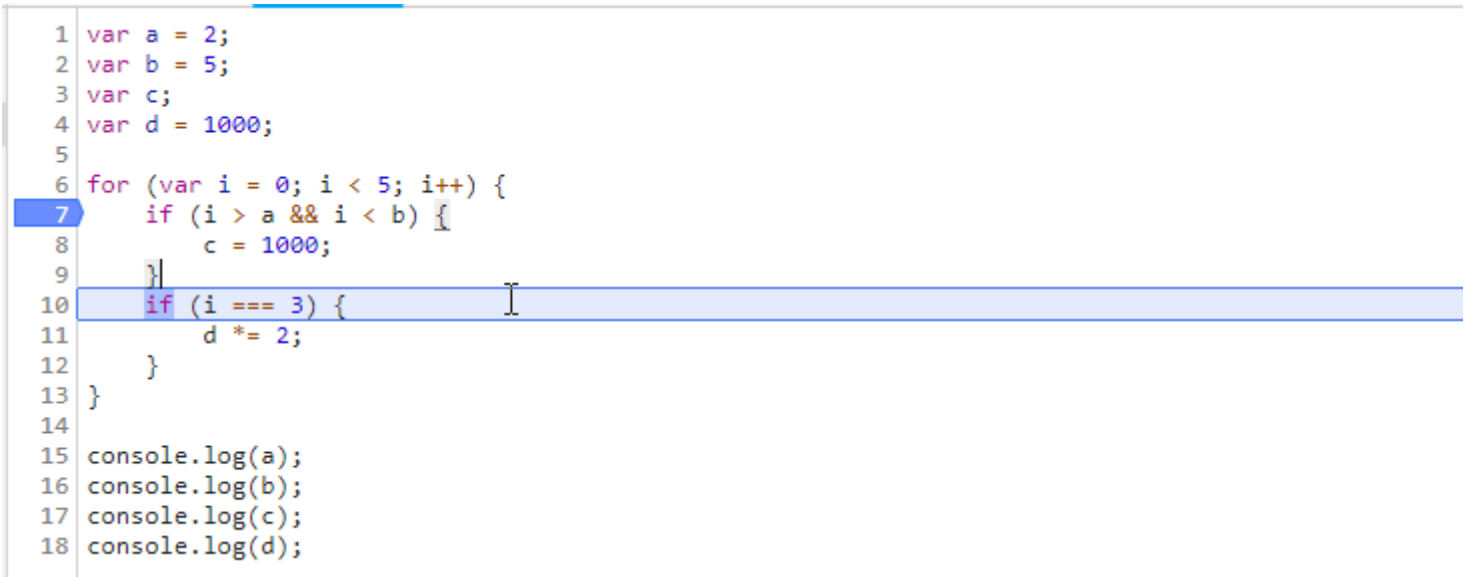
Image 8: d = 1000, according to line 4

Now we have 7 special icons in our sources window



- **The first icon allows us to resume script execution till the end or the next breakpoint**
- **The second icon allows us to step through the loop step by step observing how the variables get changed**

Now we can click the second icon to jump into the next point in the code.



We jumped to the next if statement instead of jumping inside the first if statement to execute the code c = 1000;, this is because the if condition is currently false (i which is 0 is not greater than a which is 2). and in the next if statement we jumped to, the condition is also false because i is not equals to 3; 0 !=== 3. Now we can start another round for the loop setting i to 1. Remember we are dealing with 0 1 2 3 4.

```
1  var a = 2;
2  var b = 5;
3  var c;
4  var d = 1000;
5
6  for (var i = 0; i < 5; i++) {
7      if (i > a && i < b) {
8          c = 1000;
9      }
10     if (i === 3) {
11         d *= 2;
12     }
13 }
14
15 console.log(a);
16 console.log(b);
17 console.log(c);
18 console.log(d);
```

Paused on breakpoint

Watch

a: 2

Call Stack

(anonymous) script.js:7

Scope

Global Window

Breakpoints

script.js:7
if (i > a && i < b) {

XHR/fetch Breakpoints

DOM Breakpoints

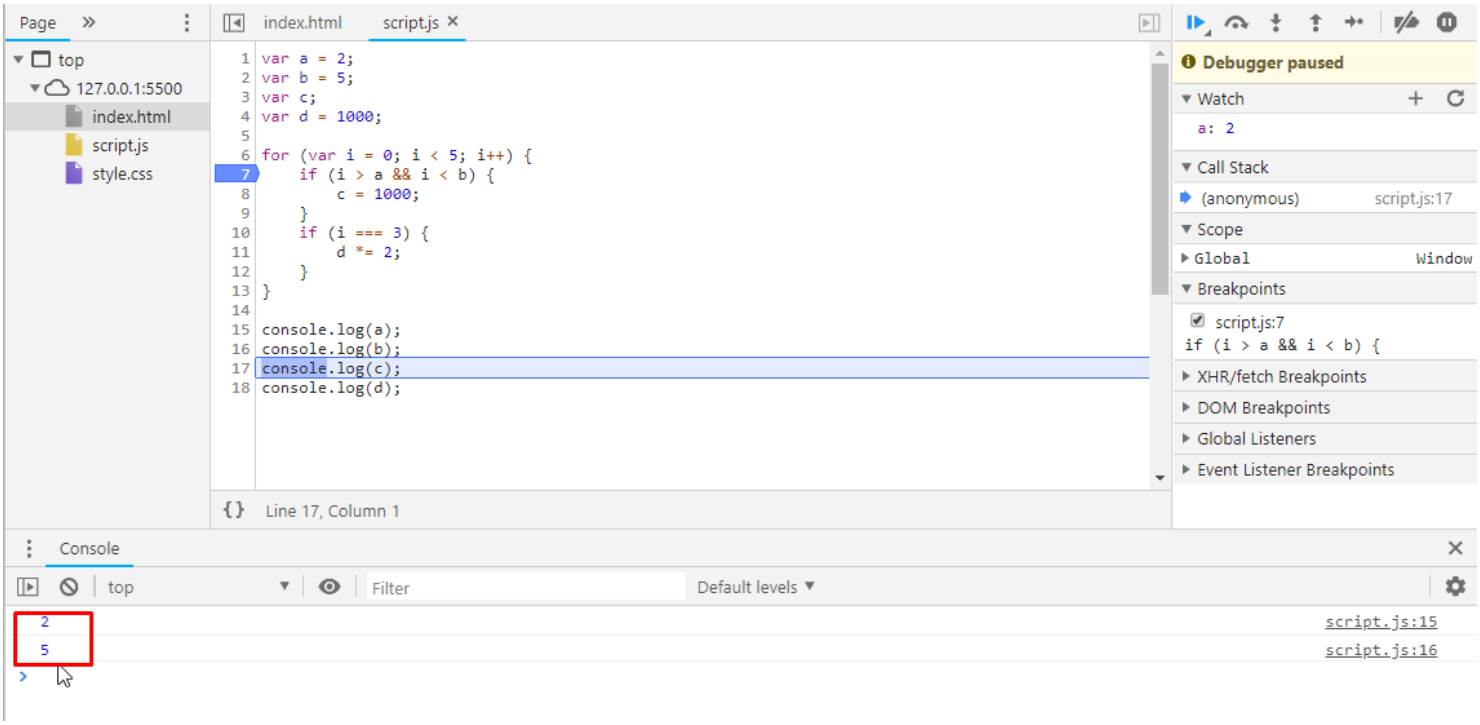
Click the 2nd icon multiple times to start the second round where i = 1. Which we can see above. The same thing will still happen in this round because 1 is not greater than 2 and so we won't enter line 8 till this condition is met on the 4th round where i will be 3…..so we keep clicking the 2nd icon and jumping from the two if statements till this condition is fulfilled.

```
1  var a = 2;
2  var b = 5;
3  var c;
4  var d = 1000;
5
6  for (var i = 0; i < 5; i++) {
7      if (i > a && i < b) {
8          c = 1000;
9      }
10     if (i === 3) {
11         d *= 2;
12     }
13 }
14
15 console.log(a);
16 console.log(b);
17 console.log(c);
18 console.log(d);
```

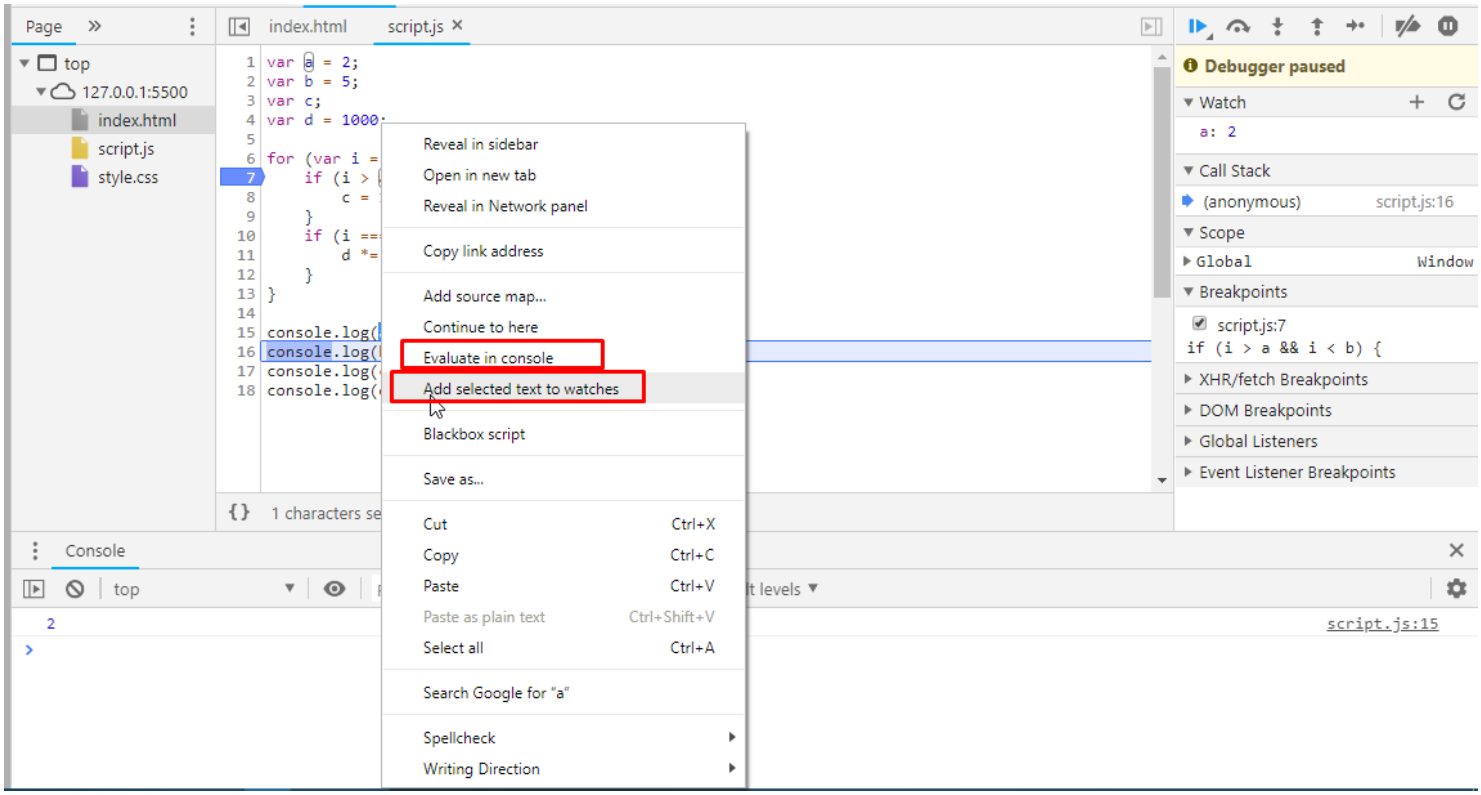
Now we have entered inside line 8 because the condition required to enter line 8 which we can find in line 7 is now true i.e. 3 is greater than 2 and 3 is less than b which is 5 because we are now in line 8, c should now be 1000.

```
1  var a = 2;
2  var b = 5;
3  var c;
4  var d = 1000;
5
6  for (var i = 0; i < 5; i++) {
7      if (i > a && i < b) {
8          c = 1000;
9      }
10     if (i === 3) {
11         d *= 2;
12     }
13 }
14
15 console.log(a);
16 console.log(b);
17 console.log(c);
18 console.log(d);
```

And after looping through all the values 0 1 2 3 4, we should enter line 15 to 18 where our values are printed and we should see values getting printed in the console



You can also watch your variables as they get updated during run time or evaluate them in the console and later change their values



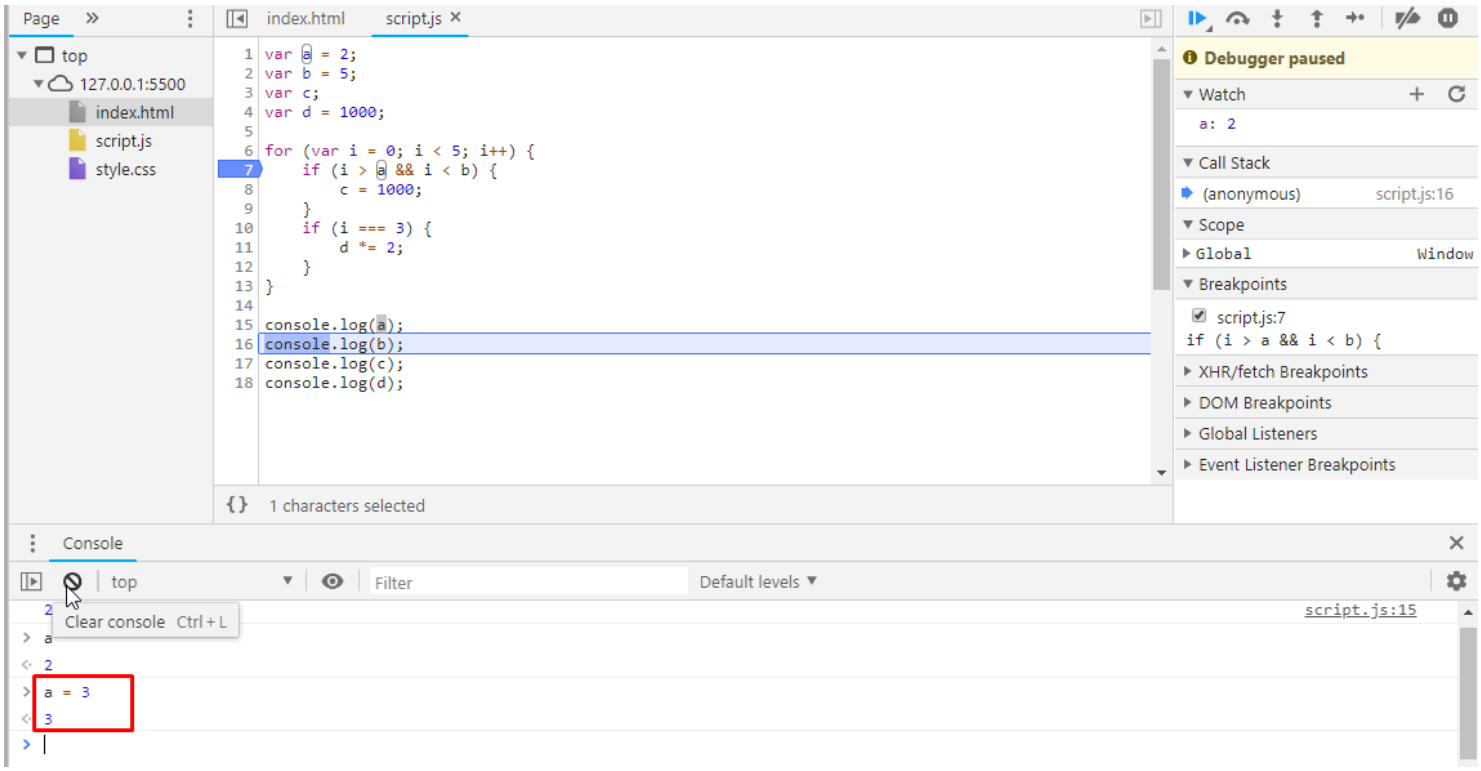


Image 9: the value of a we have changed to 3

We can add any of the variable to our watchlist.

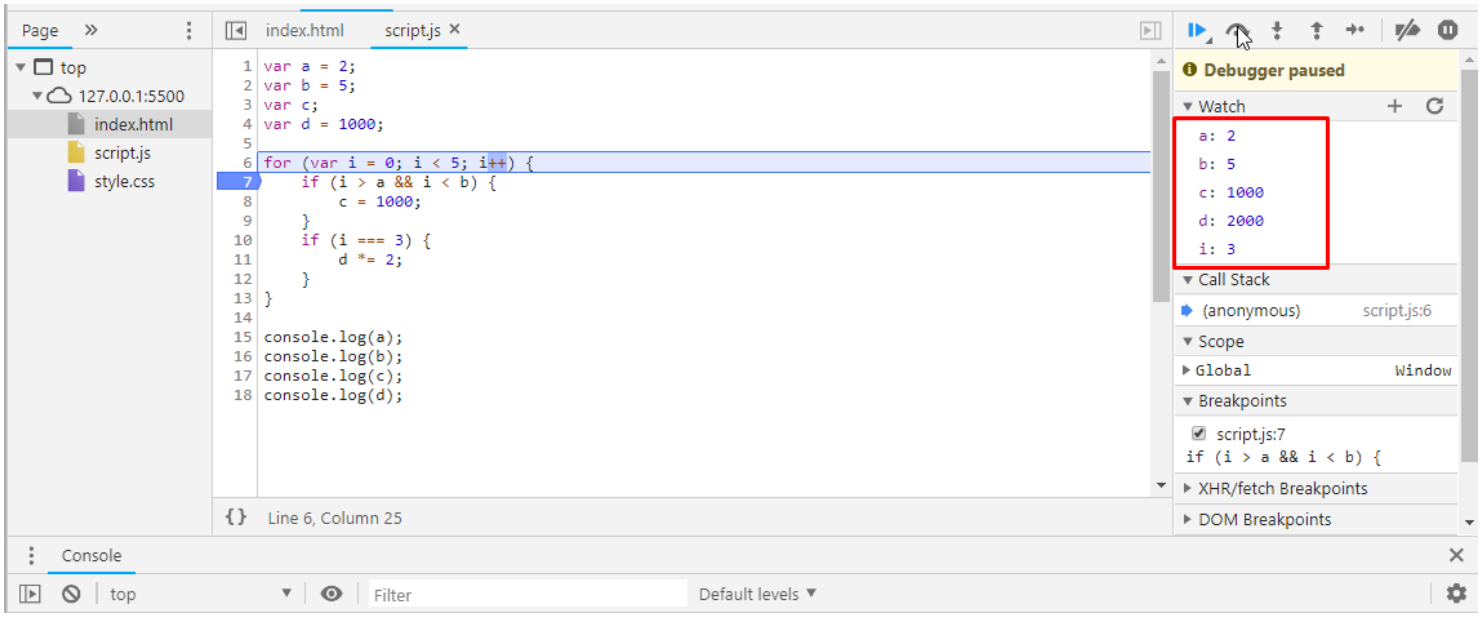


Image 10: all our variables are added to watchlist and we can observe how their values get updated at run time.

LOGGING DATA TO THE CONSOLE: - Another powerful tool in our JS arsenal is logging our code result to the console.

Script.js

```
var a = 2;
var b = 5;
var c;
var d = 1000;

for (var i = 0; i < 5; i++) {
    if (i > a && i < b) {
        c = 1000;
    }
    if (i === 3) {
        d *= 2;
    }
}
// logging to the console
console.log(a);
console.log(b);
console.log(c);
console.log(d);
// logging to the console
```

HANDLING ERRORS WITH TRY AND CATCH: - When writing codes, there might be scenario where you won't know if it will return an error at runtime. For example, if you are reaching out to a server and at runtime the server might be offline. JS offers the try & catch to deal with this.

Script.js

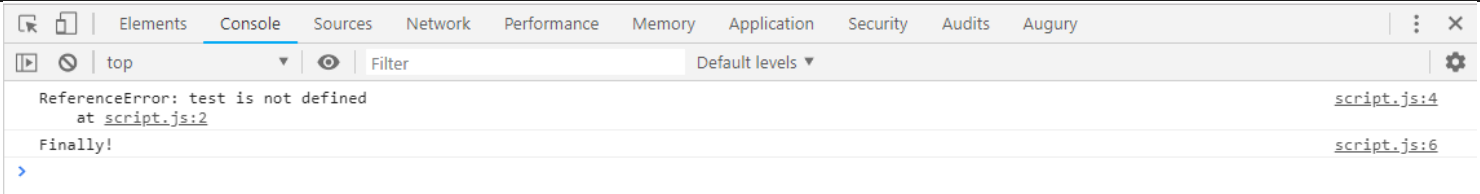
```
try {
// try to execute the code we have here
} catch (error) {
  console.log(error); // catch the error that may arise and log it to the console
} finally {
```

```
  console.log('Finally!'); // finally run this code at the end of it all
```

Let's try to compute a code with an error and try & catch it

Script.js

```
try {
 test(); //lets try to execute a function that doesnt exist and catch the error
} catch (error) {
  console.log(error); // catch the error that may arise and log it to the console
} finally {
  console.log('Finally!'); // finally run this code in the console at the end of it all
}
```



Our error has been caught in the console. This is important when we have code which we are not sure if it will run fine or not.

FUNCTIONS

INTRODUCTION: - We have covered the basic usage of functions. In this module let us cover the advanced topic on functions.

CLOSURES: - Closures are functions which are defined in another function which is aware of its environment and stores it. Definitely this might not be clear so let's examine a real world usage.

Script.js

```
function generator(input){ //the first function
  var number = input;
//another function inside the first function. NB: this function doesnt have a name.
  return function(){
    // the function here is our closure and its aware of its environment : its aware of
'input' & 'number'
  };
}
```

Let's execute some codes

Script.js

```
function generator(input){ //the first function
  var number = input;
//another function inside the first function. NB: this function doesnt have a name.
  return function(){
    return number * 2;
  };
}

var calc = generator(900);
console.log(calc);
```

```
f (){
    return number * 2;
  }
```

script.js:10

We might be expecting to get 1800 or an error as result but surprisingly we get something else. The result we get is the closure (the function we have inside another function). Since we are returning this function, that is why we are getting it as a result.

Now let's examine how we can execute the operation we have inside the closure i.e. return number * 2.

Script.js

```
function generator(input){ //the first function
  var number = input;
//another function inside the first function. NB: this function doesnt have a name.
  return function(){
    return number * 2;
  };
}

var calc = generator(900);
console.log(calc()); //we are using the calc() to execute the operation we have in the closure
```

```
1800
```

script.js:10

remember we said closure stores its environment. Let's define another variable and give it another value

Script.js

```
function generator(input){ //the first function
  var number = input;
//another function inside the first function. NB: this function doesnt have a name.
  return function(){
    return number * 2;
  };
}

var calc = generator(900);
var anotherCalc = generator(1000); //now we defining a new variable and giving it a new value to overide the exisiting one
console.log(calc()); //we are using the calc() to execute the operation we have in the closure
```

```
1800
```

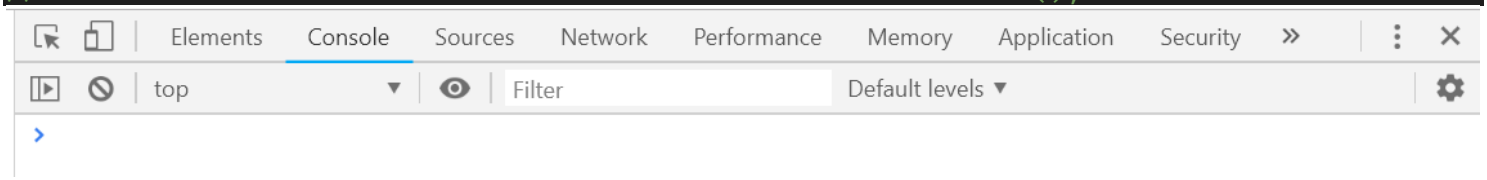
script.js:11

We can see that even though a new value has been assigned, JS still used '900', this is because closure stores its environment.

IMMEDIATELY INVOKED FUNCTION EXECUTIONS IIFEs: - Besides closures, we can write functions that will immediately get executed when JS run through our JS file. This functions are called *Immediately invoked function* **executions IIFEs. Let's see how this works and why we could need it.**

Script.js

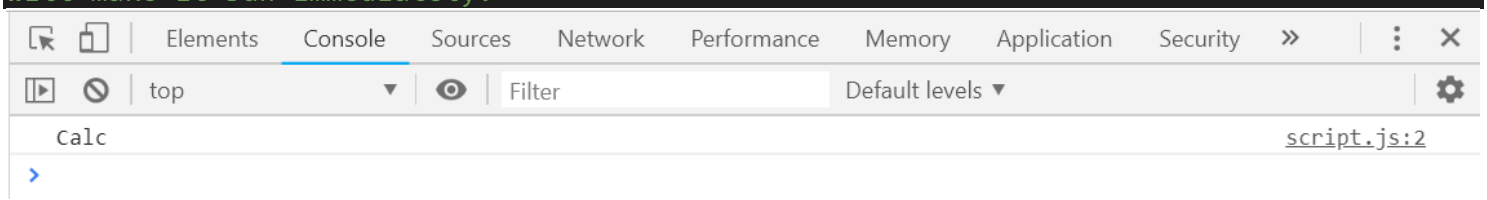
```
function calc(){
  console.log('Calc');
}
//this won't run in the console bcos we will need to add calc();
```



we get nothing in the console because this function needs calc(); to be executed but we can turn it to an IIFE so it get executed immediately.

Script.js

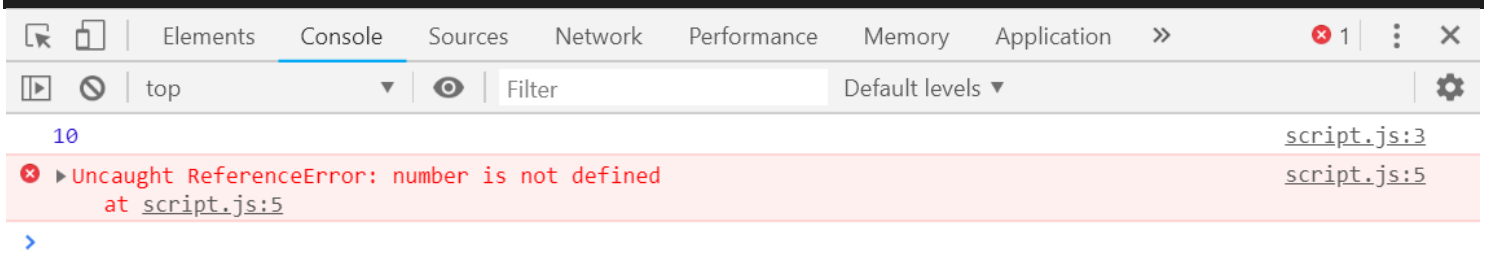
```
(function calc(){
  console.log('Calc');
})();
//enclosing the function inside a bracket and adding another bracket (); outside it will make it run immediately.
```



Now the question to be asked is – Why do we need this?

Script.js

```
(function calc(){
  var number = 10;
  console.log(number);
})();
console.log(number); //trying to see if we can still access 'number'
```

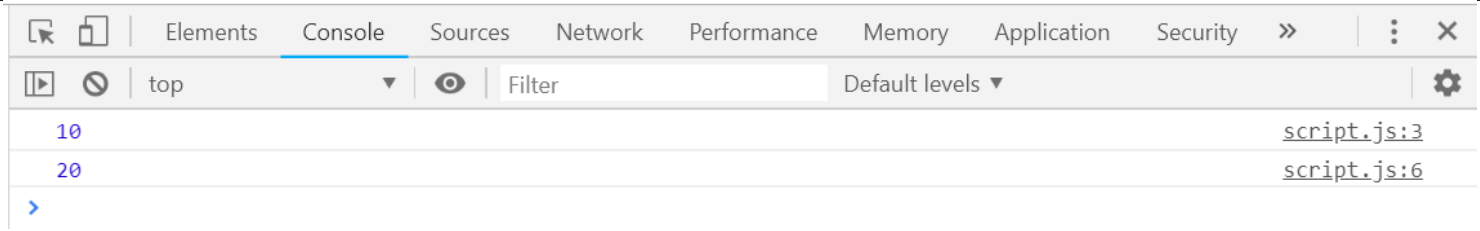


So back to our question, IIFEs are useful so that we don't get our global scope polluted with variables. Every function creates its own local scope and in this example of IIFEs, the variable number is not available outside the function so this way we can create another variable with the same name as 'number' without any conflict.

Script.js

```
(function calc(){
  var number = 10;
```

```
  console.log(number);
})();
var number = 20;
console.log(number);
```

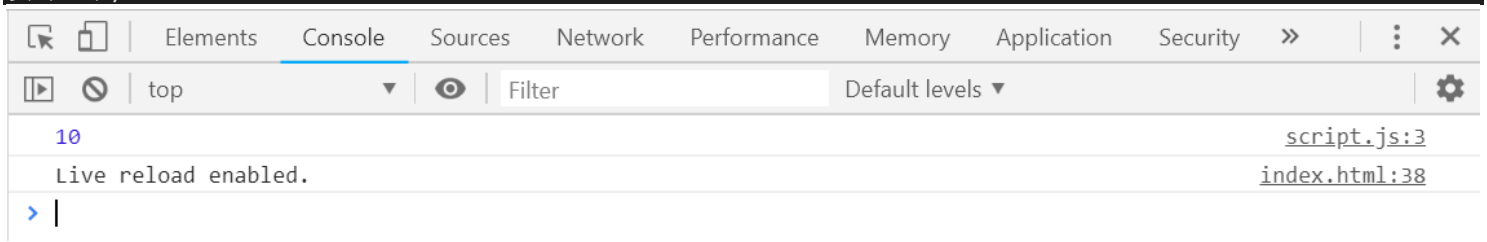


So IIFEs ensures we don't have our variables available everywhere. Now we are not saying you should write all your functions as IIFEs but it might be useful if global scope pollution is an issue in your application.

We can also pass arguments inside IIFEs

Script.js

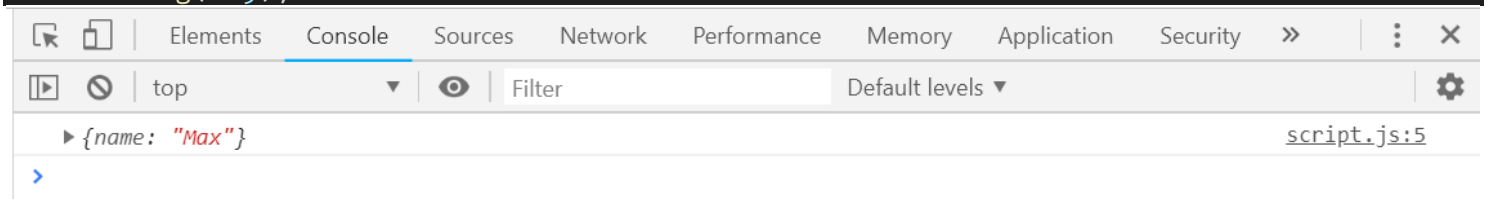
```
(function calc(input){
  var number = input;
  console.log(number);
})(10);
```



We can access global scope while working with IIFEs

Script.js

```
var obj = {}; //this is the global scope which we can access from the IIFEs
(function calc(input){
  obj.name = 'Max';
})(obj);
console.log(obj);
```



BUILT IN METHODS & PROPERTIES: - We want to learn about some important methods and properties we can use with functions.

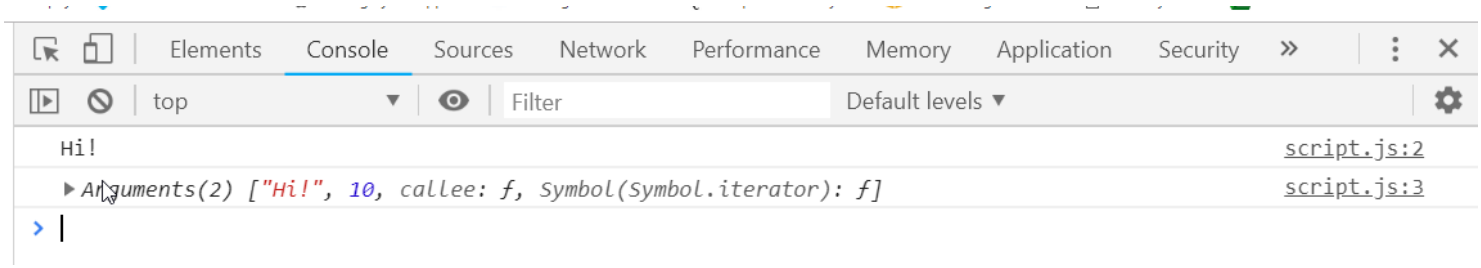
ARGUMENTS

Script.js

```
function message(message){
  console.log(message);
  console.log(arguments);
```

```
}
message('Hi!', 10);
```

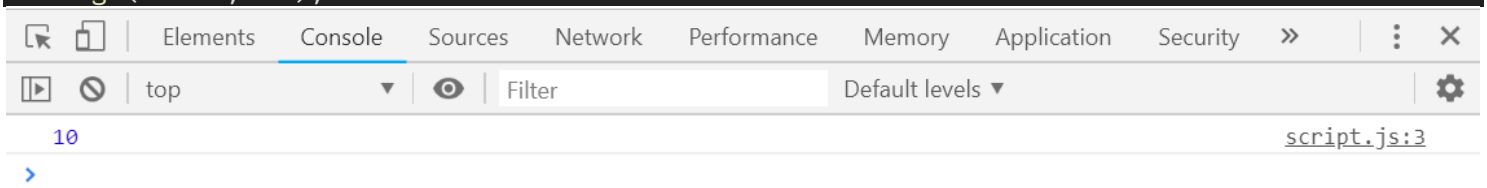
Here we are using 'arguments' and we get an object which appear like an array which has values and index which is incrementing. The string 'Hi' & 10 here are the arguments passed.



We can access the arguments we have in the objects using their index number. Let's check for the second argument '10'.

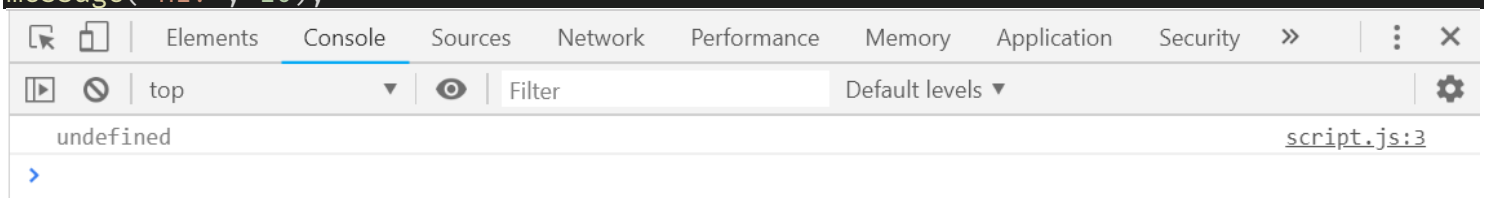
Script.js

```
function message(message){
  // console.log(message);
  console.log(arguments[1]); //checking the second arguments '10'
}
message('Hi!', 10);
```



Script.js

```
function message(message){
  // console.log(message);
  console.log(arguments[2]); //we are checking for an argument with the index of 2
which we dont have
}
message('Hi!', 10);
```



So the arguments here help to expand the functionality of our function considering the number of arguments we've got and use them in the end.

We can check for the number of arguments we have present in our function as well.

Script.js

```
function message(message){
  // console.log(message);
  console.log(arguments.length); //we are checking for the number of arguments we have
}
message('Hi!', 10);
```

```
2                                                    script.js:3
>
```

We have 2 arguments present.

We can also check for the name of our function.

Script.js

```
function message(message){
  console.log(message);
}
console.log(message.name);
```

```
message                                              script.js:4
>
```

We might be wondering why we need to access the name of the function when we already have it but it might be useful if we are passing our function dynamically or in this case below.

Script.js

```
function message(message){
  console.log(message);
}
var msg = message; //here we are passing the variable 'msg' to our function name
console.log(msg.name); // now we are accessing the name of the function
```

```
message                                              script.js:5
>
```

as we can see, we still get back the name of our function even though we assigned it to a new variable.

We can also check for the length of arguments the function expects us to pass.

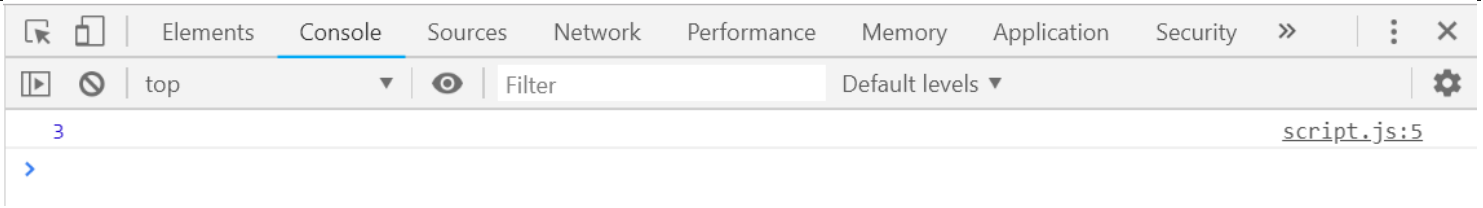
Script.js

```
function message(message){ //the length will be based on the number of arguments we
have here
  console.log(message);
}
var msg = message;
console.log(msg.length);
```

```
1                                                    script.js:5
>
```

Script.js

```
function message(message, numbers, age){ //the length will be based on the number of
arguments we have here
  console.log(message);
}
var msg = message;
console.log(msg.length);
```



Now that we passed 3 different arguments inside the function, the length has changed to 3.

BUILT IN OBJECTS & FUNCTIONS

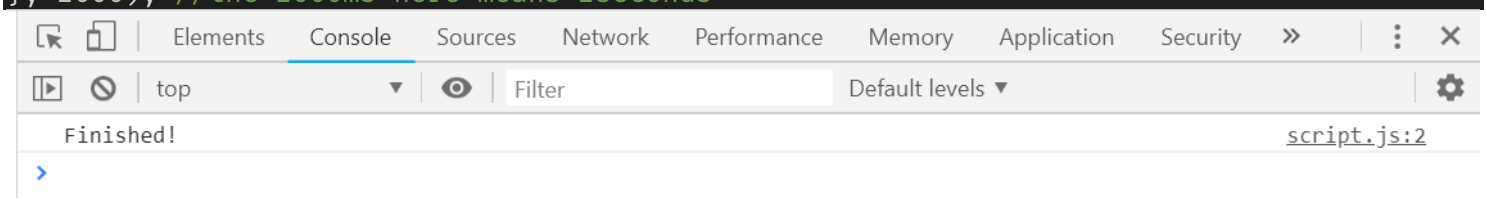
INTRODUCTION: - We want to learn about some built in objects and methods which makes JS so powerful.

TIMERS & INTERVALS: - We will start with two of the most important built in functions right away.

We will start with the setTimeout( ); this method we wait for the timeout you specified and then execute some code. So this method here expects two arguments: the functions you want to execute and the number of milliseconds the timer should wait before executing the function.

Script.js

```
setTimeout(function() {
  console.log('Finished!');
}, 2000); //the 2000ms here means 2seconds
```



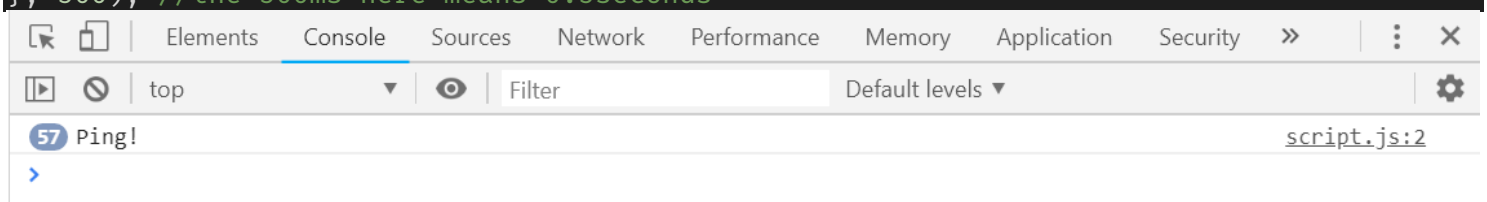
We won't experience any output for 2 seconds and after 2 seconds we get 'finished' as an output.

setInterval( );

This will execute your function every milliseconds you specify.

Script.js

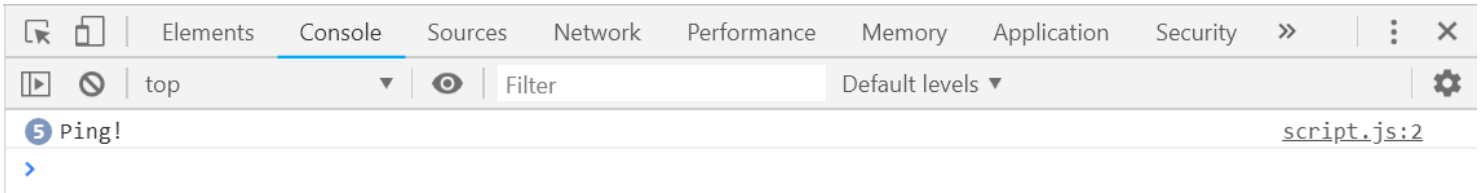
```
setInterval(function() {
  console.log('Ping!');
}, 500); //the 500ms here means 0.5seconds
```



This ping will run non-stop, but how can we stop it.

Script.js

```
var interval = setInterval(function() {
  console.log('Ping!');
}, 500);
//this will stop the setInterval after 2.5seconds
setTimeout(function () {
  clearInterval(interval);
}, 2500);
```



Since our setInterval is getting executed every 500milliseconds and we clearing it after 2500milliseconds, that's 2500/500 which is 5 times.

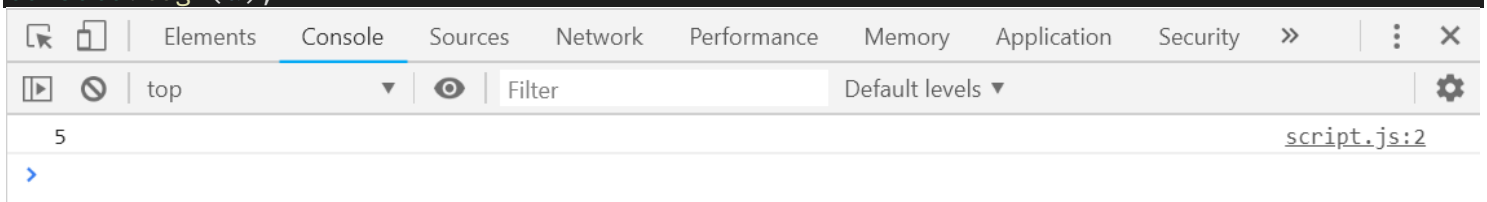
TRANSFORMING FORMAT VALUES: - Another built in methods is functions and methods which allows you to transform certain values.

parseInt( )

This tries to transform a string into a number.

Script.js

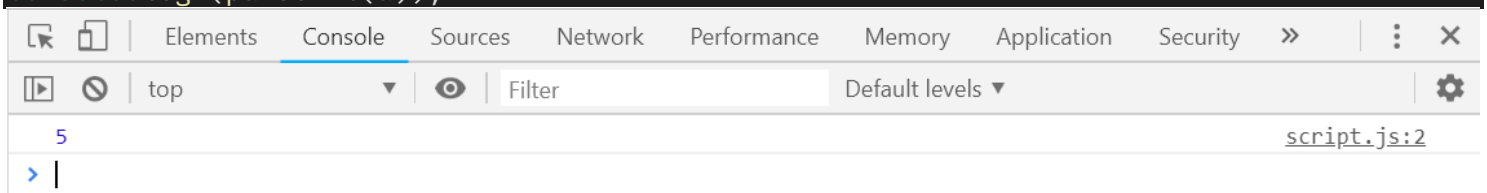
```
var a = '5';
console.log (a);
```



Currently we have 5 in the console as string, now let us convert the string into a number.

Script.js

```
var a = '5';
console.log (parseInt(a));
```



Now our string is now a number.

What about converting a text into a number

Script.js

```
var a = 'test';
console.log (parseInt(a));
```

```
NaN                                                    script.js:2
>
```

we got not a number because this is a text which JS could convert.

.toString: - This converts a number to string.

Script.js

```
var a = 10;
console.log (a.toString());
```

```
10                                                     script.js:2
>
```

10 converted to string.

.toFixed: - This will round up a decimal to whole number depending on the number of decimal places you specify in the bracket in front

Script.js

```
var a = 10.7;
console.log (a.toFixed());
```

```
11                                                     script.js:2
>
```

Script.js

```
var a = 10.4;
console.log (a.toFixed(2)); //rounding of 10.4 to a whole number in 2 decimal places
```

```
10.40                                                  script.js:2
>
```

NB: by default, it will transform it to a whole number without decimal place.

STRING FUNCTIONS: - Speaking of string/text, there a lot of things you can do with that.

We can get the length of the string i.e. the number of characters in the string including the whitespace

String.js

```
var string = "Any text";
console.log(string.length);
```

```
8                                                    script.js:2
>
```

we have 8 characters in total including the whitespace.

Just like an array, strings have index which can allow us access each character in the text.

Script.js

```
var string = "Any text";
console.log(string[2]);
```

```
y                                                    script.js:2
>
```

we get 'y' which is the letter having index of 2 because index starts with 0.

Alternative to this, we can also use another keyword 'charAt'

Script.js

```
var string = "Any text";
console.log(string.charAt(2));
```

```
y                                                    script.js:2
>
```

.concat

This will add a new string of text to our string variable value.

Script.js

```
var string = "Any text";
console.log(string.concat(' add a new string!'));
```

```
Any text add a new string!                           script.js:2
>
```

.toUpperCase

This will convert the string to uppercase.

You can also use lowercase.

Script.js

```
var string = "Any text";
console.log(string.toUpperCase());
```

```
ANY TEXT                                                    script.js:2
>
```

.split( )

This splits a string into substring using a specified separator (this can be any of the text) and return them as array.

Script.js

```
var string = "Any text";
console.log(string.split(' ')); //here we are splitting using the whitespace
```

```
▸(2) ["Any", "text"]                                        script.js:2
>
```

Script.js

```
var string = "Any text";
console.log(string.split('x')); //here we are splitting using the letter 'x'
```

```
▸(2) ["Any te", "t"]                                        script.js:2
>
```

.trim( )

This will remove white spaces at the end and beginning of strings. This can be useful when handling user inputs in forms where user might have extra whitespace before or after their name which might create a problem if you submit the text like that to a database.

Script.js

```
var string = "Any text          ";
console.log(string.trim());
```

```
Any text                                                    script.js:2
>
```

As you can see, the extra whitespaces at the end of the text has been trimmed and the one between the text was left alone.

You can check out other string functions in the cheat sheet.

THE MATH OBJECT: - In JS, there are also some important built in object you may use and one of them is the math object. The math object gives us some built in functions and properties connected to mathematics.

The Math.PI : this gives us access to the pi value in mathematics.

Script.js

```
var pi = Math.PI;
console.log(pi);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top  Filter  Default levels
3.141592653589793                                                                    script.js:2
>
```

Math.E

Script.js

```
var e = Math.E;
console.log(e);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top  Filter  Default levels
2.718281828459045                                                                    script.js:2
>
```

.abs: - to get the absolute value ignoring minus sign for example.

Script.js

```
var e = -3;
console.log(Math.abs(e));
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top  Filter  Default levels
3                                                                                    script.js:2
>
```

.round: - this will round up the number from decimal to whole number.

Script.js

```
var e = 1.567;
console.log(Math.round(e));
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top  Filter  Default levels
2                                                                                    script.js:2
>
```

.ceil: - This returns the next integer greater than the current one. For 1.27 it will return 2, 4.56 returns 5

Script.js

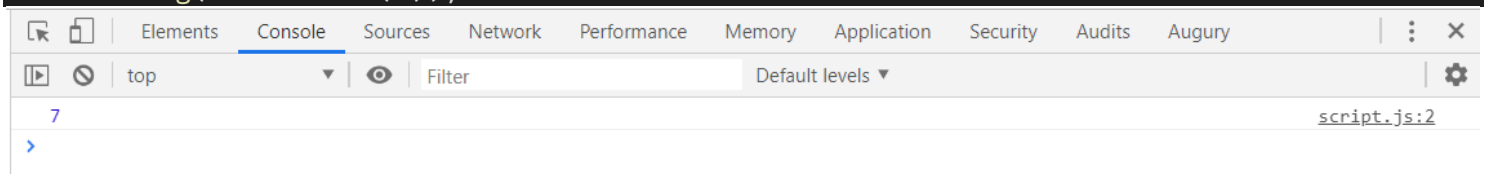
```
var e = 7.567;
console.log(Math.ceil(e));
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top  Filter  Default levels
8                                                                                    script.js:2
>
```

.floor: - this round the number down to whole number without using the numbers after decimal

Script.js

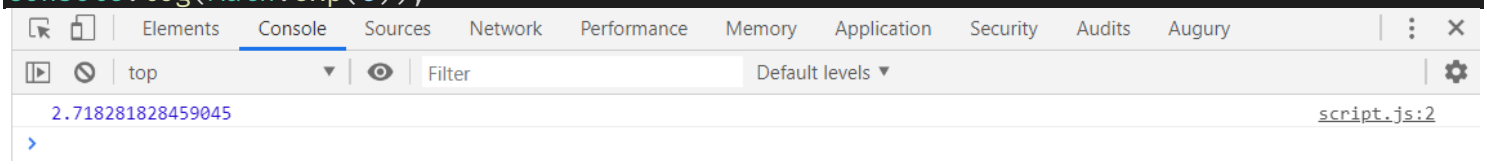
```
var e = 7.567;
console.log(Math.floor(e));
```



.exp: - exponential

Script.js

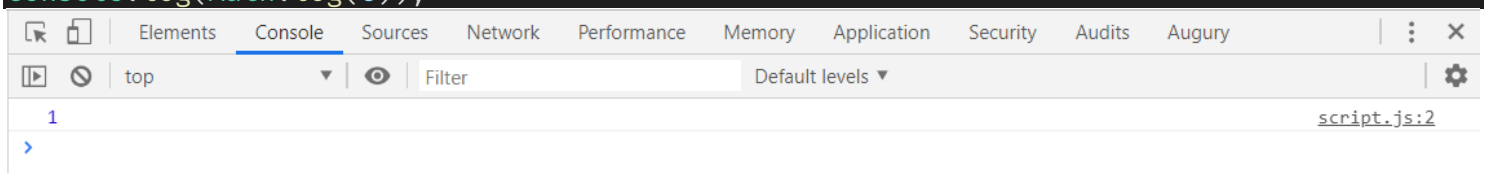
```
var e = 1;
console.log(Math.exp(e));
```



.log: - returns the natural logarithm base e of a number

Script.js

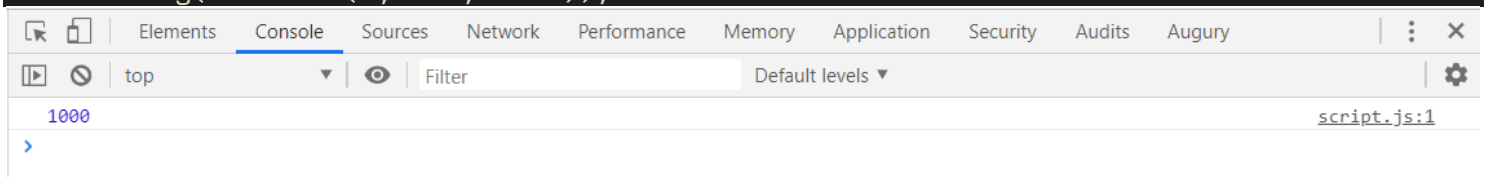
```
var e = Math.E;
console.log(Math.log(e));
```



.max: - this returns the highest possible value in a given array of numbers

Script.js

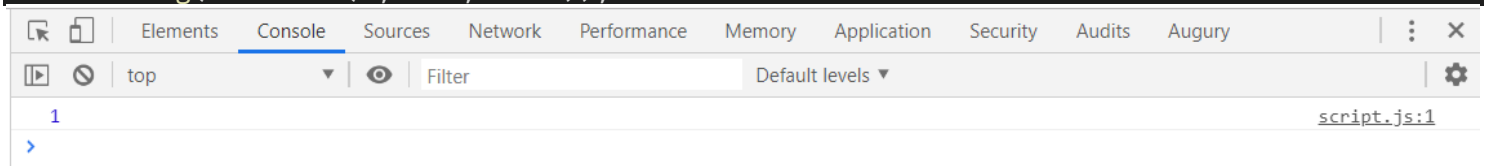
```
console.log(Math.max(1, 100, 1000));
```



.min: - this returns the lowest possible value in a given array of numbers

Script.js

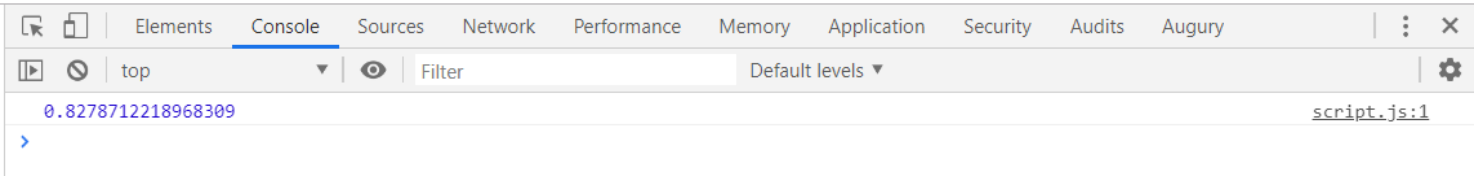
```
console.log(Math.min(1, 100, 1000));
```



.random: - this returns a random number decimal number between 0 and 1

Script.js

```
console.log(Math.random());
```

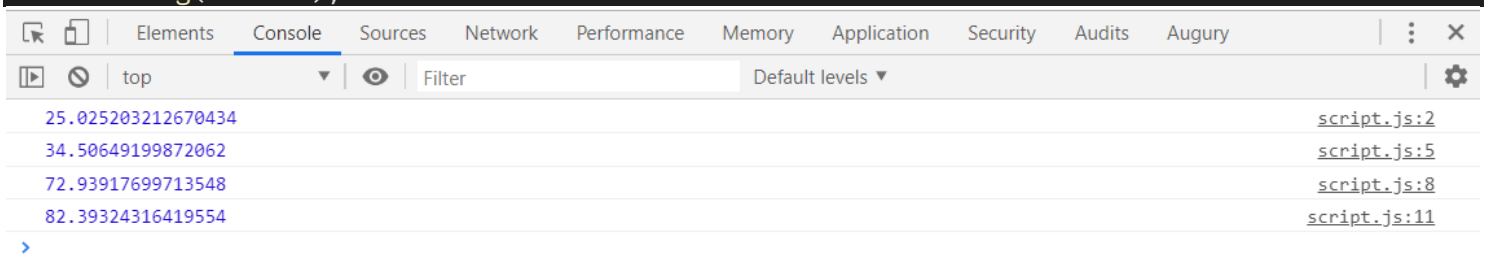


Random number has its usefulness in apps for example if you are designing a quiz app, you might need to use a random number but often times we might need a random number that won't be between 0 and 1 and you might also want the random number not to be in floating/decimal figure.

Let's start with how we can generate random number between 0 and 100. NB: This will be 1 – 99.999999

Script.js

```
var random = Math.random() * 100; //multiply it by the number u want it to be the
highest
console.log(random);
//WE ARE TRYING TO GENERATE ANOTHER RANDOM NUMBER
var random = Math.random() * 100; //multiply it by the number u want it to be the
highest
console.log(random);
//WE ARE TRYING TO GENERATE ANOTHER RANDOM NUMBER
var random = Math.random() * 100; //multiply it by the number u want it to be the
highest
console.log(random);
//WE ARE TRYING TO GENERATE ANOTHER RANDOM NUMBER
var random = Math.random() * 100; //multiply it by the number u want it to be the
highest
console.log(random);
```



Now let's ensure we start from 1 and not 0 and also include 100 as a possible number, then we will also eliminate decimal.

Script.js

```
var random = Math.floor(Math.random() * 100) + 1;
console.log(random);
//
var random = Math.floor(Math.random() * 100) + 1;
console.log(random);
//
var random = Math.floor(Math.random() * 100) + 1;
console.log(random);
//
var random = Math.floor(Math.random() * 100) + 1;
console.log(random);
//Math.floor will round down our results to whole number without approximation. so
0.9999 will return 0 but 0 wont be possible because we have added + 1 which will ensure
we start from 1 to 100
```

```
24                                  script.js:2
2                                   script.js:5
17                                  script.js:8
16                                  script.js:11
>
```

THE DATE OBJECT: - This allows you to easily create dates, work with them, pass them etc.

Date( )

Script.js

```
var today = new Date();
console.log(today);
```

```
Tue Mar 19 2019 10:32:56 GMT+0100 (West Africa Standard Time)   script.js:2
>
```

This the default format for generating today's date and time but you may want to generate a specific date maybe in the past or future. To do this, you will need to pass an argument to the date constructor.

Script.js

```
var today = new Date(2017, 3, 7); //2017, 3, 7 will return april 7th 2017. april will be 3 not 5 because JS counts from 0 so January is 0 not 1
console.log(today);
```

```
Fri Apr 07 2017 00:00:00 GMT+0100 (West Africa Standard Time)   script.js:2
>
```

Script.js

```
var today = new Date(2017, 0, 7); //2017, 3, 7 will return april 7th 2017. april will be 3 not 5 because JS counts from 0 so January is 0 not 1
console.log(today);
```

```
Sat Jan 07 2017 00:00:00 GMT+0100 (West Africa Standard Time)   script.js:2
>
```

Creating a date that doesn't exist like 32nd of the month, JS will round it up to the next month.

Script.js

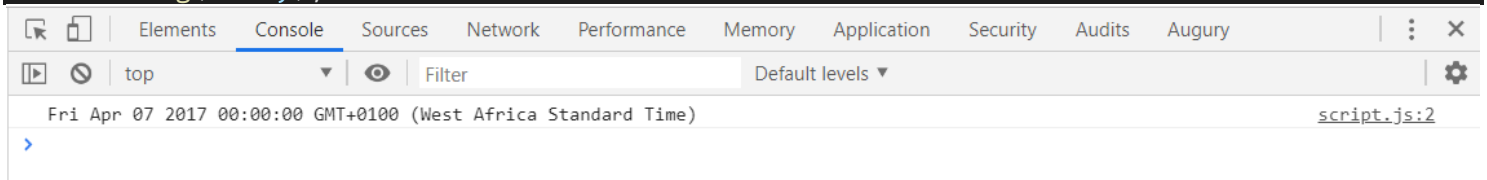
```
var today = new Date(2017, 0, 32); //2017, 3, 7 will return april 7th 2017. april will be 3 not 5 because JS counts from 0 so January is 0 not 1
console.log(today);
```

```
Wed Feb 01 2017 00:00:00 GMT+0100 (West Africa Standard Time)   script.js:2
>
```

You can also pass your date arguments as a string and in this case, the month will not start from 0 but from 1.

Script.js

```
var today = new Date('2017/04/07'); //date argument getting passed as a string where 04 is now april
console.log(today);
```

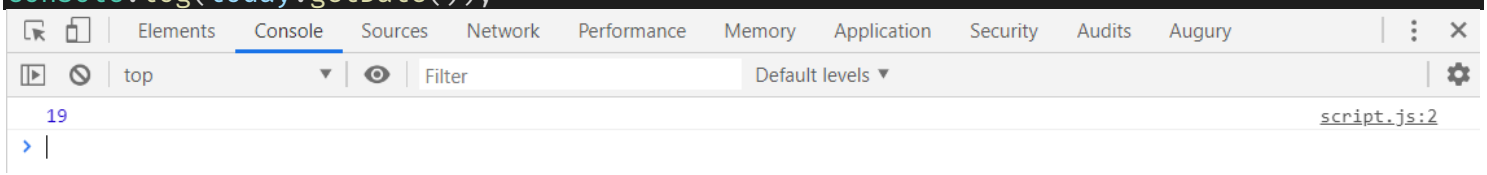


In the cheat sheet attached to this episode, you will find valid format for dates in JS.

We can get the current day in the month using the code below.

Script.js

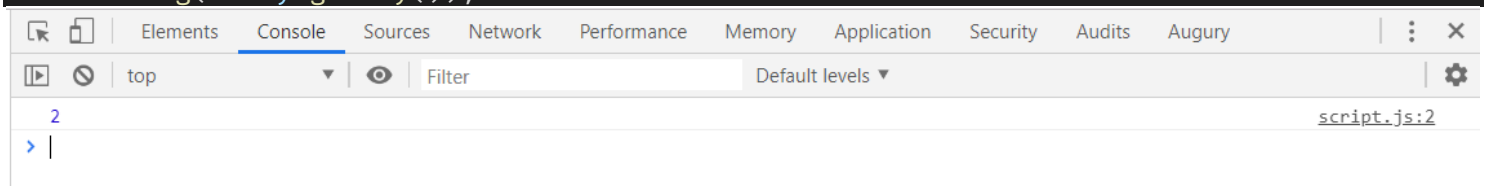
```
var today = new Date();
console.log(today.getDate());
```



You can get current index number of the day of the week, where Sunday = 0, Monday = 1, Tuesday = 2 etc.

Script.js

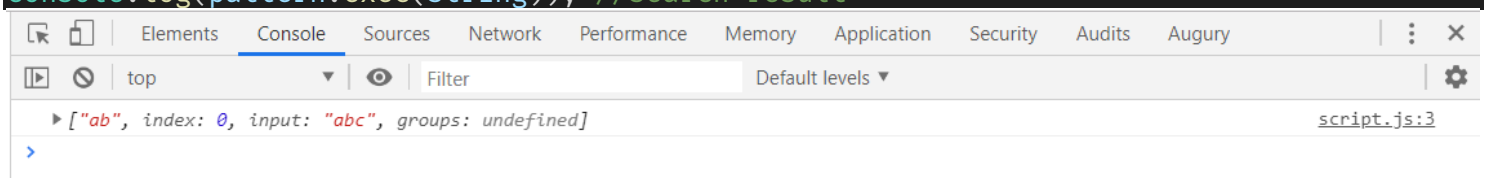
```
var today = new Date();
console.log(today.getDay());
```



REGULAR EXPRESSIONS: - The last built in object is the regular expression. Regular expression are patterns use to match character combination in strings. So basically it's a search method to search for characters in a string.

Script.js

```
var string = 'abc';
var pattern = /ab/; //search for ab in the string
console.log(pattern.exec(string)); //search result
```



"ab" was found from the string and returned as an array.

Let's try to search for characters that doesn't exist in a string.

Script.js

```
var string = 'ac';
var pattern = /ab/; //search for ab in the string
console.log(pattern.exec(string)); //search result
```

```
null                                                    script.js:3
>
```

we get null because ab cannot be found in ac.

Script.js

```
var string = 'abc';
var pattern = /ac/; //search for ac that follows each other as a pattern in the string
console.log(pattern.exec(string)); //search result
```

```
null                                                    script.js:3
>
```

here we also get null because though a & c are present inside abc but they don't follow each other. Although there is wildcard regular expression that looks for the characters not minding their arrangement but this one we are using looks for the characters as well as their pattern.

Script.js

```
var string = 'abc';
var pattern = /ab/; //search for ac that follows each other as a pattern in the string
console.log(pattern.test(string)); //test if the pattern matches the string
```

```
true                                                    script.js:3
>
```

This tested if the pattern /ab/ matches the string "abc".

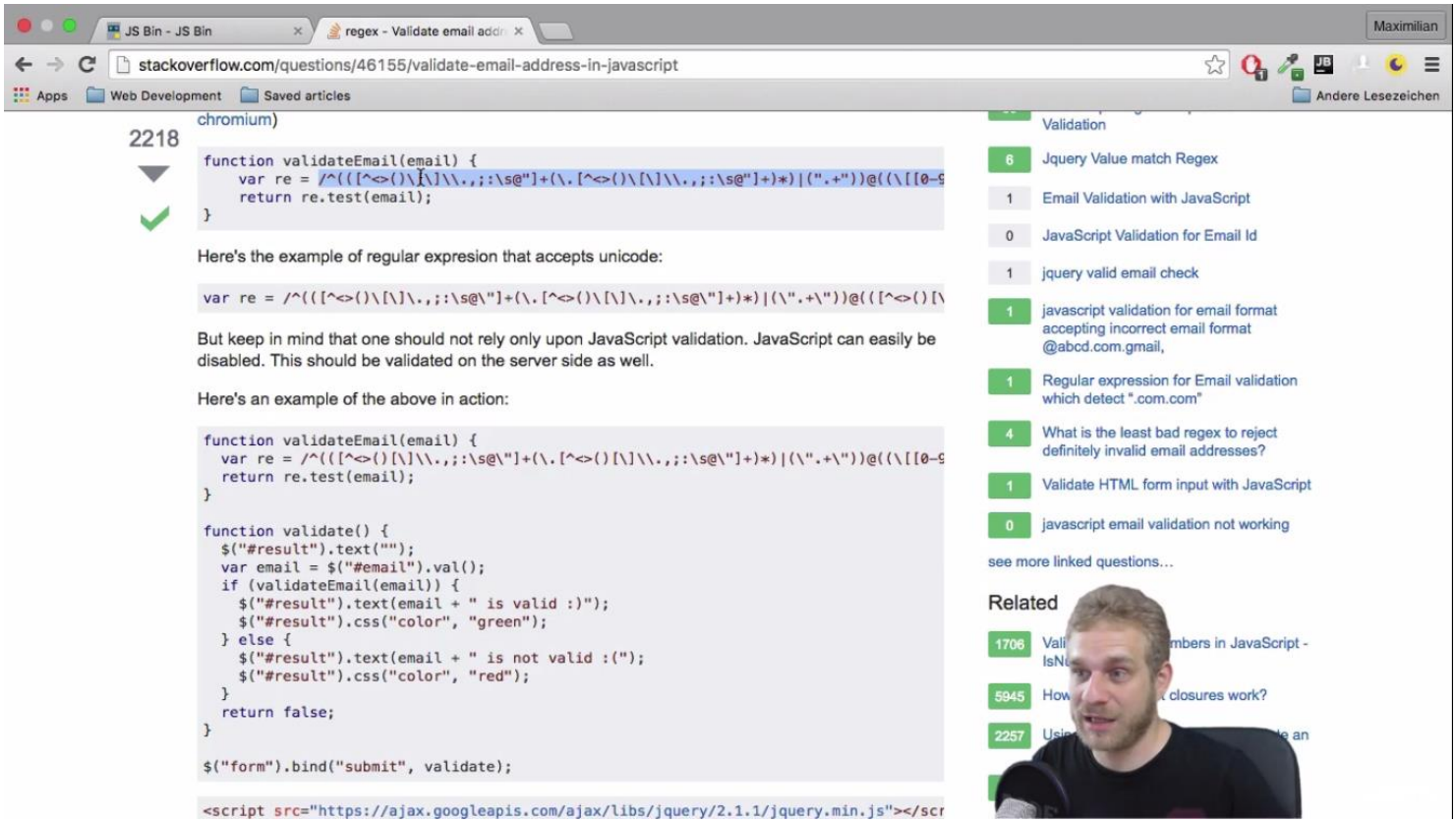
There is also another way of finding and logging characters

Script.js

```
var string = 'abc';
var pattern = /ab/; //search for ac that follows each other as a pattern in the string
console.log(string.match(pattern));
```

```
▶ ["ab", index: 0, input: "abc", groups: undefined]     script.js:3
>
```

Regular expressions is deeper than this one episode. You can learn more on google with the search tag. 'validate email regex'



So regular expression validation for email will look for patterns in the email address typed to ensure it matches the one defined in the code.

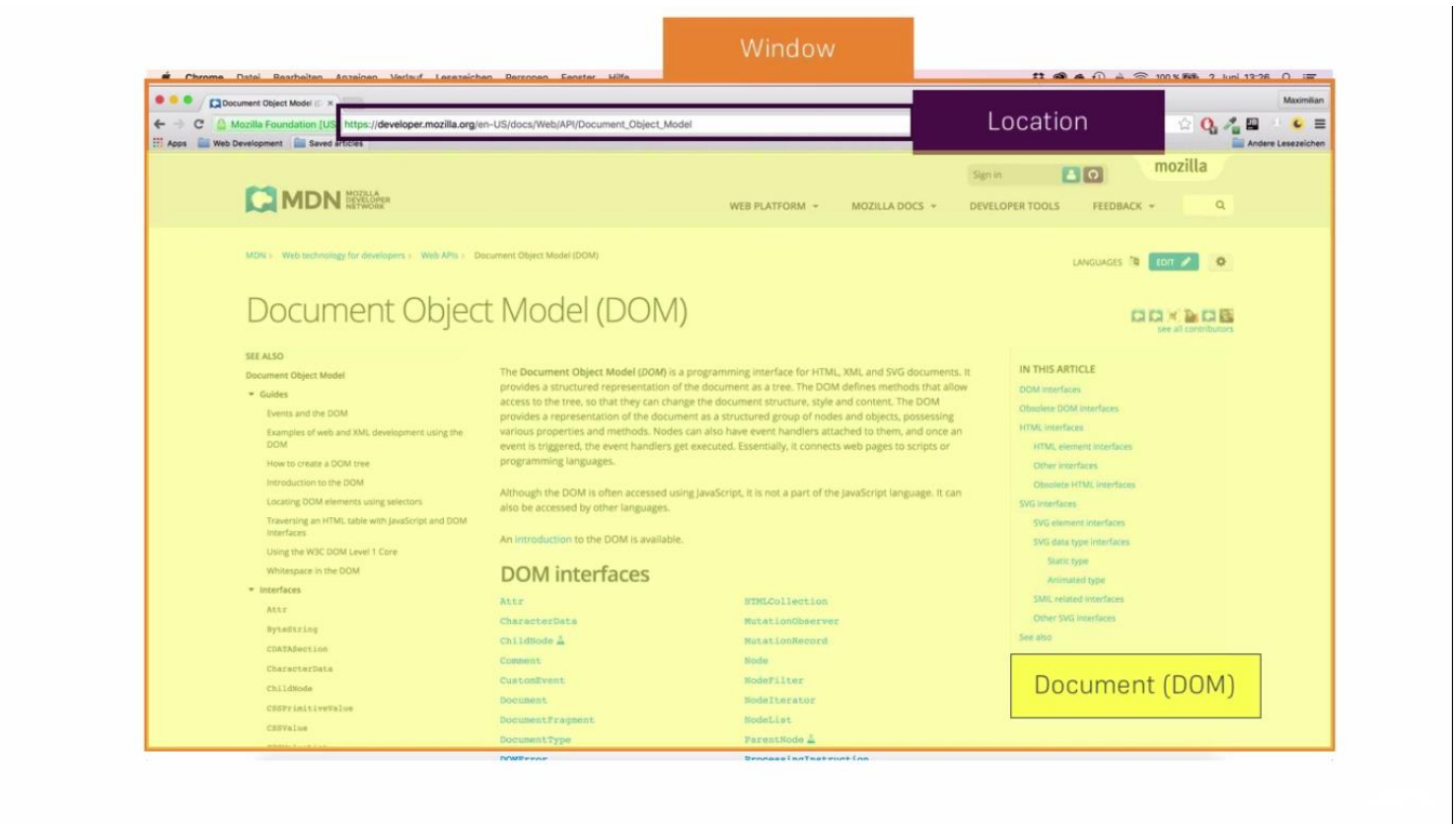
What's the use of regular expression?

- **You can use it to validate user input in a form e.g. to ensure the email address the user entered is a valid email address. This way you ensure the user is not playing a trick or forgetting the @ sign for example.**
- **It can also be useful for finding certain character/text from a string of text.**

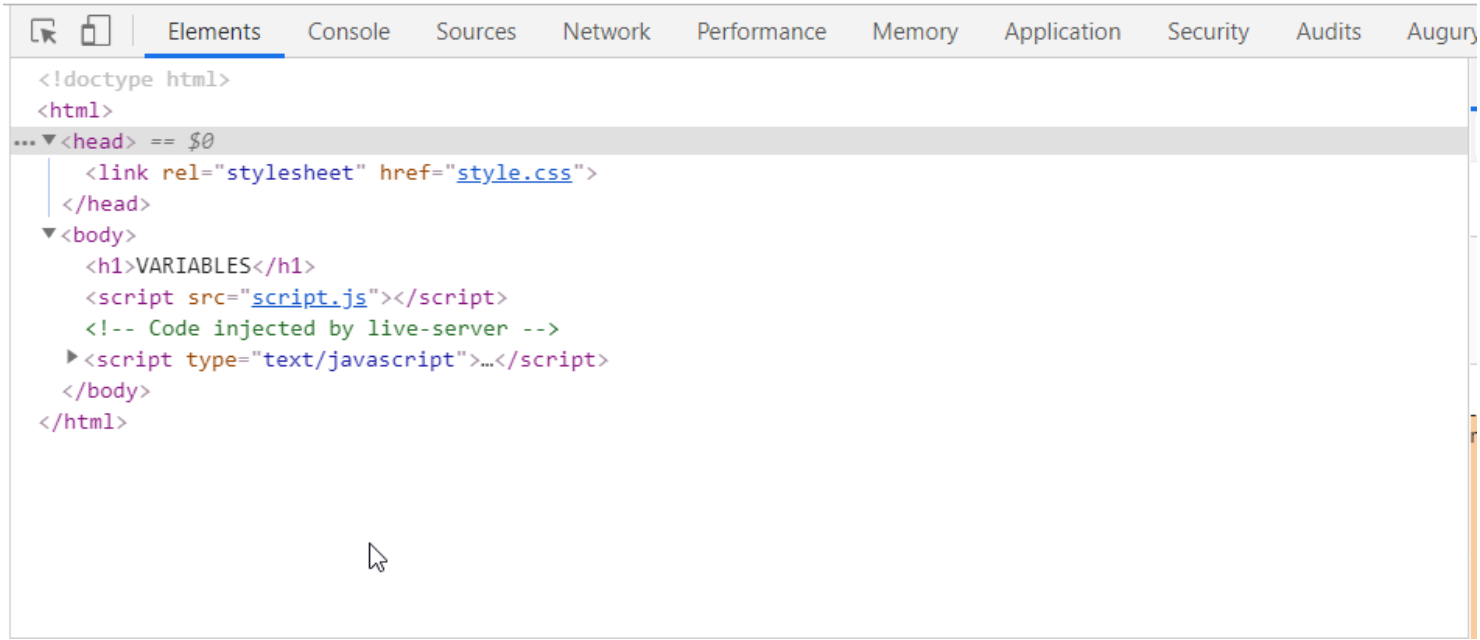
WORKING WITH THE WINDOW AND DOCUMENT OBJECT MODEL.

INTRODUCTION: - Here we will learn about the interaction between the JS and the html code because at the end, your JS code will need to interact with the html code maybe to evaluate some code, change some html elements at run time.

What is the DOM? The DOM can be easily explained with the diagram below, we have the window, the location and the DOM. The DOM is where all our html codes are rendered. The location is the web address which we can also interact with to change the location/website.



We can also interact with the DOM and change some things in there. You can access the DOM with the developer tools in chrome. The DOM is the html codes under the elements tab.

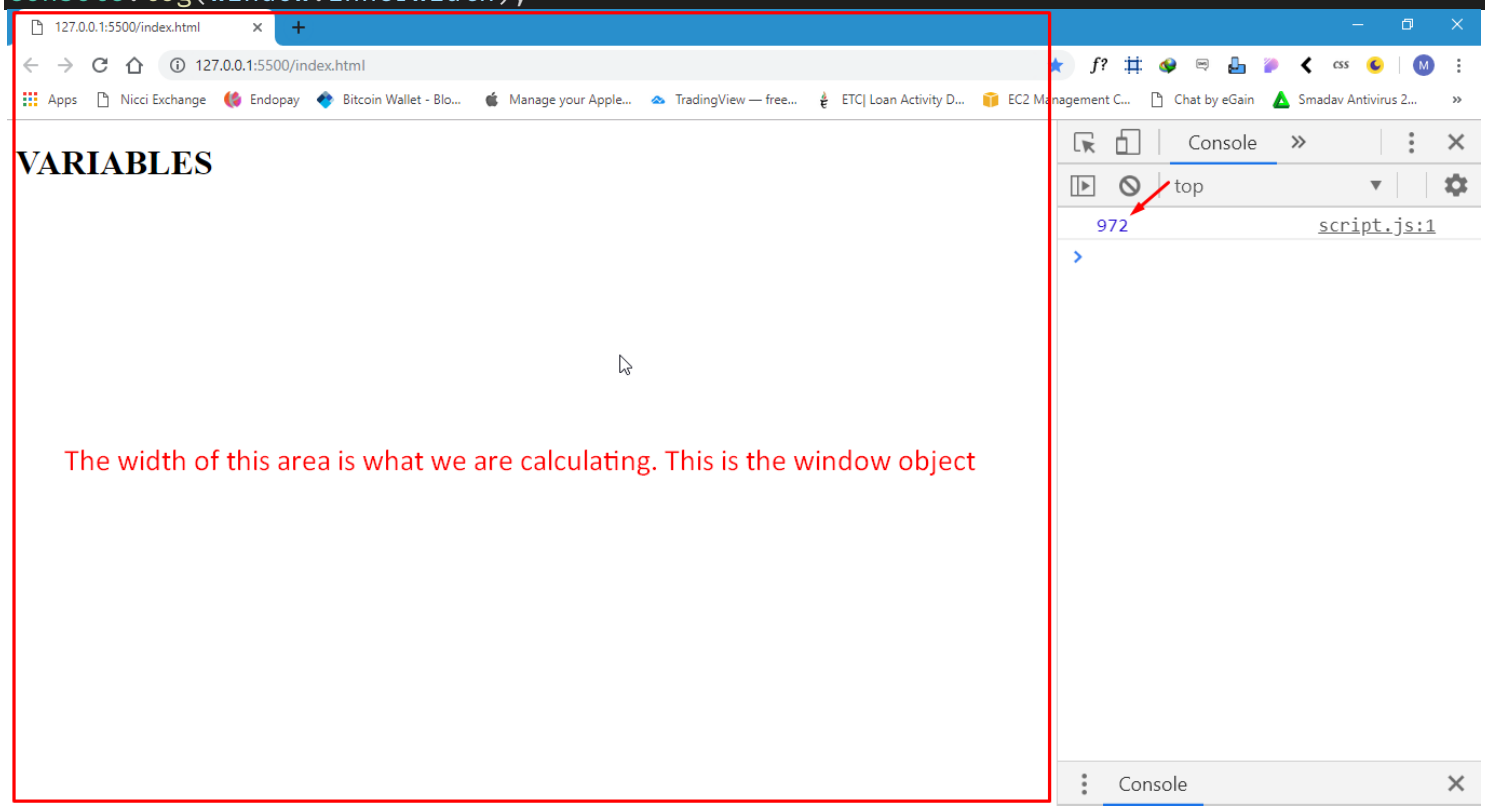


The diagram above is the DOM.

THE WINDOW OBJECT: - This is the overall holding container inside which we have location and the DOM and remember we said the global object is the window object in the browser. There are many codes you can run using the window object. For example, you can get the width of the window object i.e. the width of the interface where the browser is loaded, this will depend on how big or small your developer interface is because the bigger the width of your console, the small the width of the window object and vice versa.

Script.js

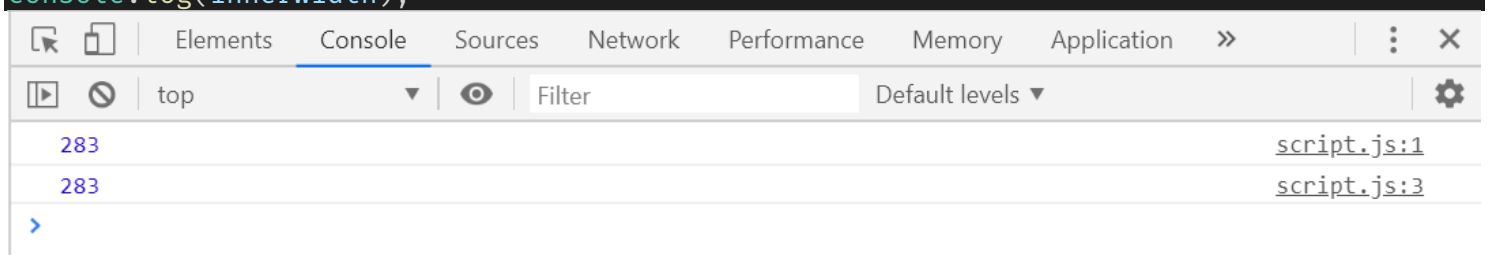
```
console.log(window.innerWidth);
```



You can as well use the 'innerWidth' without the window

Script.js

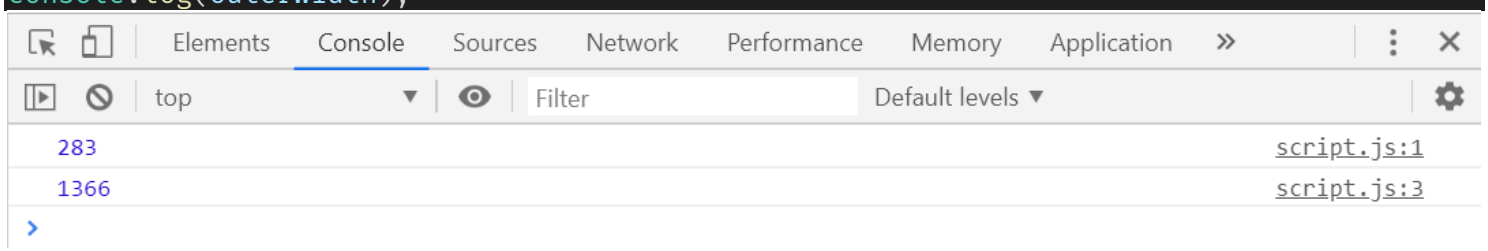
```
console.log(window.innerWidth);
//OR
console.log(innerWidth);
```



outerWidth: - this is the full size of our window

script.js

```
console.log(window.innerWidth);
//OR
console.log(outerWidth);
```



We also have the innerHeight and outerHeight that works in the same manner.

Script.js

```
console.log(window.innerHeight);
//OR
console.log(outerHeight);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  »
top ▼  Filter  Default levels ▼
635                                                    script.js:1
738                                                    script.js:3
>
```

You can as well get all the properties and methods of the window object in the console.

Script.js

```
console.log(window);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top ▼  Filter  Default levels ▼
▶clearInterval: ƒ clearInterval()
▶clearTimeout: ƒ clearTimeout()
▶clientInformation: Navigator {vendorSub: "", productSub: "20030107", vendor: "Google Inc.", maxTouchPoints: 0, hardwareConcurrency: 4, …}
▶close: ƒ ()
 closed: false
▶confirm: ƒ confirm()
▶createImageBitmap: ƒ createImageBitmap()
▶crypto: Crypto {subtle: SubtleCrypto}
▶customElements: CustomElementRegistry {}
 defaultStatus: ""
 defaultstatus: ""
 devicePixelRatio: 1
▶document: document
▶external: External {}
▶fetch: ƒ fetch()
▶find: ƒ find()
▶focus: ƒ ()
 frameElement: null
▶frames: Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
▶getComputedStyle: ƒ getComputedStyle()
▶getSelection: ƒ getSelection()
▶history: History {length: 1, scrollRestoration: "auto", state: null}
▶indexedDB: IDBFactory {}
 innerHeight: 635
 innerWidth: 319
 isSecureContext: true
 length: 0
▶localStorage: Storage {length: 0}
▶location: Location {replace: ƒ, href: "http://127.0.0.1:5500/index.html", ancestorOrigins: DOMStringList, origin: "http://127.0.0.1:5500", proto…
▶locationbar: BarProp {visible: true}
▶matchMedia: ƒ matchMedia()
▶menubar: BarProp {visible: true}
▶moveBy: ƒ moveBy()
▶moveTo: ƒ moveTo()
 name: ""
```

You can store values in your localStorage which can then be called and used throughout your applications.

Script.js

```
localStorage.setItem('key1', 1000); //Here we storing the key (key1) and the value
(1000)
console.log(localStorage.getItem('key1')); //now we are calling the value of the key
stored
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  »
top ▼  Filter  Default levels ▼
1000                                                   script.js:2
>
```

You can also store in sessionStorage as well.

Script.js

```
sessionStorage.setItem('key1', 956); //Here we storing the key (key1) and the value
(956)
console.log(sessionStorage.getItem('key1')); //now we are calling the value of the key
stored
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
956                                                                    script.js:2
>
```

The difference between local and session storage is that the localStorage will remain but the sessionStorage will be deleted once a user close his or her browser.

The window object also has the location property, this is the web address.

Script.js

```
console.log(window.location);
```

```
                                                                       script.js:1
▼Location {replace: ƒ, href: "http://127.0.0.1:5500/index.html", ancestorOrigins: DOMStringList, origin: "http://
127.0.0.1:5500", protocol: "http:", …}
  ▶ancestorOrigins: DOMStringList {length: 0}
  ▶assign: ƒ assign()
   hash: ""
   host: "127.0.0.1:5500"
   hostname: "127.0.0.1"
   href: "http://127.0.0.1:5500/index.html"
   origin: "http://127.0.0.1:5500"
   pathname: "/index.html"
   port: "5500"
   protocol: "http:"
  ▶reload: ƒ reload()
  ▶replace: ƒ ()
   search: ""
  ▶toString: ƒ toString()
  ▶valueOf: ƒ valueOf()
   Symbol(Symbol.toPrimitive): undefined
  ▶__proto__: Location
```

You can also access the dom from the console.

Script.js

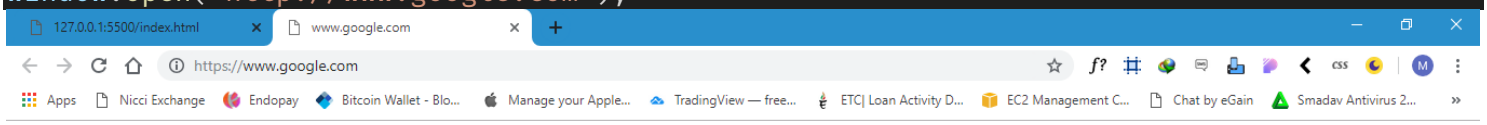
```
console.log(window.document);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▼#document                                                             script.js:1
   <!doctype html>
   <html>
   ▼<head>
      <link rel="stylesheet" href="style.css">
    </head>
   ▼<body>
      <h1>VARIABLES</h1>
      <script src="script.js"></script>
      <!-- Code injected by live-server -->
    ▶<script type="text/javascript">…</script>
    </body>
   </html>
>
```

You can also write a window object built in code to open a website address.

Script.js

```
window.open('http://www.google.com');
```



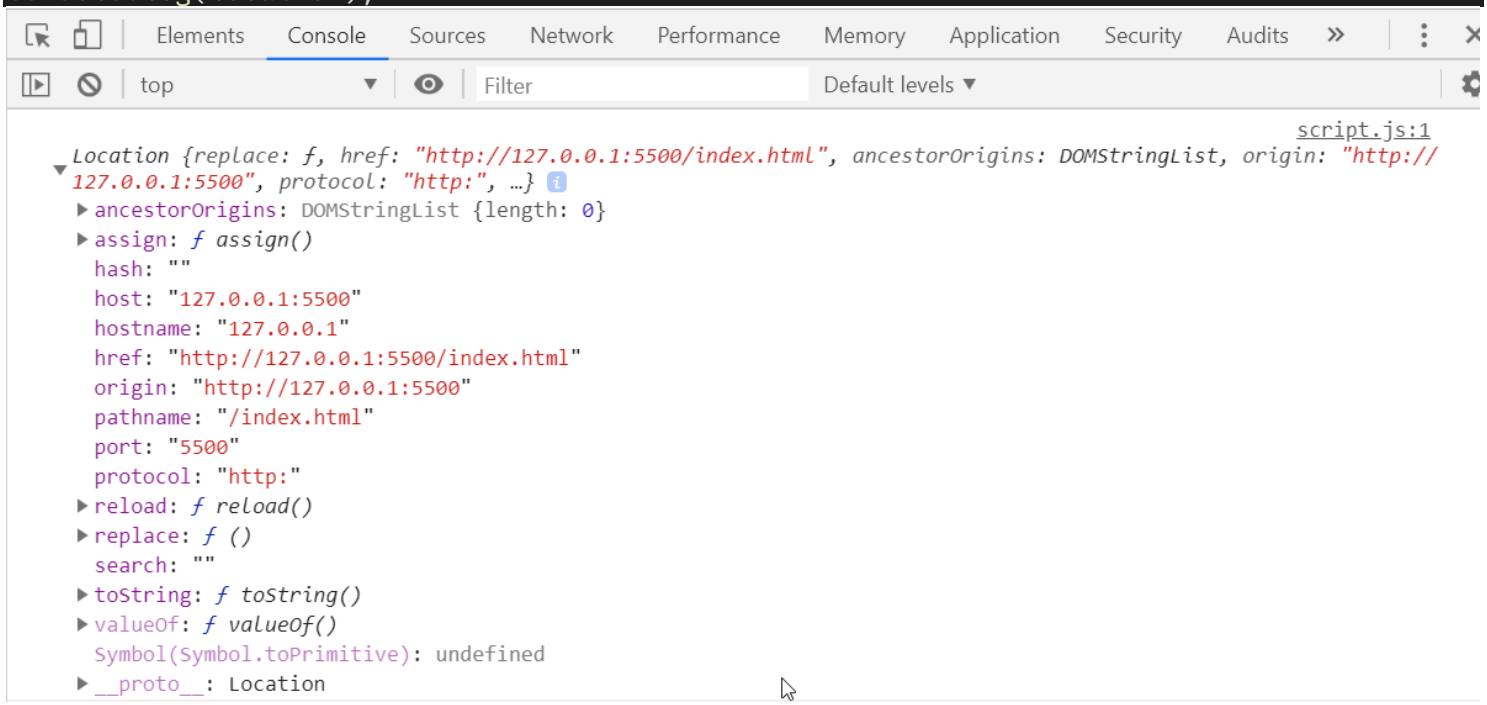
Check out the cheat sheet for more methods and the properties to dive deeper.

NB: - By default we are running on the window object so you do not need to add window. to execute your codes.

THE LOCATION OBJECT: - Now that we have seen the window object, let's see the location object which we had been shown before.

Script.js

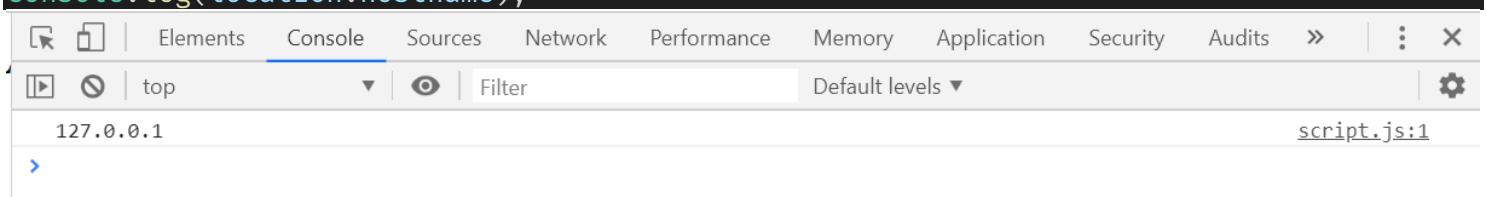
```
console.log(location);
```



You can access each of this properties one by one.

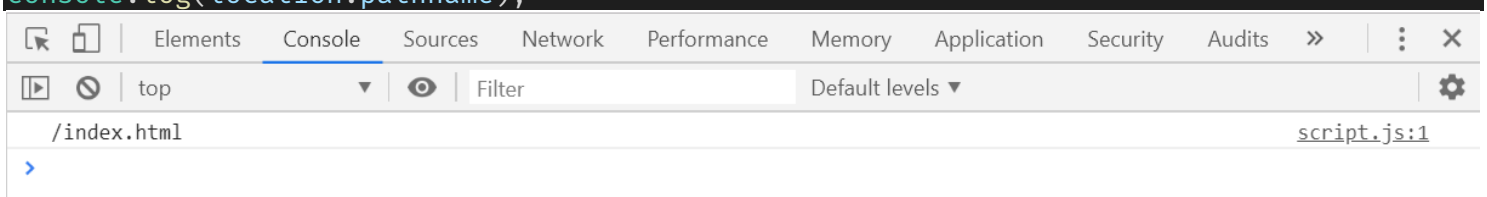
Script.js (accessing the hostname)

```
console.log(location.hostname);
```



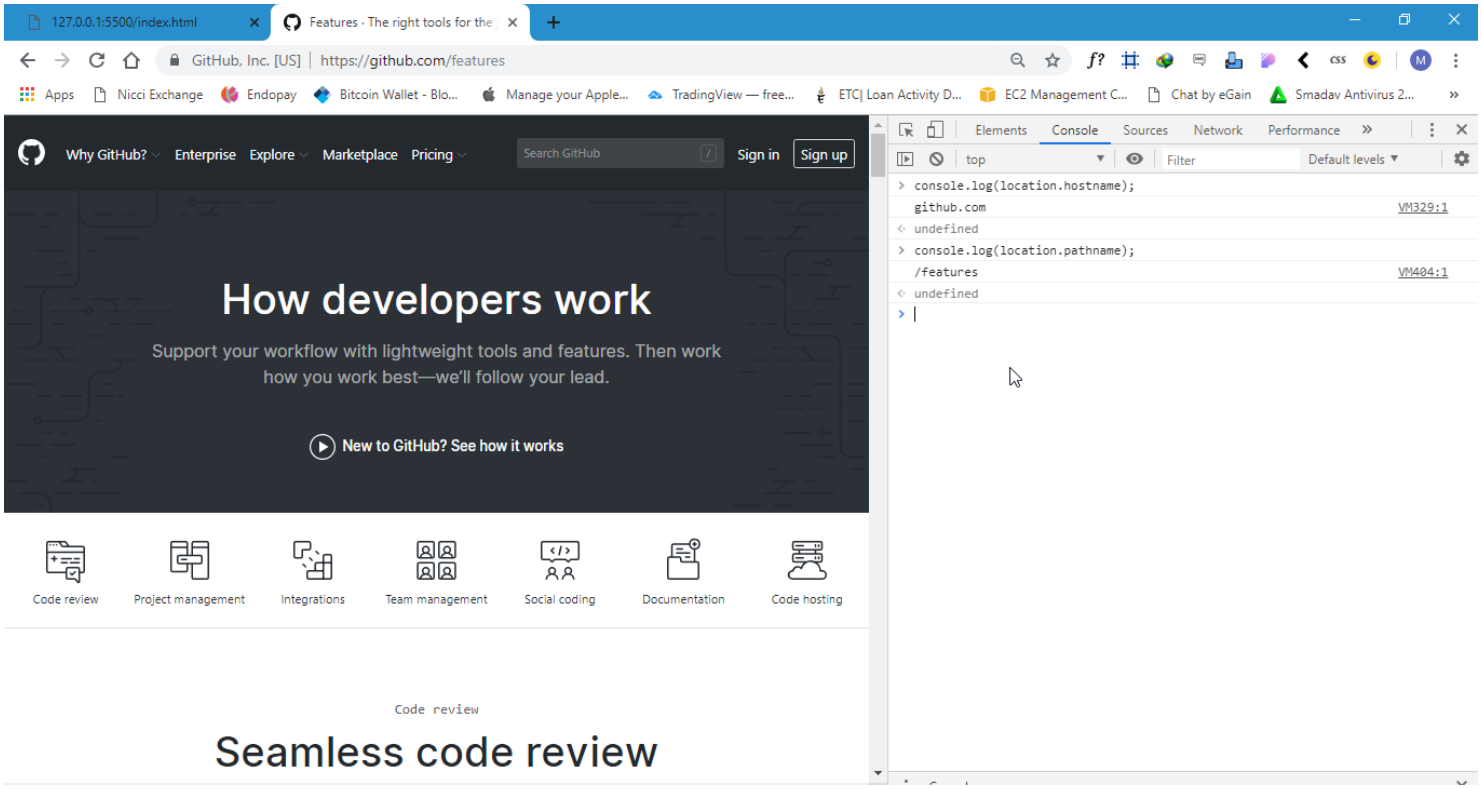
Script.js (accessing the pathname)

```
console.log(location.pathname);
```



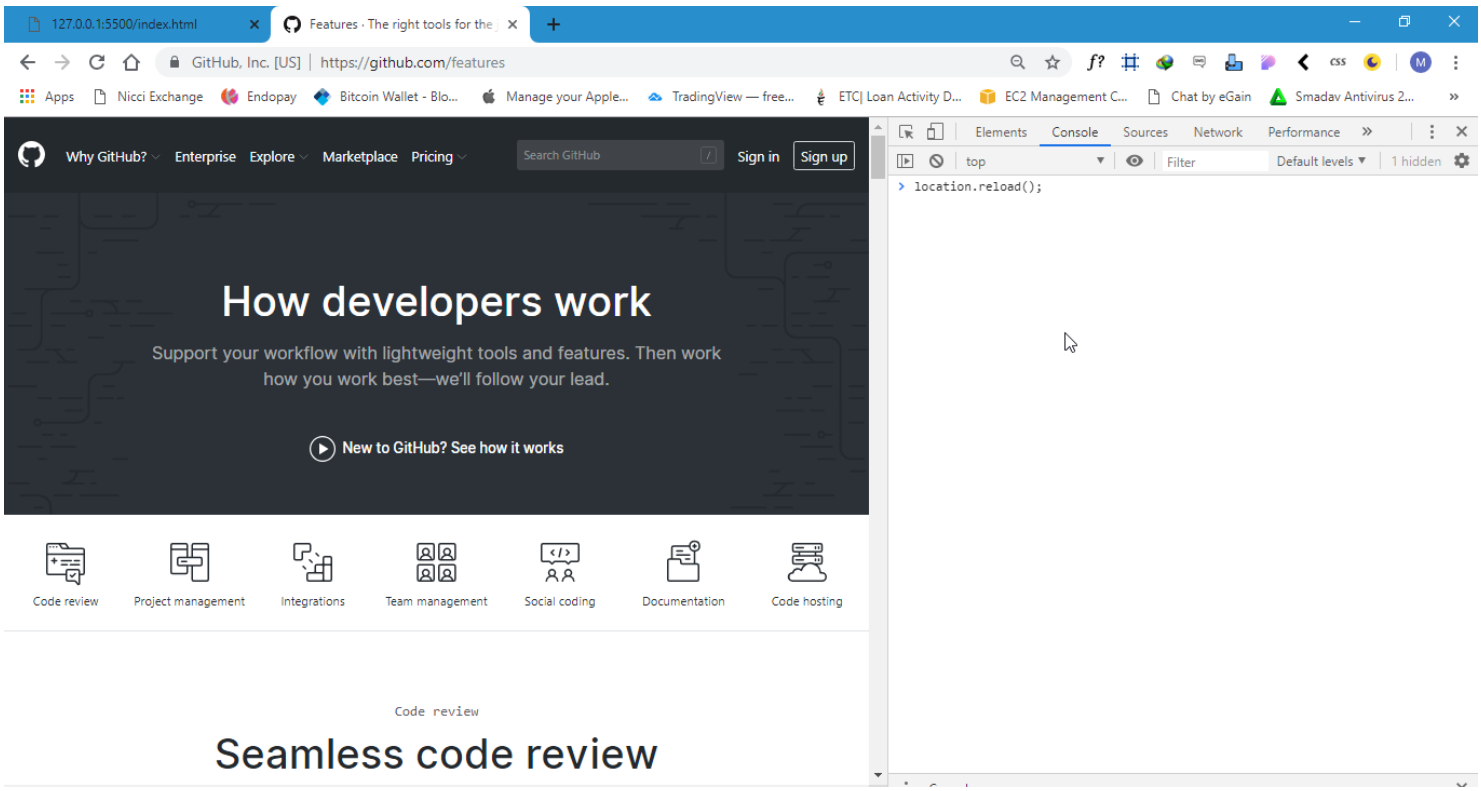
You can also run all this codes on a live website.

- **Open the website**
- **Open the developer tools and run any of the codes**

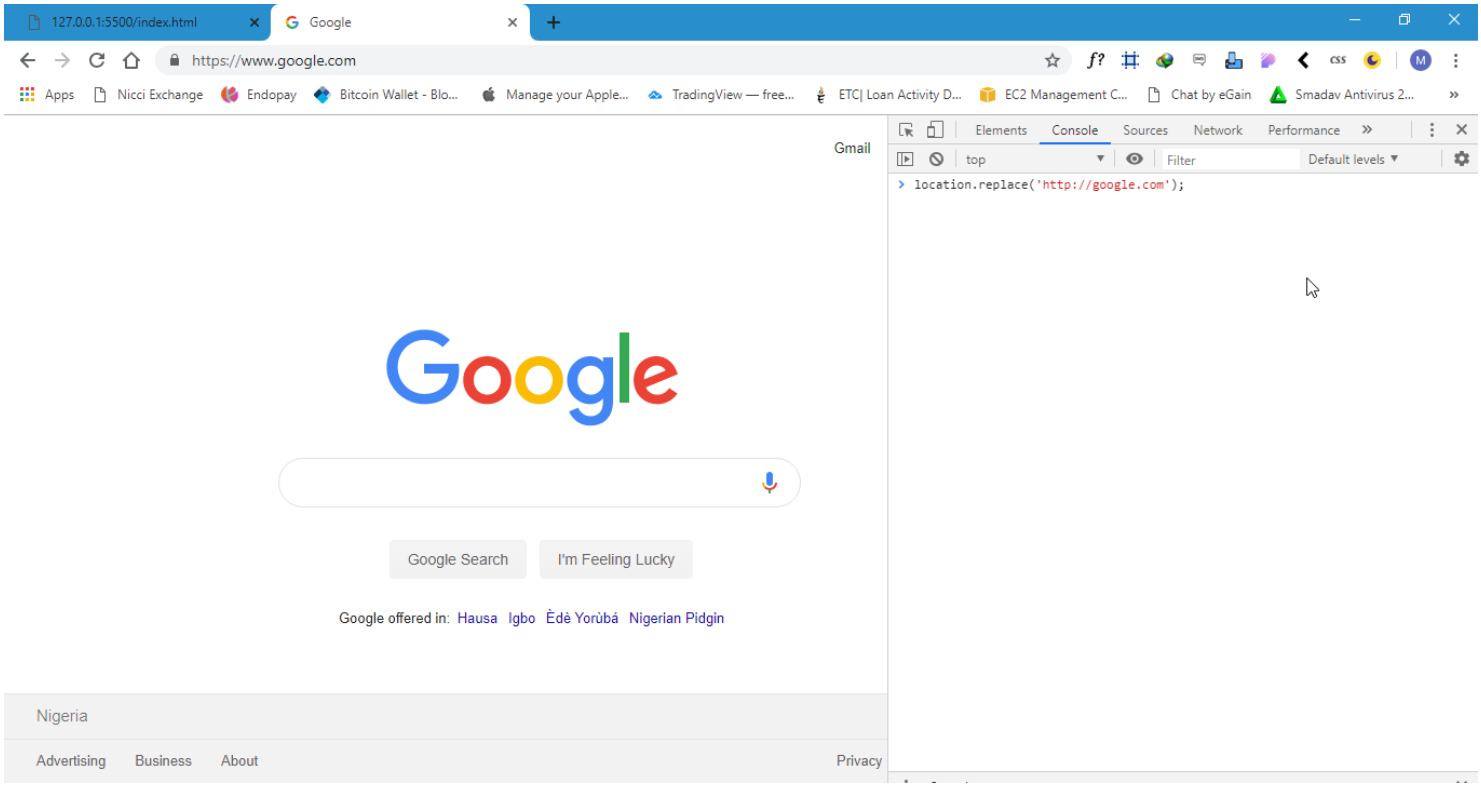


You can also use:

Location.reload(): this will reload the page



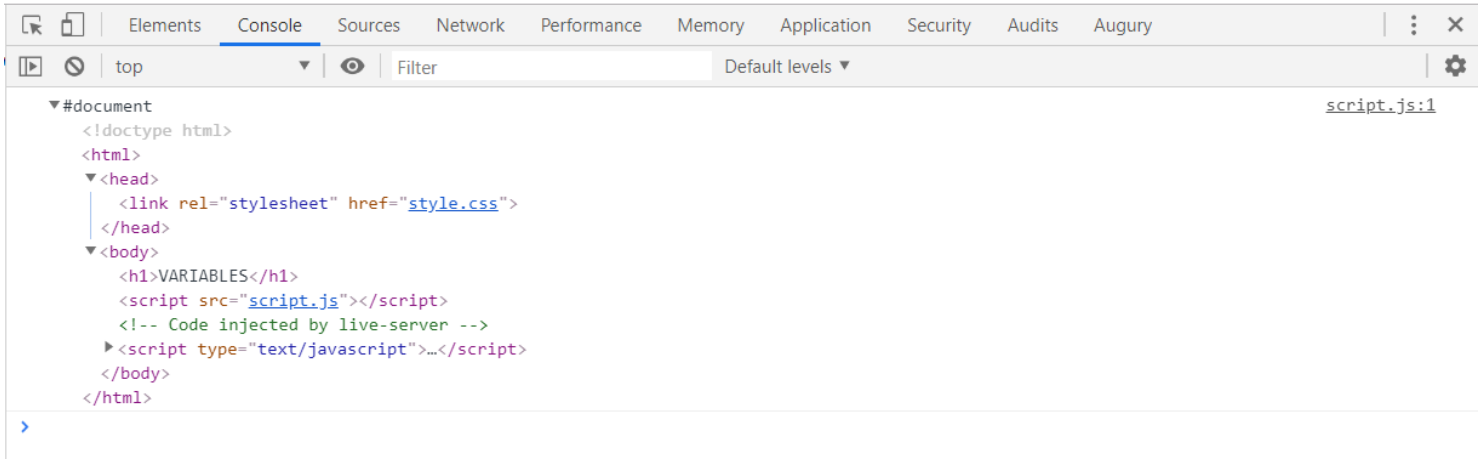
Location.replace(): this will replace the website with another



THE DOCUMENT OBJECT AND HOW TO INTERACT WITH IT: - Now that we have touched the window object and the location object, it is time to touch the document object. This you will use a lot because you will want to interact with the DOM and the html codes.

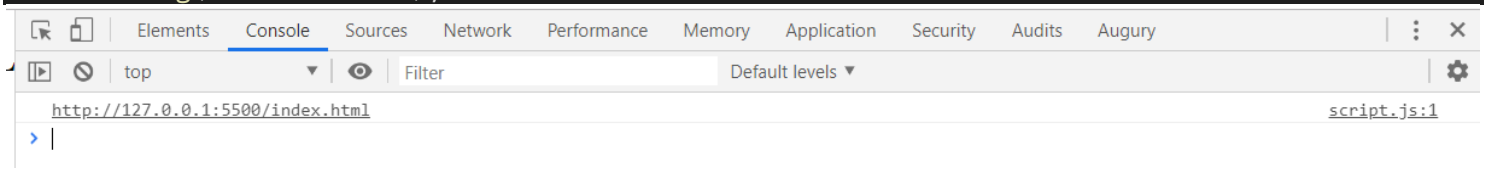
Script.js (access the document object)

```
console.log(document);
```



Script.js (document url)

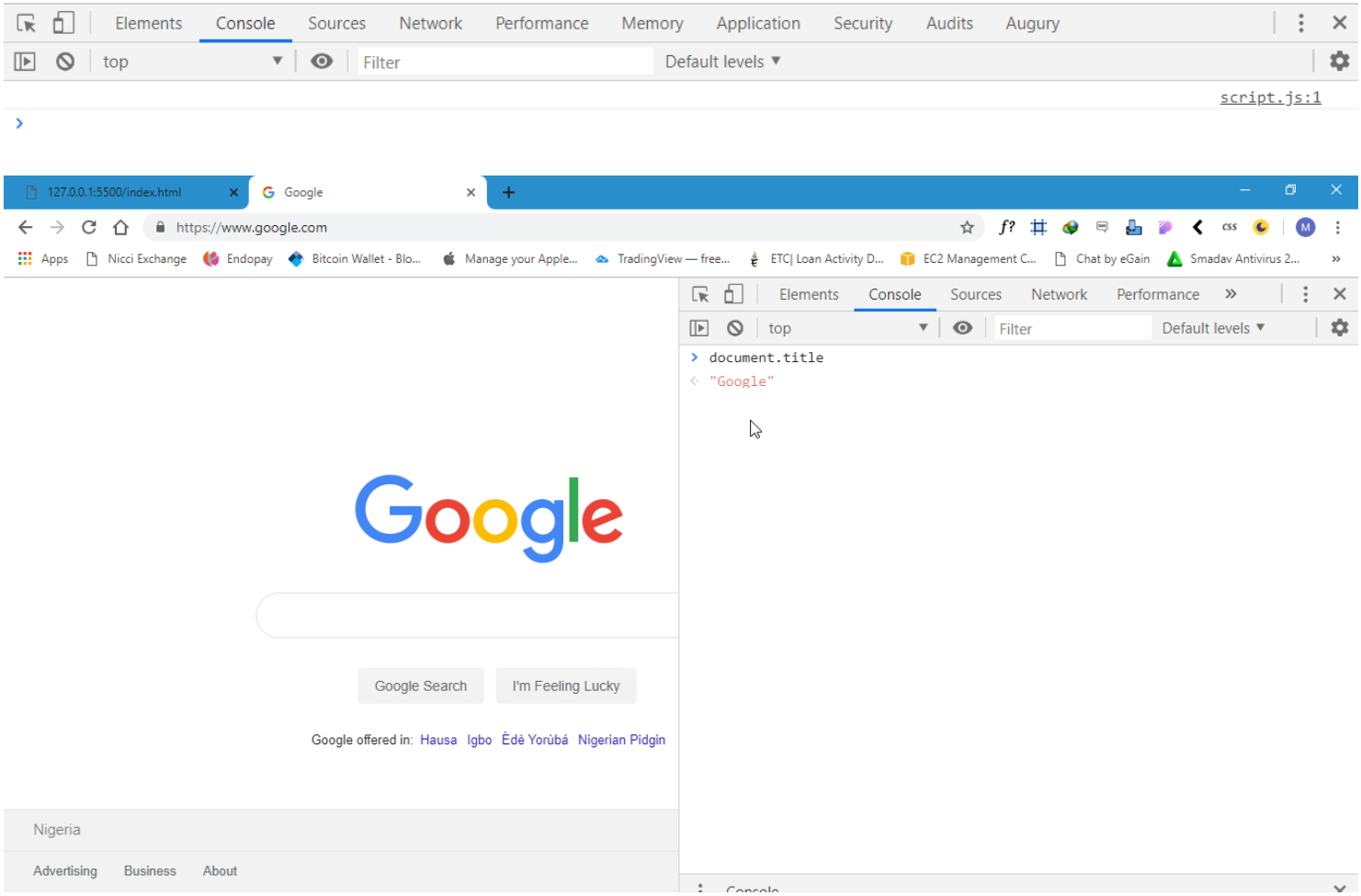
```
console.log(document.URL);
```



Script.js (document title)

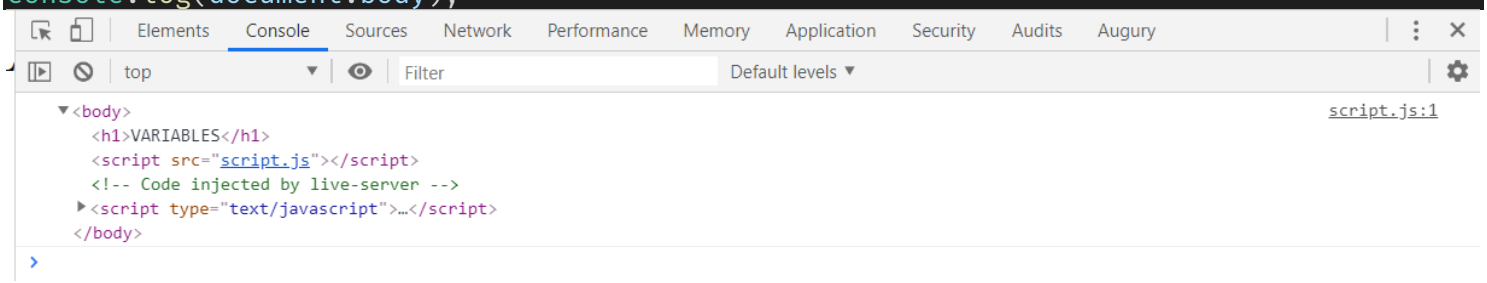
```
console.log(document.title);
```

Our document doesn't have a title currently so we get empty title in the console but we can try this on a livepage.



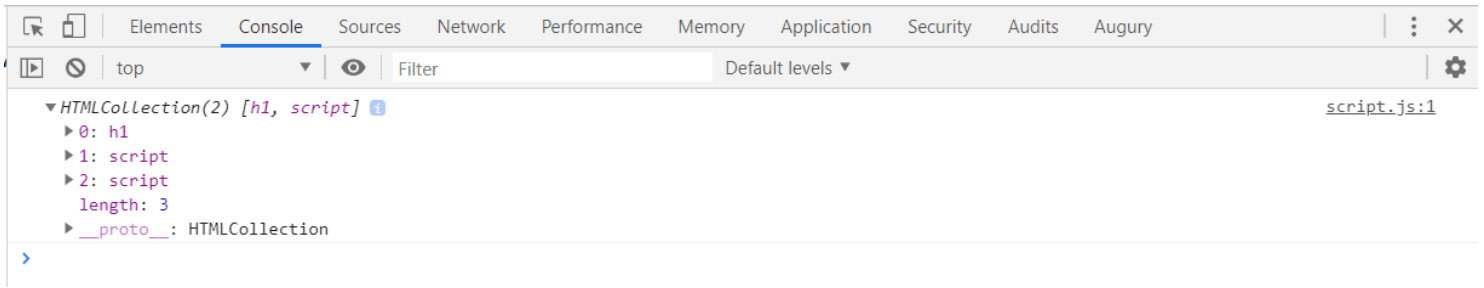
Script.js (document body)

```
console.log(document.body);
```



Script.js (document body children)

```
console.log(document.body.children);
```



This returns an array of all the html tags represented as objects.

We can access each of the tags in the html document using their index number.

Script.js

```
console.log(document.body.children[0]);
```

```
<h1>Heading</h1>
```

This returns the first tag in the body.

Script.js

```
console.log(document.body.children[1]);
```

```
<h2>Sub-heading</h2>
```

We can also log their content as plain text

Script.js

```
console.log(document.body.children[1].textContent);
```

```
Sub-heading
```

You cannot only access the content; you can as well change them to something else

Script.js (changing the h2 element)

```
document.body.children[1].textContent = 'Something else!';
```

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Heading</h1>
    <h2>Sub-heading</h2>
    <script src="script.js"></script>
  </body>
</html>
```

```
document.body.children[1].textContent = 'Something else!';
```

Heading

Something else!

Adding css styles to html element.

Script.js

```
document.body.children[1].style.backgroundColor = 'red';
```

Heading

Sub-heading

TRAVERSING THE DOM: - There are more convenient ways of interacting with the DOM but before doing that, we need to know how we can navigate between elements in the DOM.

Let's consider an unordered list below

```html
index.html
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
   <ul>
     <li> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
     <li> <a href="#">Link3</a> </li>
     <li> <a href="#">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

```js
script.js
console.log(document.body.children[0].children[0]);
```

The first child (children[0]) here is the <ul> while the first child inside this children is the Link1. We can navigate to this using the link below.

Script.js

```js
console.log(document.body.children[0].children[0]);
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels

```
▼<li>                                                    script.js:1
   <a href="#">Link1</a>
 </li>
>
```

There are other ways of doing this without using the body.chidren[0]

Script.js (firstChild)

```js
console.log(document.body.firstChild);
```

Elements Console Sources Network Performance Memory Application Security Audits

top Filter Default levels

```
▼#text                                                   script.js:1
   assignedSlot: null
   baseURI: "http://127.0.0.1:5500/index.html"
 ▶ childNodes: NodeList []
   data: "↵    "
   firstChild: null
   isConnected: true
   lastChild: null
   length: 4
 ▶ nextElementSibling: ul
 ▶ nextSibling: ul
   nodeName: "#text"
   nodeType: 3
   nodeValue: "↵    "
 ▶ ownerDocument: document
 ▶ parentElement: body
 ▶ parentNode: body
   previousElementSibling: null
   previousSibling: null
   textContent: "↵    "
   wholeText: "↵    "
 ▶ __proto__: Text
>
```

This will output the first child in the body which is the space/whitespace before the <ul>. We might be wondering that we need the first tag not the whitespace. To access the first tag or first html element then we use 'firstElementChild'

Script.js (firstElementChild)

```
console.log(document.body.firstElementChild);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
▼<ul>                                                                    script.js:1
  ▶<li>…</li>
  ▶<li>…</li>
  ▶<li>…</li>
  ▶<li>…</li>
  </ul>
>
```

Now this returns the first html element inside the body tags which is the <ul>.

lastElementChild: this will return the last element in the html

script.js

```
console.log(document.body.lastElementChild);
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits
top  Filter  Default levels
<script src="script.js"></script>                                        script.js:1
>
```

This is correct because it is the last html element before the closing body tag.

```
index.html ×
1  <!DOCTYPE html>
2  <html>
3
4    <head>
5      <link rel="stylesheet" href="style.css">
6    </head>
7
8    <body>
9     <ul>
10      <li> <a href="#">Link1</a> </li>
11      <li> <a href="#">Link2</a> </li>
12      <li> <a href="#">Link3</a> </li>
13      <li> <a href="#">Link4</a> </li>
14    </ul>
15    <script src="script.js"></script>
16   </body>
17 </html>
```

Script.js (nested first element child)

```
console.log(document.body.firstElementChild.firstElementChild);
```

```
▼<li>
  <a href="#">Link1</a>
</li>
>
```

script.js:1

We can get the second link (Link 2) which is the second nested element.

Script.js

```
console.log(document.body.firstElementChild.firstElementChild.nextElementSibling);
```

```
▼<li>
  <a href="#">Link2</a>
</li>
>
```

script.js:1

This is how you can tranverse and navigate between elements in the DOM.

parentElement: - This will return the <ul> because it is the parent inside which the links are contained.

Script.js

```
console.log(document.body.firstElementChild.firstElementChild.nextElementSibling.nextElementSibling.parentElement);
```

```
▼<ul>
  ▶<li>…</li>
  ▶<li>…</li>
  ▶<li>…</li>
  ▶<li>…</li>
</ul>
>
```

script.js:1

SELECTING ELEMENTS: - As we have mentioned before, this way of navigating our elements is a bit complex because we always have a long JS statement and you need to go deep inside the html elements to count how many you need to get to the correct element. There are better ways of doing it but we will start with the mediocre way first before moving to the optimal way.

MEDIOCRE WAY

getElementByTagName: - You can use this syntax to get element by their tag name.

script.js

```
console.log(document.getElementsByTagName('li'));
```

- Link1
- li 304 × 18
- Link4

```
▼HTMLCollection(4) [li, li, li, li]
  ▶0: li
  ▶1: li
  ▶2: li
  ▶3: li
   length: 4
  ▶__proto__: HTMLCollection
>
```

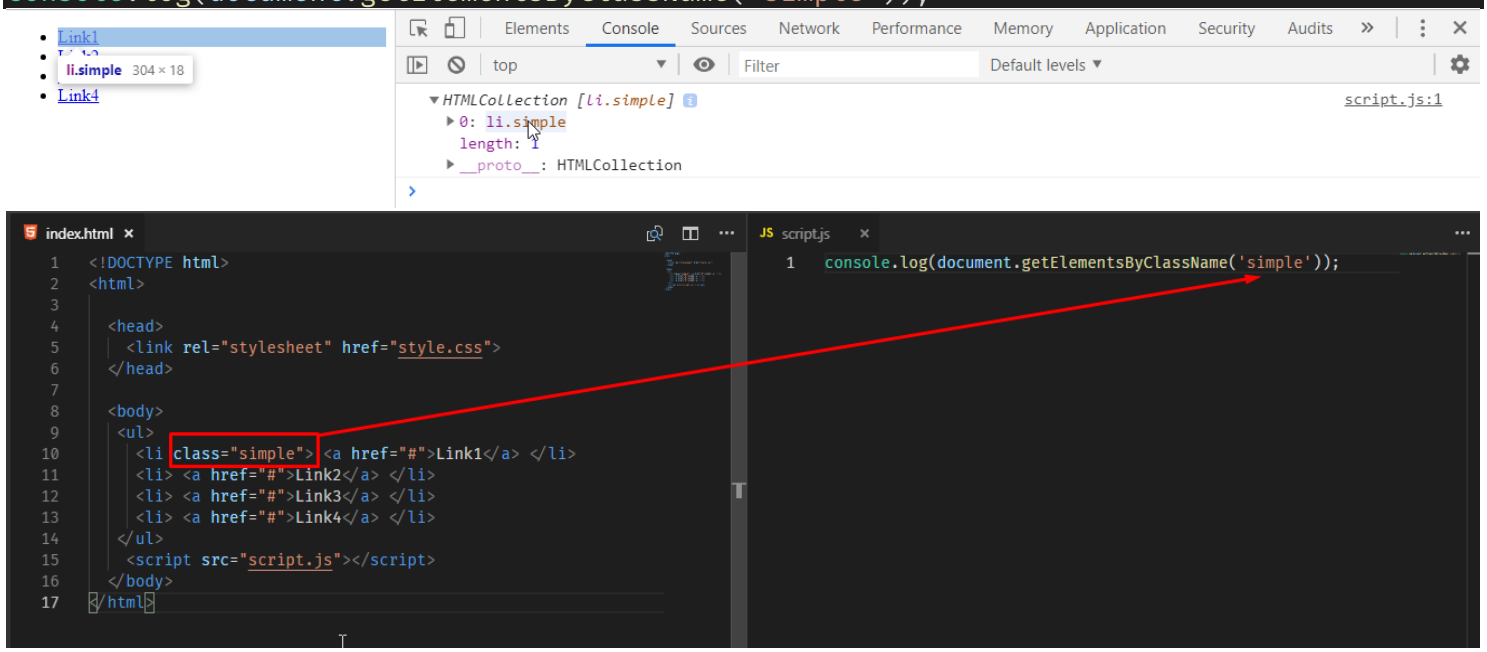
script.js:1

This results to an array of all the links.

We also have the getElementbyClassName which will allow us to get elements by specifying their class name.

Script.js

```
console.log(document.getElementsByClassName('simple'));
```

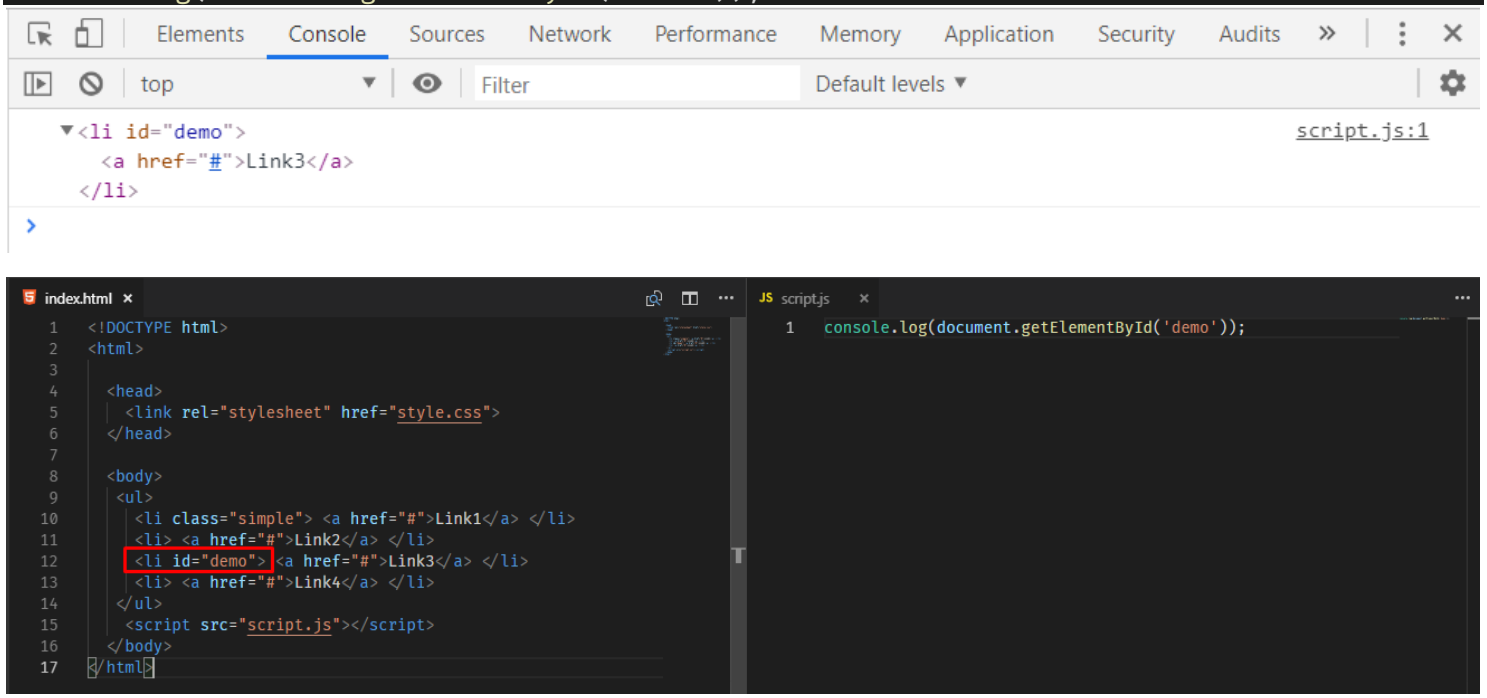


You also have other way of using element

- **getElementById: Useful if you assign id to your elements.**
- **getElementByName: Useful if you assign names to your elements.**

Script.js (getElementById)

```
console.log(document.getElementById('demo'));
```



So this are better ways of accessing your html elements.

SELECTING ELEMENTS WITH QUERY SELECTOR: - We have checked the mediocre way of selecting your html elements, now we can check the best way using JS query selector.

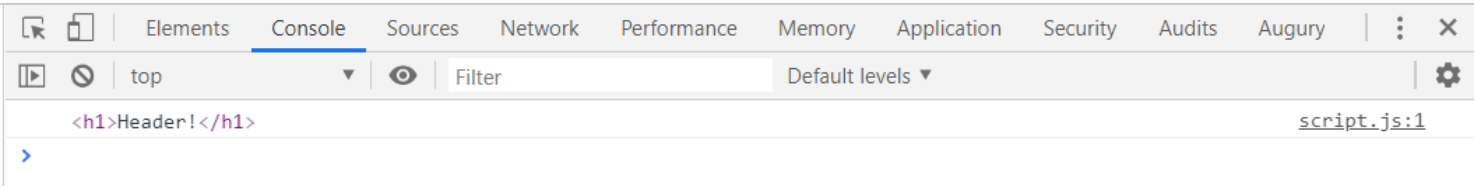
How do you use the query selector to select elements? You use the query selector to select elements as you will use css to select elements using their id, class or tag name.

Index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
   <ul>
     <li class="simple"> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
     <li id="demo"> <a href="#">Link3</a> </li>
     <li> <a href="#">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

Script.js (selection by tag)

```
console.log(document.querySelector('h1'));
```



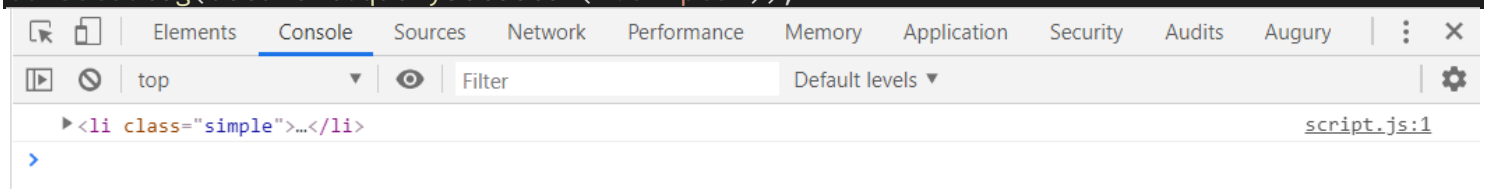
Selection by class

Index.html

```
<!DOCTYPE html>

<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
   <ul>
     <li class="simple"> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
     <li id="demo"> <a href="#">Link3</a> </li>
     <li> <a href="#">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

Script.js

```
console.log(document.querySelector('.simple'));
```



querySelector will only return the first element that matches the query.

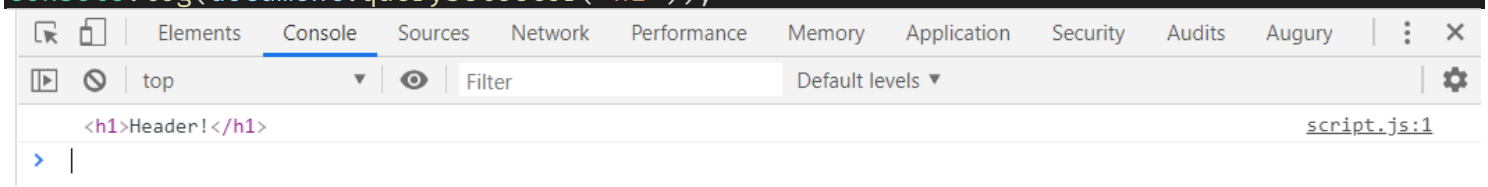
Index.html (this contains two h1 tags)

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
    <h1>Another Header!</h1>
   <ul>
     <li class="simple"> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
```

```
    <li id="demo"> <a href="#">Link3</a> </li>
    <li> <a href="#">Link4</a> </li>
  </ul>
   <script src="script.js"></script>
 </body>
</html>
```

Script.js

```
console.log(document.querySelector('h1'));
```

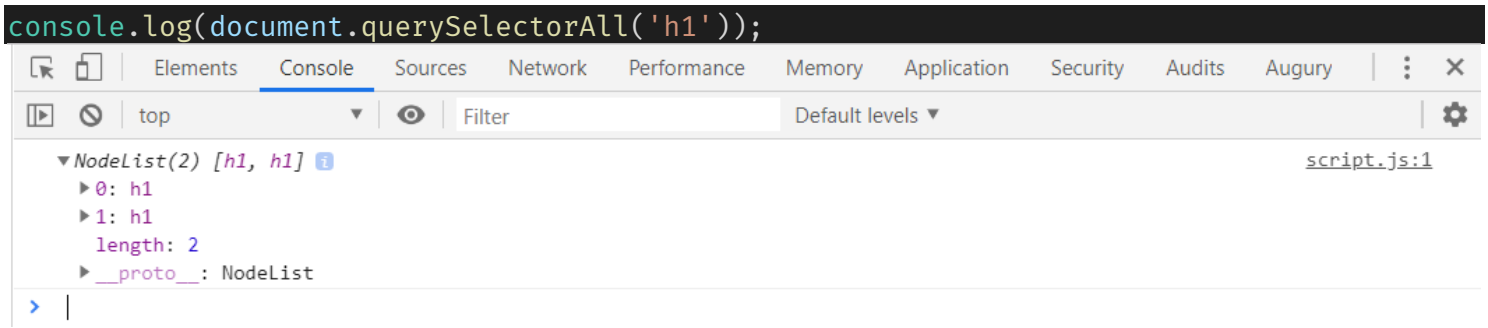


As we can see, queryselector returns the first match only.

To get all the elements that matches this query as an array, we will need to use 'querySelectorAll'

Script.js

```
console.log(document.querySelectorAll('h1'));
```



Now we can see the two h1 tag. This works for .classname and #id also

Index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
    <h1>Another Header!</h1>
   <ul>
    <li class="simple"> <a href="#">Link1</a> </li>
    <li> <a href="#">Link2</a> </li>
    <li id="demo"> <a href="#">Link3</a> </li>
    <li class="simple"> <a href="#">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

Script.js

```
console.log(document.querySelectorAll('.simple'));
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top  Filter  Default levels
▼NodeList(2) [li.simple, li.simple]                                    script.js:1
  ▶0: li.simple
  ▶1: li.simple
   length: 2
  ▶__proto__: NodeList
>
```

Query selector by id

Index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
    <h1>Another Header!</h1>
   <ul>
     <li class="simple"> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
     <li id="demo"> <a href="#">Link3</a> </li>
     <li class="simple"> <a href="#">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

Script.j

```
console.log(document.querySelectorAll('#demo'));
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top  Filter  Default levels
▶NodeList [li#demo]                                                    script.js:1
>
```

To learn more, learn about how css selector works and check the cheat sheet attached to the episode.

Query selector styles

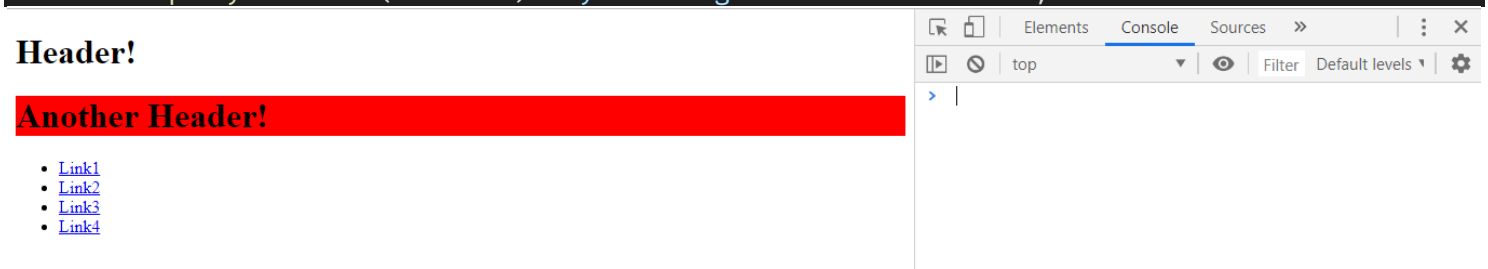
Index.html

```
<!DOCTYPE html>
<html>

  <head>
```

```
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
    <h1 id="demo">Another Header!</h1>
   <ul>
     <li class="simple"> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
     <li> <a href="#">Link3</a> </li>
     <li class="simple"> <a href="#">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

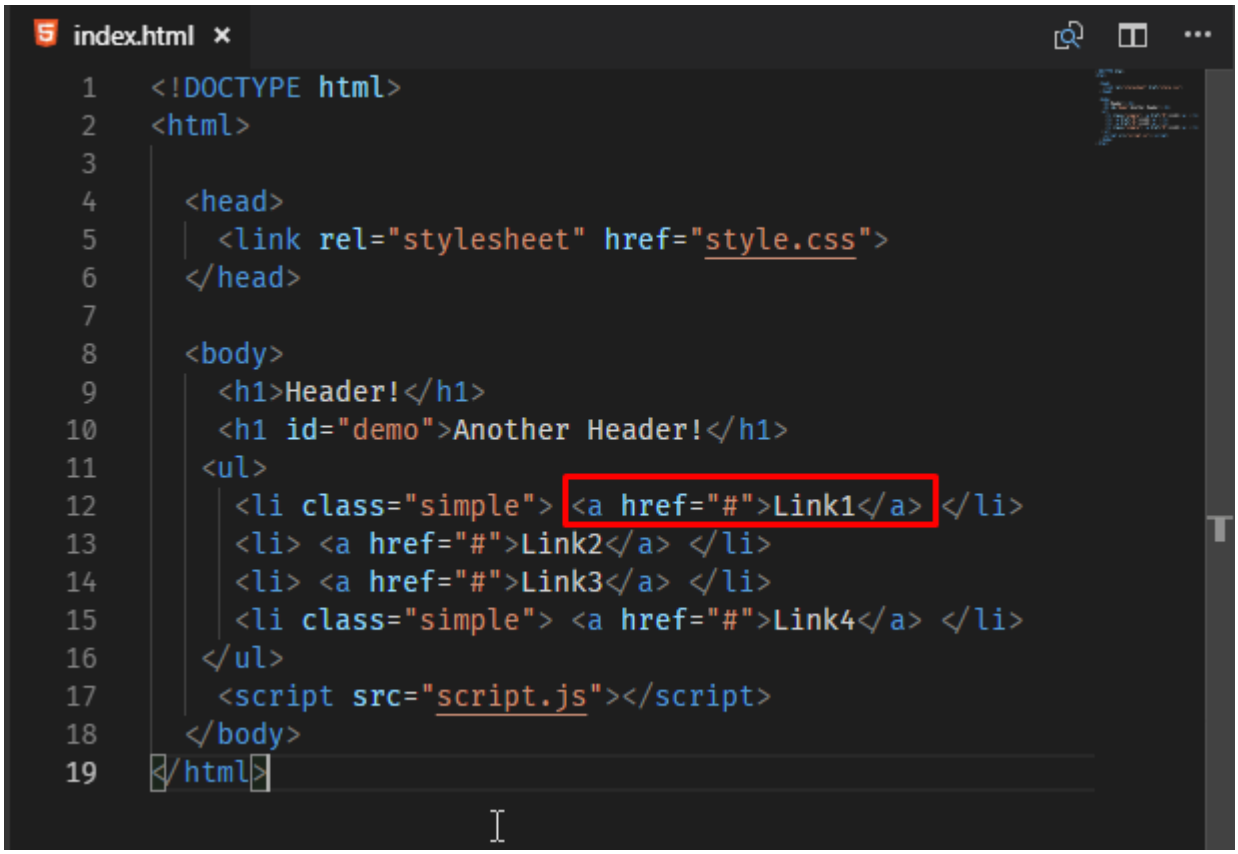
Script.js

```
document.querySelector('#demo').style.backgroundColor = 'red';
```



SELECTING ELEMENTS – EXERCISE: - It's important to practise this, so let us play around with this a little bit more.

- **Let us select the first anchor tag ahref in our document and change the background to red.**



Script.js

```
document.querySelector('a').style.backgroundColor = 'red';
//querySelector will select only the first element
//('a') represent the first anchor tag
```

Header!

Another Header!

- Link1
- Link2
- Link3
- Link4

How can we change the background color of the 4th link to red? There are two ways to do that.

Index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
    <h1 id="demo">Another Header!</h1>
   <ul>
     <li class="simple"> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
     <li> <a href="#">Link3</a> </li>
     <li> <a href="#" class="link4">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

Here we assign a special class to the href to differentiate it.

Script.js

```
document.querySelector('.link4').style.backgroundColor = 'red';
```

Header!

Another Header!

- Link1
- Link2
- Link3
- Link4

OR

We can use another method just incase the 4th link is having the same class name with another link.

Index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
    <h1 id="demo">Another Header!</h1>
   <ul>
     <li class="simple"> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
     <li> <a href="#">Link3</a> </li>
     <li> <a href="#" class="simple">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

Script.js

```
document.querySelectorAll('.simple')[1].style.backgroundColor = 'red';
//here we are using the index number to select the element she want.
```

Header!

Another Header!

- Link1
- Link2
- Link3
- Link4

What if we want to change link 3 text to Hello.

Index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
    <h1 id="demo">Another Header!</h1>
   <ul>
     <li class="simple"> <a href="#">Link1</a> </li>
     <li> <a href="#">Link2</a> </li>
     <li> <a href="#" class="link3">Link3</a> </li>
     <li> <a href="#" class="simple">Link4</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
```

```
</html>
```

Script.js

```
document.querySelector('.link3').textContent = 'Hello!';
```

Header!

Another Header!

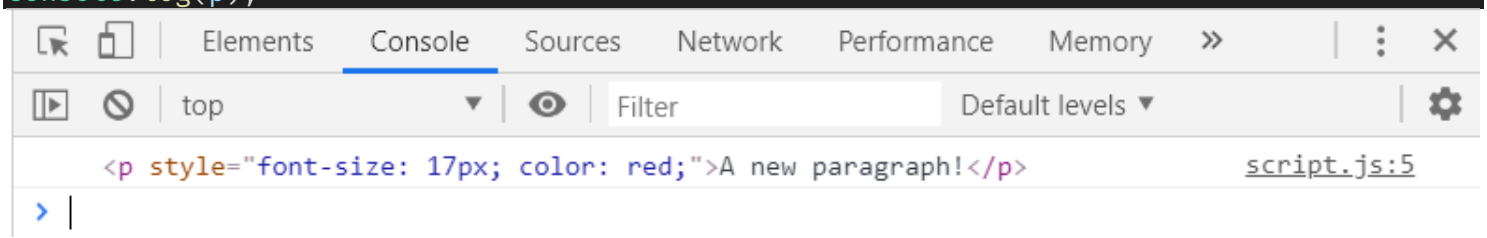
- Link1
- Link2
- Hello!
- Link4

CREATING AND INSERTING ELEMENTS: - We have seen how to get element and change them but sometimes you might want to create or destroy them.

Creating elements: let us try to create a paragraph

Script.js

```
var p = document.createElement('P');
p.textContent = 'A new paragraph!';
p.style.fontSize = '17px';
p.style.color = 'red';
console.log(p);
```



Now let us make it appear in the DOM and below the first link.

Script.js

```
var p = document.createElement('P');
p.textContent = 'A new paragraph!';
p.style.fontSize = '17px';
p.style.color = 'red';

var a = document.querySelector('a'); //this calls on the element below which we want to add the paragraph

a.appendChild(p); //this add the paragrap inside the element in this case the paragraph will be inside the Link 1
```

Header!

Another Header!

- Link1

  A new paragraph!

- Link2
- Link3
- Link4

As we can see, the paragraph was added insid the link and not below it, this is what appendChild does. We need to add it below it.

Script.js

```
var p = document.createElement('P');
p.textContent = 'A new paragraph!';
p.style.fontSize = '17px';
p.style.color = 'red';

var li = document.querySelector('li'); //this will call on the first list

li.appendChild(p); //this add the paragraph below the first list and not inside the link
```

Header!

Another Header!

- Link1

  A new paragraph!

- Link2
- Link3
- Link4

What if you want to put it before the first link

Script.js

```
var p = document.createElement('P');
p.textContent = 'A new paragraph!';
p.style.fontSize = '17px';
p.style.color = 'red';

var li = document.querySelector('li');
var a = li.firstElementChild;

li.insertBefore(p, a);
```

Header!

Another Header!

- A new paragraph!

  Link1
- Link2
- Link3
- Link4

DELETING ELEMENTS: - Removing a child is pretty straight-forward though not that convinent since it is not all browsers that supports the new method. So let us first examine the first method that works across all browsers.

We have 3 links in our app, let's see how to get rid of the second link "Link2".

Index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>Header!</h1>
    <h1 id="demo">Another Header!</h1>
   <ul>
     <li> <a href="#">Link1</a> </li>
     <li class="simple"> <a href="#">Link2</a> </li>
     <li> <a href="#" class="simple">Link3</a> </li>
   </ul>
    <script src="script.js"></script>
  </body>
</html>
```

Script.js

```
// var a = document.querySelector('.simple');
var a = document.querySelectorAll('li')[1];
a.parentElement.removeChild(a);
//the code in the comment will also work
```

Header!

Another Header!

- Link1
- Link3

Image 11: Link2 removed

A faster script can be seen below.

Script.js

```
// var a = document.querySelector('.simple');
var a = document.querySelectorAll('li')[1];
a.remove();
```

Header!

Another Header!

- Link1
- Link3

This works but not in all browsers so use with caution.

ELEMENTS & NODES: - The DOM consists of nodes, it is madeup of nodes and some nodes happens to be elements as well e.g. our ul list, h1 tag etc. so in the script to remove child, u can also use parentNode instead of parentElement.

Script.js

```
// var a = document.querySelector('.simple');
var a = document.querySelectorAll('li')[1];
a.parentNode.removeChild(a);
```

Header!

Another Header!

- Link1
- Link3

The difference between Node and elements is that, the node node can also be some things in the dom which are not elements.

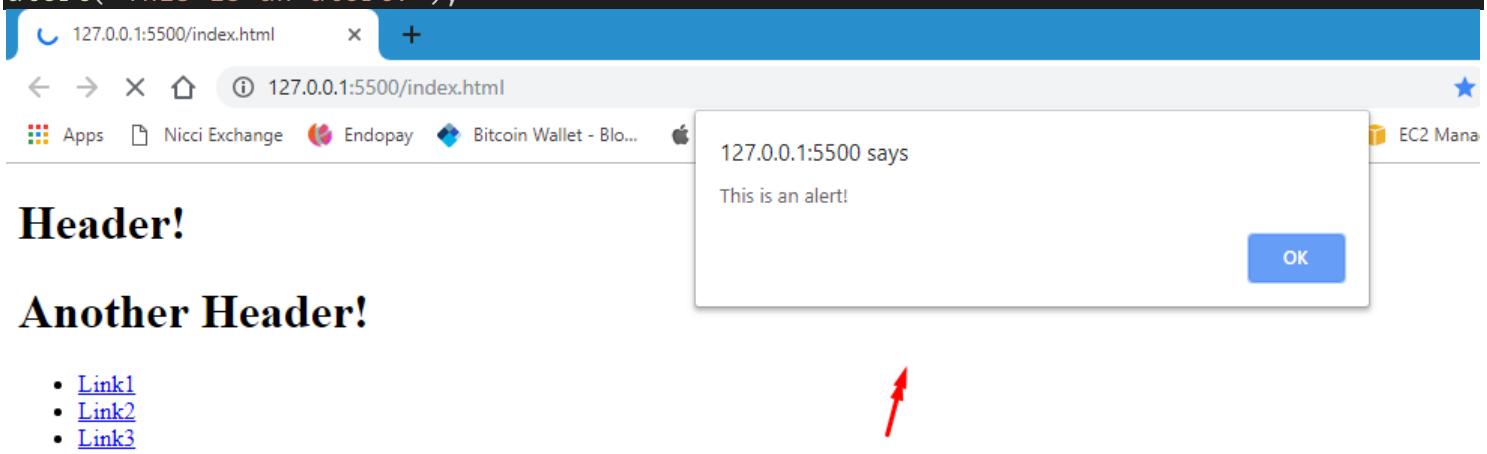
DOM INTERACTION SUMMARY: - So we have learnt a lot about selecting elements, creating them, deleting them etc. this are things you do with your applications and this is what creates a good user experience and makes your page reactive with the possibility of changing things without having to reload it or get a new page from the server. You can check the cheat sheet attached to this episode for more information and other areas that was not touched.

DIALOGS: - We want to learn about some built in dialogs the windows object offers. Dialogs are basically the pop ups the user sees.

Alert( ) : - this is a prominent one that is mostly use for debugging instead of console log.

Script.js

```
alert('This is an alert!');
```

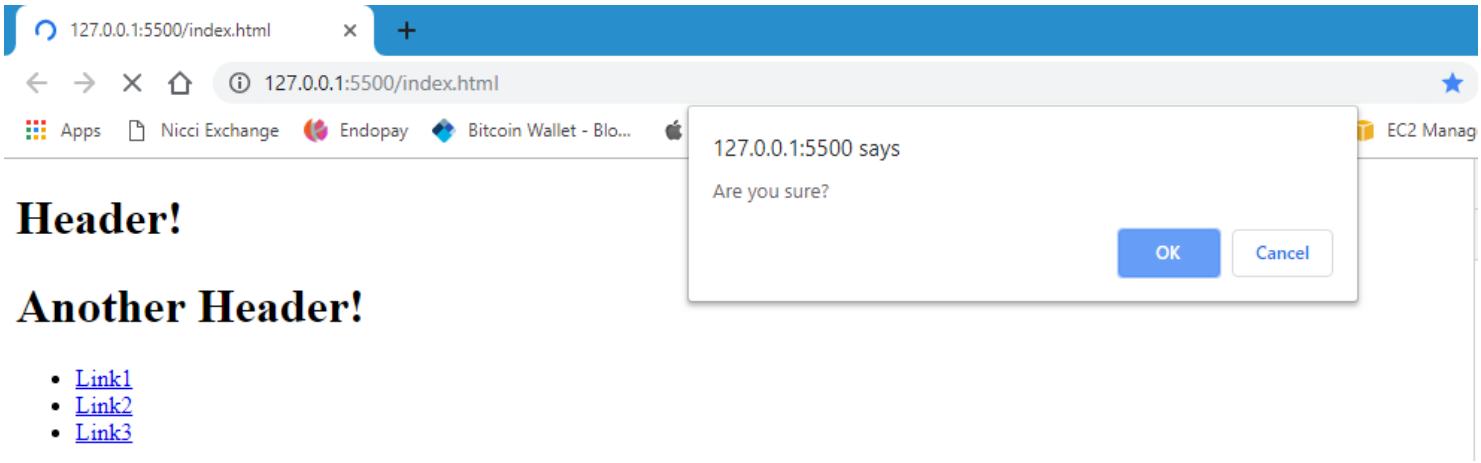


That is how to use an alert which is rarely used instead of better ways to present messages like modals and so on.

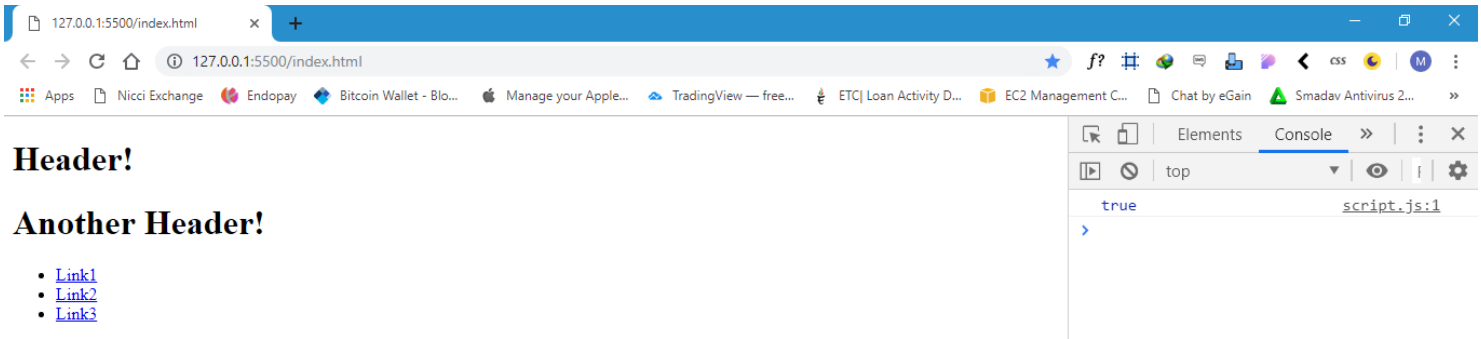
Confirm( ): - this will ask the user to select a choice – ok or cancel. You can as well log this because it will return what the user chooses.

Script.js

```
console.log(confirm('Are you sure?'));
```



if the user clicks ok, we get 'true' in the console and vice versa

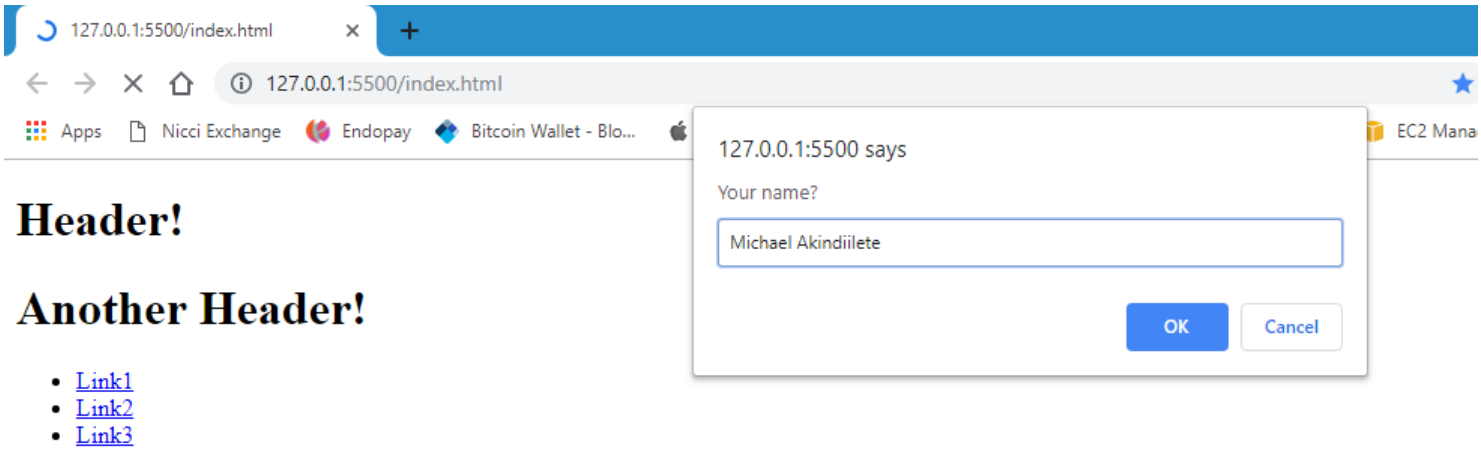


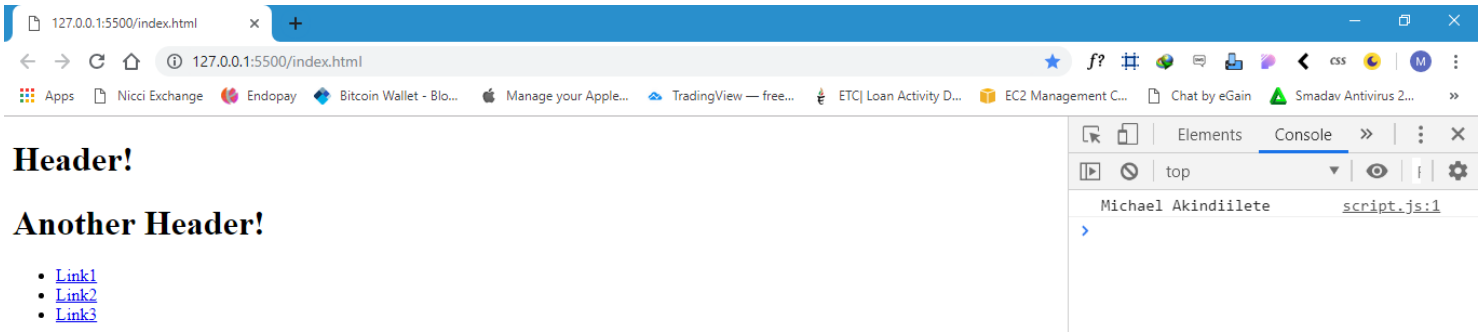
Also use with caution because there are more friendly ways of presenting dialogs nowadays.

Prompt( ): - you can use this to collect input which can then be stored or used as a variable.

Script.js

```
console.log(prompt('Your name?'));
```





Dialog summary: - Dialogs are like 10year old way of getting pop up done, there are better ways now like JS modals and bootstrap modals.

EVENTS

INTRODUCTION: - Until now, we have covered a lot of pieces that make up an awesome JS applications, we have learnt a lot but one key thing is still missing; we are not able to react to user interactions, we are not able to listen to events and that is important and therefore this module is all about events.

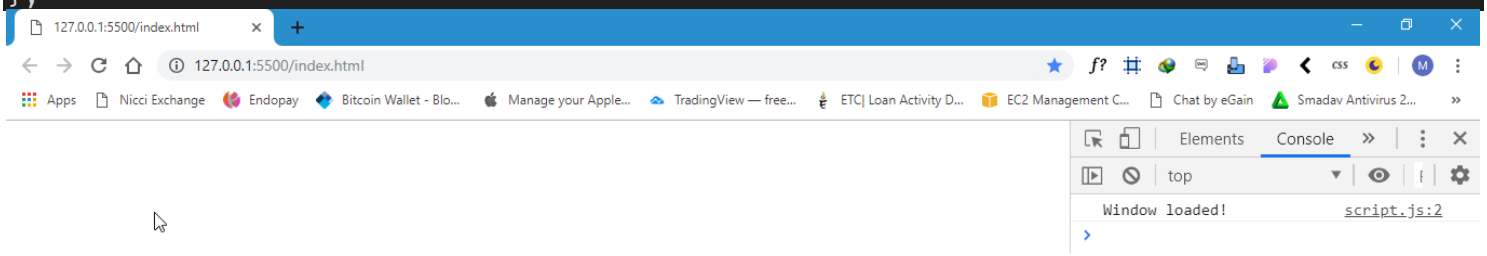
THE EVENT OBJECT: - How do JS event work? On the mozilla javascript resource for events *https://developer.mozilla.org/en-US/docs/Web/API/Event* **. So how can you listen or handle events?**

EVENTS HANDLERS: -

Window.onload: - this is an event handler which executes a function when the window is loaded

Script.js

```
window.onload = function () {
    console.log('Window loaded!')
};
```



We get window loaded in the console as soon as we load the window.

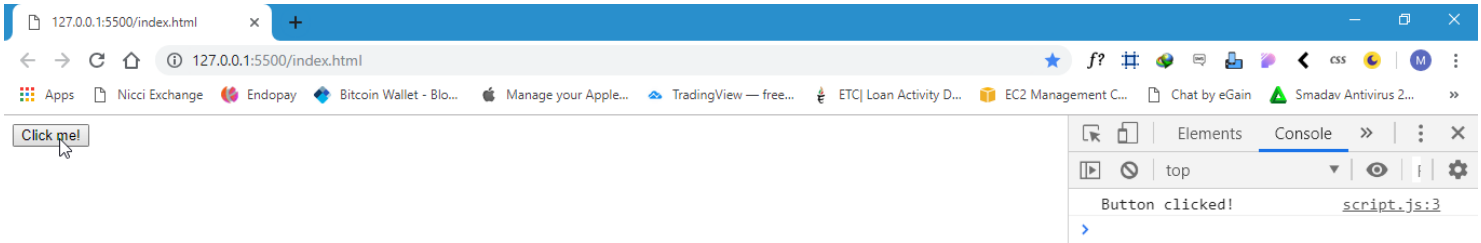
We can also handle other events like button clicks using the .onclick handler.

Index.html

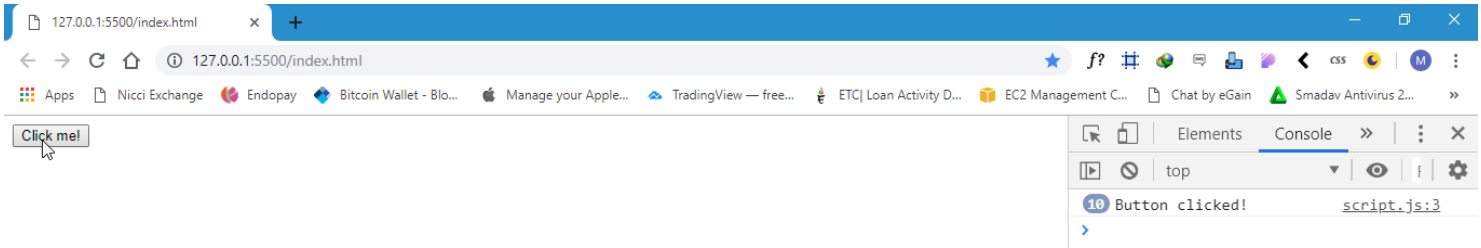
```
<!DOCTYPE html>
<html>

  <head>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <button>Click me!</button>
    <script src="script.js"></script>
  </body>
</html>
```

Script.js

```
var btn = document.querySelector('button'); //we first select the element using
queryselector
btn.onclick = function() {
    console.log('Button clicked!');
}; //we then attached onclick handler to our btn variable and assign it to a function
```



At the click of the button, we get the function executed which is to print a message to the console and the function will be executed everytime the button is clicked.



Button clicked 10 times.

How to know which functions are available on a specific html?

You can log the element to the console to view all the properties and you will identify the events handlers starting with the on at the beginning an then the name of the event.

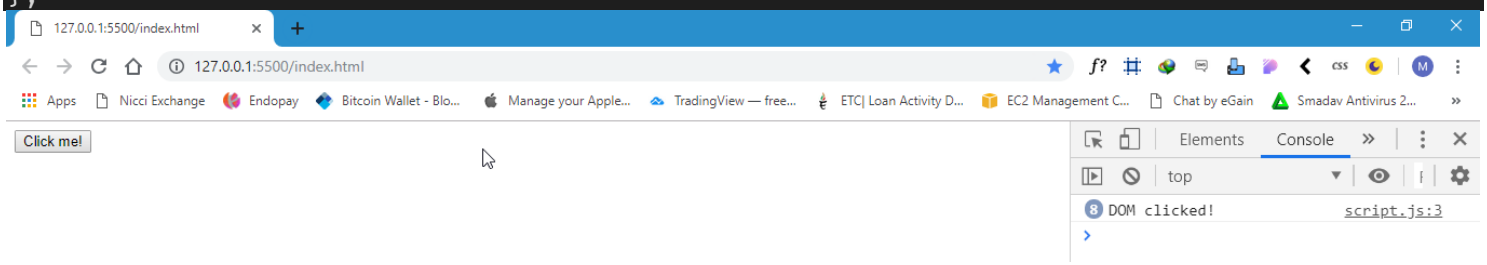
You can also google for javascript and then element name and then events and you should find list of events you can use.

You can not only listen to events on button and links alone, you can add this to any html element even if it's not clickable normally.

You can set an event to the document itself and no matter where you click in the DOM, the function will be fired.

Script.js

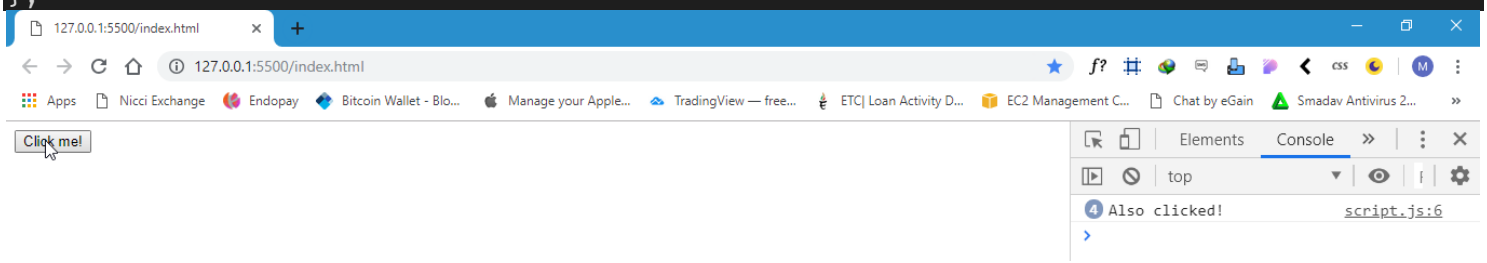
```
var btn = document.querySelector('button');
document.onclick = function() {
    console.log('DOM clicked!');
};
```



There is one major disadvantage of this approach: - If you have multiple functions you want to call, this appraoch will always overwrite the first function with the last so we need a better alternative where we can listen to miltiple clicks and execute multiple functions

Script.js

```
var btn = document.querySelector('button');
btn.onclick = function() {
    console.log('Clicked!');
};
btn.onclick = function() {
    console.log('Also clicked!');
};
```



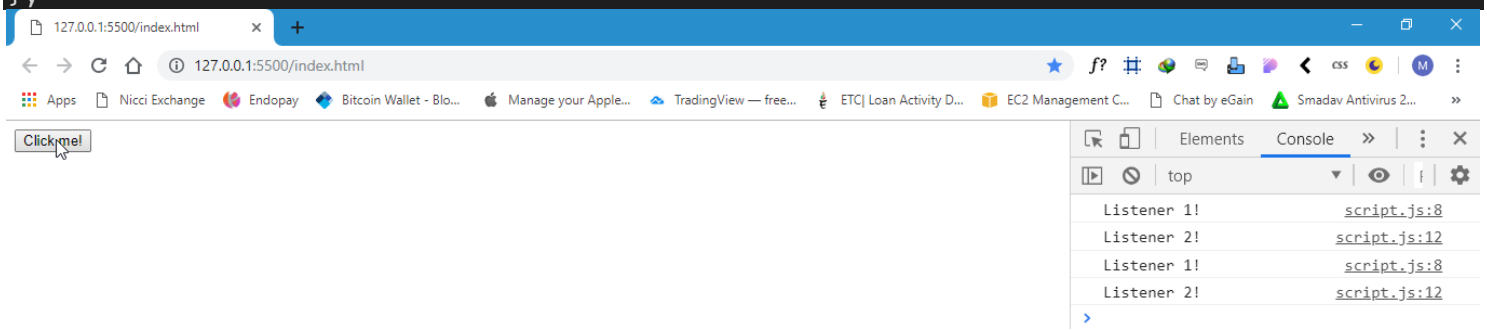
here we can see that the first function clicked was overwritten with Also clicked.

EVENT LISTENERS: - This is the alternative we were talking about "Event listeners". This might sounds similar to "event handlers" but it is a different thing in JS.

Script.js

```
var btn = document.querySelector('button');

btn.addEventListener('click', listener1); //the event type, the function name

btn.addEventListener('click', listener2); //the event type, the function name

function listener1() {
    console.log('Listener 1!');
};
```

```
function listener2() {
    console.log('Listener 2!');
};
```



now we can see that we have the two functions getting fired.

You can also remove certain event after specific setTimeout.

Script.js

```
var btn = document.querySelector('button');

btn.addEventListener('click', listener1); //the event type, the function name

btn.addEventListener('click', listener2); //the event type, the function name

setTimeout( function (){
    btn.removeEventListener('click', listener1); //here we JS also expects the type of
event and event name as argument bcos it is possible to have multiple events type with
the same name
}, 2000);

function listener1() {
    console.log('Listener 1!');
};

function listener2() {
    console.log('Listener 2!');
};
```

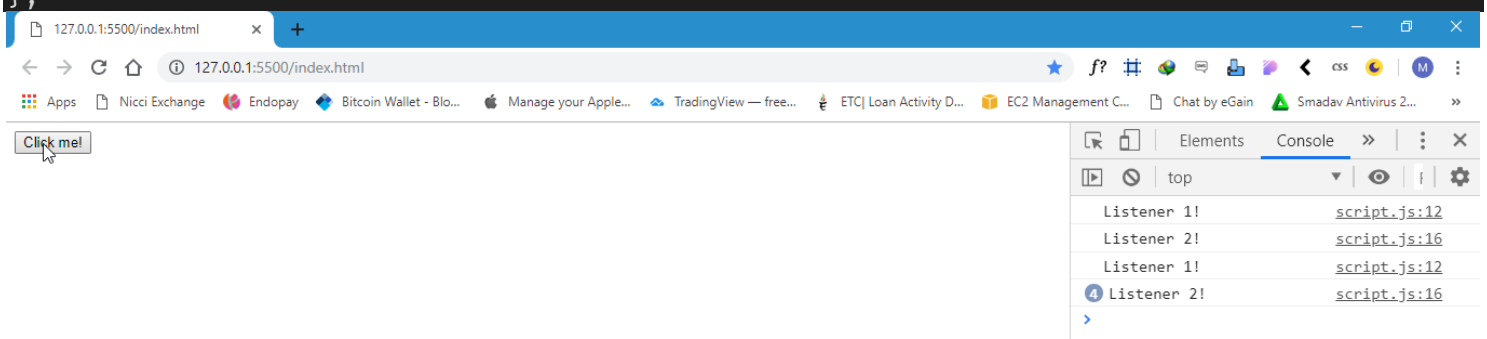


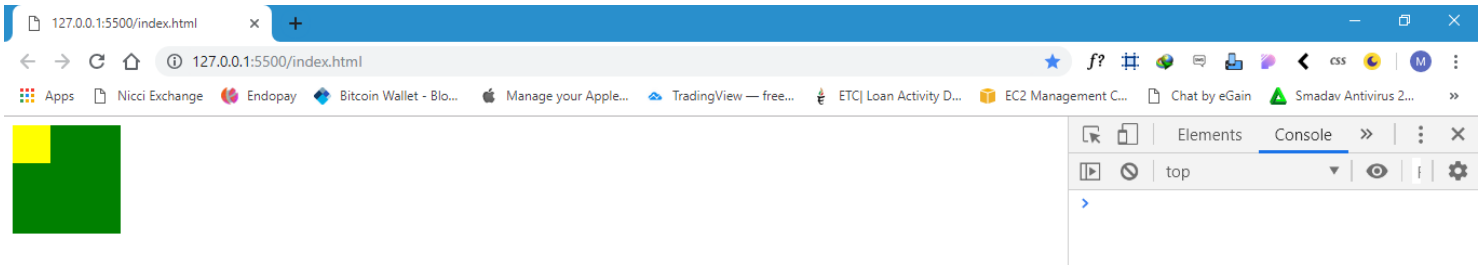
Image 12: when we run the app, we gets the two events but after 2 seconds, the first listener was removed.

EVENT BEHAVIOR: - In JS events, there are certain behaviors you need to know.

Index.html

```
<!DOCTYPE html>
<html>

  <head>
```

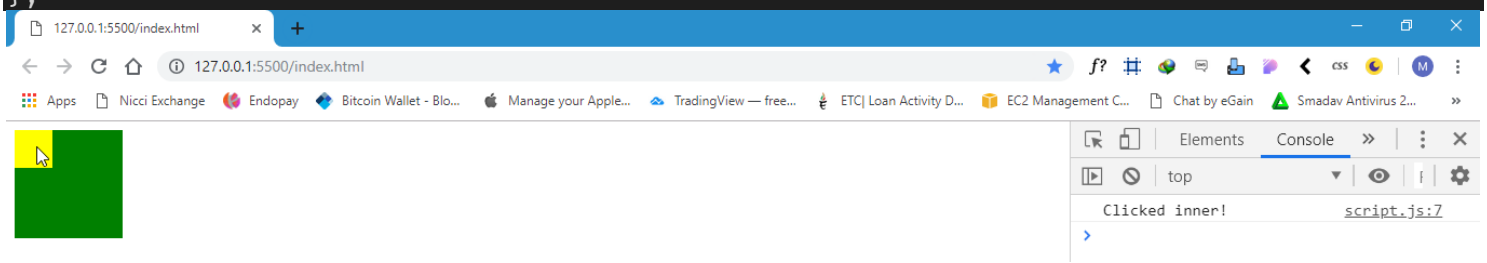
```html
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <div style="width: 100px; height: 100px; background-color:green" id="outer">
      <div style="width: 35px; height: 35px; background-color:yellow" id="inner">
      </div>
    <script src="script.js"></script>
  </body>
</html>
```



the html code gives us this shape where we have the yellow square id as inner and the green square id as outer. We can add an event to the yellow square.

Script.js

```js
var inner = document.querySelector('#inner');

inner.addEventListener('click', innerlistener); //the event type, the function name


function innerlistener() {
    console.log('Clicked inner!');
};
```

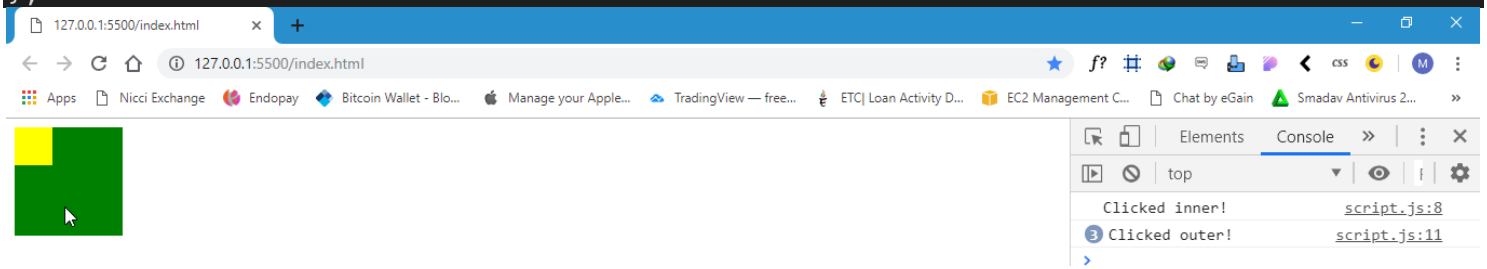


when we clicked the yellow small square, we get the message prints in the console but when we click the outer, we get nothing. We can also add event for the green as well.

Script.js

```js
var inner = document.querySelector('#inner');
var outer = document.querySelector('#outer');

inner.addEventListener('click', innerlistener);
outer.addEventListener('click', outerlistener);

function innerlistener(event) { //the event argument is optional
    console.log('Clicked inner!');
};
function outerlistener(event) {
    console.log('Clicked outer!');
```

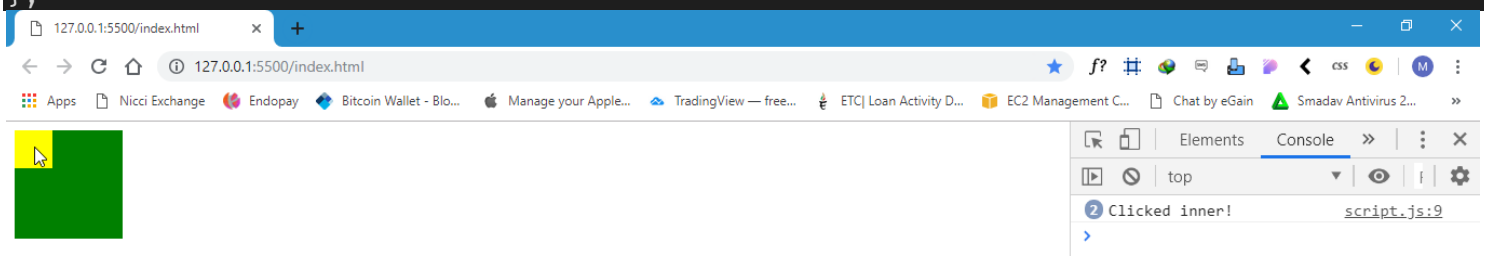
```
};
```



Now when we click the yellow, we get both the inner and outer messages printed but when we click the green, we get only the outer printed. Why? This is because the yellow is inside the green, the green stays ontop so when we click the yellow, we technically cliked the green as well because there is no way we can click the yellow without clicking the green but the green can be clicked without the yellow. Now to get each of them to be clicked separately then we need to amend the code.

Script.js

```
var inner = document.querySelector('#inner');
var outer = document.querySelector('#outer');

inner.addEventListener('click', innerlistener);
outer.addEventListener('click', outerlistener);

function innerlistener(event) { //the event argument is optional
    event.stopPropagation();//this stops the event from reaching to any other objects
other than the current object
    console.log('Clicked inner!');
};
function outerlistener(event) {
    console.log('Clicked outer!');
};
```



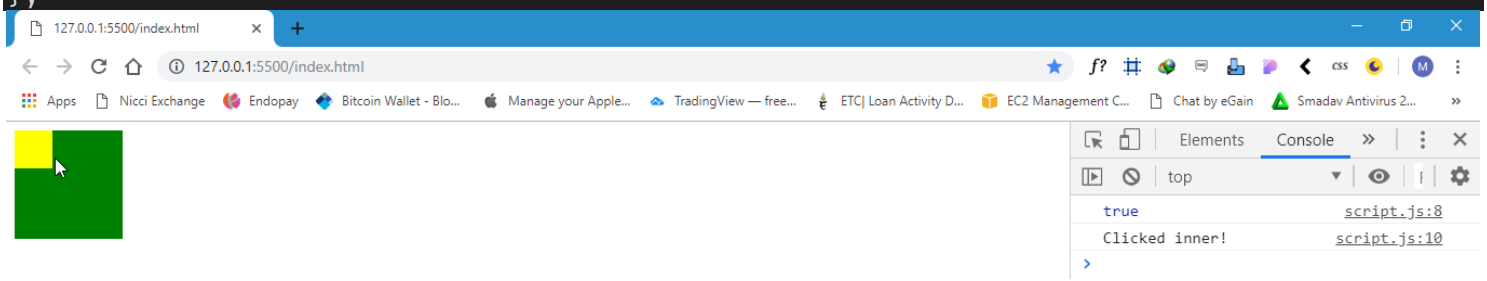
now when we click the yellow, we only get inner printed.

We can also check if the event will trigger another.

Script.js

```
var inner = document.querySelector('#inner');
var outer = document.querySelector('#outer');

inner.addEventListener('click', innerlistener);
outer.addEventListener('click', outerlistener);

function innerlistener(event) {
    console.log(event.bubbles);//this checks if the event will trigger the next
    event.stopPropagation();
    console.log('Clicked inner!');
};
function outerlistener(event) {
```

```
    console.log('Clicked outer!');
};
```



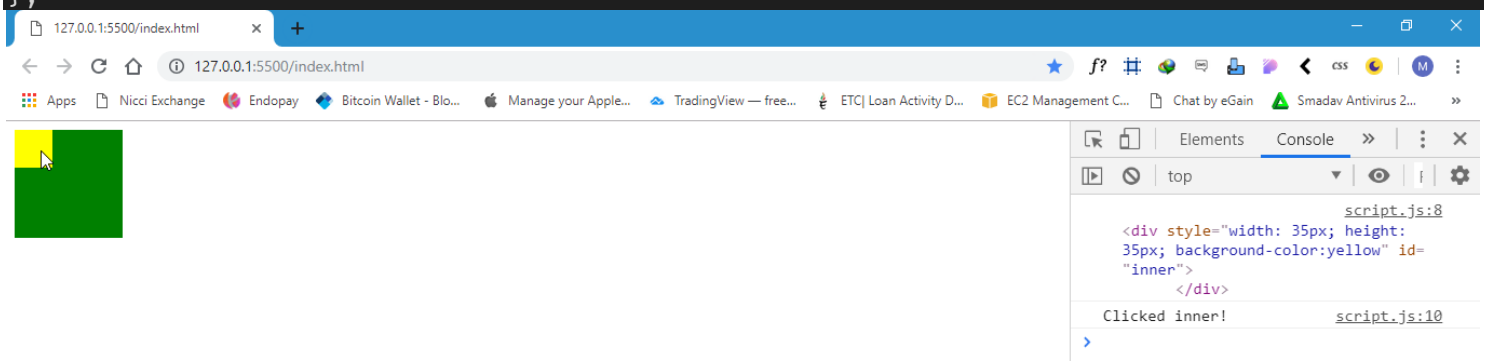
when we see true then we know we need to stop the bubbles.

EVENT OBJECT PROPERTIES: - The event object also has some other nice properties.

Event.target

Script.js

```
var inner = document.querySelector('#inner');
var outer = document.querySelector('#outer');

inner.addEventListener('click', innerlistener);
outer.addEventListener('click', outerlistener);

function innerlistener(event) {
    console.log(event.target);//this returns the dom we are targettting
    event.stopPropagation();
    console.log('Clicked inner!');
};
function outerlistener(event) {
    console.log('Clicked outer!');
};
```

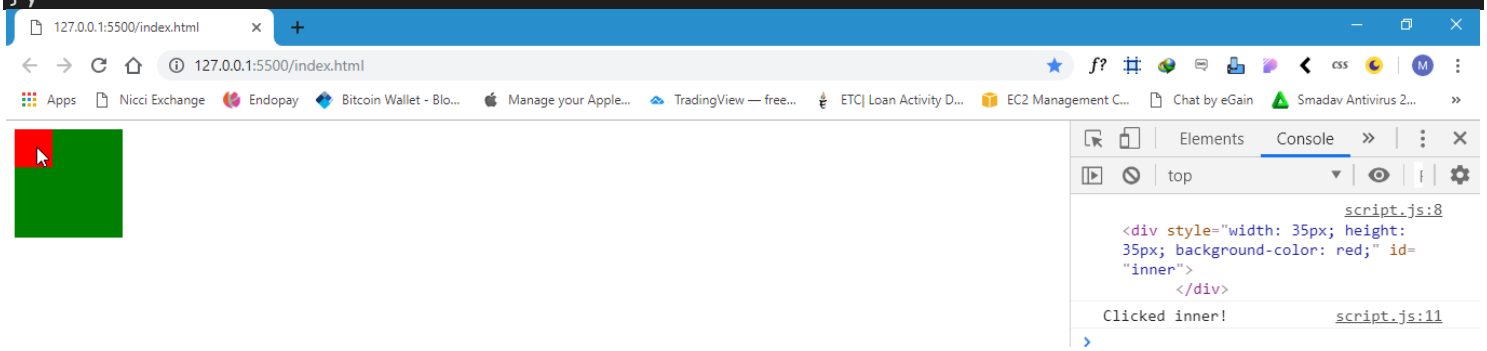


You can use this target to add some tweaks like add some css styles since the event.target is an element.

Script.js

```
var inner = document.querySelector('#inner');
var outer = document.querySelector('#outer');

inner.addEventListener('click', innerlistener);
outer.addEventListener('click', outerlistener);

function innerlistener(event) {
    console.log(event.target);//this returns the dom we are targettting
    event.target.style.backgroundColor = 'red';
```

```
    event.stopPropagation();
    console.log('Clicked inner!');
};
function outerlistener(event) {
    console.log('Clicked outer!');
};
```

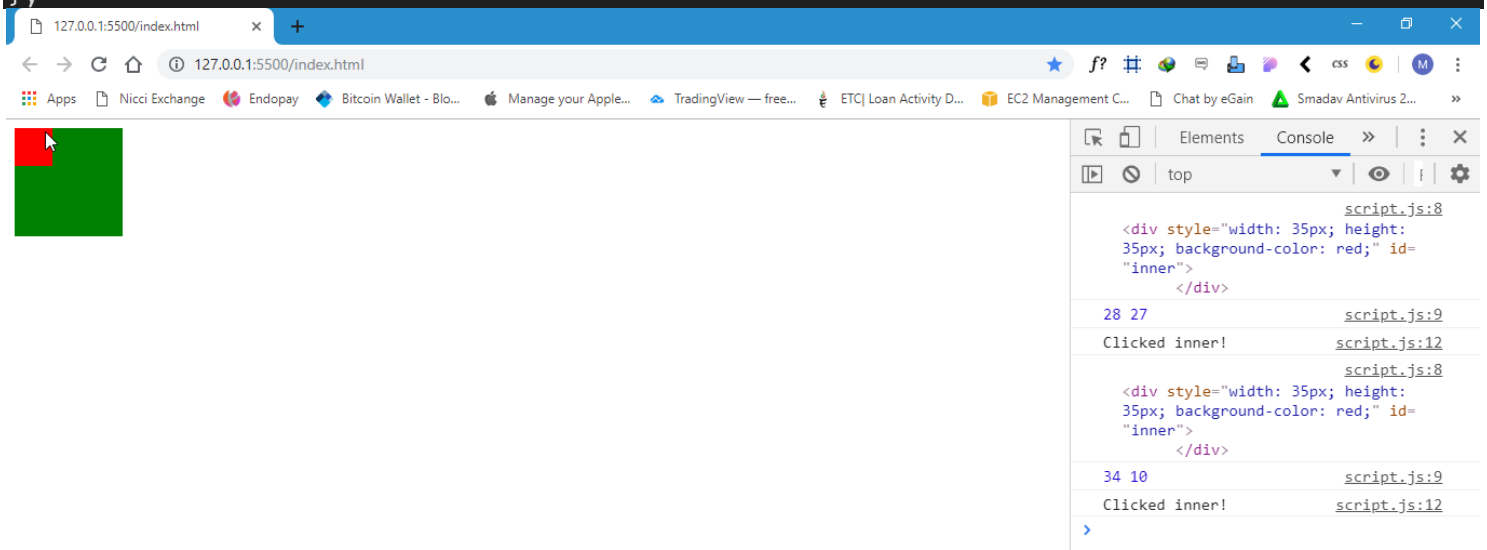


on clicking the yellow box, the event.target changed it to red.

We can also log the screen positions of where the click happens.

Script.js

```
var inner = document.querySelector('#inner');
var outer = document.querySelector('#outer');

inner.addEventListener('click', innerlistener);
outer.addEventListener('click', outerlistener);

function innerlistener(event) {
    console.log(event.target);//this returns the dom we are targettting
    console.log(event.clientX, event.clientY);//this returns the X and Y position of
where the event happens
    event.target.style.backgroundColor = 'red';
    event.stopPropagation();
    console.log('Clicked inner!');
};
function outerlistener(event) {
    console.log('Clicked outer!');
};
```

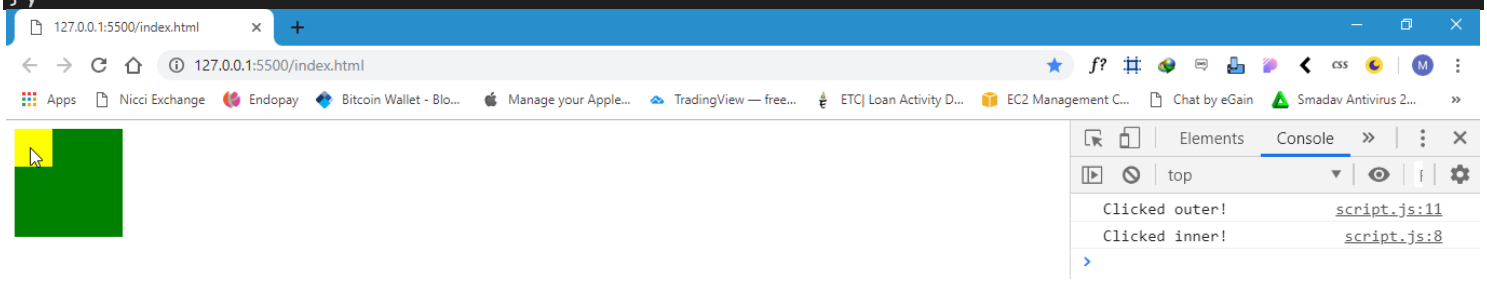


the positions will change as we click new positions. You can visit Mozilla event resource page for more.

CHANGING PROPAGATION ORDER: - We have learnt how to stop propagation in other not to fire the inner and outer events but what if we don't want to stop the propagation but we want to change the other to determine which of the two events comes first. In this case for the outer to come first before the inner.

Script.js

```
var inner = document.querySelector('#inner');
var outer = document.querySelector('#outer');

inner.addEventListener('click', innerlistener);
outer.addEventListener('click', outerlistener, true); //the true here will make the
outer fire first

function innerlistener(event) {
    console.log('Clicked inner!');
};
function outerlistener(event) {
    console.log('Clicked outer!');
};
```



often times we won't need this functionality but it is good for us to know how we can do it in case it is needed.

JAVASCRIPT AND HTTP REQUEST AJAX

INTRODUCTION: - This allows you to reach out to the web to load or post data from within JS. For example, this is what allows us to update a page with new tweets, new news etc. that you fetched through background with JS and then render it to the view or the web page. Normally you will use libraries or frameworks to handle this but it is good to learn the basics. We will also check resources where we can learn more.

SETUP AND SENDING A GET REQUEST: - For the setup of our http request, we will be using a fake online rest api from json placeholder https://jsonplaceholder.typicode.com/ **.**

Script.js

```
var http = new XMLHttpRequest();
var url = 'http://jsonplaceholder.typicode.com/posts';
var method = 'GET';

http.open(method, url); //this opens or launch the request
http.onreadystatechange = function() {
    if (http.readyState === XMLHttpRequest.DONE && http.status === 200) {
        console.log(http.responseText);
} else if (http.readyState === XMLHttpRequest.DONE && http.status !== 200) {
     console.log('Error!');
}
};
http.send();
```

```
/*
line 1 - this define a variable 'http' and assign the value 'new XMLHttpRequest' to it
line 2 - This define a variable that reached out to the fake server
line 3 - this define a variable and assign the value to the type we want 'GET' (to get
data)
line 5 - this launch the http server
line 7 - if the request is done and the status is 200 which means 'successful'
line 8 - log the response
line 9 - if the request is done but the status is not successful
line 10 - log the error
line 13 - now send the request

*/
```

Elements Console Sources Network Performance Memory Application Security Audits Augury

top Filter Default levels

script.js:8


```
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum
rerum est autem sunt rem eveniet architecto"
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor beatae ea dolores neque\nfugiat blanditiis voluptate
porro vel nihil molestiae ut reiciendis\nqui aperiam non debitis possimus qui neque nisi nulla"
  },
  {
    "userId": 1,
    "id": 3,
    "title": "ea molestias quasi exercitationem repellat qui ipsa sit aut",
    "body": "et iusto sed quo iure\nvoluptatem occaecati omnis eligendi aut ad\nvoluptatem doloribus vel accusantium quis
pariatur\nmolestiae porro eius odio et labore et velit aut"
  },
  {
    "userId": 1,
    "id": 4,
    "title": "eum et est occaecati",
    "body": "ullam et saepe reiciendis voluptatem adipisci\nsit amet autem assumenda provident rerum culpa\nquis hic commodi
nesciunt rem tenetur doloremque ipsam iure\nquis sunt voluptatem rerum illo velit"
  },
  {
    "userId": 1,
    "id": 5,
    "title": "nesciunt quas odio",
    "body": "repudiandae veniam quaerat sunt sed\nalias aut fugiat sit autem sed est\nvoluptatem omnis possimus esse voluptatibus
quis\nest aut tenetur dolor neque"
  },
```


We get back this dummy data indicating a userId, Id, title and body. All this data can now be used in our application assuming it is a valid data.

We can transform this into a JS object and now use the data we get back like accessing them with their index number and properties.

Script.js

```
var http = new XMLHttpRequest();
var url = 'http://jsonplaceholder.typicode.com/posts';
var method = 'GET';

http.open(method, url); //this opens or launch the request
http.onreadystatechange = function() {
    if (http.readyState === XMLHttpRequest.DONE && http.status === 200) {
        console.log(JSON.parse(http.responseText)); //use the JSON.parse to transform
the data to object
} else if (http.readyState === XMLHttpRequest.DONE && http.status !== 200) {
```

```
    console.log('Error!');
}
};
http.send();
/*
line 1 - this define a variable 'http' and assign the value 'new XMLHttpRequest' to it
line 2 - This define a variable that reached out to the fake server
line 3 - this define a variable and assign the value to the type we want 'GET' (to get
data)
line 5 - this launch the http server
line 7 - if the request is done and the status is 200 which means 'successful'
line 8 - log the response
line 9 - if the request is done but the status is not successful
line 10 - log the error
line 13 - now send the request

*/
```

This will give you back an array of objects

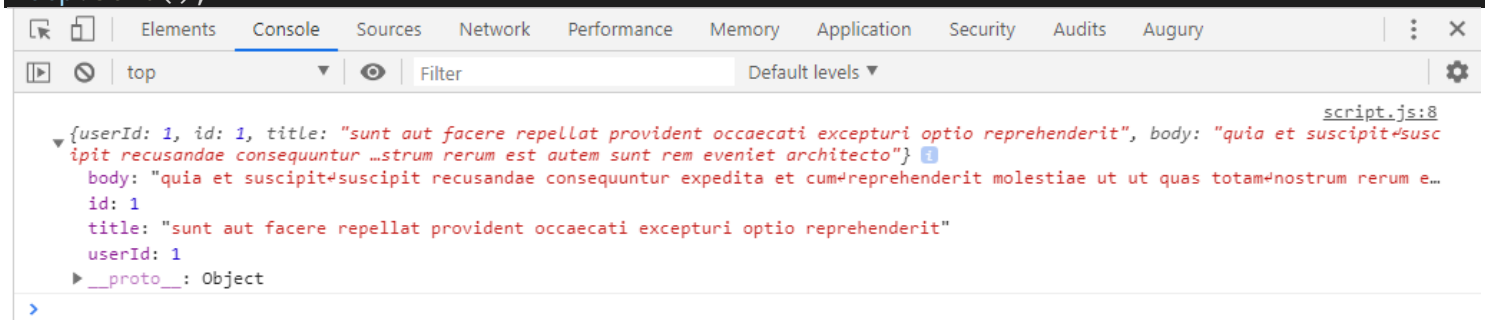
```
script.js:8
(100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}
, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}
, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}
, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
▶0: {userId: 1, id: 1, title: "sunt aut facere repellat provident occaecati excepturi optio reprehenderit", body: "quia et susci…
▶1: {userId: 1, id: 2, title: "qui est esse", body: "est rerum tempore vitae↵sequi sint nihil reprehend…aperiam non debitis poss…
▶2: {userId: 1, id: 3, title: "ea molestias quasi exercitationem repellat qui ipsa sit aut", body: "et iusto sed quo iure↵volupt…
▶3: {userId: 1, id: 4, title: "eum et est occaecati", body: "ullam et saepe reiciendis voluptatem adipisci↵sit … ipsam iure↵quis…
▶4: {userId: 1, id: 5, title: "nesciunt quas odio", body: "repudiandae veniam quaerat sunt sed↵alias aut fugi…sse voluptatibus q…
▶5: {userId: 1, id: 6, title: "dolorem eum magni eos aperiam quia", body: "ut aspernatur corporis harum nihil quis provident …s↵…
▶6: {userId: 1, id: 7, title: "magnam facilis autem", body: "dolore placeat quibusdam ea quo vitae↵magni quis e…t excepturi ut q…
▶7: {userId: 1, id: 8, title: "dolorem dolore est ipsam", body: "dignissimos aperiam dolorem qui eum↵facilis quibus…↵ipsam ut co…
▶8: {userId: 1, id: 9, title: "nesciunt iure omnis dolorem tempora et accusantium", body: "consectetur animi nesciunt iure dolor…
▶9: {userId: 1, id: 10, title: "optio molestias id quia eum", body: "quo et expedita modi cum officia vel magni↵dolorib…it↵quos …
▶10: {userId: 2, id: 11, title: "et ea vero quia laudantium autem", body: "delectus reiciendis molestiae occaecati non minima…lu…
▶11: {userId: 2, id: 12, title: "in quibusdam tempore odit est dolorem", body: "itaque id aut magnam↵praesentium quia et ea odit…
▶12: {userId: 2, id: 13, title: "dolorum ut in voluptas mollitia et saepe quo animi", body: "aut dicta possimus sint mollitia vo…
▶13: {userId: 2, id: 14, title: "voluptatem eligendi optio", body: "fuga et accusamus dolorum perferendis illo volupta…molestiae…
▶14: {userId: 2, id: 15, title: "eveniet quod temporibus", body: "reprehenderit quos placeat↵velit minima officia do…usandae qui…
▶15: {userId: 2, id: 16, title: "sint suscipit perspiciatis velit dolorum rerum ipsa laboriosam odio", body: "suscipit nam nisi …
▶16: {userId: 2, id: 17, title: "fugit voluptas sed molestias voluptatem provident", body: "eos voluptas et aut odit natus earum…
▶17: {userId: 2, id: 18, title: "voluptate et itaque vero tempora molestiae", body: "eveniet quo quis↵laborum totam consequatur …
▶18: {userId: 2, id: 19, title: "adipisci placeat illum aut reiciendis qui", body: "illum quis cupiditate provident sit magnam↵e…
▶19: {userId: 2, id: 20, title: "doloribus ad provident suscipit at", body: "qui consequuntur ducimus possimus quisquam amet si……
▶20: {userId: 3, id: 21, title: "asperiores ea ipsam voluptatibus modi minima quia sint", body: "repellat aliquid praesentium do…
▶21: {userId: 3, id: 22, title: "dolor sint quo a velit explicabo quia nam", body: "eos qui et ipsum ipsam suscipit aut↵sed omni…
▶22: {userId: 3, id: 23, title: "maxime id vitae nihil numquam", body: "veritatis unde neque eligendi↵quae quod architecto…t vel…
▶23: {userId: 3, id: 24, title: "autem hic labore sunt dolores incidunt", body: "enim et ex nulla↵omnis voluptas quia qui↵volupt…
▶24: {userId: 3, id: 25, title: "rem alias distinctio quo quis", body: "ullam consequatur ut↵omnis quis sit vel consequunt…nostr…
▶25: {userId: 3, id: 26, title: "est et quae odit qui non", body: "similique esse doloribus nihil accusamus↵omnis dol…is cum ut …
```

Now you can use the properties or call any of the data using their index number.

Script.js (calling the first object in the array)

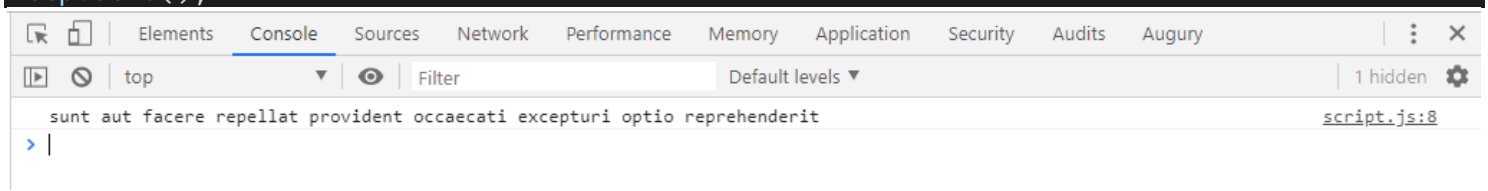
```
var http = new XMLHttpRequest();
var url = 'http://jsonplaceholder.typicode.com/posts';
var method = 'GET';

http.open(method, url); //this opens or launch the request
http.onreadystatechange = function() {
    if (http.readyState === XMLHttpRequest.DONE && http.status === 200) {
        console.log(JSON.parse(http.responseText)[0]); //we are calling the first
object in the array
} else if (http.readyState === XMLHttpRequest.DONE && http.status !== 200) {
```

```
    console.log('Error!');
}
};
http.send();
```



Script.js (calling the title of the array 0)

```
var http = new XMLHttpRequest();
var url = 'http://jsonplaceholder.typicode.com/posts';
var method = 'GET';

http.open(method, url); //this opens or launch the request
http.onreadystatechange = function() {
    if (http.readyState === XMLHttpRequest.DONE && http.status === 200) {
        console.log(JSON.parse(http.responseText)[0].title); //we are calling the title
of the first object in the array
} else if (http.readyState === XMLHttpRequest.DONE && http.status !== 200) {
     console.log('Error!');
}
};
http.send();
```



POST REQUEST: - Not only can we get data; we can also post data.

Script.js

```
var http = new XMLHttpRequest();
var url = 'http://jsonplaceholder.typicode.com/posts';
var method = 'POST'; //changing the method to post data
var data = 'title=Post%20Title&body=Body'; //this code represent - 'title = Post
Title', 'body = Body'

http.open(method, url); //this opens or launch the request
http.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded'); // setting
our content type
http.onreadystatechange = function() {
    if (http.readyState === XMLHttpRequest.DONE && http.status === 201) { //we change
the status to 201 since this is the status we get when we create or post something.
        console.log(JSON.parse(http.responseText));
} else if (http.readyState === XMLHttpRequest.DONE && http.status !== 201) {
     console.log('Error!');
}
```

```
};
http.send(data);// we pass the data we want to send which we defined as variable
```

```
Elements  Console  Sources  Network  Performance  Memory  Application  Security  Audits  Augury
top  Filter  Default levels
▼{title: "Post Title", body: "Body", id: 101}                                    script.js:10
   body: "Body"
   id: 101
   title: "Post Title"
  ▶__proto__: Object
>
```

The data we declared was stored successfully and we got it back. Though this wasn't stored on any server. It's just dummy but it is important to know how it works. More information on how to post and get data can be found in the cheat sheet.

LIBRARIES, FRAMEWORKS AND MODULES

INTRODUCTION: - We have learnt the basis of JS and how it works. At a time in your development, you will add a library of complete framework or if you have a bigger app of many modules, you might add a modularity.

JS LIBRARIES:

- **Jquery: - this uses pure js but adds many convenient properties and methods. Jquery is almost like its own language and sometimes mistaken for that but at the end it's just JS with a lot of helpers. You can access the jquery documentation to learn about how it works and all the methods and properties you can use.**

FRAMEWORKS: Sometimes you don't want a library like jquery but you want to build a real single page application (SPAs) where the whole user experience is handled by javascript.

- **Angular: - this is a javascript framework that uses typescript, a subset of JS.**

WRITING MODULAR CODE: This allows you to split up your application to several files/modules and then if you need some file you quickly through in an import at the top and have all the code from this file available.

- **Webpack**
- **Systemjs**
- **Jspm.io**

COURSE PROJECT

INTRODUCTION: - We have covered a lot in this course about JS to write your own application. It is always a great thing to have some kind of project where all those things you have learnt comes together. Though we cannot cover every piece of that but we will show the most important ones and so we get the overall feeling on how such a project looks like and that will enable you to transfer the knowledge gained throughout this course into any project and work with the dom/website and the user.

SETUP: - We will be using plunker for this but you are free to use your code editor.

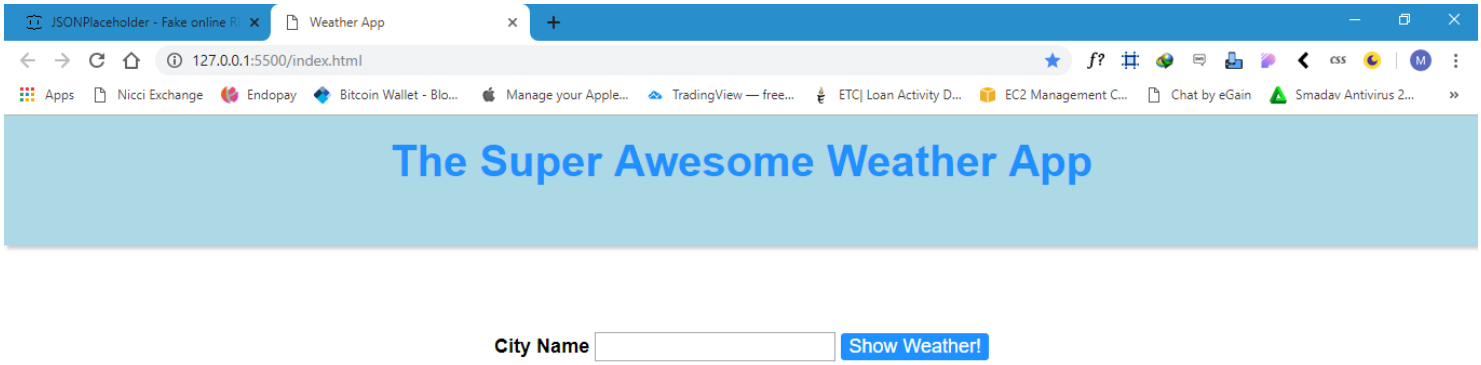
- **Firstly, we will apply the css styling which can be found in the resources attached to the project.**
- **Secondly, we will apply our html codes which we can also find attached.**
- **Create a file name 'app.js'**

- **Create a file name 'init.js'**

We will be building a basic weather app where we can enter a city name and get the weather information of that city.

App.js file: - in this file, we want to hook up the 'show weather' button to fire an event that we can actually see or load the weather for a certain city. And we also want to reach out to some server to fetch the weather and to then display it on the web page.

Init.js file: - this will get us all the html elements we need for the application



Index.html

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Weather App</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
<header>
    <h1>The Super Awesome Weather App</h1>
</header>
<div class="main">
    <div id="search">
        <label for="city">City Name</label>
        <input type="text" id="city">
        <button type="button">Show Weather!</button>
    </div>
    <div id="load">Loading...</div>
    <div id="weather">
        <h1 id="weatherCity">City Name</h1>

        <div id="weatherDescription">Weather Description</div>
        <div id="weatherTemperature">Temperature</div>

    </div>
</div>
<script src="init.js"></script>
<script src="weather-data.js"></script>
```

```
<script src="app.js"></script>
</body>
</html>
```

Style.css

```
/* Styles go here */

body {
    padding: 0;
    margin: 0;
    font-family: Helvetica Neue, Helvetica, Arial;
}

header {
    height: 100px;
    background-color: lightblue;
    text-align: center;
    padding-top: 20px;
    box-shadow: 3px 3px 3px #ccc;
}

header h1 {
    font-size: 40px;
    font-weight: bold;
    color: dodgerblue;
    margin: 0;
}

.main {
    text-align: center;
    padding-top: 80px;
}

#search {
    font-size: 18px;
}

#search label {
    font-weight: bold;
}

#search input {
    font-size: inherit;
    font-family: inherit;
}

#search button {
    font-family: inherit;
    font-size: inherit;
    background-color: dodgerblue;
    border: 1px solid dodgerblue;
    border-radius: 3px;
```

```
    cursor: pointer;
    color: white;
}

#search button:hover {
    background-color: deepskyblue;
}

#weather {
    padding: 30px;
    border: 1px solid #ccc;
    width: 500px;
    margin: 100px auto auto;
    background-color: deepskyblue;
    box-shadow: 2px 2px 3px #ccc;
    color: white;
    box-sizing: border-box;
    display: none;
    font-weight: bold;
}

#weather h1 {
    font-size: 32px;
    margin-top: 0;
    margin-bottom: 30px;
}

#weatherDescription {
    float: left;
}

#weatherTemperature {
    float: right;
}

#load {
    padding: 30px;
    width: 500px;
    margin: 100px auto auto;
    display: none;
}
```

FETCHING USER INPUT

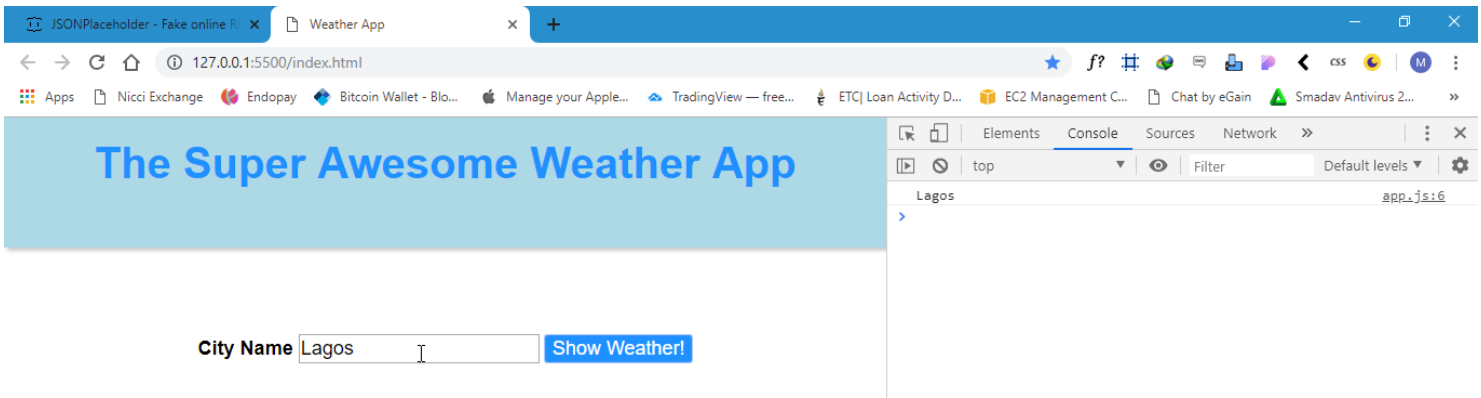
Now we need to be able to select the button and the input field elements so we can then add an event listeners to listen to click on the button and then fire an event that log the city the user enters in the console

Init.js

```
"use strict"

var searchButton = document.querySelector('button');//we are selecting the button
var searchCity = document.querySelector('#city');//we are selecting the input field
```

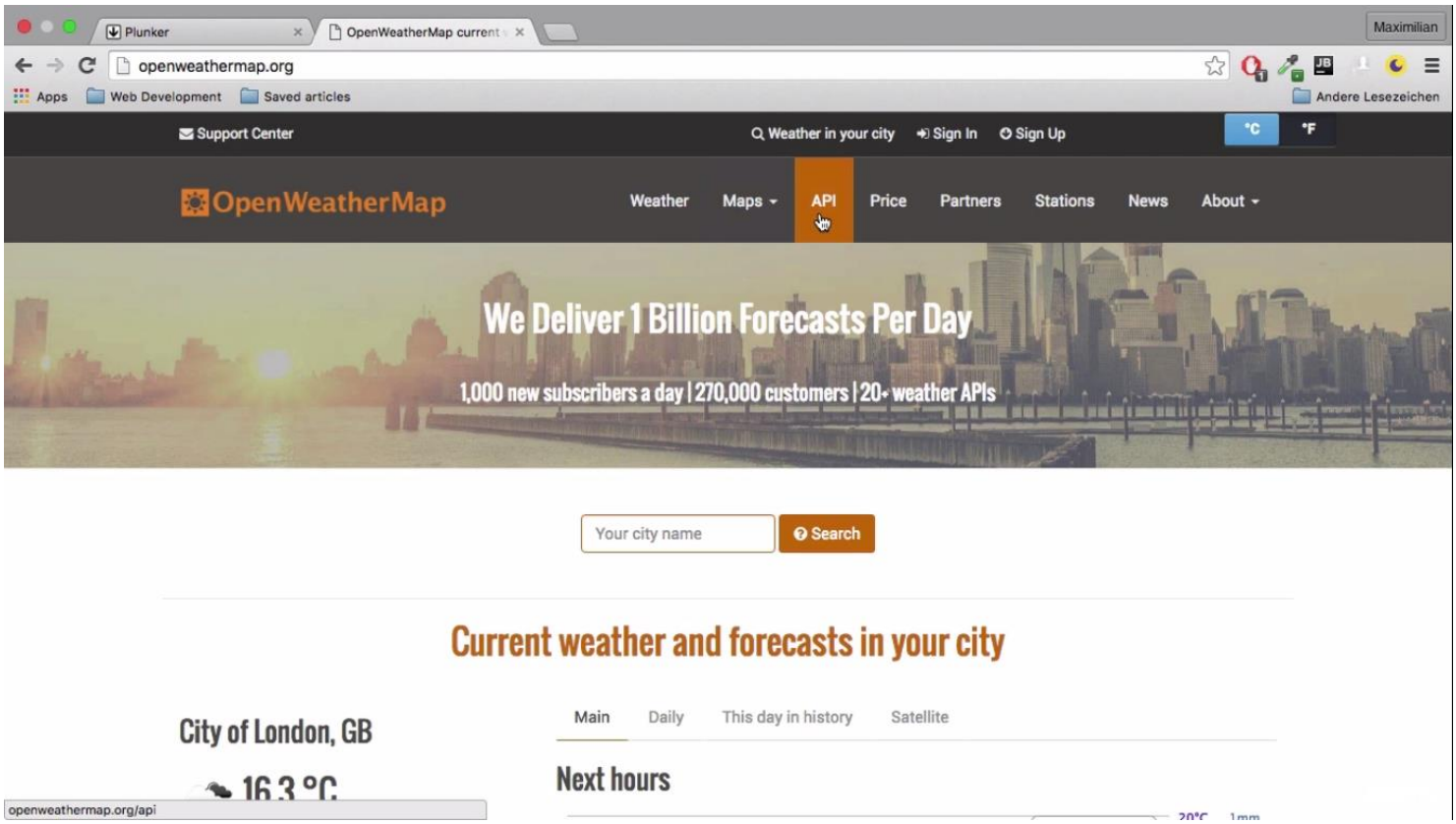
app.js

```
"use strict"

searchButton.addEventListener('click', searchWeather);

function searchWeather(){
    console.log(searchCity.value); //this will allows us to get the value the users
enters in the input field
};
```



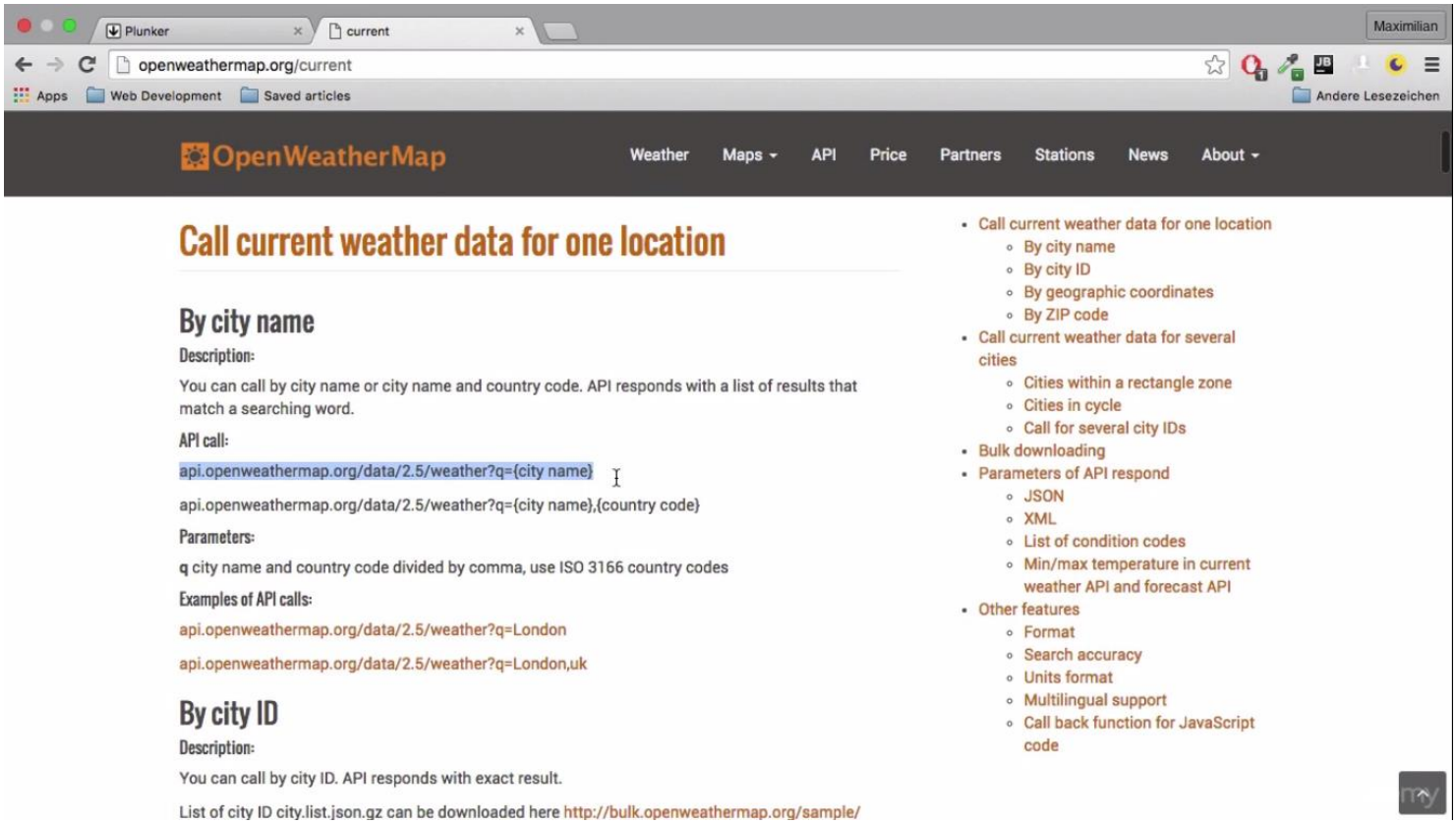
Now that we are able to get the value of the input the user enters, we need to send it to a service which actually gives us the weather and the user the weather data we get back in whichever form we get back to actually output it in the dom below the inline form.

HTTP REQUEST SETUP: - Now we need to be able to get the weather and for this we will be using a weather api from a service.

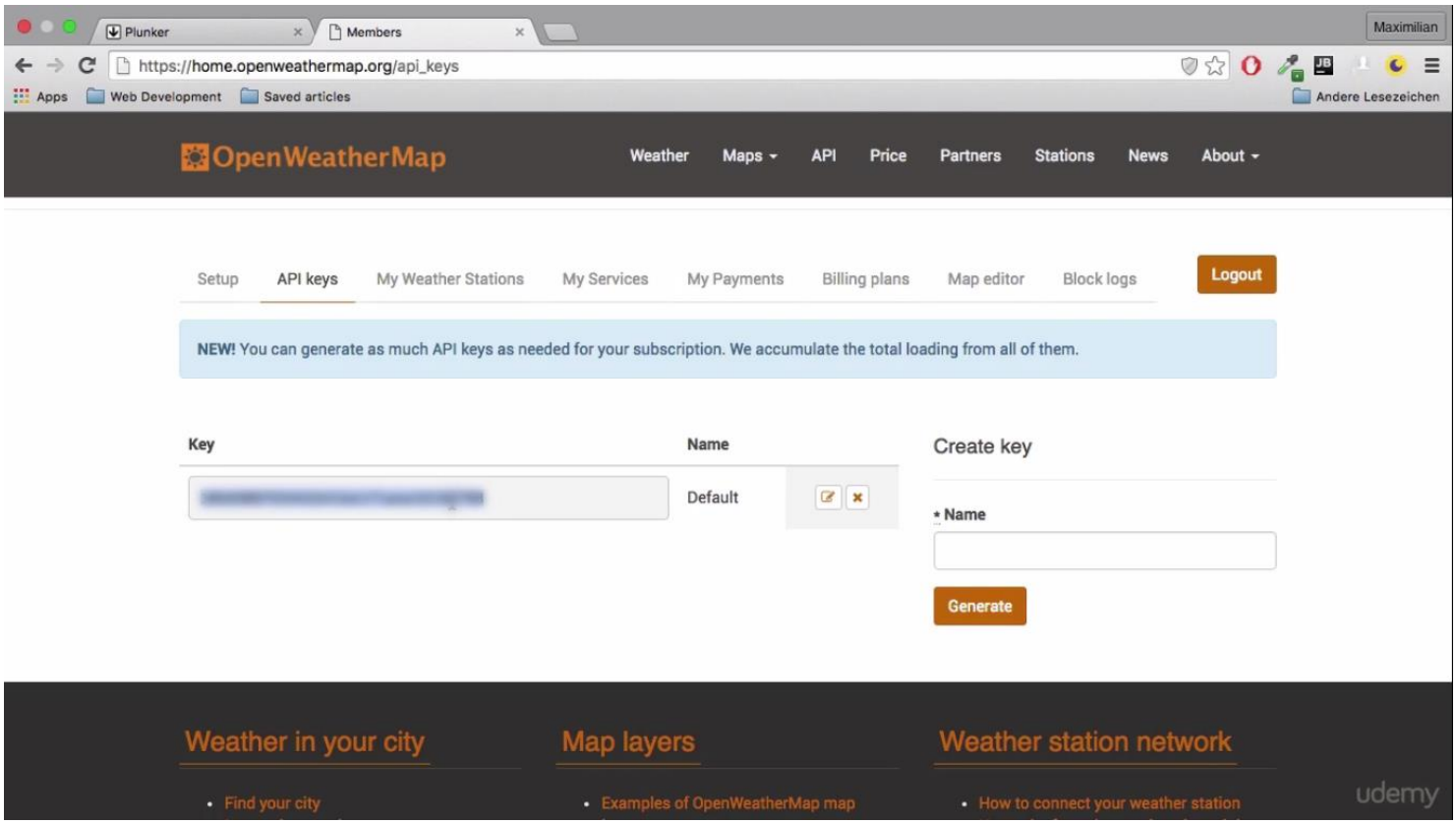


We will need to signup in other to be able to get their api.

We will be using the simple api we have below



After signing up, you will be given a key which can then append later to your api



So let us setup our http request to get the weather data from there.

Script.js

```
"use strict"

searchButton.addEventListener('click', searchWeather);
```

```
function searchWeather(){
    var cityName = searchCity.value;// assigning a variable to the value we get from
the input field
    if(cityName.trim().length == 0){
        alert('Please enter a city name!');
    }; // we are checking to sure the user enters a name using the length of the input
and we are using trim() to remove any whitespace
    var http = new XMLHttpRequest; //setting up our httprequest
    var apiKey = ''; //your api key will be stored here
    var url = 'http://api.openweathermap.org/data/2.5/weather?q=' + cityName; //the
link can be copied from the website and we adding the cityName from the user input.
    var method = 'GET';
    http.open(method, url);
    http.onreadystatechange = function() {
        if (http.readyState == XMLHttpRequest.DONE && http.status === 200) { //if the
request is done and the status is 200 then we want tp execute a block of code

        } else if (http.readyState == XMLHttpRequest.DONE && http.status !== 200) {
            alert('Something went wrong!'); //else we return an error
        }
    };
};
```

CREATING A WEATHER OBJECT: - Now it is time to work on the the logic we do execute if we get the weather data. This is the logic that will be inside the if statement.

App.js

```
"use strict"

searchButton.addEventListener('click', searchWeather);

function searchWeather(){
    var cityName = searchCity.value;
    if(cityName.trim().length == 0){
        alert('Please enter a city name!');
    };
    var http = new XMLHttpRequest;
    var apiKey = '';
    var url = 'http://api.openweathermap.org/data/2.5/weather?q=' + cityName;
    var method = 'GET';
    http.open(method, url);
    http.onreadystatechange = function() {
        if (http.readyState == XMLHttpRequest.DONE && http.status === 200) {
            var data = JSON.parse(http.responseText);
        } else if (http.readyState == XMLHttpRequest.DONE && http.status !== 200) {
            alert('Something went wrong!');
        }
    };
};
/*
line 6 - assigning a variable to the value we get from the input field.
```

```
line 9 - we are checking to sure the user enters a name using the length of the input
and we are using trim() to remove any whitespace.
line 10 - setting up our httprequest.
line 11 - your api key will be stored here.
line 12 - the link can be copied from the website and we adding the cityName from the
user input.
line 16 - if the request is done and the status is 200 then we want to turn the server
response into a JSON.
line 19 -  else we return an error.
*/
```

Here we are configuring our weather data base on the response we are expecting from the server. The response we are expecting is something of this nature below:

JSON

Example of API response:


```
{"coord":
{"lon":145.77,"lat":-16.92},
"weather":[{"id":803,"main":"Clouds","description":"broken clouds","icon":"04n"}],
"base":"cmc stations",
"main":{"temp":293.25,"pressure":1019,"humidity":83,"temp_min":289.82,"temp_max":295.37
"wind":{"speed":5.1,"deg":150},
"clouds":{"all":75},
"rain":{"3h":3},
"dt":1435658272,
"sys":{"type":1,"id":8166,"message":0.0166,"country":"AU","sunrise":1435610796,"sunset"
"id":2172797,
"name":"Cairns",
"cod":200}
```


Here we have many datas from which we only need to use the temperature 'temp' and description 'description'.

Weather-data.js

```
"use strict"
function Weather(cityName, description) {
    this.cityName = cityName;
    this.description = description;
    this._temperature = '';
}
Object.defineProperty(Weather.prototype, 'temperature', {
    get: function(){
        return this._temperature;
    },
    set: function(value) {
        this._temperature = (value * 1.8 + 32).toFixed(2) + 'F.';
    }
});
/*
line 3 -  this will store the value of the city te user enters.
line 4 -  this will store the value of the description we get.
line 5 - this will store the value of the temperature we get from the server which will
be in celsius.
line 8 - we are using the getter and so when we call the temperature property, we get
back the function.
```

```
line 11 -  we are using the setter to convert the temperature we get from server from
celsius to fahenrent.
*/
```

Init.js

```
"use strict"

var searchButton = document.querySelector('button');
var searchCity = document.querySelector('#city');
/*
line 3 - we are selecting the button.
line 4 - we are selecting the input field.
*/
```

FETCHING AND HANDLING WEATHER DATA: - We want to fetch and handle the weather data we get from the server and according to the picture below, that is the format of the data we will be getting back from the server but we are only interested in the description and temperature.

JSON

Example of API response:


```
{"coord":
{"lon":145.77,"lat":-16.92},
"weather":[{"id":803,"main":"Clouds","description":"broken clouds","icon":"04n"}],
"base":"cmc stations",
"main":{"temp":293.25,"pressure":1019,"humidity":83,"temp_min":289.82,"temp_max":295.37
"wind":{"speed":5.1,"deg":150},
"clouds":{"all":75},
"rain":{"3h":3},
"dt":1435658272,
"sys":{"type":1,"id":8166,"message":0.0166,"country":"AU","sunrise":1435610796,"sunset":
"id":2172797,
"name":"Cairns",
"cod":200}
```


Weather-data.js

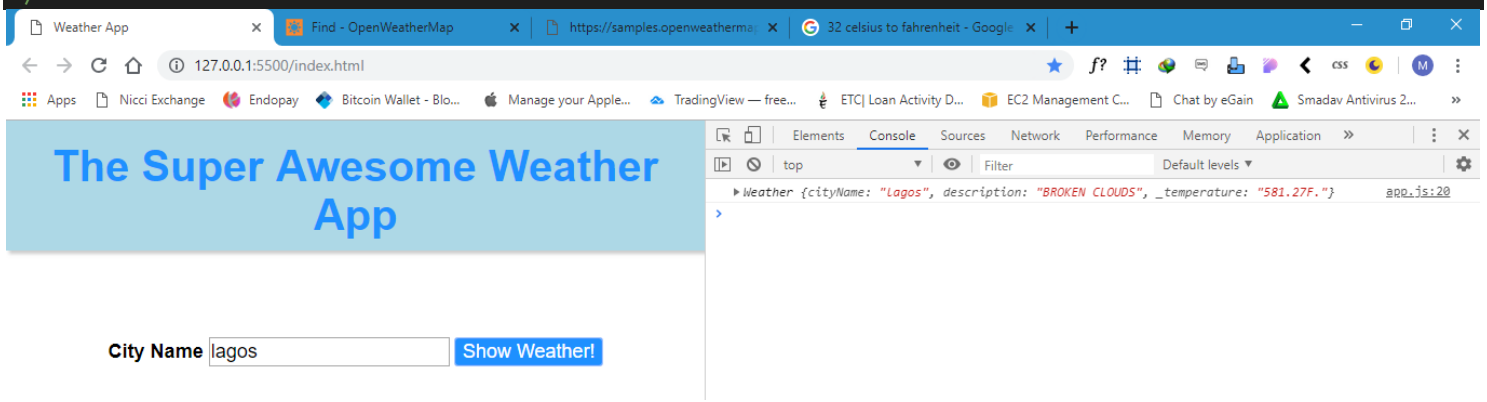
```
"use strict"
function Weather(cityName, description) {
    this.cityName = cityName;
    this.description = description;
    this._temperature = '';
}
//THE FOLLOWING LINES (7 - 15) is only used to transform the temperature data from the
default celsius we will get from server to fahrenheit. WITHOUT THE FOLLOWING, THE APP
WILL STILL RUN FINE.
Object.defineProperty(Weather.prototype, 'temperature', {
    get: function(){
        return this._temperature;
    },
    set: function(value) {
        this._temperature = (value * 1.8 + 32).toFixed(2) + 'F.';
    }
});
/*
line 3 -  this will store the value of the city te user enters.
line 4 -  this will store the value of the description we get.
```

```
line 5 - this will store the value of the temperature we get from the server which will
be in celsius.
line 7 - the 'temperature' used here will be called in app.js line 19
line 8 - we are using the getter and so when we call the temperature property, we get
back the function.
line 11 -  we are using the setter to convert the temperature we get from server from
celsius to fahenrent.
*/
```

App.js

```
"use strict"

searchButton.addEventListener('click', searchWeather);

function searchWeather(){
    var cityName = searchCity.value;
    if(cityName.trim().length == 0){
        alert('Please enter a city name!');
    };
    var http = new XMLHttpRequest;
    var apiKey = 'ac4587120ccf60df5d4a4a98a8c3f90b';
    var url = 'http://api.openweathermap.org/data/2.5/weather?q=' + cityName +
'&unitmetrics&appid=' + apiKey;
    var method = 'GET';
    http.open(method, url);
    http.onreadystatechange = function() {
        if (http.readyState == XMLHttpRequest.DONE && http.status === 200) {
            var data = JSON.parse(http.responseText);
            var weatherData = new Weather(cityName,
data.weather[0].description.toUpperCase());
            weatherData.temperature = data.main.temp;
            console.log(weatherData);
        } else if (http.readyState == XMLHttpRequest.DONE && http.status !== 200) {
            alert('Something went wrong!');
        }
    };
    http.send();
};
/*
line 6 - assigning a variable to the value we get from the input field.
line 7 - we are checking to sure the user enters a name using the length of the input
and we are using trim() to remove any whitespace.
line 10 - setting up our httprequest.
line 11 - your api key will be stored here.
line 12 - the link can be copied from the website and we adding the cityName from the
user input and then a couple of string and our api key.
line 16 & 17 - if the request is done and the status is 200 then we want to turn the
server response into a JS object.
line 18 - we are getting our city name and the weather description which we changed to
uppercase
line 19 - we are accessing the temperature data
```

```
line 20 - we want to log the weather data we get from the server to the console for
now.
line 21 -  else we return an error.
*/
```



DISPLAYING DATA: - Now we need to display the data we got from the server into our DOM and not the console and to do this, we will placing each property of the data in the html div we have created for them

```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Weather App</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
<header>
    <h1>The Super Awesome Weather App</h1>
</header>
<div class="main">
    <div id="search">
        <label for="city">City Name</label>
        <input type="text" id="city">
        <button type="button">Show Weather!</button>
    </div>
    <div id="load">Loading ... </div>
    <div id="weather">
        <h1 id="weatherCity">City Name</h1>

        <div id="weatherDescription">Weather Description</div>
        <div id="weatherTemperature">Temperature</div>

    </div>
</div>
<script src="init.js"></script>
<script src="weather-data.js"></script>
<script src="app.js"></script>
</body>
</html>
```

Now we need to select/each of this elements with query selector

Init.js

```javascript
"use strict"

var searchButton = document.querySelector('button');
var searchCity = document.querySelector('#city');
var loadingText = document.querySelector('#load');
var weatherBox = document.querySelector('#weather');
var weatherCity = document.querySelector('#weatherCity');
var weatherDescription = document.querySelector('#weatherDescription');
var weatherTemperature = document.querySelector('#weatherTemperature');

/*
```

```
line 3 - we are selecting the button.
line 4 - we are selecting the input field.
line 5 - we are selecting the loading text that loads before our page is displayed.
line 6 - we are selecting the container that contains all our data.
line 7 - we are selecting the name of the city.
line 8 - we are selecting the weather description html element.
line 9 - we are selecting the weather temperature html element.
*/
```

Weather-data.js

```
"use strict"
function Weather(cityName, description) {
    this.cityName = cityName;
    this.description = description;
    this._temperature = '';
}
//THE FOLLOWING LINES (7 - 15) is only used to transform the temperature data from the
default celsius we will get from server to fahrenheit. WITHOUT THE FOLLOWING, THE APP
WILL STILL RUN FINE.
Object.defineProperty(Weather.prototype, 'temperature', {
    get: function(){
        return this._temperature;
    },
    set: function(value) {
        this._temperature = (value * 1.8 + 32).toFixed(2) + 'F.';
    }
});
/*
line 3 -  this will store the value of the city te user enters.
line 4 -  this will store the value of the description we get.
line 5 - this will store the value of the temperature we get from the server which will
be in celsius.
line 7 - the 'temperature' used here will be called in app.js line 19
line 8 - we are using the getter and so when we call the temperature property, we get
back the function.
line 12 - Celsius to fahrenheit conversion
*/
```

App.js

```
"use strict"

searchButton.addEventListener('click', searchWeather);

function searchWeather(){
    loadingText.style.display = 'block';
    weatherBox.style.display = 'none';
    var cityName = searchCity.value;
    if(cityName.trim().length == 0){
        alert('Please enter a city name!');
    };
    var http = new XMLHttpRequest;
```

```
    var apiKey = 'ac4587120ccf60df5d4a4a98a8c3f90b';
    var url = 'http://api.openweathermap.org/data/2.5/weather?q=' + cityName +
'&unitmetrics&appid=' + apiKey;
    var method = 'GET';
    http.open(method, url);
    http.onreadystatechange = function() {
        if (http.readyState == XMLHttpRequest.DONE && http.status === 200) {
            var data = JSON.parse(http.responseText);
            var weatherData = new Weather(cityName,
data.weather[0].description.toUpperCase());
            weatherData.temperature = data.main.temp;
            weatherCity.textContent = weatherData.cityName;
            date.textContent = new Date();
            weatherDescription.textContent = weatherData.description;
            weatherTemperature.textContent = weatherData._temperature;
            loadingText.style.display = 'none';
            weatherBox.style.display = 'block';
        } else if (http.readyState == XMLHttpRequest.DONE && http.status !== 200) {
            alert('Something went wrong!');
        }
    };
    http.send();
};
/*
line 6 - 'block' will display the message 'Loading...' before our data get fetched from
the server.
line 7 - 'none' we hiding our weather box while the loading... is displayed
line 8 - assigning a variable to the value we get from the input field.
line 9 - we are checking to sure the user enters a name using the length of the input
and we are using trim() to remove any whitespace.
line 12 - setting up our httprequest.
line 13 - your api key will be stored here.
line 14 - the link can be copied from the website and we adding the cityName from the
user input and then a couple of string and our api key.
line 18 & 19 - if the request is done and the status is 200 then we want to turn the
server response into a JS object.
line 20 - we are getting our city name and the weather description which we changed to
uppercase
line 21 - we are accessing the temperature data
line 22 - 24 - we are extracting the text content of the weather data and assigning
them to their respective html elements
line 25 - we are hiding the loading.... while the data is displayed
line 26 - we are showing the weather box while the loading is hidden
line 28 -  else we return an error.
*/
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Weather App</title>
<!--Links-->
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    <link rel="stylesheet" href="style.css">
    <link href="https://fonts.googleapis.com/css?family=Merienda+One" rel="stylesheet">
    <link href="https://fonts.googleapis.com/css?family=Lobster" rel="stylesheet">
    <link href="https://fonts.googleapis.com/css?family=Acme" rel="stylesheet">
    <link rel="stylesheet"
href="https://use.fontawesome.com/releases/v5.8.1/css/all.css" integrity="sha384-
50oBUHEmvpQ+1lW4y57PTFmhCaXp0ML5d60M1M7uH2+nqUivzIebhndOJK28anvf"
crossorigin="anonymous">

</head>
<body>
    <div class="dark-overlay">
<!--HEADER-->
<section id="pageHeader">
<header>
<img src="./michaelz.png" alt="" class="navbar-brand  img-fluid  float-left ml-5 brand"
width="50" height="50" >
<div class="row">
<div class="col-md-6">
<p class="text-white mr-5 pt-3">Michaelzgraphix weather app!</p>
</div>
<div class="col-md-6 pt-3 d-none d-md-block">
<a href="http://michaelzgraphix-recipe-book.s3-website-us-west-2.amazonaws.com/"
class="float-right text-white link mr-5"> <i class="fas fa-coffee"></i> Checkout my
recipe app!</a>
</div>
</div>
</header>
</section>
<!--JAVASCRIPT-->
<section id="javascript">
<div class="container">
<div class="main">
<div class="row justify-content-center mt-5">
<div class="col-md-6 col-sm-12">
<div class="input-group">
<input id="city"type="text" class="form-control mr-2" placeholder="City Name">
<div class="input-group-append">
<button id="button" class="btn btn-info" type="button"> <i class="fas fa-hand-
pointer"></i> Show Weather!</button>
</div>
</div>

<div id="load">Loading...</div>
<div id="weather" class="slideRight">
<h1 id="weatherCity">City Name</h1>
<div id="weatherDescription">Weather Description</div>
```

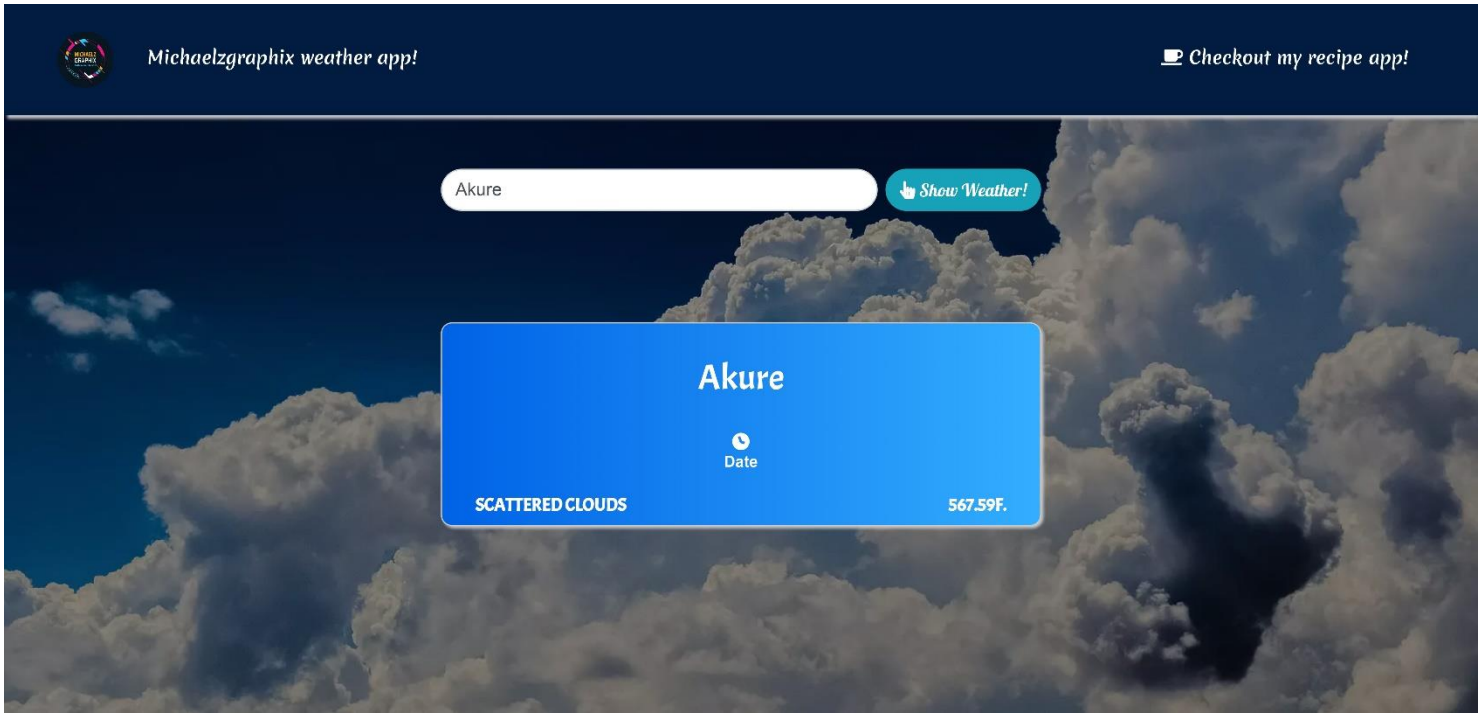
```
<div id="weatherTemperature">Temperature</div>
</div>
</div>
</div>
</div>
</div>
</section>


<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
<script src="init.js"></script>
<script src="weather-data.js"></script>
<script src="app.js"></script>
</div>
</body>
</html>
```

Style.css

```
/* Styles go here */

body {
    padding: 0;
    margin: 0;
    font-family: Helvetica Neue, Helvetica, Arial;
    background-image: url("/clouds-2329680_1920.jpg");
    background-attachment: fixed;
    background-repeat: repeat-x;
    background-position: 50% 50%;
    background-size: 100%;
    -moz-background-size: cover;
    -o-background-size: cover;
    -webkit-background-size: cover;
}

header {
    height: 100px;
    background-color: #011c40;
    text-align: left;
    padding-top: 20px;
    box-shadow: 3px 3px 3px #ccc;
    font-family: 'Merienda One', cursive;
}


header p {
```

```
    color: rgb(222, 226, 230);
    margin: 0;
    font-family: 'Merienda One', cursive;
}

.link {
    color: rgb(222, 226, 230);
    margin: 0;
    font-family: 'Merienda One', cursive;
}

.dark-overlay {
    background-color: rgba(0, 0, 0, 0.4);
    position: absolute;
    top: 0;
    left: 0%;
    width: 100%;
    height: 100%; }





  /* Add media queries for responsiveness - when the screen is 500px wide or less,
stack the links on top of each other */
  @media screen and (max-width: 500px) {
    .header a {
      float: none;
      display: block;
      text-align: left;
    }
    .header-right {
      float: none;
    }
  }
}

.main {
    text-align: center;
    padding-top: 80px;
}

#search {
    font-size: 18px;
}

#search label {
    font-weight: bold;
}

#search input {
    font-size: inherit;
    font-family: inherit;
}
```

```
#search button {
    font-family: inherit;
    font-size: inherit;
    background-color: dodgerblue;
    border: 1px solid dodgerblue;
    border-radius: 3px;
    cursor: pointer;
    color: white;
}

#search button:hover {
    background-color: deepskyblue;
}

#weather {
    padding: 30px;
    margin: 100px auto auto;
    border: 1px solid #ccc;
    background: linear-gradient(to right, #0062E6, #33AEff);
    box-shadow: 2px 2px 3px #ccc;
    color: white;
    box-sizing: border-box;
    display: none;
    font-weight: bold;
    text-align: center;
    border-radius: 10px;
}

.slideRight{
    animation-name: slideRight;
    -webkit-animation-name: slideRight;

    animation-duration: 1s;
    -webkit-animation-duration: 1s;

    animation-timing-function: ease-in-out;
    -webkit-animation-timing-function: ease-in-out;

    visibility: visible !important;
}

@keyframes slideRight {
    0% {
        transform: translateX(-150%);
    }
    50%{
        transform: translateX(8%);
    }
    65%{
        transform: translateX(-4%);
    }
    80%{
        transform: translateX(4%);
```

```
    }
    95%{
        transform: translateX(-2%);
    }
    100% {
        transform: translateX(0%);
    }
}

@-webkit-keyframes slideRight {
    0% {
        -webkit-transform: translateX(-150%);
    }
    50%{
        -webkit-transform: translateX(8%);
    }
    65%{
        -webkit-transform: translateX(-4%);
    }
    80%{
        -webkit-transform: translateX(4%);
    }
    95%{
        -webkit-transform: translateX(-2%);
    }
    100% {
        -webkit-transform: translateX(0%);
    }
}


#weather h1 {
    font-size: 32px;
    margin-top: 0;
    margin-bottom: 30px;
    font-family: 'Acme', sans-serif;
}

#weatherDescription {
    float: left;
    font-family: 'Acme', sans-serif;
}

#weatherTemperature {
    float: right;
    font-family: 'Acme', sans-serif;
}

#date {
    text-align: center;
    font-size: 14px;
}

#load {
```

```
    padding: 30px;
    margin: 100px auto auto;
    display: none;
    font-size: 80px;
    font-weight: bold;
    color: #ccc;
    text-align: center;
}


#button {
    border-radius: 30px;
    font-family: 'Lobster', cursive;
}

#city {
    border-radius: 30px;
}

#city:hover {
    border: dodgerblue 1px solid;
}
```