

History of JavaScript:

- In 1995, the browser that dominated the market was Netscape Navigator. However, with only static HTML pages, they were eager to improve the experience. They decided they needed a scripting language that would let developers make the internet more dynamic. A Netscape employee named Brendan Eich made the first version in 10 days. After a few iterations, they named the language JavaScript since Java was popular back then. Otherwise the languages don't have much connection.
- Soon, **JavaScript (JS)** was becoming popular. A standard was created by ECMA for scripting languages. The standard was based on JS. **ECMAScript (ES)** is a scripting-language specification standardized by Ecma International. It was created to standardize JavaScript to help foster multiple independent implementations. ES is the standard, while JS implements that standard.
- As time went on, ES's standards improved.
- ES3 (1999) is the baseline for modern day Javascript.
- A bunch of others, like Mozilla, started to work hard on ES5, which was released in 2009.
- Although more versions of the standard have been released, we will mostly talk about new features up to ES6 (2015).
- Browsers adopt new ES standards slowly, but ES6 is mostly completely adopted by modern browsers.

Basic JS:

- **Introduction:**
- JavaScript is the programming language of HTML and the Web.
- HTML is used to define the content of web pages.
- CSS is used to specify the layout of web pages.
- JavaScript is used to program the behavior of web pages.
- Javascript is an interpreted programming language that is used to make webpages interactive.
- It's object based.
- It runs on the client's browser.
- It uses events and actions to make webpages interactive.
- **Linking JavaScript in HTML:**
- In HTML, JavaScript code or a link to the javascript file is inserted between <script> tag.
- The <script> tag can be placed in either the <head> tag or the <body> tag or both.
- E.g.
 1. Linking to an external javascript file called index.js:
`<script src="path/to/javascript/file"> </script>`
 2. Putting inline javascript:
`<script>
// Some JavaScript code
</script>`
- **Equality:**
- There are two main ways of checking for equality in Javascript. We can use either the == operator or the === operator. However, the == operator will sometimes produce odd results. The === operator checks for type equality as well as content. A strict comparison (===) is only true if the operands are of the same type and the contents match. However, (==) converts the operands to the same type before making the comparison. Therefore, it is recommended that you only use the === operator and to never use the == operator.

- E.g.
`console.log(1 == 1); // expected output: true`
`console.log('1' == 1); // expected output: true`
`console.log(1 === 1); // expected output: true`
`console.log('1' === 1); // expected output: false`
- **Output:**
- JavaScript can output data in different ways:
 1. Writing into an HTML element, using **innerHTML**.
 2. Writing into the HTML output using **document.write()**.
 3. Writing into an alert box, using **window.alert()**.
 4. Writing into the browser console, using **console.log()**.
- Using innerHTML:
- To access an HTML element, JavaScript can use the **document.getElementById(id)** method. This is the most common way of obtaining an element to modify. Since IDs in HTML are guaranteed to be unique, we can access a pre-existing element in an HTML form using this strategy.
- The id attribute defines the HTML element. The innerHTML property defines the HTML content.
- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="test.js"></script>
  </head>
  <body>
    <h1>My First Web Page</h1>

    <p>My First Paragraph</p>

    <p id="demo"> Sample Text </p>
    <button onclick=changetext()> Click me</button>

  </body>
</html>
```

```
function changetext(){
  document.getElementById("demo").innerHTML = 5 + 6;
}
```



Originally, the page displays “Sample Text”, but after you click on the button, it changes to “11” because I modified the value of id=“demo” using `document.getElementById("demo").innerHTML`.

- **Note:** There is a similar command called `document.getElementsByClassName()` however since classes are not unique, this command will always return the result in an array.
- Using `document.write()`:
- For testing purposes, it is convenient to use `document.write()`.
- **Note:** Using `document.write()` after an HTML document is loaded, will delete all existing HTML.
- E.g. Consider the code and output below:

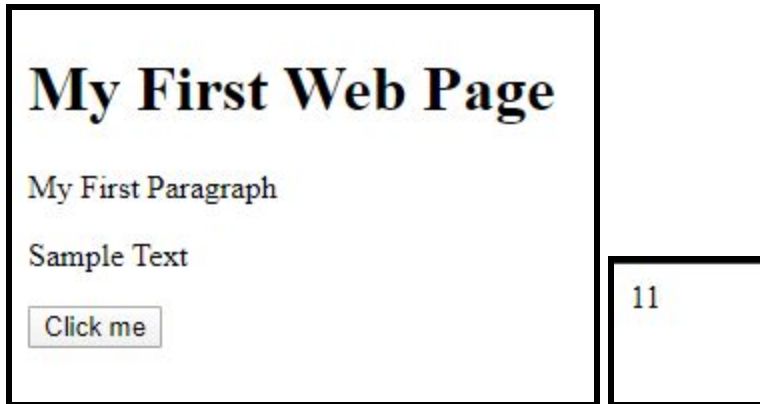
```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="test.js"></script>
  </head>
  <body>
    <h1>My First Web Page</h1>

    <p>My First Paragraph</p>

    <p> Sample Text </p>
    <button onclick=changetext()> Click me</button>

  </body>
</html>
```

```
function changetext(){
  document.write(5 + 6);
}
```



Originally, there's a lot of output shown on the web page, but after you click on the button, `document.write(5+6)` will delete all the existing HTML and replace it with 11.

- Using `window.alert()`:
- You can use an alert box to display data.
- E.g. Consider the code and output below:

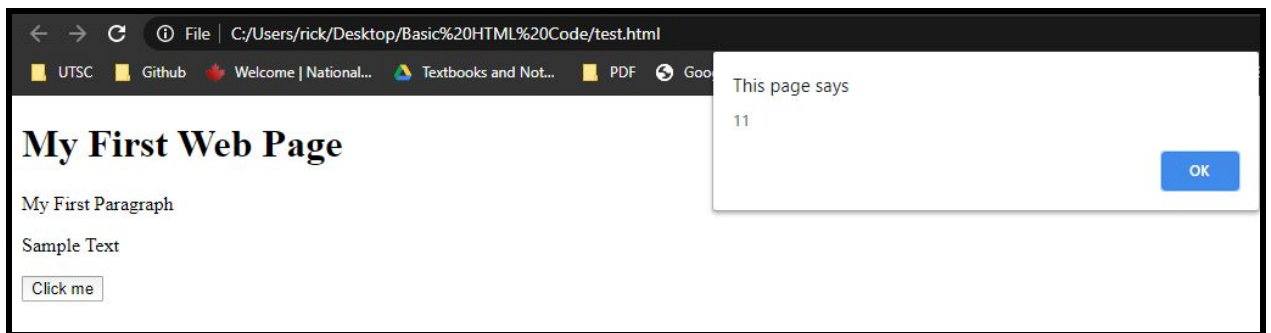
```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="test.js"></script>
  </head>
  <body>
    <h1>My First Web Page</h1>

    <p>My First Paragraph</p>

    <p> Sample Text </p>
    <button onclick=changetext()> Click me</button>

  </body>
</html>
```

```
function changetext(){
  window.alert(5 + 6);
}
```



When you click on the button, `window.alert()` will create a pop-up box with whatever is in the parentheses. In this case, I have `window.alert(5+6)`, so the pop-up box contains 11.

- Using `console.log()`:
- For debugging purposes, you can use the `console.log()` method to display data.
- E.g. Consider the code and output below:

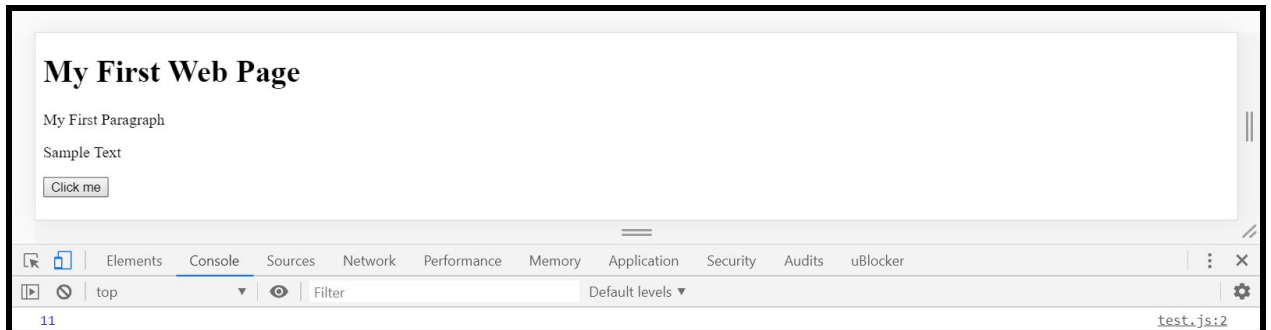
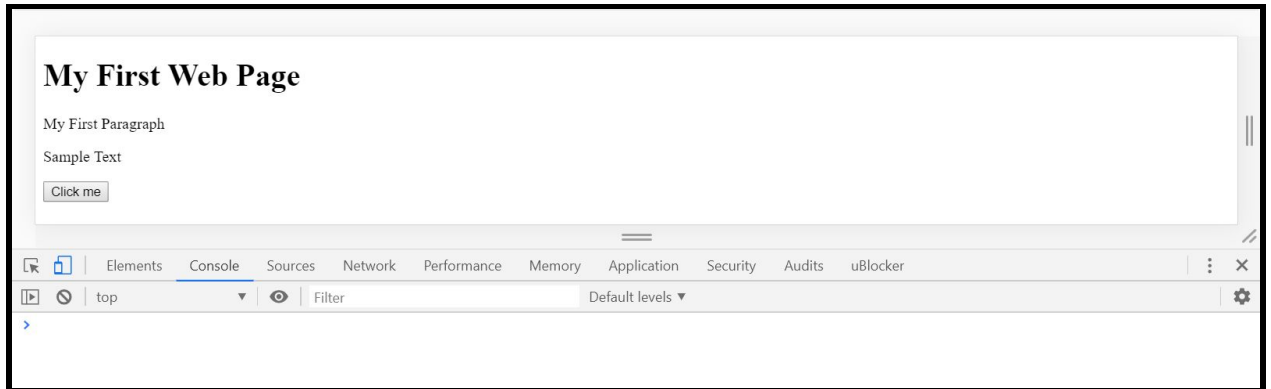
```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="test.js"></script>
  </head>
  <body>
    <h1>My First Web Page</h1>

    <p>My First Paragraph</p>

    <p> Sample Text </p>
    <button onclick=changetext()> Click me</button>

  </body>
</html>

function changetext(){
  console.log(5 + 6);
}
```



Originally, there's nothing in the console log, but when you click on the button you'll see an 11 there.

- **Comments:**
- Single line comments are denoted by *//*.
- Multi-line comments are denoted by */* */*.
- **Operators:**
- **Table of JavaScript Arithmetic Operators:**

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

- **Table of JavaScript Assignment Operators:**

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

- **Table of JavaScript Comparison Operators:**

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

- **Table of JavaScript Logical Operators:**

Operator	Description
&&	logical and
	logical or
!	logical not

- **Table of JavaScript Type Operators:**

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

- **Variables:**

- In JS, we can declare variables using the **var** keyword.
- You can create the variable first and then define it later or do both at once.
- E.g. In (1), we are creating the variable x and then defining it later while in (2), we are doing both at the same time.
 1. `var x`
`x = 4`
 2. `var x = 4`
- Variables declared using var have **function scope**. This means that they can be accessed within the function they are declared in. Furthermore, this includes any other nested blocks like loops, if-statements, nested functions, etc. This is known as **lexical scope**.
- E.g. Consider the code and the output below:

```
function f1(){
  var a = 1

  console.log(a)

  if (true){
    a = 2
    console.log(a)
  }

  console.log(a)
}

f1()
```

1
2
2

Because “a” is a variable, it has function scope, which means that it can be accessed anywhere within the function it is declared in. Furthermore, we are able to access “a” in the if statement. This is an example of lexical scope.

- **Note:** If a variable is declared outside a function, it has **global scope**. This means that all scripts and functions on a web page can access it.
- **Note:** If a variable is declared inside a function, it has **local scope**. They can only be accessed from within the function.

- E.g. Consider the code and the output below:

```
var b = 3;

function f1(){
  console.log(b)

  if (true){
    var a = 2
    console.log(a)
  }

  console.log(a)
}

f1()
console.log(a)
```

3
2
2
✖ ▶ Uncaught ReferenceError: a is not defined at test.js:16

In this example, the variable “b” has global scope, while the variable “a” has local scope. Hence, we are able to access “b” inside f1, but we are not able to access “a” outside of f1.

- **Note:** If you assign a value to a variable that has not been declared, it will automatically become a global variable.
- E.g. Consider the code and the output below:

```
function f1(){
  if (true){
    a = 2
    console.log(a)
  }

  console.log(a)
}

f1()
console.log(a)
```

2
2
2

In this example, we didn’t declare “a” with the keyword var, and as a result, it became a global variable. This is why we are able to access “a” outside of f1.

- **Hoisting:**
- **Hoisting** is JavaScript’s default behavior of moving declarations to the top.
- Remember that var declarations and definitions are separate. All var variables and function declarations are ‘hoisted’ up to the top of their function scope or global scope if it’s not in function, while variable definitions stay in place.

- E.g. Consider the code and output below:

```
function f1(){
  console.log(a)
  var a = 3
}

f1()
```

undefined

What's happening is this:

```
function f1(){
  var a // Declaring "a" at the top of the function.
  console.log(a) //While a has been declared, it hasn't been defined, so its value is undefined
  a = 3 // Definition stays in place.
}
f1()
```

- E.g. Consider the code and output below:

```
function f1(){
  if (true){
    var a = 3
    console.log(a)
  }
  console.log(a)
}

f1()
```

3
3

This is what's really happening:

```
function f1(){
  var a; // Declaring "a" at the top of the function.
  if (true){
    a = 3 // Defining "a"
    console.log(a)
  }
  console.log(a)
}
f1()
```

- E.g. Consider the code and output below:

```
f1()
function f1(){
  var a = 3
  console.log(a)
}
```

3

What's happening is this:

```
function f1(){  
  var a = 3  
  console.log(a)  
}  
f1()
```

Here, f1 is being hoisted to the top of the global scope. This is why we can call it before we have declared it.

- **Use Strict:**
- The keyword **"use strict"** defines that JavaScript code should be executed in "strict mode".
- The "use strict" directive was new in ECMAScript version 5.
- The purpose of "use strict" is to indicate that the code should be executed in "strict mode".
- It helps you to write cleaner code, like preventing you from using undeclared variables.
- Strict mode is declared by adding **"use strict"** to the beginning of a script or a function.
- If it is declared at the beginning of a script, it has global scope. If it is declared inside a function, it has local scope.
- Strict mode makes it easier to write "secure" JavaScript.
- Strict mode changes previously accepted "bad syntax" into real errors.
- As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.
- In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties. However, in strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.
- E.g. Consider the code and output below:

```
function f1(){  
  a = 3  
  console.log(a)  
}  
f1()
```

3

Now, let's put "use strict" at the top of the file:

```
"use strict"  
function f1(){  
  a = 3  
  console.log(a)  
}  
f1()
```

✖ ▶ Uncaught ReferenceError: a is not defined
at f1 (test.js:3)
at test.js:6

We now get an error, because we haven't declared the variable "a".

- Still, with var it's not always easy. There are two new ways to declare variables in ES6 (2015): let and const. The important difference is that they both have **block scope**, meaning that only the current block can access them. However, lexical scope still applies, meaning that any inner block can also access.
- **Let:**
- Let is another way to create a variable. The main difference between let and var is that let has block scope while var has function scope.
- E.g. Consider the code and output below:

```
"use strict"
function f1(){
    if (true){
        var a = 3
    }
    console.log(a)
}
f1()
```

3

```
"use strict"
function f1(){
    if (true){
        let a = 3
    }
    console.log(a)
}
f1()
```

✖ ▶ Uncaught ReferenceError: a is not defined
at f1 (test.js:7)
at test.js:9

The reason why when we did “var a = 3”, it was fine is because var has function scope. This means it can be accessed anywhere in the function. However, let has block scope, which means that it can only be accessed anywhere in its block, which is the if-statement in this case. Hence, when we did console.log(a) outside of the if-statement, it caused an error.

- E.g. Consider the code and output below:

```
"use strict"
function f1(){
    let a = 3
    if (true){
        console.log(a)
    }
}
f1()
```

3

In this case, the block that “a” is in is f1. Hence, anything inside f1 can access “a”. This is why when we do console.log(a) in the if-statement, it doesn't cause an error.

- **Const:**
- Const also has block scope, like let. It is used for variables that will not be re-assigned. I.e. Variables defined with const behave like let variables, except they cannot be reassigned.
- JavaScript const variables must be assigned a value when they are declared.
- **Note:** Const does not define a constant value. It defines a constant reference to a value. Because of this, we cannot change constant primitive values, but we can change the properties of constant objects.
- If we assign a primitive value to a constant, we cannot change the primitive value.
- E.g. Consider the code and output below:

```
"use strict"
function f1(){
  const a = 3
  console.log(a)
  a = 5; // Will cause an error.
}
f1()
```

```
3
✖ ▶ Uncaught TypeError: Assignment to constant variable.
   at f1 (test.js:5)
   at test.js:7
```

- However, we can change the properties of a constant object. But we can not reassign a constant object.
- E.g. Consider the code and output below:

```
"use strict"
function f1(){
  const school = {"Name": "UTSC", "Address": "1265 Military Trail, Scarborough"}
  console.log(school)
  school.Name = "UTSG" // Will not cause an error. You can change a property.
  school.Principal = "William Gough" // Will not cause an error. You can add a property.
  console.log(school)
}
f1()
```

```
▶ {Name: "UTSC", Address: "1265 Military Trail, Scarborough"}
▶ {Name: "UTSG", Address: "1265 Military Trail, Scarborough", Principal: "William Gough"}
```

Here, I am changing the Name property and adding a new property, called Principal, for the school object. Notice that I did not receive any errors.

- E.g. Consider the code and output below:

```
"use strict"
function f1(){
  const school = {"Name": "UTSC", "Address": "1265 Military Trail, Scarborough"}
  school = {"Name": "UTSG", "Address": "1265 Military Trail, Scarborough"}
}
f1()
```

✖ ▶ Uncaught TypeError: Assignment to constant variable.
 at f1 (test.js:4)
 at test.js:6

Here, I am reassigning the object school, which gives me an error.

- We can change the elements of a constant array, but cannot reassign a constant array.
- E.g. Consider the code and output below:

```
"use strict"
function f1(){
  const colors = ["red", "blue", "yellow"]
  console.log(colors)
  colors[0] = "green"
  console.log(colors)
}
f1()
```

▶ (3) ["red", "blue", "yellow"]
 ▶ (3) ["green", "blue", "yellow"]

Here, I am changing the first element of the colors array, which is allowed.

- E.g. Consider the code and output below:

```
"use strict"
function f1(){
  const colors = ["red", "blue", "yellow"]
  colors = ["blue", "red", "yellow"]
}
f1()
```

✖ ▶ Uncaught TypeError: Assignment to constant variable.
 at f1 (test.js:4)
 at test.js:6

Here, I am reassigning the array colors, which gives me an error.

- **Rule:** In this class and everywhere else, default to using const unless you know you will have to re-assign a variable.

- **Rule:** Do not use var, but know how it works for backwards-compatibility.
- **Data Types:**
- JavaScript has **dynamic data types**. This means that the same variable can be used to hold different data types.
- The different data types are listed below:
 1. **String:**
 - A string is a series of characters.
 - Strings are written with quotes. You can use single or double quotes.
 - E.g.


```
var carName1 = "Volvo XC60"; // Using double quotes
var carName2 = 'Volvo XC60'; // Using single quotes
```
 - You can use quotes inside a string, as long as they don't match the quotes surrounding the string.
 - E.g.


```
var answer1 = "It's alright"; // Single quote inside double quotes
var answer2 = "He is called 'Johnny'"; // Single quotes inside double quotes
var answer3 = 'He is called "Johnny"'; // Double quotes inside single quotes
```
 2. **Numbers:**
 - JavaScript has only one type of numbers.
 - Numbers can be written with, or without decimals.
 - E.g.

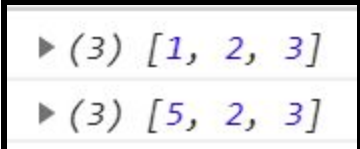

```
var x1 = 34.00; // Written with decimals
var x2 = 34; // Written without decimals
```
 - Extra large or extra small numbers can be written with scientific (exponential) notation.
 - E.g.


```
var y = 123e5; // 12300000
var z = 123e-5; // 0.00123
```
 3. **Boolean:**
 - Booleans can only have two values: true or false.
 4. **Objects:**
 - JavaScript objects are written with curly braces {}.
 - Object properties are written as name:value pairs, separated by commas.
 - E.g.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```
 5. **Arrays:**
 - JavaScript arrays are written with square brackets.
 - Array items are separated by commas. Array indexes are zero-based, which means the first item is [0], second is [1], and so on.
 - E.g.

```
const cars = ["Saab", "Volvo", "BMW"];
```
 - Arrays are mutable. You can change the elements of arrays like such:


```
const a = [1,2,3]
console.log(a)
a[0] = 5
console.log(a)
```



Here, we changed index 0 (the first element) of array a to 5.

- To get the length of an array, you can use the `.length` function.
Syntax: **array_name.length**
- E.g. Consider the code and output below:

```
const a = [0, 2, 4]
console.log(a.length)
```



6. Typeof Operator:

- You can use the JavaScript **typeof operator** to find the type of a JavaScript variable.
- The typeof operator returns the type of a variable or an expression.
- E.g.
typeof "" // Returns "string"
typeof "John" // Returns "string"
typeof "John Doe" // Returns "string"
typeof 0 // Returns "number"
typeof 314 // Returns "number"
typeof 3.14 // Returns "number"
typeof (3) // Returns "number"
typeof (3 + 4) // Returns "number"

7. Undefined:

- In JavaScript, a variable without a value, has the value **undefined**. The type is also undefined.
- E.g. **var car; // Value is undefined, type is undefined**
- Any variable can be emptied, by setting the value to undefined. The type will also be undefined.
- E.g. **car = undefined; // Value is undefined, type is undefined**
- You can empty an object by setting it to undefined.
- E.g.
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = undefined; // Now both value and type is undefined

8. Empty Values:

- An empty value has nothing to do with undefined.
- An empty string has both a legal value and a type.
- E.g. **var car = ""; // The value is "", the typeof is "string"**

9. Null:

- In JavaScript null is "nothing". It is supposed to be something that doesn't exist.
- In JavaScript, the data type of null is an object.
- You can empty an object by setting it to null.
- E.g.
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null; // Now value is null, but type is still an object
- Undefined and null are equal in value but different in type.

10. Primitive Data:

- A **primitive data value** is a single simple data value with no additional properties and methods.

- The typeof operator can return one of these primitive types:
 - String
 - Number
 - Boolean
 - undefined

- E.g.

```
typeof "John" // Returns "string"
typeof 3.14 // Returns "number"
typeof true // Returns "boolean"
typeof false // Returns "boolean"
typeof x // Returns "undefined" (if x has no value)
```

11. Complex Data:

- The typeof operator can return one of two complex types:
 1. Function
 2. Object
- The typeof operator returns "object" for objects, arrays, and null.
- The typeof operator does not return "object" for functions. Instead, it returns "function" for functions.
- **Note:** The typeof operator returns "object" for arrays because in JavaScript arrays are objects.
- E.g.


```
typeof {name:'John', age:34} // Returns "object"
typeof [1,2,3,4] // Returns "object"
typeof null // Returns "object"
typeof function myFunc(){} // Returns "function"
```
- **Note:** JavaScript evaluates expressions from left to right.
- **Note:** When adding a number and a string, JavaScript will treat the number as a string.
- **If Statements:**
- Syntax:


```
if (condition){
  // Code inside the if block.
}
else if (condition){
  // Code inside the else if block.
}
else {
  // Code inside the else block.
}
```
- You must have an if block, you can have as many else if blocks as you want, the minimum number of else if blocks is 0, and you can have at most 1 else block, the minimum number of else blocks you can have is 0.
- The conditions are evaluated from top to bottom, and once the first condition is true, the code inside that block will run and once it's done running, it will exit the if-statement.
- **For loops:**
- Syntax:


```
for (statement 1; statement 2; statement 3){
  // Code to run inside for loop.
}
```

- Statement 1 is executed one time before the execution of the code block.
- Statement 2 defines the condition for executing the code block.
- Statement 3 is executed every time after the code block has been executed.
- E.g.

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

Here, statement 1 is "let i = 0", statement 2 is "i < 5" and statement 3 is "i++".

- **For In Loops:**
- The JavaScript for/in statement loops through the properties of an object.
- E.g. Consider the code and output below:

```
"use strict"
function f1(){
  const person = {"Firstname":"Rick", "Lastname":"Lan"}
  for (const item in person){
    console.log(item + ": " + person[item])
  }
}
f1()
```

```
Firstname: Rick
Lastname: Lan
```

- **For/Of Loop:**
- The JavaScript for/of statement loops through the values of an iterable objects such as Arrays, Strings, Maps, NodeLists, and more.
- The for/of loop has the following syntax:

```
for (variable of iterable) {
  // code block to be executed
}
```

variable: For every iteration the value of the next property is assigned to the variable. Variables can be declared with const, let, or var.

iterable: An object that has iterable properties.

- E.g. Consider the code and output below:

```
"use strict"
function f1(){

    // Using for/of loop to loop through an array.
    const color = ["red", "blue", "yellow"]
    for (const item of color){
        console.log(item)
    }

    // Using for/of loop to loop through a string.
    const string = "ABCDEF"
    for (const letter of string){
        console.log(letter)
    }
}
f1()
```

red
blue
yellow
A
B
C
D
E
F

- **While loops:**
- The while loop loops through a block of code as long as a specified condition is true.
- Syntax:

```
while (condition) {  
    // code block to be executed  
}
```

- E.g. Consider the code and output below:

```
"use strict"
function f1(){
    let i = 0
    while(i < 10){
        console.log(i)
        i++
    }
}
f1()
```

0
1
2
3
4
5
6
7
8
9

- **Do/While Loop:**
- The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- Syntax:

```
do {
    // code block to be executed
}
while (condition);
```

- E.g. Consider the code and output below:

```
"use strict"
function f1(){
    let i = 0
    do{
        console.log(i)
        i++
    }while(i > 5)
}
f1()
```

0

- **Functions:**
- A **function** is a block of code designed to perform a particular task.
- A function is executed when something **invokes** it (calls it).
- A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses (). Furthermore, the code to be executed by the function is placed inside curly brackets {}.
- Syntax:

```
function function_name(parameter1, parameter2, ..., parameterN){
    // Code inside the function
}
```
- To invoke a function, put the name of the function, followed by parentheses () and with all the parameters inside the ().
Note: Accessing a function without () will return the function object instead of the function result.
- E.g. Consider the code and output below:

```
"use strict"
function f1(a, b, c){
    console.log(a+b+c)
}
f1(1, 2, 3) // Invokes function f1.
```

6

- E.g. Consider the code and output below:

```
"use strict"
function f1(a, b, c){
    return(a+b+c)
}
console.log(f1(1, 2, 3)) // Prints the return value of f1.
console.log(f1) // Prints the function object.
```

```

6
f f1(a, b, c){
  return(a+b+c)
}

```

Here, in the first console.log(), because I have the parentheses with the arguments inside, it will print out the return value of f1. However, in the second console.log(), I only have the function name, so it only prints the function object.

- Functions in JS are **first-class objects**, meaning that they can be stored in a variable, passed as an argument to a function and returned from a function. I.e. They are essentially used as a value anywhere values are used.
- Functions can be passed around without names. These are known as **anonymous functions**. We can call them using **Immediately Invoked Function Expressions**. **Immediately-Invoked Function Expressions (IIFE)** are functions that are executed immediately after being defined. We can make any function expression an IIFE by wrapping it in parentheses, and adding a following pair of parentheses at the end.
- E.g. Consider the code and output below:

```

(function (){
  console.log('a')
  console.log('b')
})();

```

a
b


```

(function test(){
  console.log('a')
  console.log('b')
})();

```

a
b

You can make both anonymous and non-anonymous functions IIFEs.

- **Closure:**
- **Closure** is a feature in JavaScript where an inner function has access to the outer (enclosing) function's variables. It allows function/block scopes to be preserved even after they finish executing.

- E.g. Consider the code and output below:

```
function outer(){
  const a = 3
  function inner(){
    console.log(a)
  }
  return inner
}
const x = outer()
console.log(x)
x()
```

```
f inner(){
  console.log(a)
}
3
```

Here, when we do `const x = outer()`, `x` gets the inner function object since `outer` returns `inner`, not `inner()`. Hence, when we do `console.log(x)`, it prints the code for `inner()`. Next, when we do `x()`, we're running `inner()`. Note that `outer()` has already been exited, so normally, any variables in it that's outside of `inner()` should be inaccessible, but because of closure, `inner()` still has access to variable "a". `inner()` will carry with it all variables in the scope when it was defined, in this case, "a".

- E.g. Consider the code and output below:

```
function outer(){
  let a = 3
  function inner(){
    console.log(a)
  }
  a = 5;
  return inner
}
const x = outer()
console.log(x)
x()
```

```
f inner(){
  console.log(a)
}
5
```

The variable "a" can still change in `outer()`, and `inner()` will register those changes in the carried scope until `outer()` returns.

- **Objects:**
- An object in JS is simply a set of key-value pairs.
- Keys are called **properties**.
- Values can be of any type, even complex data structures.
- Syntax: `var obj_name = {key1:value1, key2:value2, ...}`
- E.g.
`var car = {type:"Fiat", model:"500", color:"white"};`
- You can access object properties in two ways:
 1. `objectName.propertyName`
 2. `objectName["propertyName"]` (You must have the quotation marks.)

- E.g. Consider the code and output below:

```
"use strict"
function f1(){
  const person = {"FirstName":"Rick", "LastName":"Lan"}
  console.log(person)
  // Accessing the properties of person.
  console.log(person.FirstName, person.LastName)
  // Accessing the properties of person in another way.
  console.log(person["FirstName"], person["LastName"])
}
f1()
```



```
▶ {FirstName: "Rick", LastName: "Lan"}
Rick Lan
Rick Lan
```

- Properties can be added and changed.
Note: If you created an object using `const`, you can still change and/or add properties. You just can't reassign the object.
- Objects can also have **methods**. Methods are actions that can be performed on objects. A method is a function stored as a property. You access an object method with the following syntax: `objectName.methodName()`. If you access a method without the `()` parentheses, it will return the function definition.

- E.g. Consider the code and output below:

```
"use strict"
function f1(){
  const person = {
    "FirstName": "Rick",
    "LastName": "Lan",
    "FullName": function(){
      return this.FirstName + " " + this.LastName
    }
  }
  console.log(person)
  // Accessing the properties of person.
  console.log(person.FirstName, person.LastName)
  // Accessing the properties of person in another way.
  console.log(person["FirstName"], person["LastName"])
  // Prints the return value of the FullName method.
  console.log(person.FullName())
  // Prints the function definition.
  console.log(person.FullName)
}
f1()
```

```
► {FirstName: "Rick", LastName: "Lan", FullName: f}
Rick Lan
Rick Lan
Rick Lan
f (){
    return this.FirstName + " " + this.LastName
}
```

- **This:**
- The JavaScript **this** keyword refers to the object it belongs to. It refers to the containing object of the call-site of a function, not where the function is defined.
I.e. **This** is the object that is executing the current function.
- **This** is context-dependent. The value of **this** is not obvious from reading function definition.
- If a code is being executed as part of a simple function call, then **this** refers to a global object. The window object is the global object in the case of the browser. If strict mode is enabled for any function, then the value of **this** will be marked as undefined as in strict mode. The global object refers to undefined in place of the windows object.

- E.g. Consider the code and output below:

```
"use strict"
function f1(){
    console.log(this)
}
f1()
```

```
undefined
```

Here, we are using "use strict", so the value of **this** is undefined.

```
function f1(){
    console.log(this)
}
f1()
```

```
▶ Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, ...}
```

Here, we removed "use strict", so the value of **this** is the window object.

- When a function is invoked with the **new** keyword, then the function is known as a **constructor function** and returns a new instance. In such cases, the value of **this** refers to a newly created instance.
- E.g. Consider the code and output below:

```
function Person(fn, ln) {
    this.first_name = fn;
    this.last_name = ln;

    this.displayName = function() {
        console.log("Name:" + " " + this.first_name + " " + this.last_name);
    }
}

let person = new Person("John", "Reed");
person.displayName(); // Prints Name: John Reed
let person2 = new Person("Paul", "Adams");
person2.displayName(); // Prints Name: Paul Adams
```

```
Name: John Reed
```

```
Name: Paul Adams
```

- E.g. Consider the code and output below:

```
function person(){
    console.log(this)
}
```

```
let x = new person()
```

```
▶ person {}
```

- In JavaScript, the property of an object can be a method or a simple value. When an object's method is invoked, then **this** refers to the object which contains the method being invoked.
- E.g. Consider the code and output below:

```
function Person() {
  const person = {
    "Fn": "Rick",
    "Ln": "Lan",
    "fullname": function(){
      console.log(this.Fn + " " + this.Ln) // Here, this refers to the person object.
    }
  }

  person.fullname()
}

Person()
```

Rick Lan

- E.g. Consider the code and output below:

```
const person = {
  firstname: "rick",
  lastname: "lan",
  fullName: function(){
    console.log(this)
  }
}

person.fullName()
```

► {firstname: "rick", lastname: "lan", fullName: f}

Here, it shows that **this** is referring to the person object.

- Summary of **this**:

Item	What this refers to
Method (A function in an object)	Refers to the object the method is in.
Function without "Use Strict"	Refers to the global variable (In most browsers, the global variable is Windows)
Function with "Use Strict"	Refers to "undefined"
Constructor functions	Refers to the newly created instance.

- A function in JavaScript is also a special type of object. Every function has call, bind, and apply methods. These methods can be used to set a custom value to this in the execution context of the function.
- Call:
- The call() method is a predefined JavaScript method.
- With the call() method, you can write a method that can be used on different objects. I.e. It can be used to invoke (call) a method with an owner object as an argument (parameter).
I.e. With call(), an object can use a method belonging to another object.
- Syntax: `function_name.call(obj1, obj2, ..., argument1, argument2, ...)`
- E.g. Consider the code and output below:

```
const person = {
  fullName: function(){
    console.log(this.firstname + " " + this.lastname)
  }
}

const person1 = {'firstname': "Rick", 'lastname': "Lan"}

person.fullName.call(person1)
```

Rick Lan

Here, `this` refers to person1.

- E.g. Consider the code and output below:

```
const person = {
  fullName: function(school){
    console.log(this.firstname + " " + this.lastname + " " + school)
  }
}

const person1 = {'firstname': "Rick", 'lastname': "Lan"}

person.fullName.call(person1, "UTSC")
```

Rick Lan UTSC

Here, we added arguments to our function. All arguments must go after all objects. Furthermore, `this` is still referring to person1.

- Apply:
- With the apply() method, you can write a method that can be used on different objects.
- The apply() method is similar to the call() method. The difference is that the call() method takes arguments separately while the apply() method takes arguments as an array.
- The apply() method is very handy if you want to use an array instead of an argument list.

- E.g. Consider the code and output below:

```
const person = {
  fullName: function(){
    console.log(this.firstname + " " + this.lastname)
  }
}

const person1 = {'firstname': "Rick", 'lastname': "Lan"}

person.fullName.apply(person1)
```

Rick Lan

Here, I don't have arguments, so apply behaves the same way as call().

- E.g. Consider the code and output below:

```
const person = {
  fullName: function(school){
    console.log(this.firstname + " " + this.lastname + " " + school)
  }
}

const person1 = {'firstname': "Rick", 'lastname': "Lan"}

person.fullName.apply(person1, ["UTSC"])
```

Rick Lan UTSC

Here, I do have arguments, and I enter the arguments in an array as opposed to a list of arguments as done in call().

- Bind:
- Bind creates a new function that will force the **this** inside the function to be the parameter passed to bind().
I.e. With bind(), you are binding an object to a function and referencing it using the **this** keyword.
- The bind method returns a new function, allowing you to pass in a this array and any number of arguments. Use it when you want that function to later be called with a certain context like events.
- Use bind() when you want that function to later be called with a certain context, useful in events. Use call() or apply() when you want to invoke the function immediately, and modify the context.

- E.g. Consider the code and output below:

```
const customer1 = { name: 'Leo' };
const customer2 = { name: 'Nat' };

function greeting(text) {
  console.log(text + " " + this.name);
}

const helloLeo = greeting.bind(customer1);
const helloNat = greeting.bind(customer2);

console.log(helloLeo)
console.log(helloNat)

helloLeo('Hello'); // Prints Hello Leo
helloNat('Hello'); // Prints Hello Nat
```

```
f greeting(text) {
  console.log(text + " " + this.name);
}
f greeting(text) {
  console.log(text + " " + this.name);
}
Hello Leo
Hello Nat
```

- **Events:**
- An HTML **event** can be something the browser does, or something a user does.
- Here are some examples of HTML events:
 - An HTML web page has finished loading.
 - An HTML input field was changed.
 - An HTML button was clicked.
- Table of HTML events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page