Basics of JS

JavaScript - An Introduction

JavaScript is a text-based programming language used both on the client-side and server-side. It allows you to make web pages that are interactive in nature. JavaScript gives web pages interactive elements that engage a user. Suppose you want a user to pause or play a music video, you need interactive buttons, which send the command of pausing and playing to your website. Common examples of JavaScript that you might use every day include the search box on Amazon, songs on music websites or a profile search box on Instagram.

To recap, JavaScript adds a certain behavior to your websites.

For many years, JavaScript was only functionable on some browsers. Microsoft's Internet Explorer had no support for JavaScript until much later. Instead, Microsoft created its own proprietary client-side script called JScript. In the early days of Web development, programmers who wished to create dynamic websites were often forced to choose one browser family over the other. This was less than ideal because it made the Internet less universally accessible.

What can it do for you?

1. JavaScript loads content into the document (DOM) if and when the user needs it, without having the need to reload the entire web page — this is commonly referred to as Ajax. **(You will learn this soon enough)**
2. JavaScript can test for what is possible in your browser and react accordingly — this is called Principles of unobtrusive JavaScript. It is also known as defensive scripting.

3. JavaScript can help fix browser problems or patch holes in browser support. Common use of this is fixing CSS.
4. While JavaScript is a client-side language, some of its most powerful features involve asynchronous interaction with a remote server. Asynchronous simply means that JavaScript is able to communicate with the server in the background without interrupting the user interaction taking place in the foreground.

   Take a search engine for example. Today, search engines almost all have an autocomplete function. The user begins typing a word into the search box and a list of possible search terms or phrases appears below. The experience is seamless. Suggested search terms appear without reloading the page.

   In the background, JavaScript reads the letters as the user types, sends those letters to a remote server and the server sends suggestions back.

   The software on the server side analyzes the words and runs algorithms to anticipate the user's search term. Such programs are diabolically large and complex. The JavaScript on the client's machine is as simple and small as possible so as not to slow down the user's interaction. The communication between JavaScript and the server-side program is limited by the user's bandwidth. This is why developers prioritize efficiency in JavaScript functions and make the amount of data communicated between the programs as small as possible.

   Only once the user selects a search term does the entire page reload and produce the search results. Engines such as Google have reduced or eliminated the need to reload, even for that step. They simply produce results using the same asynchronous process, **which you will learn in the Backend Development part of this course.**

JS DATA TYPES

There are **7 data types** available in JavaScript. You need to remember that JavaScript is **loosely typed**(or **dynamic language**), so any value can be assigned to variables in JavaScript.

According to latest ECMAScript standard there are **6 primitive data types** and **1 object** -

- **Number** - represents integer and floating values
- **String** - represents textual data
- **Boolean** - logical entity with values as 'true' and 'false'
- **Undefined** - represents variable whose value is not yet assigned
- **Null** - represents intentional absence of value
- **Symbol** - represents unique value
- **Object** - represents key value pair

All of them except Object have **immutable values**, i.e. the values which cannot be changed.

Some important information about the data types have been described below -

- **Number** represents variables whose value is either an integer or float. It can have numbers in the range between **$-(2^{53}-1)$** and **$(2^{53}-1)$**. Other than integer and float numbers, it has three symbolic values: `+Infinity`, `-Infinity`, and `NaN`. It uses 2 constants - `Number.MAX_VALUE` and `Number.MIN_VALUE`. Both of these constants lie between +Infinity and -Infinity. Eg., 10, -25.2, 8.321.

- **String** represents textual data. String contains elements that can be accessed using the **index**. The **first element has index 0**. Eg., "hello", "1234", "12here". You can access each element of the string like - `str="HelloWorld"`, then `str[1]` will output 'e' on the console.

SOME IMPORTANT VALUES

Below are some important values that are defined in brief-

- **undefined**

'undefined' is the value assigned to the variable that has not yet been assigned any value. We can also **explicitly assign an 'undefined' value** to a variable, but that does not make any sense due to its meaning.
Eg., `var a;` defines a variable that has not been assigned any value. If you write on the console as - `console.log(a);`, then **'undefined'** is printed.

- **null**

  'null' is the value that represents a reference that points to a non-existent object or address. This means that there is an absence of a value. The **data type for null value is "Object".**

- **NaN**

  'NaN' means **Not-A-Number**. So, if any e***xpression fails to return a number***, then ***'NaN'*** is printed on the console. Eg., (`12 - "abc"`) cannot be evaluated to a number, so 'NaN' is printed on the console. The ***data type for 'NaN' is "Number".***

- **Infinity**

  Infinity is a variable in global scope. Its ***value is greater than any other number***. It literally works as Infinity, i.e. it has the same property as in Mathematics. Any numeric expression evaluated to a number outside the range of the number, prints 'Infinity' on the console.

TYPEOF OPERATOR

'typeof' operator is used to ***return the name of the data type to which the value belongs to***. It returns the name of the data type in the form of a string. There are two syntax for using **typeof** operator -

```
typeof operand
```

OR

```
typeof(operand)
```

The parentheses are optional. Operand can be a ***variable*** or ***some value*** or ***an expression.***
Below are some of the data types and their return values of 'typeof' -

S.No.	Data Type	Return value of 'typeof'
1.	Number	"number"
2.	String	"string"
3.	Boolean	"boolean"
4.	Undefined	"undefined"
5.	Null	"object"
6.	Symbol	"symbol"
7.	Object	"object"

EXTRA:
You can read typeof in detail from the below link -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof

OPERATORS IN JS

Operators are used to perform some action on operands. Operand can be of any data type, eg. number, boolean, string, etc..

Some of the operators are described below -

- **Arithmetic Operators**

  Arithmetic operators take numerical values as operands and evaluate them to a single numerical value. Some of the arithmetic operators are -

  - **Addition(+)** - It adds the numerical operands and is also used for string concatenation. It will add numerical and string operand to a number if the string can be converted to a number. Else it concatenates them.
  - **Subtraction(-)** - It subtracts the two numeric operands. If any one of them is not a number or cannot be converted to a number, then **'NaN'** is printed.

- **Division(/)** - It divides the first operand with the second operand. If the second operator is '+0' or '-0', then **'Infinity'** and **'-Infinity** are printed respectively. If they are not divisible **'NaN'** is printed. Try dividing 0 by 0 and see what happens!
- **Multiplication(*)** - It multiplies the two numeric operands. Any number multiplied with Infinity prints **'Infinity'**. See what happens when **Infinity** is multiplied with **0**.
- **Remainder(%)** - It finds the remainder left after division. If one of the operands is **'Infinity'** or **'NaN'**, then **'NaN'** is printed.

EXTRA:

You can learn more about arithmetic operator from this link - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators

- **Assignment Operators**

Assignment operators are used to **assign value of the right operand/expression to the left operand.** The simplest assignment operator **is equal (=),** which assigns the right operand value to left operand.
Eg., `var x=10; var y=x;` will assign value 10 to y.

Other assignment operators are ***shorthand operations of other operators*.** They are called **compound assignment operators.** Some of them with their meaning (i.e. the extended version of these operations) are provided below -

S.No.	Name	Shorthand Operator	Meaning
1.	Addition Assignment	x +=y	x = x + y
2.	Division Assignment	x /=y	x = x / y
3.	Exponentiation Assignment	x **=y	x = x ** y
4.	Right Shift Assignment	x >>=y	x = x >> y
5.	Bitwise XOR Assignment	x ^=y	x = x ^ y

EXTRA:

To look for other assignment operators, visit the link below - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Overview

- **Increment and Decrement Operator**

The **increment operator** increments the value of the numeric operand by one.
The **decrement operator** decreases the value of the numeric operand by one.

There are two ways to use increment and decrement -

- **Using postfix (x++ or x--)** - the value is returned first and then the value is incremented or decremented.
- **Using prefix (++x or --x)** - the value is first incremented or decremented and then returned.

Eg., `var x=10; console.log(++x);` will print **11** on the console.

- **Comparison Operators**

The **comparison operators** are used to ***compare two values with each other.***
The **equality operator (==)** is used to ***compare the two values***, if they are equal or not. But the values are not directly compared. First, ***they are converted to the same data type and then the converted content is compared.***
Eg., `"1" == 1` evaluates to `true`, even though the first one is a 'string' and the other is a 'number'.
There is another comparison operator **(===)** known as a **strict equality operator**. It ***checks both the data type and the content***. If the data type is not equal, it returns false.
So `"1" === 1` now evaluates to `false`.

Other comparison operators are -

- **Inequality (!=)** - It returns the opposite result of the equality operator.
- **Strict Inequality (!==)** - It returns the opposite result of the strict equality
- **Greater Than (>)** - It returns true if left operand is greater than the right one
- **Greater Than or Equal (>=)** - It returns true if the left operand is greater than or equal to the right one.
- **Less Than (<)** - It returns true if left operand is less than the right one
- **Less Than or Equal (<=)** - It returns true if the left operand is less than or equal to the right one.

EXTRA:

You can get a more detailed comparison and working of equality operators from the link below -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

- **Logical Operators**

The **logical operators** use boolean values as their operands. These operands are mostly expressions that evaluate to ***'true'*** or ***'false'***.

There are three logical operators -

- **Logical AND (&&)** - returns ***'true'*** if both operands/expression are true, else ***'false'***. If the first expression is false, the second expression is not evaluated and ***'false'*** is returned. Eg., `false && true` returns `false`.
- **Logical OR (||)** - returns ***'true'*** if any of the operands/expression is true, else ***'false'.*** Eg., `false || true` returns `true`.
- **Logical NOT (!)** - returns the opposite boolean value to which the expression is evaluated to. Eg., , `!false` returns `true`.

EXTRA:

The below link provides a detailed view of the logical operators -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators

- **Bitwise Operators**

The **bitwise operators** treat operands as sequences of 32 bits binary representation(0 and 1).

The several bitwise operators are -

- **Bitwise AND (&)** - returns 1 for each bit position where both bits are 1s. Eg., (5 & 13 = 5) is evaluated as (0101 & 1101 = 0101).
- **Bitwise OR (|)** - returns 1 for each bit position where either bit is 1. Eg., (5 | 13 = 13) is evaluated as (0101 | 1101 = 1101).
- **Bitwise XOR (^)** - returns 1 for each bit position where either bit is 1 but not both. Eg., (5 ^ 13 = 8) is evaluated as (0101 ^ 1101 = 1000).
- **Bitwise NOT (~)** - returns the inverted bits of operand. This means that 0 becomes 1 and vice versa.
- **Left Shift (<<)** - shifts bits to the left and inserts 0s from right.
- **Sign-propagating Right Shift (>>)** - shifts bits to the right and inserts either 0s or 1s from the left according to the sign of the number ('0' for positive and '1' for negative).
- **Zero-fill Right Shift (>>>)** - shifts bits to the right and inserts 0s from left.

EXTRA:

You can get details of bitwise operator from the below link -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators

NOTE: *You need to be careful while working with operators because they work in precedence. So you might use round brackets to group them according to your priority of execution. Also, the round brackets (called as* ***grouping operator****) has the highest precedence. You can check the link below to see the precedence of the operators -* *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#Table*

EXTRA:

You can read about all the operators from the link below - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Operators

CONDITIONALS

Conditional statements are used to ***run different sets of statements according to the conditions.*** This means that based upon different conditions, we can perform different actions. These are the same as used in other languages.

Some of the conditional statements have been described below -

- **If statements**

  The **'if'** statement specifies a block of statements (JavaScript code) that gets executed when the **'condition'** specified in the 'if' statement evaluates to true.

  The syntax for the 'if' statement is -

```
if (condition) {
   ...STATEMENTS...
}
```

  First, the condition is evaluated and if it is `true`, then the statements inside it are executed. Else if the condition is `false`, statements inside **'if'** are not executed.

- **if-else Statements**

  The **'if'** statement specifies a block of statements (JavaScript code) that gets executed when the **'condition'** specified in the **'if'** statement evaluates to true. Else, if the ***condition is false*** then the statements inside the ***'else' block are executed.***
  The syntax for the 'if-else' statement is -

```
if (condition) {
   ...STATEMENTS...
} else {
   ...SOME OTHER STATEMENTS...
}
```

- **else-if statements**

  The above two works for only two options i.e. whether the condition is `true` or `false`. Using **else-if** we can define ***more than one conditions*** and form a chain. The first condition that returns `true` gets executed, else the last 'else' block is executed.

```
if (condition - 1) {
   ...STATEMENTS...
} else if (condition - 2) {
   ...SOME OTHER STATEMENTS...
}...
...
...
else if (condition - (n)) {
   ...SOME OTHER STATEMENTS...
} else {
   ...EXECUTE IF NO CONDITION IS SATISFIED...
}
```

  Now the first condition is checked. If it evaluates to ***true***, it gets executed and other ***'else-if'*** blocks are not checked. If the first condition is ***false***, then the second condition is checked.

  Like this, each ***'if'*** condition is checked one by one, until one is ***true*** or ***'else'*** block is reached. If any one of them is ***true***, the block associated with it gets executed. Else the ***'else'*** block is executed.

- **Switch statements**

  The **switch** statements are similar to **else-if** statements, but ***works on a single expression***. This expression evaluates to different values which decide which block of code needs to be executed.

  The syntax for 'switch' statement is **-**

```
switch (expression) {
```

```
    case choice1:
        Run
        if expression = choice1
        break;
    case choice2:
        run
        if expression = choice2
        break;
        ...
        ...
        ...
        case choice(n):
            run
        if expression = choice(n)
        break;
    default:
        run
        if no choice is found
}
```

The expression gives some result which is checked against the choices. If any of them matches, its set of statements get executed. Else the last **'default'** block gets executed.

We have provided a ***'break***;' statement after each case is completed, to top the execution of the ***'switch'*** block. If **break** is not present, then the case after the one that matches also gets executed until ***'break***;' is found or switch block does not end.

EXTRA:

You can read about these conditional statements from the link below -
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals

LOOPS

Loops are used to do **something repeatedly.** Let's say that you need to print 'n' numbers, then you can use a loop to do so. There are many different kinds of loops, but they almost do the same thing. Their use varies depending on the type of situation, where one loop will be easy to implement over another.

The various loops that JavaScript provide are described below -

- **for statement**
  A **for** loop is used to ***repeat something until the condition evaluates to false***. The syntax for **'for'** loop is -

```
for ( [initializationStatement] ; [condition] ; [updateStatement] )
{
   ... STATEMENTS ...
}
```

  The **`initializationStatement`** used to initialize loop counters.
  The **`condition`** is the expression that is evaluated to boolean value 'true' or 'false'.
  The **`updateStatement`** is used to update the loop counters.

  Some points to remember about for loop in JS are -

  - The **`initializationStatement`** is optional and you can initialize these statements before the 'for' loop.
  - You can provide more than one **`initializationStatement`** using **comma**(",")as separator.
  - The loop executes until the condition becomes false. If you omit the **`'condition'`**, then it is evaluated to be true.
  - The **`updateStatement`** is also optional and can be present inside the loop. If you do not provide any update, the loop will execute infinitely.

  Eg., for(var i=0; i<=10; i++) { ... }; is loop where we have initialization, condition and update statements.
  Another example where there is no initializer, the loop will look like this - for(; i<=10; i++) { ... };. Here you can see a semicolon(;) at the start and then the condition. You need to provide semicolons for each part of the loop, even if the options are missing. Eg., for(vari=0; ; i++);

- **while statement**
  The **while** statement ***executes the statements until the condition is not false***. The syntax for **'while'** loop is -

```
while (condition) {
   ...STATEMENTS...
}
```

First, the **`condition`** is evaluated and if it is **true**, then the statements are executed and again the **`condition`** is tested. The execution stops when the **`condition`** returns **false**.

***NOTE**: You have to provide an update expression inside the loop, so that it does not repeat infinitely.*

- **do...while statement**
  The **do-while** loop is similar to while loop, except that ***the statements are executed at least once.*** The syntax for ***'do-while'*** loop is-

```
do {
   ...STATEMENTS...
} while (condition);
```

First, the statements are executed without checking the **`condition`**. After that the **`condition`** is checked and if it is **true** the statements are executed again.

Advantages of JS

1. JavaScript is fast and reliable on the client side, as it runs immediately within the browser. It is unhindered by network calls to a backend server, unless outside resources are utilized.
2. It is very easy to learn and implement
3. It is a popular scripting language, accepted universally on the web.
4. JavaScript plays nicely with other languages and can be used in a huge variety of applications.
5. Using JavaScript, we may create rich interfaces.
6. It adds dynamic nature to your web pages and provides a direct interface between actions made by the user with the desired effect.

Disadvantages of JS

1. Since the script code is executed on a user's computer, in some cases it may be exploited for phishing or hacking purposes. A skilled programmer can inject malicious code into the script.
2. JavaScript is sometimes interpreted differently by different browsers. This makes it somewhat difficult to write cross-browser code.

Functions and Arrays

FUNCTIONS

JavaScript is based on **functional programming.** Therefore, functions are fundamental building blocks of JavaScript.

Function contains a set of statements that ***perform some task.***

You define a function using the **'function'** keyword**.** The syntax for creating a function is –

```
function functionName(parameters) {
   // SET OF STATEMENTS
}
```

You can call the above function as –

```
functionName(arguments);
```

The function execution stops either when all the statements have been executed or a return statement is found. The **return statement** stops the execution of the function and returns the value written after the return keyword. . A function may or may not return some value after its execution.

Function Arguments

JavaScript is a dynamic language and therefore it allows passing different number of arguments to the function and does not give error in these condition -

- **Passing fewer arguments -** In this case, when few arguments are passed, the other parameters that do not get any value assigned to them get value **'undefined'** by default.
- **Passing more arguments -** In this case, when more arguments are passed, the extra arguments are not considered.

Eg., when you have a add function given below –

```
function add(a, b, c) {
   return a + b + c;
```

```
}
console.log(add(10, 20)); // Prints - NaN
console.log(add(10, 20, 30, 40)); // Prints - 60
```

- **Arguments Object**

  Although we will study about Objects in the next session, in context to functions you can have access to arguments passed to function.

  The **argument** object is used to store the arguments passed to the function in an array like object. You can either use the parameter name or argument object to access the values. It is helpful in cases when you don't know the number of arguments passed to the function.

  The number of arguments can be found using - "**arguments.length**". The arguments can be accessed using the brackets notation as used in arrays - "**arguments[i]**", where **'i'** is a number starting from 0.

  Eg., the below code prints all the values of the passed to function –

```
function printAll() {
   for (var i = 0; i < arguments.length; ++i) {
       console.log(arguments[i]);
   }
}
printAll('mango', 'apple'); // Prints - mango apple
printAll('fire', 'water', 'ice', 'gas'); // Prints - fire water ice
gas
```

- **Default Parameters**

  In JavaScript, if you ***pass less arguments*** than in the function, the ***remaining parameters*** default to **'undefined'**. But you can also set your own default values to them.

  Eg., the below function uses the default parameters when their values is not passed–

```
function findInterest(p, r = 5, t = 1) {
   console.log("Interest over", t, "years is:", (p * r * t) / 100);
}
findInterest(1000); // Prints - Interest over 1 years is: 50
findInterest(1000, 7); // Prints - Interest over 1 years is: 70
findInterest(1000, 8, 2); // Prints - Interest over 2 years is: 160
```

- **Rest Parameters**

The **rest parameter** syntax **(...variableName)** is used to represent an indefinite number of arguments. It is defined in an array like structure.

Eg., let's say in a situation when we want to add at least three numbers, then we can use something like the below function –

```
function addAtLeastThree(a, b, c, ...numbers) {
   var sum = a + b + c;
   for (var i = 0; i < numbers.length; ++i) {
       sum += numbers[i];
   }
   return sum;
}
console.log(addAtLeastThree(10, 20, 30, 40, 50));
// Prints - 150
```

- **Hoisting**

JavaScript provides a very interesting feature called **hoisting**. This means that you can use a variable or function even before it's declaration.

Hoisting is a mechanism in JavaScript where variables and function declarations are moved to the top of their scope before code execution.

***NOTE**: If you use a variable or function and do not declare them somewhere in the code, then it will give an error.*

You can use hoisting with both variables and function as shown below

- **Variable Hoisting**

  Variable hoisting means that you can use a variable even before it has been declared.

  Eg., you can use variable as -

```
console.log(a); // Prints - undefined
...
/* ... Other JavaScript Statements ... */
...
var a = 10;
```

  The above prints **'undefined'** because only the variable declaration is moved to the top and not its definition.

- **Function Hoisting**

  Function hoisting means that you can **use function even before function declaration** is done.
  Eg., you can use function like this –

```
console.log(cube(3)); // Prints - 27
...
/* ... Other JavaScript Statements ... */
...
function cube(n) {
   return n * n * n;
}
```

  But you cannot use function hoisting when you have used function expression. If you use function expression and use function hoisting, then the result will be **undefined**!

  For eg:

```
console.log(a);
var a = function(){
    console.log("Inside function");
}
//output => undefined
```

SCOPE

The **scope** of a variable is part of code where that variable is accessible. **Scope of variable** depends where they are defined.

- **Global Scope**

  When a variable is ***defined*** globally(i.e. not in any function), ***it can be used anywhere in the code.*** For eg.,

```
var i = 10;
function abc() {
   console.log(i); // This will print 10
}
console.log(i); // This will print 10
```

- **Function Scope**

  When a variable is ***defined*** inside a function, it is accessible only within that function. Eg., see the below function –

```
var i = 0;
function abc() {
   var j = 1;
   console.log(i); // This will print 0
   console.log(j); // This will print 1
}
console.log(j);
// This statement will throw an error as scope of j is only within
the function abc.
```

  Now let's see what happens when there are different variables with the same name in different scopes. For eg.,

```
var i = 0;
function abc() {
   var i = 1;
   console.log(i);
}
```

Here, within the function 'abc' we have 2 variables ‘i’ i.e. one whose scope is global and other whose scope is within function 'abc'.
Output of the above code will be **1** as preference will be given to a variable that is in local scope(here scope of function 'abc').
If we remove line ***'var i = 1**;*', then the interpreter will look outside the function and output will be **0**.

FUNCTION WITHIN FUNCTION

You can define a function within a function which we can also call a **nested function**. The nested function ***can be called inside the parent function only.***

The nested function can ***access the variables of the parent function and as well as the global variables.***

But the parent function cannot access the variables of the inner function

This is useful when you want to create a variable that needs to be passed to a function. But using a global variable is not good, as other functions can modify it. So, ***using nested functions will prevent the other functions from using this variable.***

Eg., using a count variable which can only be accessed and modified by the inner ***increaseCount()*** function -

```
function totalCount() {

    var count = 0;
    function increaseCount() {
        count++;
    }
    increaseCount();
    increaseCount();
    increaseCount();
    return count;
}

console.log(totalCount()); // Prints - 3
```

As all the variables of the **totalCount()** function **are available to it’s all child functions,** the **increaseCount()** function **can access** the variable **count** created **inside the totalCount()** function and **modify its value.**
Any other functions **outside the scope** of the **totalCount()** function **cannot access the variables created inside it** therefore, they **can’t access** the **count** variable.

- **Scope Chain**

  Lets now discuss how the interpreter looks for a variable. This is decided according to the Scope Chain -

  When a ***variable is used inside a function,*** the JavaScript interpreter looks for that variable inside the function. If the variable is not found there, then it looks for it in the outer scope i.e. in the parent function. If it is still not found there, it will move to the outer scope of the parent and check it; and do the same until the variable is not found. The search goes on till the global scope and if the variable is not present even in global scope then the interpreter will throw an error.

  Eg., the below function shows how a variable is accessed when used inside a inner function -

```
function a() {
   var i = 20;

   function b() {
       function c() {
           console.log(i); // Prints - 20
       }
       c();
   }
   b();
}
a();
```

FUNCTION DECLARATION AND EXPRESSION

Functions in JavaScript can be defined in two ways –

- **Function Definition** - Creating a function using function keyword and function name.
- **Function Expression** - Creating a function as an expression and storing it in a variable.

We have discussed function declaration and expression in detail below -

- **Function Definition**

  A function definition is creating a function in a normal way, which we have read until now. The syntax is -

```
function functionName(parameters) {
   // SET OF STATEMENTS
```

```
}
```

Here, the parameters take values differently for different types of variables. We can either pass primitive value or non-primitive value –

- If the ***value passed as an argument is primitive***, then it gets passed by value. This means that the changes to the argument does not reflect the changes globally and only remains local.
  Eg.,

```
function abc(b) {
   b = 20;
   console.log(b); // Prints - 20
}
var a = 10;
abc(a);
console.log(a); // Prints - 10
```

- If the ***value passed as an argument is non-primitive***, then it gets passed by reference. This means that change is visible outside the function.
  Eg.,

```
function abc(arr2) {
   arr2[1] = 50;
   console.log(arr2); // Prints - Array [10, 50, 30]
}
var arr1 = [10, 20, 30];
abc(arr1);
console.log(arr1); // Prints - Array [10, 50, 30]
```

- **Function Expression**

We have discussed that variables can take primitive and non primitive values. So function is one of the possible values that a variable can have. Function expression is used to assign the function to a variable.

Eg, the below code uses the function expression with function name –

```
var factorial = function fac(n) {
   return n < 2 ? 1 : n * fac(n - 1);
}
console.log(factorial(3)); // Prints - 6
```

However, note that the name “fact” that this function has can be used only inside the function to refer to itself, it can’t be used outside the function.

The function expression as shown above is named i.e. the function being assigned has a name. We can have **anonymous** function expressions as well i.e. it does not have a name.

The syntax is –

```
var variableName = function(parameters) {
   // SET OF STATEMENTS
}
```

Eg.,

```
var factorial = function fac(n) {
   var ans = 1;
   for (var i = 2; i <= n; i++) {
       ans *= i;
       return ans;
   }
   console.log(factorial(3)); // Prints - 6
}
```

PASSING FUNCTION AS ARGUMENT

Functions in JavaScript are basically objects. So we can also pass a function as argument to another function.

There are two ways to pass a function –

- First, pass function as argument like - `function abc(arg1, functionName);`
- Second, define function as an argument like - `function abc(arg1, function() { ... });`

Example,

```
function abc(a, b, compute) {
```

```
    compute();
}

function multiple(a, b) {
    console.log(a * b);
}

function add(a, b) {
    console.log(a + b);
}
abc(5, 2, multiple); // Prints - 10
abc(5, 2, add); // Prints - 3
```

The function passed is also called a **callback function**. A callback is a piece of code that is passed as an argument to other code, which is expected to execute the argument(function) at some convenient time.

But ***why do we need to use callbacks?*** - JavaScript is an event driven language, meaning that instead of waiting for a response from a function, it keeps executing the code in sequence. So if you want to execute something after some line of code, then callbacks are very useful. We will see callbacks in upcoming sections.

ARRAYS

Array is an **ordered collection of data**(can be primitive or non-primitive), used to store multiple values. This helps in storing an indefinite number of values.
Each item/value has an **'index'** attached to it, using which we can access the values. In JavaScript ***'index'*** starts at ***0***.

The array also contains a property **'length'** that stores the number of elements present inside the array. It changes its value dynamically as the size of the array changes.

- **Creating an Array**

  There are two ways to create an array -

- **Using square brackets** - We can create an empty array as - `var arrayName = []` and array with initial values as - `var arrayName = [value1, value2, ..., valueN]`

- **Using Array Object** - We can create an empty array as - `var arrayName = new Array()` and array with some length as - `var arrayName = new Array(N)`, where 'N' is length of array.

  Another way to create an array using an 'Array' object is providing the values in it like -
  `var arrayName = new Array(value1, value2, ..., valueN)`. This will create an array with these elements.

  Array is heterogeneous, meaning it can ***contain different types of value*** at the same time.
  Also the array can store primitive and non-primitive values.
  Eg.,

```
var array = ["hammer", 85, { name: "Preeti" }, [0, 2, 6]]
```

- **Accessing Elements in Array**

  You can **access the individual elements** of the array **using the square bracket notation** like - `array[1]` will return '85'. This also allows you to modify the value like - `array[1] = 20` and the array now becomes -

```
["hammer", 20, {name: "Preeti"}, [0, 2, 6]]
```

  When using an **array inside an array**, you can access the value of the inner array directly like - `array[3][1]` will return **'2'**.

  If you ***access the array outside it's range***, i.e. you pass an invalid index(whether negative or greater than the length of array), then ***'undefined'*** is returned.

- **Placing Elements at Outside Array Range**

  When you use index outside the range to add elements to an array, the array behaves in a different manner.

When a value is assigned to a positive **index outside the range of the array**, then the array stores this value at the specified index and all other indices before it are empty. The length of the array also changes to 'index+1'.

When a ***negative value is used***, then the array stores the element as a key-value pair, where the negative index is the key and the element to be inserted is the value.

You can play around it and check the output.

EXTRA*: You must give a read to the below links on how arrays are stored in JavaScript - *https://stackoverflow.com/questions/20321047/how-are-javascript-arrays-represented-in physical-memory

FUNCTIONS ON ARRAYS

Below are some important methods that can be used on arrays -

- **push method**

  The **'push()'** method is used to ***add one or more elements to the end of the array*** and ***returns the new length of array.*** It uses the **'length'** property to add elements. If the array is empty, then the **'push()'** method will create a **'length'** property and also add an element to it.
  In case, you are adding multiple elements, separate them using a **comma(,).**

  The syntax is - `array.push(element1, ... , elementN)`

- **pop method**

  The '**pop()**' method is used to ***remove the last element from the array*** and ***return that element***. It also **decreases** the ***'length'*** of the array by 1.
  Using pop on an empty array returns ***'undefined'***.

  The syntax is - `array.pop()`

- **shift method**

  The '**shift()**' method is used to ***remove the first element from an array*** and ***return that element.*** If the ***'length'*** property is 0, ***'undefined'*** is returned.

The syntax is - `array.shift()`

- **unshift method**

  The '**unshift()**' method is used to ***add one or more elements to the beginning of the array*** and ***returns the new length of array***. In case, you are adding multiple elements, separate them using a **comma(,).**

  The syntax is - `array.unshift(element1, ... , elementN)`

- **indexOf Method**

  The '**indexOf()**' method is used to ***return the first index at which the given element is found*** in the array. If the element is not found, then '-1' is returned.
  By default the whole array is searched, but ***you can provide the start index from which the search should begin***. It is **optional**. If the index provided is negative, then the offset is set from the end of the array and search in the opposite direction is done.

  The syntax is - `array.indexOf(element, fromIndex)`

  You can check the examples from the below link -
  https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf#Examples

- **splice method**

  The **'splice()**' method is used to ***remove or replace or add elements to an array***. If an element is removed, it returns the array of deleted elements. If no elements are removed, an empty array is returned.

  The syntax is - `array.splice(start, deleteCount, item1, ..., itemN)`

  You can check the examples of adding, removing and replacing from this link -
  https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Array/splice#Examples

- **reverse method**

The **'reverse'** method is used to ***reverse the content of the array*** and ***return the new reversed array.*** This means that the first element becomes last and vice-versa.

The syntax is - `array.reverse()`

- **sort method**

The '**sort()**' method is used to ***sort the elements of an array*** and ***return the sorted array.*** The sort is done by converting the elements to string and then comparing them.

The syntax is - `array.sort([compareFunction])`

Here, '***compareFunction(a, b)***' is optional and you can provide it to sort arrays according to the defined function.
You can visit the below link, to know more about 'compareFunction' -
https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Array/sort#Description

- **join method**

The **'join()'** method is used to **concatenate all the elements in an array** and **return a new string.**
By default, they are separated by **comma(,),** but you can also provide your own separator as a string. It is optional to provide a separator.

The syntax is - `array.sort(separator)`

If an element is ***'undefined'*** or ***'null'***, then it is converted into an empty string.

- **toString method**

The **'toString()'** method is used to **return the array in the form of a string.** The string contains all the elements separated by comma.

The syntax is - `array.toString()`

EXTRA:
You can find other methods that array uses from the below link -

https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Array#Methods_2

ITERATING OVER ARRAYS

Iterating over the arrays helps in accessing the values and also manipulates each one of them individually, using a lesser line of code. We have used two methods to iterate over the arrays -

- **For loop**

  The 'for' loop is used normally to iterate over all the values of the array.
  Eg.,

```
var arr = [10, 20, 30];
for (var i = 0; i < arr; ++i) {
   console.log(arr[i] * 2);
}
```

  The above code will print the array values as doubled - 20 40 60

- **forEach Method**

  The **'forEach()'** method calls a function once for each array element.
  The syntax is -

```
arr.forEach(function callback(currentValue, index, array) {
   /* Function Statements */
}, thisArg);
```

  You can either provide function definition as shown in the syntax above. Or you can pass the function name to it.

  Eg., the below code will print the values on the console -

```
var items = ['apple', 'banana', 'orange'];
items.forEach(function(item) {
   console.log(item);
})
```

You can use the below link to find in detail about the parameters passed to the forEach method -
https://developer.mozilla.org/enUS/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach#Description

Objects and Timing Events

OBJECTS

JavaScript objects are a collection of properties in a key-value pair. These objects can be understood with real-life objects, like similar objects have the same type of properties, but they differ from each other.

Eg., let's say a 'ball' is an object and has properties like 'shape' and 'radius'. So ***every ball will have the same properties, but different balls will have different values*** to them.

Some important points about objects are -

- Object contains ***properties separated with a comma(,).***
- Each property is represented in a ***key-value pair.***
- ***Key and value are separated using a colon(:).***
- The ***key can be a string or variable name*** that does not contain any special characters, except underscore(_).
- The ***value can contain any type of data - primitive, non-primitive,*** and even a ***function.***
- The ***objects are passed by reference to a function.***

An example of an object is -

```
var obj = {
   key1: "value1",
   key2: 12345,
   "key3": true,
   key4: function() { /* Something Here */ }
}
```

- **Creating an Object**

  The object can be created in two ways -

  - **Using curly brackets -** We can create empty object as - `var obj = {};` and an object with some initial properties as - `var obj = {key1: value1, ..., keyN: valueN}`

- **Using new operator -** We can create empty object as - `var obj = new Object();` and an object with properties as - `var obj = new Object({key1: value1, ..., keyN: valueN})`

The properties can be created at the time of creating an object and also after that. ***Both creating and accessing the properties share similar syntax.***

- **Creating and Accessing Properties**

The **properties are created in a key-value pair,** but there are some restrictions in the way some keys are created. There are two ways to create and access properties:

- **Using a dot operator -** You can use the dot operator only when the property name starts with a character. Property can be accessed like - `obj.propertyName.` Similarly, you can create property like - `obj.propertyName = value`
- **Using a square bracket -** You need to use a square bracket when the key name starts with a number. If the name contains a special character then it will be stored as a string. Property is accessed like - `obj["propertyName"].` Similarly, you create property like - `obj["propertyName"] = value`

You can also **set the function as the value** to the key. So the key then becomes the method name and ***needs parentheses to execute***. So you can execute the method like **-** `obj.methodName()` and `obj["methodName"]().`

NOTE: *If you access a property that has not been defined then 'undefined' is returned.*

- **Deleting Property**

You can ***remove the property of the object*** using the **'delete'** operator followed by the property name. You can either ***use dot operator*** or ***square bracket notation.*** The syntax is -

```
delete obj["objectName"]
```

OR

```
delete obj.objectName
```

- **How Objects Are Stored**

There are two things that are very important in objects -

- Objects are **stored in heap.**
- Objects are **reference types**.

These two are important in the regard that ***object variables point to the location*** where they are stored. This means that ***more than one variable can point to the same location***.

Until now, you are creating new objects everytime like -

```
var item1 = { name: 'banana' };
var item2 = { name: 'banana' };
```

The above two lines will create two different objects are not therefore equal -

```
item1 == item2; // Returns - false
item1 === item1; // Returns - false
```

But, if you assign one object to another like - `item2 = item1;` then the value of 'item1' gets assigned to 'item2' and therefore, they both will point to the same location -

```
item1 == item2; // Returns - true
item1 === item1; //Returns - true
```

OBJECTS

JavaScript provides a special form of a loop to traverse all the keys of an object. This loop is called for**...in**' loop.

The syntax for '***for-in***' loop is -

```
for (variable in object) {
   /* Statements */
}
```

Here the **'variable'** ***gets assigned the property name*** on each iteration and **'object'** ***is the object*** that you want to iterate. Use the ***square bracket notation with 'variable' to access the property values.***

The ***iteration may not be in a similar order as to how you see properties in the objec***t or how you have added them. Because the objects are ordered in a special manner.
The ***property names as integers are iterated first*** in ascending order. Then the other names are iterated in the order they were added.

The below code shows how you can iterate using the **'for-in'** loop -

```
for (key in obj) {
   console.log(i, ":", obj[i]);
}
```

It will print the following lines on the console -

```
key1 : value1
key2: 12345
key3: true
key4: function key4()
```

There are two more ways to access all the keys, but it will return an array of keys -

- **Object.keys(obj)**
- **Object.getOwnPropertyNames(obj)**

EXTRA: You can read about the other two ways from the links below -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames

NESTED OBJECTS

We have already discussed that the ***value of an object's property can be anything.*** So we can have ***objects inside an object*** and they are called **nested objects.** We can have ***any number nesting inside an object,*** i.e. an object can contain another object, which can also contain another object, and so on.

Eg. a nested object is -

```
var student = {
   name: "Anjali",
   class: 5,
   roll: "016-115-19",
   address: {
       city: "New Delhi",
       pincode: 110063
   }
}
```

Here, 'address' is a nested object.

- **Creating Nested Objects**

  You create a nested object as you have created other properties, but cannot create a property of the nested object. This means `obj.propertyName.nestedProperty1 = value1` ***is invalid and gives an error.***

  Instead you create nested object as -

```
obj.propertyName = { nestedProperty1 = value1, ..., nestedPropertyN = valueN }
```

- **Accessing Nested Objects**

  The nested objects can be accessed using multiple dot operator or square brackets notation like -

```
obj.propertyName.nestedProperty1
```

OR

```
obj["propertyName"]["nestedProperty1"]
```

ARRAY AS OBJECT

Arrays are actually objects. If you use the '**typeOf()**' method on an array, you will see that it will return "**object**". If you see an array on a console, **they are actually key-value pairs**, with the ***positive integers as the keys***.
Arrays can also store properties just like objects. Eg., `array["one"] = 1;` will store this property inside the array and can access it like - `array.one; or array["one"];`.

But if arrays are the same as objects, then what is the difference? Well, ***arrays are somewhat different than objects***. These difference are summarized in the points -

- Arrays ***have a 'length' property*** that objects do not have.
- You can access the values of the arrays like - ***array[0]; or array["0"];,*** whereas in objects you have to use **double quotes** ("") only.
- Only when you use an integer as a key, it will change the 'length' property.

- Adding a non-integer key will not have any effect on the length's property.

So we can say that ***arrays work both like objects and arrays*** (from other languages like Java).

***NOTE**: Length property will be set according to the maximum integer key of the array.*
For eg:

```
var arr = [1,2,3,4,5,6];
console.log(arr.length);
//output = 6
```

- **Using the for...in loop to Iterate**

  Since **arrays are also objects,** you can use a **'for-in'** loop to traverse it. Traversing the array using the 'for-in' loop is the same as traversing an object.

  There is something interesting about arrays you need to know. Let's say that you have an array like -

```
var arr = [10, 20, 30];
```

  and you add another property like -

```
arr["four"] = 40;
```

  then if you print an array on the console like - `console.log(arr);`, it will show the array as - Array(3) [10, 20, 30].

  But, it also ***contains the property*** "`four: 40`", ***but it does not show in the array***. But if you use a 'for-in' loop to traverse it, you can traverse all the properties.

```
for (var i in arr) {
   console.log(i, ":", arr[i]);
}
```

  you will see something like this on the console -

  0: 10
  1: 20
  2: 30
  there: 123

TIMING EVENTS

The **timing events** allow the ***execution of a piece of code at a specific time interval.*** These events/methods are directly available in the DOM Window object, i.e. they are there in the browser. You'll learn about DOM in the next lecture.
Therefore, these are **global methods** and can be **called using a 'window' object or without it.**
Below we have used the timing events that window provides us-

- **setTimeout()**

  The **'setTimeout()'** method is used to ***execute a piece of code after a certain amount of time.*** The piece of code is usually written inside a function.
  The ***function can be passed as a parameter or you can use an anonymous function directly as a parameter.***

  The syntax is -

```
var timeoutID = scope.setTimeout(function, delay, param1,
   param2, ...)
```

  The '**setTimeout()**' method ***returns a positive integer*** which r***epresents the ID of the timer*** created. The use of this ID will be explained later.

  The ***parameters passed*** (specified after the delay time) are ***optional*** and are accessible to the 'function'.

  The **'delay'** is ***written in milliseconds***, so '1000' represents '1' second.

  It is ***optional to use a variable to store the ID***, but it depends upon use cases which will be defined later.

- **setInterval()**

  The **'setInterval()'** method is used to ***execute a piece of code repeatedly with a fixed time interval between each call.***

  The syntax is -

```
var intervalID = scope.setInterval(function, delay, param1,
   param2, ...)
```

The meaning and use of each of the parameters are the same as that of the 'setTimeout()' method.

- **clearTimeout()**

  The **'clearTimeout()'** method is used to **cancel a timeout** established using the 'setTimeout()' method.
  The syntax is - `scope.clearTimeout(timeoutID)`
  The ***'timeoutID' is the ID that 'setTimeout()' method returns***. Passing an invalid ID to this method will not do anything.

  NOTE*: When you don't need to use the 'clearTimeout()' method, then there is no need to store the ID returned by the 'setTimeout()' method.*

- **clearInterval()**

  The **'clearInterval()'** method is used to **cancel the repeating timed action established using 'setInterval()' method.**

  The syntax is **-** `scope.clearInterval(intervalID)`

  The ***'intervalID' is the ID that 'setInterval()' method returns*****.** Passing an invalid ID to this method will not do anything.

  NOTE*: The 'setTimeout()' and 'setInterval()' method share the same pool for storing IDs, which means that you can use 'clearTimeout()' and 'clearInterval()' methods interchangeably. However, you should avoid doing so.*

DOM Events in JavaScript

DOM

JavaScript is used to ***add behavior and interactivity to the web page***. It is also possible to manipulate the web page using JavaScript.

But **how does JavaScript manipulate the web page?** - ***JavaScript accesses the DOM to manipulate the web page***. Using DOM, the JavaScript gets access to HTML as well as CSS of the web page and can also add behavior to the HTML elements.

Now, ***what is DOM? - Document Object Model is an API that represents and interacts with HTML documents***. When a page is loaded, the browser creates the DOM for the web page. The DOM represents the document as a node tree, where each node represents part of the document. It can be an element, text, etc., just how that web page was actually written.

We have mentioned that DOM is an API. So what is an API? - In simple terms API is an easy way by which you can use code written by somebody else. They make life easy for us. For eg – with the help of DOM, accessing elements in an HTML page and/or editing them or even adding new elements to the page will become quite easy for us and we don't have to write everything from scratch.

Let's explain the tree node structure of DOM - The DOM represents a document with a logical tree. The tree is a hierarchical structure, in the sense that we have tags inside tags in HTML. For example, the DOM tag is the root node, then and the tags are its children. Like this, we have tags inside both and tags as their children.

Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree; with them, you can change the document's structure, style, or content. Event listeners can be added to nodes and triggered on an occurrence of a given event.

- **WINDOW**

The **'window'** is the **object of the browser**. It is like an **API** that is used to set and access all the properties and methods of the browser like an alert box or setting a timeout.

It is **automatically created** by the browser.

- **SCREEN**

  The **screen** object ***contains information about the browser screen.*** It can be used to display screen width, height, colorDepth, pixelDepth, etc. You use the 'screen' object as - `screen.property;`

  The ***'screen' is the object of the window***, therefore it is the same as - `window.screen`

- **DOCUMENT**

  A **'document'** object represents the **HTML code** that is displayed inside a browser window. The **'document'** object has various properties that refer to other objects which allow access to and modification of the content of the web page.

  If you want to access any element in an HTML page, you always start with accessing the 'document' object. You use the 'document' object as - `document.property;`

  The 'document' is the object of the window, therefore it is the same as - `window.document`

EXTRA:
You can read about DOM from the link below -
https://developer.mozilla.org/enUS/docs/Web/API/Document_Object_Model/Introduction
You can learn about API better from the link below -
https://www.mulesoft.com/resources/api/what-is-an-api

FETCHING ELEMENTS

The web page becomes dynamic when you use JavaScript to modify its elements and add events to it. But most of the time dynamicity of web pages is more due to the change in the content of the web page.

You can use the **'document'** object to **access the elements** on your HTML page. The **'document'** object represents your web page. It ***contains methods that can be called to access the elements as objects***. You can then manipulate these elements using their objects and the changes would reflect in the elements on the web page.

You can ***access elements from following selectors -***

- document.getElementById
- document.getElementsByTagName
- document.getElementsByClassName
- document.querySelectorAll
- document.querySelector

These **selectors return an HTMLCollection** which is an ***array-like structure containing the list of nodes***. The collection returned by these selectors is **'live'**, meaning that it automatically updates when the underlying document is changed.

- **Find Element by ID**

  You can find an HTML element of a specific ID using the following statement -

```
element = document.getElementById(ID)
```

  It will **return an element object** whose ID matches the specified string or if the element is not found then it returns 'null'.
  The **'element'** variable ***gets the returned element object***.
  The **'ID'** represents the ***id of the element*** to find.

  Eg.,

```
var comment = document.getElementById("comment");
```

  It will return the element having ID 'comment'

- **Find Element by Tag Name**

  You can find all the HTML elements of same tag name from the following statement-

```
elements = document.getElementsByTagName(tagName)
```

  It will ***return a 'live' HTMLCollection*** of all those elements that have the same tag name as provided to the above statement.

If no element is found then it returns an empty HTMLCollection. The **'elements'** variable gets this ***returned array*** and you can access the individual element from it. The 'tagName' represents the ***name of the tag*** to find.

Eg.,

```
var input = document.getElementsByTagName("input");
```

It will return all the input elements present in the HTML page.

- **Find Element by Class Name**

You can find all the HTML elements of the same class name from the following statement -

```
elements = document.getElementsByClassName(names
```

It will also ***return a live HTMLCollection*** of all those elements that have the same class added to them as provided to the above statement. If no element is found then it returns an empty HTMLCollection.

The **'elements'** variable gets this ***returned array*** and you can access elements from it. The **'names'** represent a ***string containing one or more class names*** separated by whitespace.

Eg.,

```
var colors = document.getElementsByClassName("red blue");
```

It will return all the elements which have both the 'red' and 'blue' class.

- **Find Element Using Selectors**

You can find HTML elements using CSS selectors from the following statement -

```
element = document.querySelector(selectors)
```

This will ***return the first element that matches the specified group of selectors***. If no match is found 'null' is returned.
The **'selectors'** represents a ***group of selectors separated by a comma***.

Eg,.

```
var input = document.querySelector("input.email");
```

It will return the first input element having class 'email'.

There is also another query selector which ***returns all the elements that match the specified CSS selectors -***

```
elements = document.querySelectorAll(selectors)
```

This will ***return a static NodeList*** of elements that match the specified group of selectors.
Eg.,

```
var notes = document.querySelector("div.note");
```

It will return the NodeList of 'div' elements having class 'note'.

EXTRA: There is something interesting that you can read from the link below-
https://developer.mozilla.org/enUS/docs/Web/API/Document/querySelector#Usage_notes
https://developer.mozilla.org/en-US/docs/Web/API/Element/querySelectorAll

EVENTS IN JAVASCRIPT

An HTML event is something that the browser does or something a user does. You can also call ***events as actions or occurrences that happen in the browser***, to which you can respond to in some way.
An ***event can be triggered by the user action*** e.g. clicking a mouse button or tapping keyboard.

Here are some more examples of HTML events -

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Events can represent everything from ***basic user interactions to automated notifications of things*** happening in the rendering model.

To execute these events, you use functions and link them to an element. Sometimes in a function, you will see a ***parameter name such as 'event', 'evt' or 'e'.*** This is called an **event object** and it is ***automatically passed to the function.***

To get notified of DOM events, you can use two ways -

- Using addEventListener() method
- Using on<event> handlers

- **addEventListener() Method**

  The **'addEventListener()**' method sets up a function that will be called whenever the specified event is delivered to the target.

  The syntax is - `target.addEventListener(type, listener, options);`

  The **'target'** represents the element on which the event is added/attached.
  The **'type'** is a ***case-sensitive string*** representing the event type like **'click'**, **'keypress'**, etc.

  The **'listener'** is mostly a JavaScript function.

  You can know more about the syntax from the link below - https://developer.mozilla.org/enUS/docs/Web/API/EventTarget/addEventListener#Syntax

  Eg., you can show an alert box upon a mouse click on a button as -

```
var submitButton = document.getElementById("submit-button");
submitButton.addEventListener("click", function() {
   alert("Submit Button is Clicked.");
});
```

- **on<event> Handlers**

  The on<event> handlers are a group of properties offered by DOM elements to help manage how that element reacts to events.

  The syntax is - `target.on<event> = functionRef;`

  The **'target'** represents the element on which the event is added/attached.
  The **'functionRef'** represents the ***function name or a function expression***. The function receives an argument of an 'Event' object.

Examples of on<event> handlers are **'onclick'**, **'onkeypress'**, etc.
NOTE: When using on<event>, you can only have one handle for each event for an element. To use more than one event handler for the same event, 'addEventListener()' is a good choice.

- **Click Event**

The **click event** occurs when the ***user clicks on an element.*** The click event is completed when the ***mouse button is pressed and released*** on a single element.

You can use one of the following syntaxes -

```
target.onclick = functionRef;
```

OR

```
target.addEventListener("click", function() {
   // JavaScript Code
});
```

You can assign only one 'onclick' event to an element at a time.
Eg., you can use the onclick event on an input tag as -

```
function abc(event) {
   console.log("Input element contains text - ",
       event.target.value);
}
document.getElementsByTagname("input")[0].onclick = abc;
```

This will print the value written inside the first input element of the web page.

ADDING SCRIPT TO HTML

To use JavaScript in your web page, you need to insert it into your HTML page.

You can write your JavaScript code by using the **<script>** tag. You then need to write JavaScript code
in between them. This will also help in using the same script in other web pages as well.

Eg., you have to insert JavaScript like this -

```
<script type="text/javascript">
```

```
   document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
```

You can add a **'type'** attribute to mention the type of script you are using. But since ***default scripts are written in JavaScript***, you may not need to write it.

- **Adding External Scripts**

  <script> tag can be used in another manner as well. It can ***add external JavaScript files*** to the web page.

  Writing JavaScript code in external files separates it from HTML code. It makes HTML and JavaScript easier to read and maintain.
  The external JavaScript files should have the **extension - '.js'.**

  You can add JavaScript file like this -

```
<script type="text/javascript" src="myScript.js"></script>
```

  The JavaScript file name with the extension is mentioned inside the 'src' attribute. External scripts can be referenced with a full **absolute URL** or with a ***path relative*** to the current web page.

- **JavaScript in <head> and <body> Tag**

  You place scripts **inside the <head> tag**, just like the <link> tag. You can use both of the above-mentioned ways to add the script to the web page by writing them inside the <head> tag like -

```
<head>
   <!-- Other Header Tags -->
   <script type="text/javascript" src="myScript.js"></script>
</head>
```

  But you can also add script inside the tag as well like -

```
<body>
   <script type="text/javascript" src="myScript.js"></script>
   ...
   <!-- HTML CODE -->
   ...
</body>
```

But when you use the above two methods, then the ***JavaScript compilation is done first*** even before the HTML code is rendered on the web page. ***This slows down the loading time of the web page*** and also some things might not as expected as elements are not rendered at that time.

To improve the loading time of the web page, we can also make the JavaScript load and compile after the page is loaded. To do this we need to **add the script at the bottom of the <body> tag** after the HTML code like -

```
<body>
   ...
   <!-- HTML CODE -->
   ...
   <script type="text/javascript" src="myScript.js"></script>
</body>
```

Now, the ***HTML code is rendered first*** and then after that, the JavaScript is loaded

SCROLL EVENT

Scroll events allow ***reacting on a page or element scrolling***. The scroll event fires when the document view or an element has been scrolled.

Since scroll events can ***fire at a high rate,*** the ***event handler shouldn't execute computationally expensive operations*** such as DOM modifications.

Below we have discussed 'onscroll' event -

- **onscroll**

  The **'onscroll'** event occurs when the ***user scrolls an item's content***. You can ***use only one 'onscroll' event*** on an element at a time.

  You can use one of the following syntaxes -

```
target.onscroll = functionRef;
```

OR

```
target.addEventListener("scroll", function() {
   // JavaScript Code
});
```

The **'target'** represents the element on which the event is added/attached.
The '**functionRef**' represents the function name or a function expression. The function receives a parameter of a MouseEvent object.

You can look at the example from the below link -
https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_onscroll_dom

MOUSE EVENTS

The **mouse events** represent those ***events that occur due to the user interacting with a pointing device*** (such as a mouse). It consists of several events of clicking and movement of the mouse over an element.

We have discussed some of the common mouse events below -

- **onmouseenter**

  The **'onmouseenter'** event occurs when the ***mouse button is moved over an element.***

  You can use one of the following syntaxes -

```
target.onmouseenter = functionRef;
```

OR

```
target.addEventListener("mouseenter", function() {
   // JavaScript Code
});
```

The **'target'** represents the element on which the event is added/attached.
The **'functionRef'** represents the function name or a function expression. The function receives a parameter of a MouseEvent object.

Eg.,

```
var button = document.getElementById('submit-button');
button.onmouseenter = function(event) {
   console.log("Mouse entered element - ", event.target);
}
```

This will print the element on the console when the mouse enters the element.

- **onmouseleave**

  The **'onmouseleave'** event occurs when the ***mouse button is moved out of an element***.

  You can use one of the following syntaxes -

```
target.onmouseleave = functionRef;
```

  OR

```
target.addEventListener("mouseleave", function() {
   // JavaScript Code
});
```

- **onmousedown**

  The **'onmousedown'** event occurs when the ***mouse button is pressed on an element.***

  You can use one of the following syntaxes -

```
target.onmousedown = functionRef;
```

  OR

```
target.addEventListener("mousedown", function() {
   // JavaScript Code
});
```

- **onmouseup**

  The **'onmouseup'** event occurs when the **mouse button is released over an element**.

  You can use one of the following syntaxes -

```
target.onmouseup = functionRef;
```

  OR

```
target.addEventListener("mouseup", function() {
   // JavaScript Code
});
```

Here is a good example showing the use of both onmouseup and onmousedown events - https://developer.mozilla.org/enUS/docs/Web/API/GlobalEventHandlers/onmouseup#Result

KEY EVENTS

Keyboard events occur when the ***user interacts with the keyboard keys***.
There are three types of keyboard events -

- **onkeypress**

  The **'onkeypress'** event occurs when a ***key is being pressed***. The keypress event ***sends which character was entered in ASCII code.***

  This event ***does not occur*** on keys that have a ***toggle effect like 'Caps Lock' and 'Num Lock'***. However, the other two keyboard events will be dispatched for these types of keys.

  You can use one of the following syntaxes -

```
target.onkeypress = functionRef;
```

OR

```
target.addEventListener("keypress", function() {
   // JavaScript Code
});
```

- **onkeydown**

  The **'onkeydown'** event occurs when a ***key is pressed down***. The keydown ***event sends a code indicating the key*** which is pressed.

  You can use one of the following syntaxes -

```
target.onkeydown = functionRef;
```

OR

```
target.addEventListener("keydown", function() {
   // JavaScript Code
});
```

- **onkeyup**
  The **'onkeyup'** event occurs when ***a key is getting released***. The keyup ***event sends a code indicating the key.***

  You can use one of the following syntaxes -

```
target.onkeyup = functionRef;
```

  OR

```
target.addEventListener("keyup", function() {
// JavaScript Code
});
```

EXTRA:
The below link will tell you about how the event works when a key is pressed and hold down -
https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent#Usage_notes

EVENT BUBBLING

Event bubbling is defined as the ***propagation of event*** starting to ***trigger from the deepest target element to its ancestors/parents*** in the same nesting hierarchy until it reaches the outermost DOM element.

Example of how event bubbling works like -

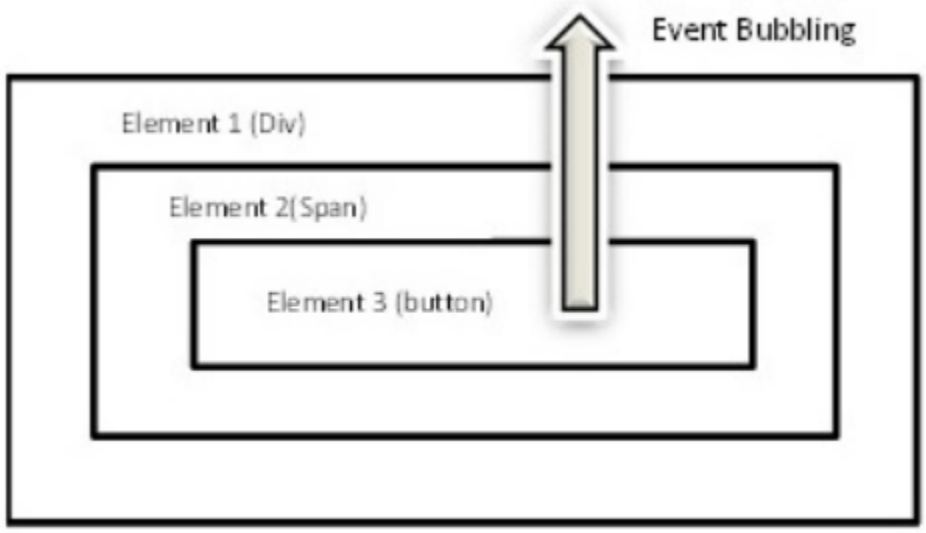
- A button is clicked and the event is directed to the button.
- If an event handler is set for that object, the event is triggered.
- The event then bubbles up to the object's parent.
- The event bubbles up from parent to parent until it is handled, or until it reaches the document object.

You can find whether an event bubbles up through the DOM or not using the **'bubbles'** property of an event. The syntax to check the bubbling of an event is -

```
event.bubbles;
```

It will ***return a boolean value*** representing if event bubbles or not.

Eg., suppose there is a structure as shown in the image below; where we have a button inside a span and which are enclosed inside a div.



If you ***click on Element 3(button)***, then if there is some event handler on it, then it is executed and the event bubbles to Element 2(span). It executes the event handler of the Element 2 if any. It then finally reaches the Element 1(div) and runs its event handler, if any.

- **Stop Event Bubbling**

  Since all the browsers ***by default support event bubbling***, it is sometimes useful to stop the event bubbling. So if you ***want only one event to run on a single element*** then the bubbling of an event is not a necessity.

  To stop the event from bubbling up, you can use any of the following approaches -

  - **stopPropagation() method :**
    To ***stop the propagation of any particular event*** to its parents you can use the '**stopPropagation()**' method. This method invokes only the event handler of the target element.
    You can stop propagation of a particular event using the following statement-

```
event.stopPropagation();
```

    Eg., we can disable event propagation after clicking on a button with ID as 'comment' like this -

```
document.getElementById("comment").onclick =
function(event) {
   alert("Comment Posted");
   event.stopPropagation();}
```

- **stopImmediatePropagation() method:**
  In DOM, ***we can have multiple handlers for the same event*** and they are independent of each other. So stopping the execution of one event handler generally doesn't affect the other handlers of the same target.

  So when you want to stop further propagations as well as to ***stop any other event handler of the same event*** from executing, then you can use **'stopImmediatePropagation()**' method.

  You can stop propagation using the following statement-

```
event.stopImmediatePropagation();
```

STRICT MODE

The strict mode was introduced in ECMAScript 5. It is a way to **add a strict checking in JavaScript** that would make fewer mistakes.

JavaScript ***allows strict mode code and non-strict mode code to coexist***. So you can add your new JavaScript code in strict mode in old JavaScript files.

Strict mode introduces several restrictions to the JavaScript code like eliminates some silent errors by throwing errors.

You can introduce a strict mode in your JavaScript code by writing this simple statement - '`use strict`'; OR "`use strict`";

You can apply strict mode to an entire script or to individual function -

- Write this at the top of the whole script to apply strict mode globally.
- Or write it inside functions to apply it to a particular function only.

Eg., you have a function as -

```
function abc(a, a) {
   console.log(a + a);
}
abc(10, 20);
```

The above code will print 40 on the console, whereas if you use strict mode as -

```
function abc(a, a) {
   'use strict';
   console.log(a + a);
}
abc(10, 20);
```

This code will produce an error - 'SyntaxError: duplicate formal argument a'.

EXTRA:

You can get a more detailed explanation on strict mode from the below link -

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

https://www.geeksforgeeks.org/strict-mode-javascript/

Closures

LET KEYWORD

The let allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used. This is unlike the var keyword, which defines a variable globally, or locally to an entire function regardless of block scope.

- **Scoping Rules**

  Variables declared by 'let' have their scope in the block for which they are defined, as well as in any contained sub-blocks. In this way, let works very much like var. The main difference is that the scope of a var variable is the entire enclosing function:

```
function varTest() {
    var x = 1;
    if (true) {
        var x = 2; // same variable!
        console.log(x); // 2
    }
    console.log(x); // 2
}

function letTest() {
    let x = 1;
    if (true) {
        let x = 2; // different variable
        console.log(x); // 2
    }
    console.log(x); // 1
}
```

Let in for loop – Consider the following for loop.

```
for (var a = 1; a < 5; a++) {
    setTimeout(function() {
        console.log(a)
    }, 1000);
}
```

This loop will print 1 2 3 4. Every round of let creates a new variable and bounds it with the closure. Let a gets a new binding for every iteration of the loop. This means that every closure, if a function is created in a loop captures a different instance.

***EXTRA**: You can read about let from the links below -*
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
https://www.geeksforgeeks.org/difference-between-var-and-let-in-javascript/

EXECUTION CONTEXT AND LEXICAL ENVIRONMENT

- **Execution Context**

  When the **JavaScript Engine** is used to run your code, your code is executed in a **specific Execution Context for each statement**. There are two main types of Execution Context in the JavaScript environment. When your code is first run, even if it's spread across multiple pages using the <script /> tag, JavaScript creates one **Global Execution Context** in which your code is placed when it executes and runs inside the browser. The second is the **Function Execution Context,** which was created when you called the function you defined.

  Each time you call a function, a new Function Execution Context is created.

```
var message = 'Hello there';

function foo(message) {
    bar(message);
}

function bar(message) {
    console.log(message);
}
foo(message);
```

  Initially, **Execution Context Stack** is empty.

When this code runs, the JavaScript engine ***creates one Global Execution Context*** and pushes it to Execution Context Stack.

When we call the function foo, Global Execution Context was paused because JavaScript is a single-threaded environment that can only execute one code at a time. After that JavaScript engine will create a new Function Execution Context for foo and push it into Execution Context Stack.

When the foo function is executed we invoked the bar inside the foo definition. JavaScript engine paused the execution context in the foo function and ***creates a new Function Execution Context for the bar()*** and pushed it into the stack.

After the bar is finished executing it will be popped out in the Execution Context Stack and go back to foo and resume its execution. The same process will be applied to the foo until we finish and go back to the Global Execution Context and resume the execution.

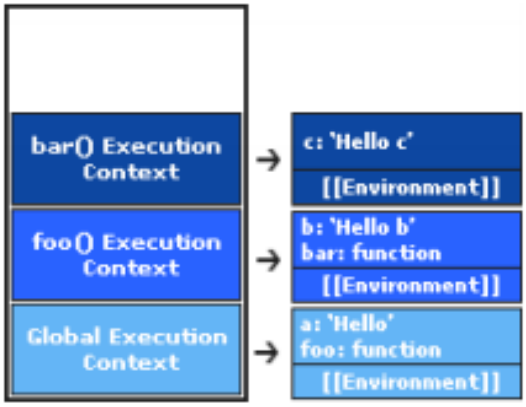
- **Lexical Environment**

Consider the following code:

```
var a = 'a';
function foo() {
    var b = 'b';
    function bar() {
        var c = 'c';
        console.log(c); // You can access me here.
        console.log(b); // You can access me too..
        console.log(a); // You can also access me..
    }
    bar();
}
foo();
```

When this code is first created and foo is stored in a global environment. Only the bar function visible as the bar() is an internal function within the foo environment, it created a new environment when we named the function foe below and saved the variable b in the foo environment. We also called the bar() function when invoking foo(), which also creates a new environment.

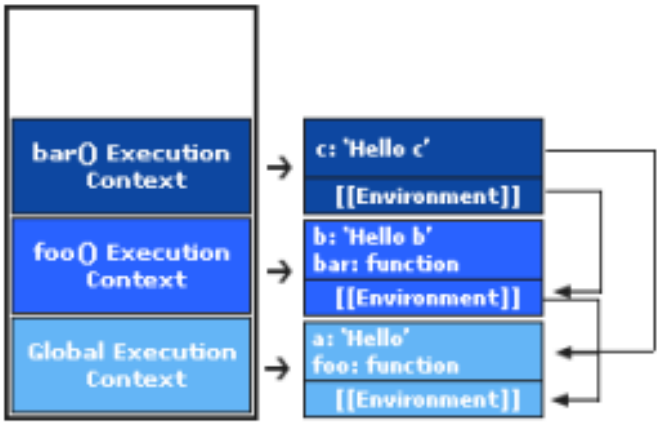
Whenever a function is called, a new function execution context is created, and a new lexical environment is added to the execution context stack.



Inside the bar function, we do logging to check if the variables that we create are visible and if we can access them in their environment.
When variable c was logged in the bar environment, it was displayed successfully obviously because it was inside his environment.

However, when we do logging the variable b and variable it was also successfully displayed. How did this happen?

This is because In the second logging we call the variable b which is not in the bar function scope. So javascript does this internally to search it in other Outer Environment until they find that variable. In our case, they found the variable b at foo function since the bar function has reference to foo function the foo function environment is not pop out in the Execution Context! When variable a was logged it was also successfully displayed because variable a is stored in the Global Execution Context. Everyone can access the Scope in the Global Execution Context.



***EXTRA**: You can read about them from the links below -*
https://blog.bitsrc.io/understanding-execution-context-and-execution-stack-in-javascript 1c9ea8642dd0

CLOSURES

A closure is a feature in JavaScript where an inner function has access to the outer (enclosing) function's variables—a scope chain.

The closure has three scope chains:
it has access to its own scope—variables defined between its curly brackets
it has access to the outer function's variables
it has access to the global variables
A closure is created when an inner function is made accessible from outside of the function that created it. This typically occurs when an outer function returns an inner function. When this happens, the inner function maintains a reference to the environment in which it was created. This means that it remembers all of the variables (and their values) that were in scope at the time. The following example shows how closure is created and used.

```
function add(value1) {
    return function doAdd(value2) {
        return value1 + value2;
    };
}
var increment = add(1);
var foo = increment(2);
// foo equals 3
```

From the above example, we can make the following observations -

The add() function returns its inner function doAdd(). By returning a reference to an inner function, a closure is created.

"value1" is a local variable of add(), and a non-local variable of doAdd(). Non-local variables refer to variables that are neither in the local nor the global scope. "value2" is a local variable of doAdd().

When add(1) is called, a closure is created and stored in "increment". In the closure's referencing environment, "value1" is bound to the value one. Variables that are bound are also said to be closed over. This is where the name closure comes from.

When increment(2) is called, the closure is entered. This means that doAdd() is called, with the "value1" variable holding the value one.

EXTRA:

You can read about closure from the links below -

https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-closureb2f0d2152b36

Constructors and Prototypes

THIS KEYWORD

this keyword in JavaScript ***refers to an object that is executing in the current code.*** Like 'window' object executes in the global scope. Similarly, ***every function while executing has a reference to its current execution context,*** which can be ***referenced by 'this'.***

In most cases, the ***value of 'this' depends on how a function is called.*** It may be different each time the function is called.

The value of 'this' also depends on whether we are working in the strict mode or not.

- **In normal/sloppy mode**

  In the ***global execution context,*** 'this' refers to the global object as the function call is bound to the window object by default.

  Inside a function, if the ***value of 'this' is not set by the call***, then it will also ***default to the 'window' object.*** Eg.,

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

  When an object has a function defined as a property to it, its 'this' refers to the object the method is called on. Eg.,

```
var obj = {
    a: 25,
    abc: function() {
        return this.a;
    }
}
console.log(obj.abc()); // Prints - 25
```

  The result of the output won't change even if the function is attached to the object afterwards the creation of the object.
  Let's say that the object defined above has a nested object as -

```
obj.b = { bcd: function() { console.log(this.a); }, a: 35 };
```

Then, inside the nested object - 'b', 'this' will refer to the object 'b'. So, if we do something like this -

```
console.log(obj.b.bcd()); // Prints - 35
```

Only the most immediate reference to the function call matters.

- **In strict mode**

Strict mode puts a restriction on the value of this, it will be undefined if in global context function is not bound to any object. Whereas in the sloppy mode it was set to 'window' object.

Inside a function, if the ***value of 'this' is not set by the call, then it will be 'undefined'.*** This happens because the function is called directly and not as a method(eg. window.abc()). Here is an example below -

Eg.,

```
function abc() {
    console.log(this);
}
abc(); // Prints – undefined
window.abc() // Prints window object
```

The function of an object behaves in the same way whether in the strict mode and or not.

- **Using call() or apply() methods**

As we know now that in strict mode if you call a function straightaway, it is not bound to any object. So you can bound a function to a particular object by using **'call()'** or '**apply()'** methods.

Eg.,

```
var obj = { a: 12, b: 13 };

function sum() {
    return this.a + this.b;
}
sum.call(obj);
sum.apply(obj);
```

The difference in '**call()'** and **'apply()'** functions lie in the way arguments are passed to the function.

In the call() method, you ***pass individual parameters*** after passing the object parameter.
In the apply() method, you pass the rest of the ***parameters as an array.***
If the ***first argument passed is some other value than an object***, then an ***attempt is made to convert it to an object.***
So, if you write code like this -

```
function abc() {
    console.log(this);
}
abc.call(7); // Prints - Number { 7 }
abc.apply(true); // Prints -Boolean { true }
```

The number is internally converted to object as - ***new Number(7)***

EXTRA: You can read an article on this keyword from the link below - https://codeburst.io/all-about-this-and-new-keywords-in-javascript-38039f71780c

CONSTRUCTOR

You can ***create objects*** in JavaScript ***using curly braces { ... }*** syntax. But what if you need to create multiple similar objects. You can either write the same syntax for every object or you can use the constructor to create objects.

Using { ... } syntax to create multiple objects can create certain inconsistencies in code - there can be spelling mistakes, the code can become difficult to maintain, changes to all the objects will be difficult.

To overcome all the above inconsistencies, ***JavaScript provides a function constructor***. The ***constructor provides a blueprint/structure for objects***. You use this same structure to create multiple objects.

Constructors technically are regular functions. There is one convention to constructors -

- The first letter of the function is capital.

Objects can be created by ***calling the constructor function with the 'new' keyword.***

Using a constructor means that -

- all of these objects will be created with the same basic structure.
- we are less likely to accidentally make a mistake than if we were creating a whole bunch of generic objects by hand.

It is important to ***always use the new keyword*** when invoking the constructor.
If ***new is not used,*** the constructor may clobber the 'this' which was accidentally passed to it.
In most cases that is the global object (window in the browser or global in Node.). Without the 'new' function will not work as a constructor.

```
function Student(first, last, age) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
}
var stu1 = new Student("John", "Doe", 50);
var stu2 = new Student("Sally", "Rally", 48);
```

The ***new keyword is the one that is converting the function call into constructor call*** and the following things happen -

1. A brand new empty object gets created
2. The empty object gets linked to the prototype property of that function. (We will read about prototypes later in this lecture.)
3. The empty object gets bound as ***this*** keyword for the execution context of that function call
4. If that function does not return anything then it implicitly returns ***this*** object.

NOTE*: The 'this' referred in the constructor bound to the new object being constructed.*

EXTRA:
You can get a good read from the link below -
https://javascript.info/constructor-new

PROTOTYPES

Prototypes are a simple way to share behavior and data between multiple objects. The prototypes are property of the Object constructor and can be seen from the code below -

```
Object.prototype
```

All the objects created in JavaScript are instances of 'Object'. Therefore, all of them share the 'Object. prototype' method. We can also override these properties, i.e. we can change them.

Why do you need prototypes? - Sometimes you want to add new properties (or methods) to all the existing objects of a given type. This is possible only by adding the new method to the prototype function.

Since, all the objects share the same prototype chain, the changes done to the Object prototype is seen by all the instances.
Eg.,

```
function Student(name, age) {
    this.name = name;
    this.age = age;
}

var stu1 = new Student("John", 50);
var stu2 = new Student("Sally", 48);
Student.prototype.getName = function() {
    return this.name;
}
```

The below statement will work and produce the name of the object like -

```
stu1.getName();. // Return “John”
```

- **_proto_ AKA Dunder Proto:**

proto points to the prototype object of the constructor function.

When an object is created in JavaScript, the JavaScript engine adds a __proto__ property to the newly created object. This _proto_ property is equal to the Object's prototype property.

Let's see an example -

```
function Student(name, age) {
    this.name = name;
    this.age = age;
}
var stu1 = new Student("John", 50);
```

If you do check the `stu1._proto_` and `Student. prototype`, then you will see that both of them are the same. Also if you apply strict equal to check if they point at the same location then it will return true.

```
Student.prototype === stu1._proto_       // Returns true
```

OBJECT

undefined: 'undefined' is the value assigned to the variable that has been declared but not initialized or defined. A method or statement also returns undefined if the variable that is being evaluated does not have an assigned value. A function returns undefined if a value was not returned.

CLASSES

Classes are introduced in ECMAScript 2015. These are ***special functions that allow one to define prototype-based classes*** with a clean, nice-looking syntax. It also introduces great new features which are useful for object-oriented programming.

An example of creating a class is -

```
class Person {
    constructor(first, last) {
        this.firstName = first;
        this.lastName = last;
    }
}
```

The way you have defined class above is known as class declaration. In order to ***create a new instance of class Person***, we do this -

```
let p1 = new Person("Rakesh", "Kumar");
```

Some points you need to remember -

- You define class members inside the class, such as methods or constructor.
- By default, the body of class is executed in strict mode.
- The constructor method is a special method for creating and initializing an object.
- You cannot use the constructor method more than once, else SyntaxError is thrown.
- Just like a constructor function, 'new' keyword is required to create a new object.

NOTE*: The type of the class is 'function', i.e. typeof(Person) will print 'function' on the console.*

Class Expression

A **class expression** is another way to define a class, which is similar to function expression. They can be named and unnamed both, like -

```
let Person = class {};
```

OR

```
let Person = class Person2 {};
```

The name given to the class expression is local to the class's body.

Hoisting

Class declarations are not hoisted. If you try to use hoisting, it will return 'not defined' error.

Inheritance

The JavaScript class also supports inheritance like other languages such as Java and C++. To inherit a class you have to use the **'extends'** keyword. Eg.,

```
class Vehicle {
    constructor(make, model, color) {
        this.make = make;
        this.model = model;
        this.color = color;
    }
    getName() {
        return this.make + " " + this.model;
    }
}
class Car extends Vehicle {
    getName() {
        return this.make + " " + this.model + " in child class.";
    }
}
let car = new Car("Honda", "Accord", "Purple");
car.getName(); // "Honda Accord in child class."
```

If you want to call the function of the base class. We use the 'super' keyword in order to call the base class methods from within the methods of the child class. Eg.,

```
class Car extends Vehicle {
    getName() {
        return super.getName() + " - called base class function from child
class.";
    }
}
```

Getter and Setter

You can also have a getter**/setter** to ***get the property value*** or to ***set the property values*** respectively. You have to use **'get'** and **'set'** methods to create a getter and setter respectively.

Eg.,

```
class Vehicle{
    constructor(model){
        this.model = model;
    }
    get model(){
        return this._model;
    }
    set model(value){
        this._model = value;
    }
}

Vehicle v = new Vehicle("dummy");
console.log(v.model); //get is invoked
v.model = "demo"; //set is invoked
```

EXTRA:

> ***Read from the link below to read about classes -***
> ***https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes***