

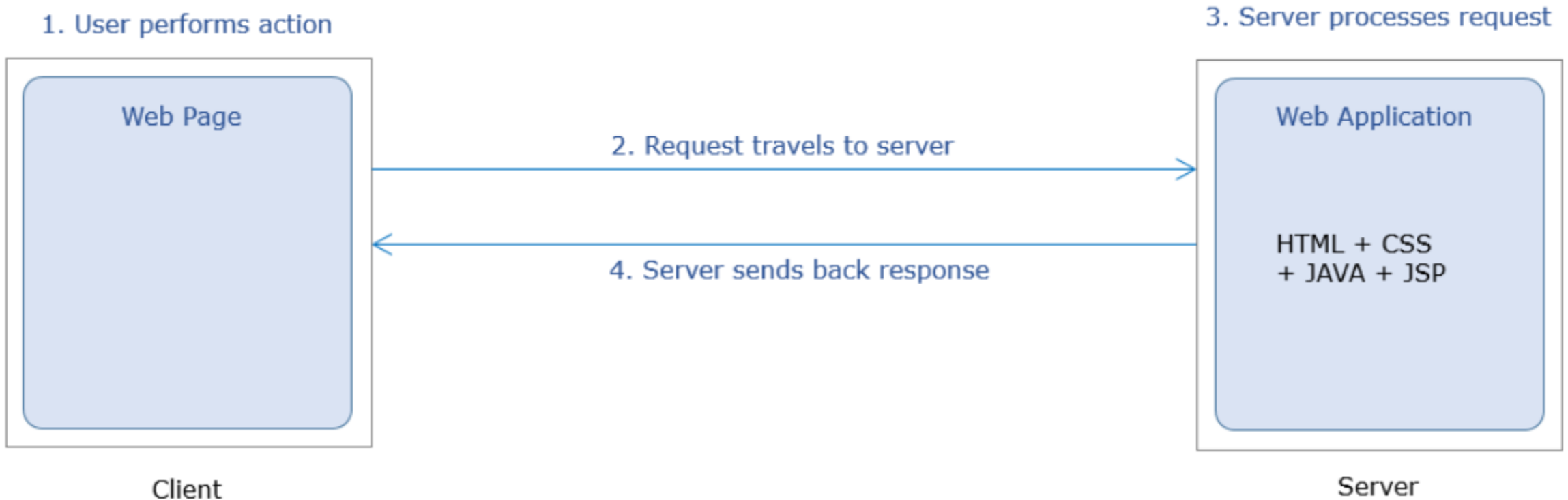
# Javascript

<https://www.linkedin.com/in/niranjana-ravichandran/>

## 1.1 Javascript Introduction

- JavaScript was introduced as a **full-fledged client-side language** used for developing web applications in 1995. JavaScript is easy to learn, debug, and test. It is an **event-based, platform-independent, and an interpreted language with all the procedural programming capabilities**.
- Web applications that were developed using server-side programming languages like Java and Dot Net involved travel of user requests all the way from the client(browser) to the server hosting the application. The multiple request-response cycles between Client and Server were consuming both time and network bandwidth.
- JavaScript got introduced as a Client-Side programming language with the capability of executing user requests on the Client-Side. This could help in reducing the number of request-response cycles between Client and Server and decreasing the network bandwidth thus reducing the overall response time.
- Later, in 1997 **ECMAScript (European Computer Manufacturers Association Script)** established a standard for the scripting languages that redefined the core features any scripting language should have and how to implement those features. From then, JavaScript evolved year after year with every new version of ECMAScript introducing new features. Developers prefer JavaScript to create dynamic, interactive, and scalable web applications as it helps developers in extending the functionalities of the web pages effectively.

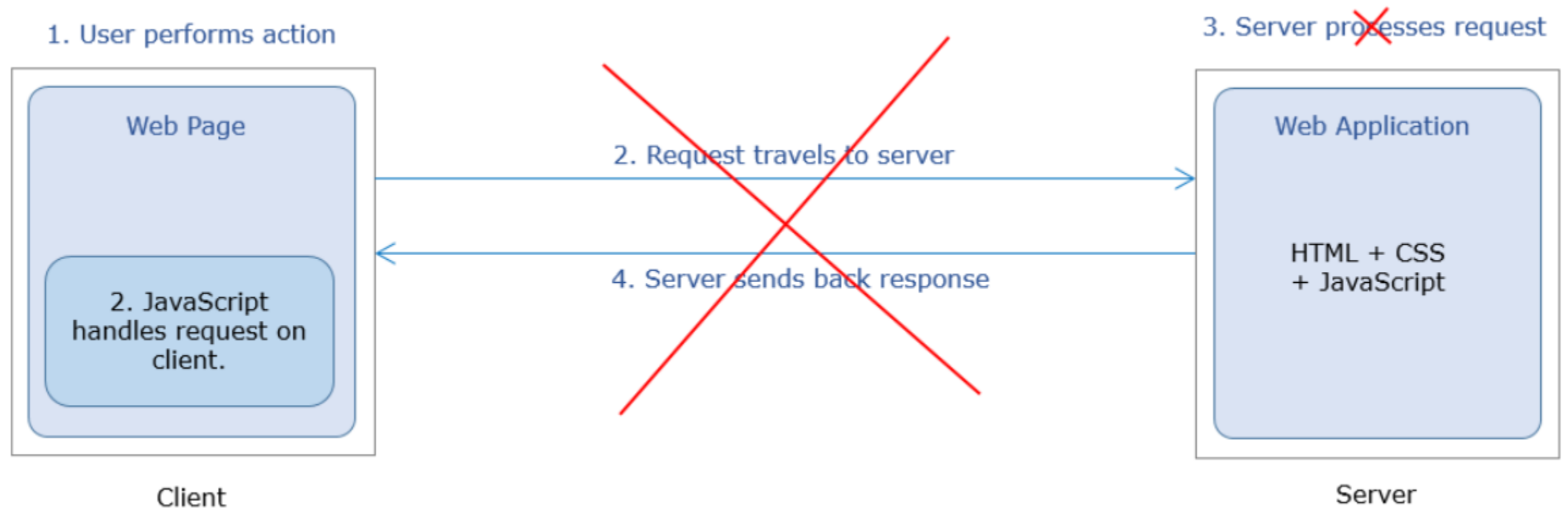
## 1.2 Server side Language



### **Shortcomings:**

- Multiple request-response cycles to handle multiple user requests
- More network bandwidth consumption
- Increased response time

## 1.3 Client side Language



### Advantages:

- No need for back-and-forth request-response cycles
- Less network bandwidth consumption
- In comparison to Java: JavaScript provides a 35% decrease in average response time and Pages being served 200ms faster.

## 1.4 What is Javascript?

JavaScript is the programming language for web users to convert static web pages to dynamic web pages.



## 1.4 What is Javascript?

JavaScript was not originally named as JavaScript. It was created as a scripting language in 1995 over the span of 10 days with the name '**LiveScript**'.

The Scripting language is the one that controls the environment in which it runs.

But now JavaScript is a full-fledged programming language because of its huge capabilities for developing web applications. It contains core language features like control structures, operators, statements, objects, and functions.

JavaScript is an **interpreted** language. The browser interprets the JavaScript code embedded inside the web page, executes it, and displays the output. It is not compiled to any other form to be executed.

All the modern web browsers come along with the JavaScript Engine; this engine interprets the JavaScript code. There is absolutely no need to include any file or import any package inside the browser for JavaScript interpretation.

### 1.4.1 Javascript Engines

JavaScript runtime Engine	Browser
Spidermonkey	Netscape Navigator Mozilla Firefox
V8	Google Chrome Opera
JavaScriptCore	Safari
Chakra and ChakraCore	Internet Explorer

## **1.5 Identifiers**

To model the real-world entities, we will need to name them to use it in the JavaScript program. Identifiers are those names that help us, in naming the elements in JavaScript.

### **Rules for Identifiers:**

- The first character of an identifier should be letters of the alphabet or an underscore (\_) or dollar sign (\$).
- Subsequent characters can be letters of alphabets or digits or underscores (\_) or a dollar sign (\$).
- Identifiers are case-sensitive. Hence, firstName and FirstName are not the same.

Reserved keywords are part of programming language syntax and cannot be used as identifiers.

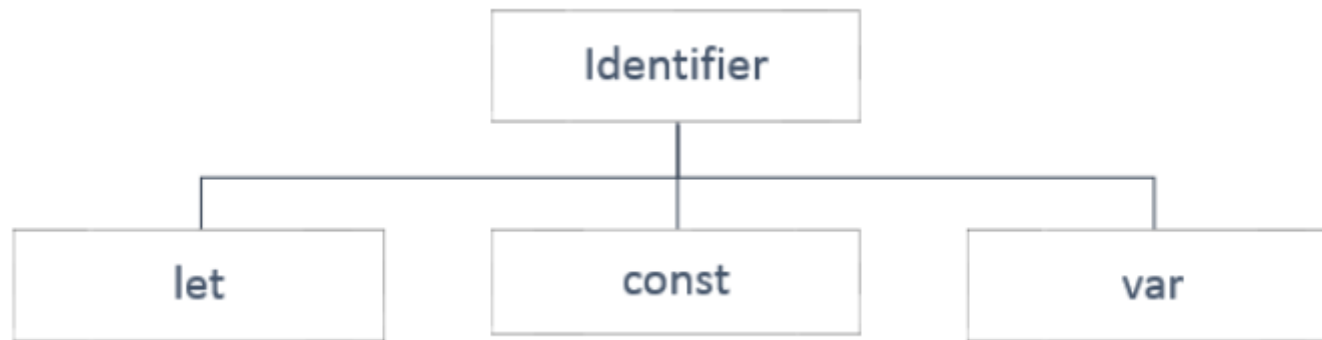
e.g., firstName



### 1.5.1 Identifier Types

The identifiers are declared into specific type based on:

- The data which an identifier will hold and
- The scope of the identifier



Scope:

Block : **A block is a set of opening and closing curly brackets.**

Function: Globally available to the Function within which it has been declared and it is possible to declare the identifier name a second time in the same function

### 1.5.2 let

- An identifier declared using 'let' keyword has a **block** scope i.e., it is available only within the block in which it is defined.
- The value assigned to the identifier can be done either at the time of declaration or later in the code and can also be altered further.

### 1.5.3 const

- The identifier that we choose to hold data that does not vary is called Constant and to declare a constant we use the '**const**' keyword followed by an identifier.
- The value is initialized during the **declaration itself and cannot be altered later**.
- When we talk about the scope of the identifiers declared using the 'const keyword, it takes the **block** scope i.e., they exist only in the block of code within which they are defined.

### 1.5.4 var

For example, if we say A4 size paper the dimension of the paper remains the same, but its color can vary.

The identifiers that we declare to hold data that vary are called Variables and to declare a variable, we optionally use the '**var**' keyword.

The value for the same can be initialized optionally. Once the value is initialized, it **can be modified** any number of times later in the program.

Talking about the scope of the identifier declared using 'var' keyword, it takes the **Function** scope i.e., It means they are only available **inside the function they're created in, or if not created inside a function, they are 'globally scoped.'**

<b>Keyword</b>	<b>Scope</b>	<b>Declaration</b>	<b>Assignment</b>
let	Block	Redeclaration not allowed	Re-assigning allowed
const	Block	Redeclaration not allowed	Re-assigning not allowed
var	Function	Redeclaration allowed	Re-assigning allowed

## 1.6 DataTypes

Data type mentions the type of value assigned to a variable.

In JavaScript, the type is not defined during variable declaration. Instead, it is determined at run-time based on the value it is initialized with.

Hence, we can say that JavaScript language is a loosely typed or dynamically typed language.

### 1.6.1 Number

To store a variable that holds a numeric value, the primitive data type number is used. In almost all the programming languages a number data type gets classified as shown below:



In JavaScript, any other value that does not belong to the above-mentioned types is not considered as a legal number. Such values are represented as **NaN (Not-a-Number)**.

### **1.6.2 String**

When a variable is used to store textual value, a primitive data type string is used. Thus, the string represents textual values. String values are written in quotes, either single or double.

Single quotes or double quotes have no special semantics over the other. Though you must be strategic in the selection between the two when you must use one string within the other.

```
let personName= "Rexha"; // Both are same  
let personName = 'Rexha';
```

```
let ownership= "Rexha's"; // Use Different Strings  
let ownership = 'Rexha"s';
```


```
let ownership= "Rexha"s"; // Error  
let ownership = 'Rexha's';
```

### 1.6.2 Accessing characters in string

Now, to access any character from within the string we need to be aware of its position in the string. Each character in the string occupies a position.

The first character exists at index 0, next at index 1, and so on.

	<b>R</b>	<b>e</b>	<b>x</b>	<b>h</b>	<b>a</b>
Index:	0	1	2	3	4

Length = 5

### 1.6.3 Literals

Literals can span multiple lines and interpolate expressions to include their results.

The template literal notation enclosed in ``(**backticks**) makes it convenient to have multiline statements with expressions and the variables are accessed using **`${}`** notation.

### 1.6.4 Boolean

When a variable is used to store a logical value that can be only true or false at all times, primitive data type boolean is used. Thus, boolean is a data type which represents only two values.

Values such as 100, -5, "Cat",  $10 < 20$ , 1,  $10 * 20 + 30$ , etc. evaluates to true whereas 0, "", NaN, undefined, null, etc. evaluates to false.

### 1.6.5 Undefined

When the variable is used to store "**no value**", primitive data type undefined is used. undefined is a data type in JavaScript that has a single value also termed as undefined. The undefined value represents "no value".

### 1.6.6 Null

The null value represents "**no object**".

If you are wondering why we would need such a data type, the answer is JavaScript variable intended to be assigned with the object at a later point in the program can be assigned null during the declaration.

### 1.6.7 Objects

The data type is said to be **non-primitive** if it is a collection of multiple values.

The variables in JavaScript may not always hold only individual values which are having one of the primitive data types.

There are times when we want to store a group of values inside a variable.

JavaScript gives non-primitive data types named Object and Array, to implement this.

### **Objects**

Objects in JavaScript are a collection of properties and are represented in the form of [key-value pairs].

The 'key' of a property is a string or a symbol and should be a legal identifier.

The 'value' of a property can be any JavaScript value like Number, String, Boolean, or another object.

JavaScript provides the number of built-in objects as a part of the language and user-defined JavaScript objects can be created using object literals.

```
{  
  key1 : value1,  
  key2 : value2,  
  key3 : value3  
};
```

```
let mySmartPhone = {  
  name: "iPhone",  
  brand: "Apple",  
  platform: "iOS",  
  price: 50000  
};
```



### 1.6.8 Array

The Array is a special data structure that is used to store an ordered collection, which we cannot achieve using the objects.

There are two ways of creating an array:

```
let dummyArr = new Array();  
//OR  
let dummyArr = [];
```

A single array can hold multiple values of different data types.

```
digits =[1,2,3,"four"];
```

## 1.8 Operators

Operators in a programming language are the **symbols** used to perform operations on the values.

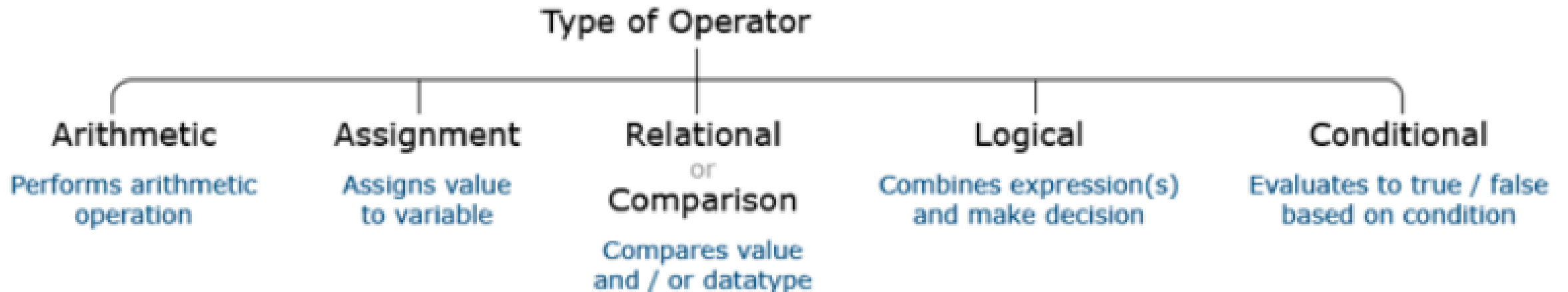
**Operands**: Represents the data.

**Operator**: which performs certain operations on the operands.

### 1.8.1 Types Of Operator

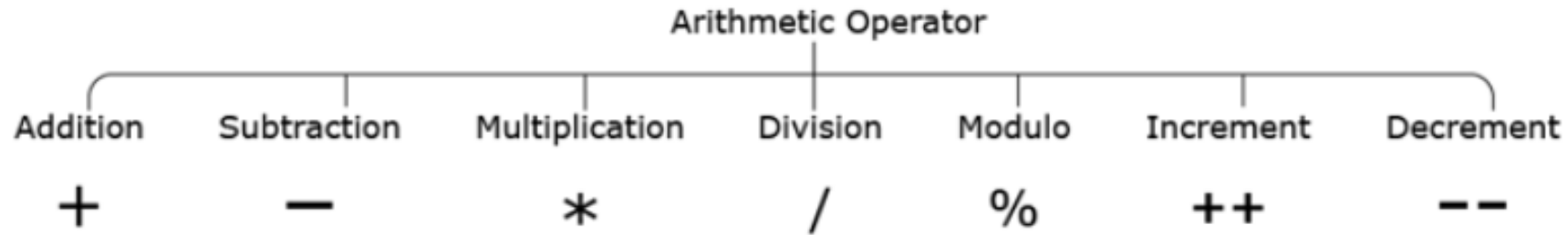
Operators are categorized into **unary, binary, and ternary** based on the number of operands on which they operate in an expression.

JavaScript supports the following types of operators.



## **1.8.2 Arithmetic Operator**

Arithmetic operators are used for performing arithmetic operations



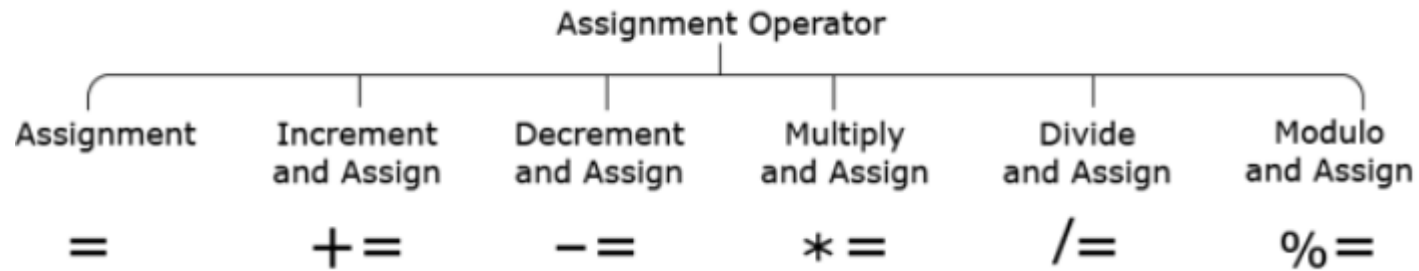
### **1.8.2.1 Arithmetic Operator on String**

Arithmetic operators are used for performing arithmetic operations

Arithmetic operator '+' when used with string type results in the concatenation.

### 1.8.3 Assignment Operator

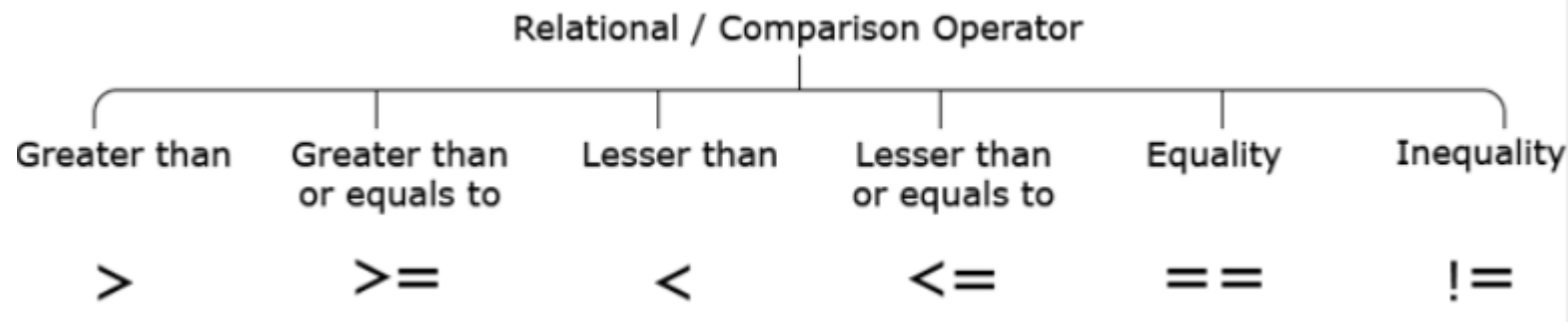
Assignment operators are used for assigning values to the variables.



### 1.8.4 Relational Operator.

Relational operators are used for **comparing values** and the **result of comparison is always either true or false**.

Relational operators shown below do **implicit data type conversion** of one of the operands before comparison.

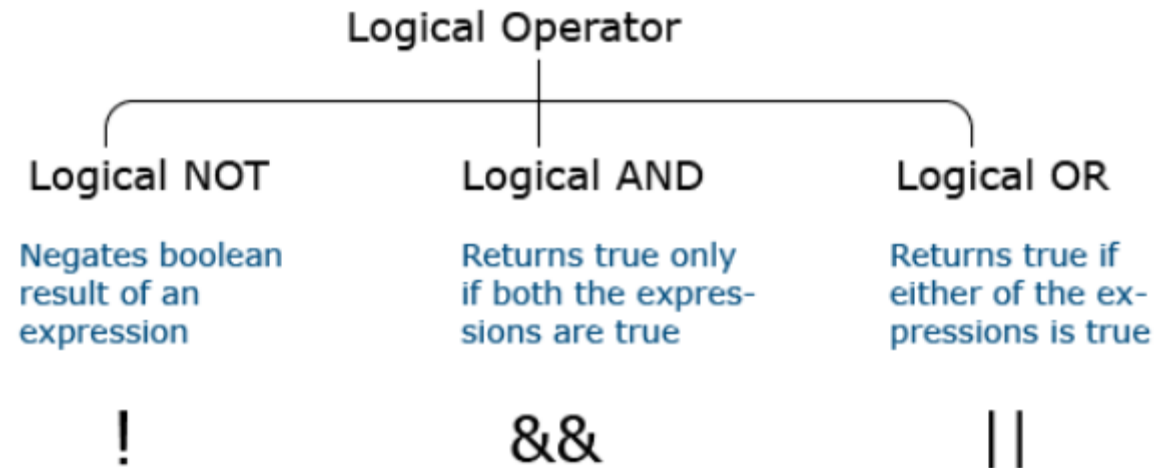


### 1.8.4 Strict Equality

**Strict equality (===)** and **Strict inequality (!==)** operators consider only values of the same type to be equal. Hence Strict equality and Strict inequality operators are highly recommended to determine whether 2 given values are equal or not.

### 1.8.5 Logical Operator

Logical operators allow a program to decide based on multiple conditions. Each operand is considered a condition that can be evaluated to true or false.



### 1.8.6 TypeOf Operator

"**typeof**" is an operator in JavaScript.

JavaScript is a loosely typed language i.e.; the type of variable is decided at runtime based on the data assigned to it. This is also called dynamic data binding.

As programmers, if required we can use this operator `typeof` to find the data type of a JavaScript variable.

```
typeof "Hello World" // String  
typeof 2.14 // number
```

## 1.9 Statement

- Statements are instructions in JavaScript that must be executed by a web browser.
- JavaScript code is made up of a sequence of statements and is **executed in the same order as they are written**.
- **White (blank) spaces** in statements are **ignored**.
- It is **optional** to end each JavaScript statement with a **semicolon**. But it is highly recommended to use it as it avoids possible misinterpretation of the end of the statements by JavaScript engine.

## 1.10 Expression

While writing client-logic in JavaScript, we often **combine variables and operators to do computations**. This is achieved by writing expressions.

```
var operator1 = 10;  
var operator2 = 50;  
var product = operator1 * operator2;
```

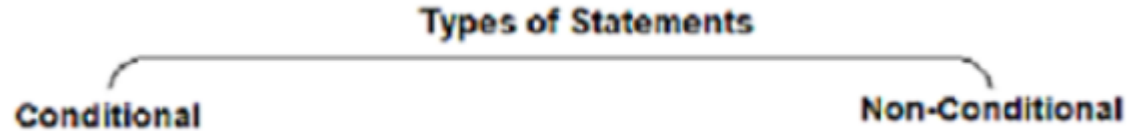


EXPRESSION

STATEMENT



### 1.9.1 Type Of Statements:



#### **Conditional Statements :**

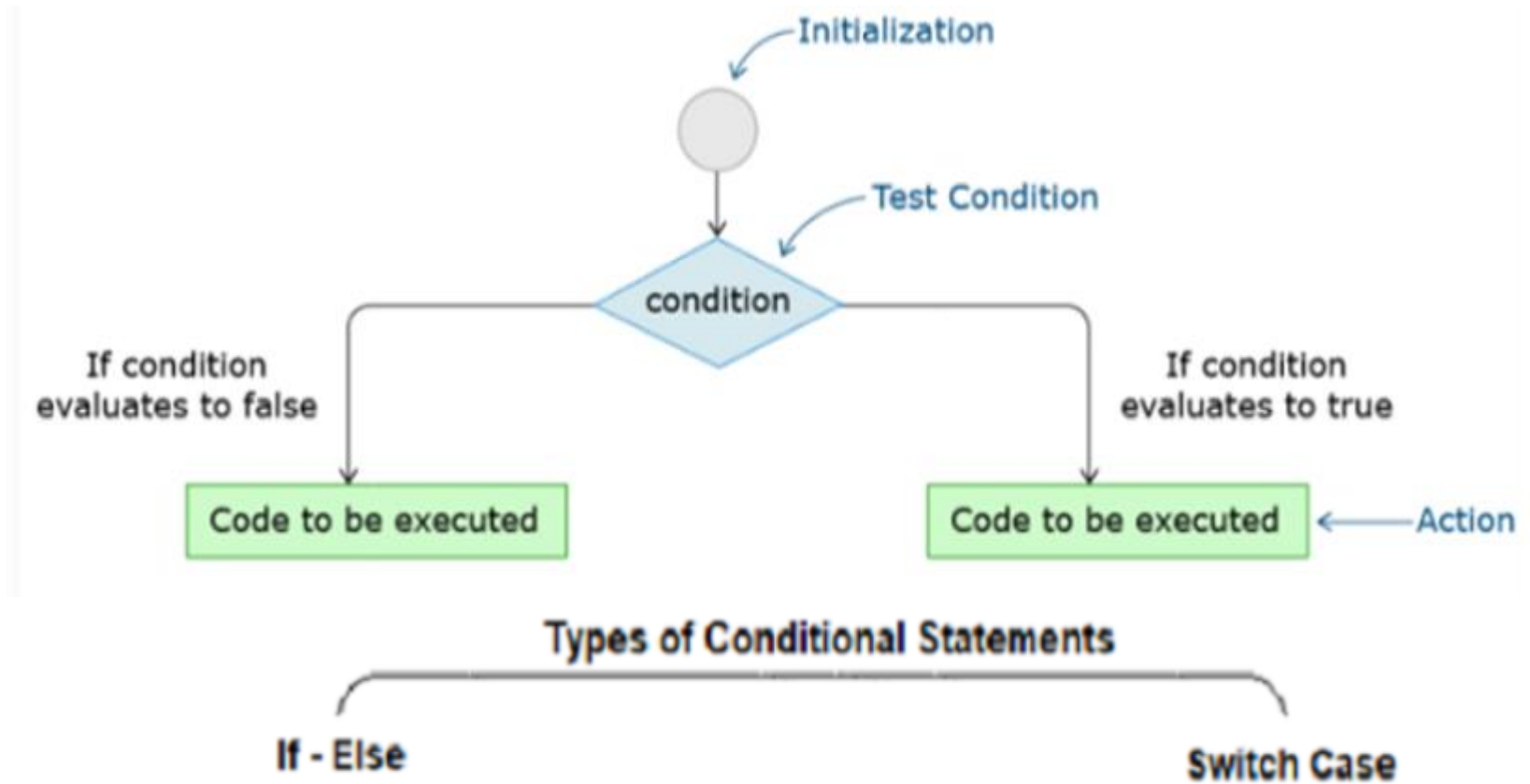
- Conditional statements help you to decide based on certain conditions.
- These conditions are specified by a set of conditional statements having Boolean expressions that are evaluated to a Boolean value true or false.

#### **Non-Conditional Statements :**

- Non-Conditional Statements are those statements that do not need any condition to control the program execution flow.

### 1.9.2 Types of Conditional Statement

Conditional statements help in performing different actions for different conditions. It is also termed as **decision-making** statements.



### 1.9.2.1 If Statement

if statement evaluates the expression given in its parentheses giving a boolean value as the result. You can have multiple if statements for multiple choice of statements inside an if statement.

We have different flavors of If-else statement:

- Simple If statement
- If -else
- If-else-if ladder

#### 1.9.2.1.1 If Statement

Inside Statements are executed if condition is true.

#### 1.9.2.1.2 If Else Statement

If condition is true, true block is executed. If false, else block is executed

#### 1.9.2.1.3 If Else If Ladder

If condition is true, if block is executed. If false, and **else if** block is true **else if** block is executed. If **else if** is also false , **else** block is executed.

## 1.9.2.2 Switch Statement

The Switch statement is used to select and evaluate one of the many blocks of code.

```
switch (expression) {  
    case value1: code block;  
                break;  
    case value2: code block;  
                break;  
    case valueN: code block;  
                break;  
    default: code block;  
}
```

**1.9.2.2.1 Switch Statement With Break** - Break statement moves the execution control out of the switch statement

**1.9.2.2.2 Switch Statement Without Break** – Control does not come out of switch statement

**1.9.2.2.3 Default Statement** – When none of case match , default is executed

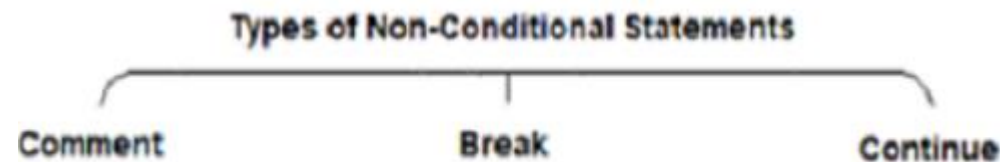
### **1.9.2.3 Ternary Operator**

It is a conditional operator that evaluates to one of the values based on whether the condition is true or false.

It happens to be the only operator in JavaScript that takes three operands. It is mostly used as a shortcut of 'if-else' condition.

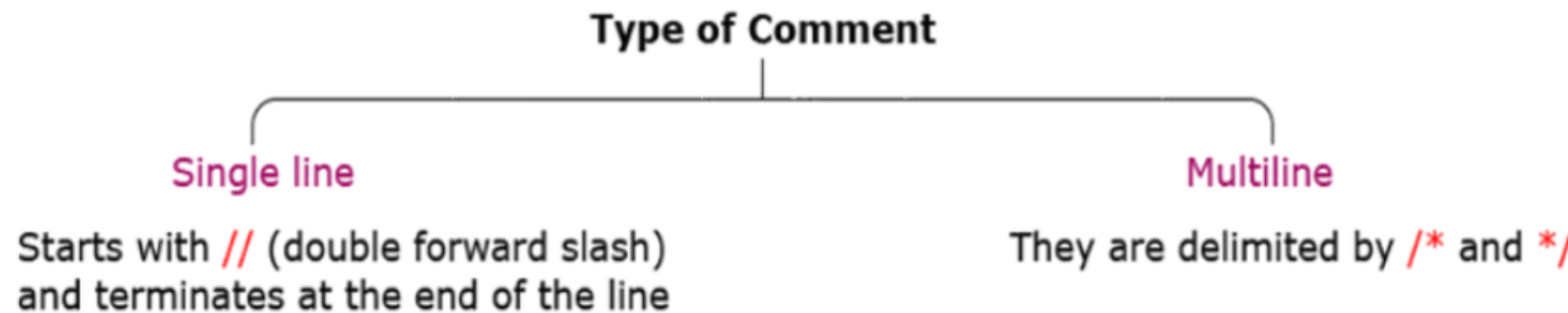
### 1.9.3 Non Conditional Operators

Non-Conditional Statements are those statements that do not need any condition to control the program execution flow.



#### 1.9.3.1 Comment

At times, we may not want to execute a certain portion of our code or maybe we want to add information in our code that explains the significance of the line of code being written.



### 1.9.3.3 Break

While we are iterating over the block of code getting executed within the loop, we may want to exit the loop if a certain condition is met.

'**break**' statement is used to terminate the loop and transfer control to the first statement following the loop.

```
// Break
var counter = 0;
for (var loop = 0; loop < 5; loop++) {
    if (loop == 3)
        break;
    counter++;
}
```

loopVar	counter
0	1
1	2
2	3
3	Loop terminated. counter = 3.

### 1.9.3.4 Continue

There are times when during the iteration of the block of code within the loop, we may want to skip the block execution for a specific value and then continue executing the block for all the other values. JavaScript gives us a 'continue' statement to handle this.

Continue statement is used to **terminate the current iteration of the loop and continue execution of the loop with the next iteration.**

```
// Continue
var counter = 0;
for (var loop = 0; loop < 5; loop++) {
    if (loop == 3)
        continue;
    counter++;
}
```

loopVar	counter
0	1
1	2
2	3
3	Iteration terminated. Hence counter is not incremented.
4	4



### **Tryout 1:**

#### **Problem Statement:**

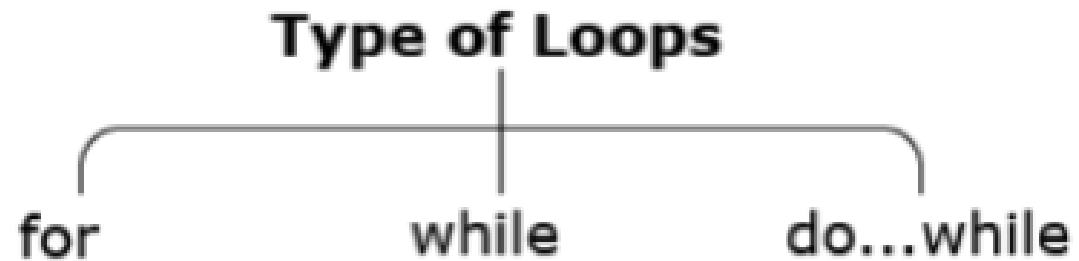
Write a JavaScript code to make online booking of theatre tickets and calculate the total price based on the below conditions:

- 1.If seats to be booked are not more than 2, the cost per ticket remains \$9.
- 2.If seats are 5 or more, booking is not allowed.
- 3.If seats to be booked are more than 2 but less than 5, based on the number of seats booked, do the following:
  1. Calculate total cost by applying discounts of 5, 7, 9, 11 percent, and so on for customer 1,2,3 and 4.

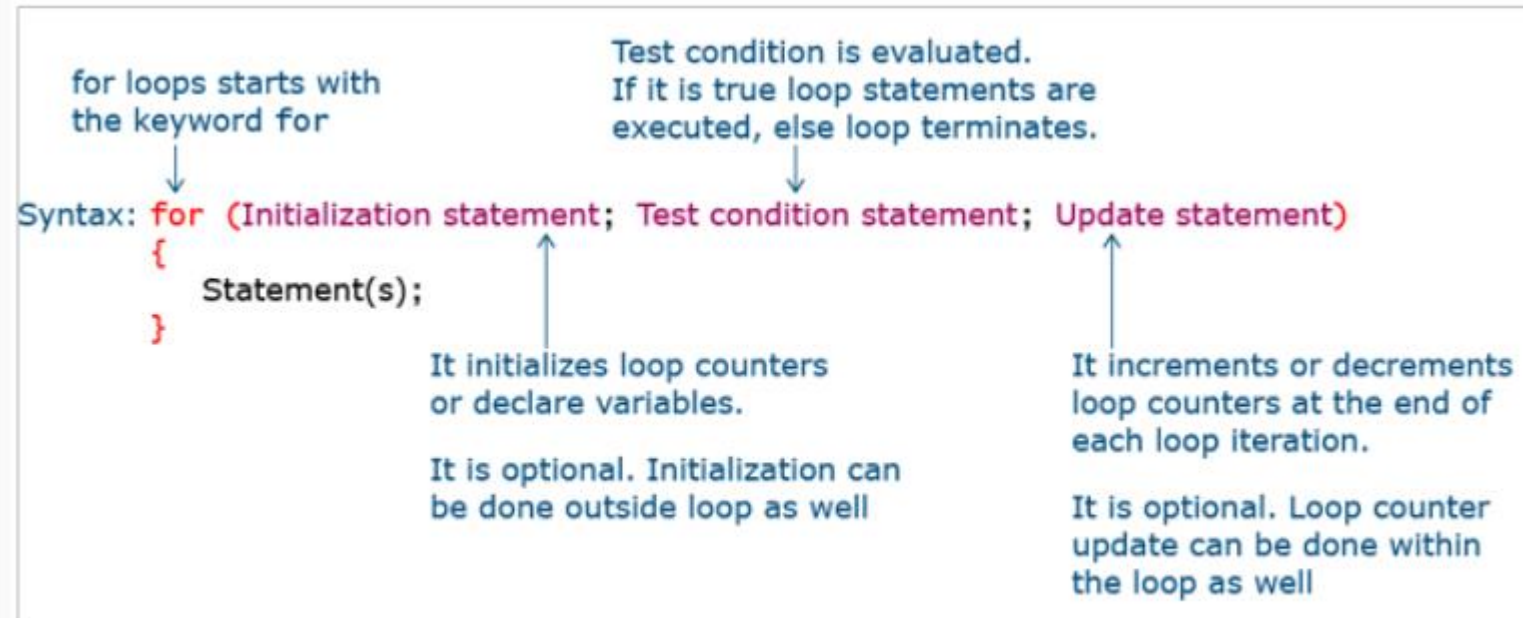
## **1.10 Loops**

To perform repeated actions , we use loops

### **1.10.1 Type of loops**

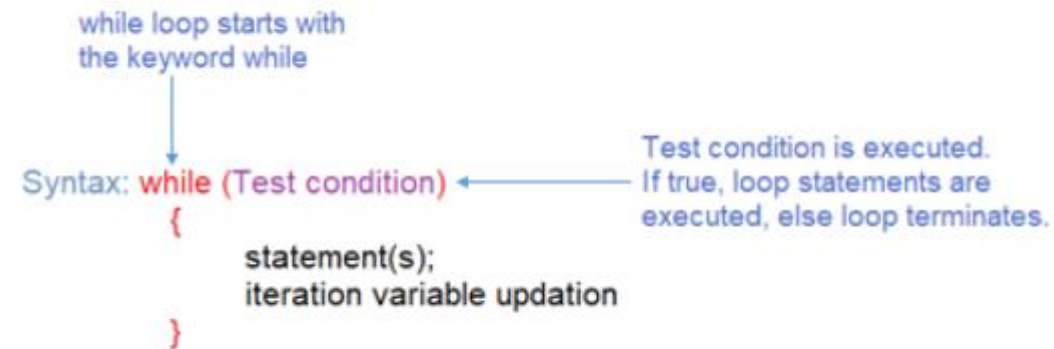


## 1.10.2 For Loop



### 1.10.3 While Loop

'while' loop is used when the block of code is to be executed as long as the specified condition is true.



The value for the variable used in the test condition should be updated inside the loop only.

### 1.10.4 Do While Loop

'do-while' is a variant of 'while' loop.

This will execute a block of code once before checking any condition.

Then, after executing the block it will evaluate the condition given at the end of the block of code.

Now the statements inside the block of code will be repeated till condition evaluates to true.

Syntax: do

{

Statement(s);

}while (Test condition)



Test condition is evaluated.  
If it is true loop statements are  
executed, else loop terminates.

## 1.11 Functions

The JavaScript engine can execute JavaScript code in two different modes:

- **Immediate mode**

- As soon as the webpage loads on the browser, JavaScript code embedded inside it, executes without any delay.

- **Deferred mode**

- Execution of JavaScript code is deferred or delayed until any user action like data input, button click, drop-down selection, etc. takes place.
- The JavaScript code that we have seen until now is running in immediate mode. As soon as we load the page in the browser, the script gets executed line by line without any delay.
- But in real-world application development, it is not possible to wait for sequential execution of the code written for the huge applications. JavaScript provides a solution to this problem in the form of JavaScript functions.
- Functions are one of the integral components of JavaScript. A JavaScript Function is a set of statements that performs a specific task. They become a reusable unit of code.

### **1.11.1 Type of Functions**

JavaScript has two types of functions.

#### **1. User-defined functions**

- JavaScript allows us to write our own functions called as user-defined functions. The user-defined functions can also be created using a much simpler syntax called arrow functions.

#### **2. Built-in functions**

- JavaScript provides several predefined functions that perform tasks such as displaying dialog boxes, parsing a string argument, timing-related operations, and so on.

### 1.11.2 Function Declaration and invocation

To use a function, we need to define or declare a function and then we can invoke it from anywhere in the program.

A function declaration also called a function definition, consists of the function keyword, followed by:

- Function name
- A list of parameters to the function separated by commas and enclosed in parentheses, if any.
- A set of JavaScript statements that define the function, also called a function body, enclosed in curly brackets {...}.

#### Function Declaration

```
function function_name(parameter 1, parameter 2 , ..., parameter n) {  
    //statements to be executed  
}
```

#### Function Invocation

```
function_name(argument 1, argument 2, ..., argument n);
```



### 1.11.3 Function Parameters

Function parameters are the variables that are defined in the function definition and the values passed to the function when it is invoked are called arguments.

In JavaScript, function definition does not have any data type specified for the parameters, and **type checking is not performed on the arguments passed to the function.**

JavaScript **does not throw any error if the number of arguments passed during a function invocation doesn't match with the number of parameters** listed during the function definition. If the number of parameters is more than the number of arguments, then the parameters that have no corresponding arguments are set to **undefined**.

#### 1.11.3.1 Default Parameters

JavaScript introduces an option to assign default values in functions.

#### 1.11.3.2 Rest Parameters

Rest parameter syntax allows us to hold an indefinite number of arguments in the form of an array.

### 1.11.4 Arrow Function

In JavaScript, functions are first-class objects. This means that you can assign a function as a value to a variable.

```
// Arrow Function
let sayHello = function () {
  console.log("Welcome to JavaScript");
};
sayHello();
```

Here a function without a name, called an anonymous function, is assigned to a variable sayHello.

Built-in functions	Description	Example
alert()	It throws an alert box and is often used when user interaction is required to decide whether execution should proceed or not.	alert("Let us proceed");
confirm()	It throws a confirm box where user can click "OK" or "Cancel". If "OK" is clicked, the function returns "true", else returns "false".	let decision = confirm("Shall we proceed?");
prompt()	It produces a box where user can enter an input. The user input may be used for some processing later. This function takes parameter of type string which represents the label of the box.	let userInput = prompt("Please enter your name:");
isNaN()	This function checks if the data-type of given parameter is number or not. If number, it returns "false", else it returns "true".	isNaN(30); //false isNaN('hello'); //true
isFinite()	It determines if the number given as parameter is a finite number. If the parameter value is NaN, positive infinity, or negative infinity, this method will return false, else will return true.	isFinite(30); //true isFinite('hello'); //false
parseInt()	<p>This function parses string and returns an integer number.</p> <p>It takes two parameters. The first parameter is the string to be parsed. The second parameter represents radix which is an integer between 2 and 36 that represents the numerical system to be used and is optional.</p> <p>The method stops parsing when it encounters a non-numerical character and returns the gathered number.</p> <p>It returns NaN when the first non-whitespace character cannot be converted to number.</p>	<pre>parseInt("10"); //10 parseInt("10 20 30"); //10, only the integer part is returned parseInt("10 years"); //10 parseInt("years 10"); //NaN, the first character stops the parsing</pre>
parseFloat()	<p>This function parses string and returns a float number.</p> <p>The method stops parsing when it encounters a non-numerical character and further characters are ignored.</p> <p>It returns NaN when the first non-whitespace character cannot be converted to number.</p>	<pre>parseFloat("10.34"); //10.34 parseFloat("10 20 30"); //10 parseFloat("10.50 years"); //10.50</pre>
eval()	It takes an argument of type string which can be an expression, statement or sequence of statements and evaluates them.	eval("let num1=2; let num2=3;let result= num1 * num2;console.log(result)");

Built-in functions	Description	Example
setTimeout()	It executes a given function after waiting for the specified number of milliseconds. It takes 2 parameters. First is the function to be executed and the second is the number of milliseconds after which the given function should be executed.	<pre>function executeMe(){   console.log("Function says hello!") } setTimeout(executeMe, 3000); //It executes executeMe() after 3 seconds.</pre>
clearTimeout()	It cancels a timeout previously established by calling setTimeout(). It takes the parameter "timeoutID" which is the identifier of the timeout that can be used to cancel the execution of setTimeout(). The ID is returned by the setTimeout().	<pre>function executeMe(){   console.log("Function says hello!") } let timerId= setTimeout(executeMe, 3000); clearTimeout(timerId);</pre>
setInterval()	It executes the given function repetitively. It takes 2 parameters, first is the function to be executed and second is the number of milliseconds. The function executes continuously after every given number of milliseconds.	<pre>function executeMe(){   console.log("Function says hello!"); } setInterval(executeMe,3000); //It executes executeMe() every 3 seconds</pre>
clearInterval()	It cancels the timed, repeating execution which was previously established by a call to setInterval(). It takes the parameter "intervalID" which is the identifier of the timeout that can be used to cancel the execution of setInterval(). The ID is returned by the setInterval().	<pre>function executeMe(){   console.log("Function says hello!"); } let timerId=setInterval(executeMe, 2000); function stopInterval(){   clearInterval(timerId);   console.log("Function says bye to setInterval()!") } setTimeout(stopInterval,5000) //It executes executeMe() every 2 seconds and after 5 seconds, further calls to executeMe() is stopped.</pre>

## 1.12 Class

In 2015 JavaScript introduced the concept of the Class.

- Classes and Objects in JavaScript coding can be created similar to any other Object-Oriented language.
- Classes can also have methods performing different logic using the class properties respectively.
- The new feature like Class and Inheritance ease the development and work with Classes in the application.
- JavaScript is an **object-based language** based on prototypes and allows you to create hierarchies of objects and to have inheritance of properties and their values.
- The Class syntax is built on top of the existing prototype-based inheritance model.

### **Classes**

- In 2015, ECMAScript introduced the concept of classes to JavaScript
- The keyword **class** is used to create a class
- The **constructor** method is called each time the class object is created and initialized.
- The Objects are a real-time representation of any entity.
- We use methods to communicate between various objects, to perform various operations.

### 1.12.1 Static Methods

Just like in other programming languages, we can create static methods in JavaScript using the **static** keyword. Static values can be accessed *only* using the class name and not using **this** keyword. Else it will lead to an error.

### 1.12.2 Inheritance

- In JavaScript, one class can inherit another class using the **extends** keyword. The subclass inherits all the methods ( **both static and non-static** ) of the parent class.
- Inheritance enables the **reusability** and **extensibility** of a given class.
- JavaScript uses prototypal inheritance which is quite complex and unreadable. But, now we have the more friendly extends keyword which makes it easy to inherit the existing classes.
- Keyword **super** can be used to refer to base class methods/constructors from a subclass

## Class Tryout

### **Problem Statement:**

Create an Employee class extending from a base class Person.

### **Approach to the solution:**

- Create a class Person with name and age as attributes
- Add a constructor to initialize the values
- Create a class Employee extending Person with additional attributes role and contact
- The constructor of the Employee to accept the name, age, role and contact where name and age are initialized through a call to super to invoke the base class constructor
- Add a method getDetails() to display all the details of Employee.

The details of the Employee are:

Name: Beron

Age: 24

Role: Technology Analyst

Phone: 9001234567

### **1.13 Events**

When the interaction happens, the event triggers. JavaScript event handlers enable the browser to handle them. JavaScript event handlers invoke the JavaScript code to be executed as a reaction to the event triggered.





## 1.13 Events

Event	Event-handler	Description
click	onclick	When the user clicks on an element, the event handler onclick handles it.
keypress	onkeypress	When the user presses the keyboard's key, event handler onkeypress handles it.
keyup	onkeyup	When the user releases the keyboard's key, the event handler onkeyup handles it.
load	onload	When HTML document is loaded in the browser, event handler onload handles it
blur	onblur	When an element loses focus, the event handler onblur handles it.
change	onchange	When the selection of checked state change for input, select or text-area element changes, event handler onchange handles it.

### 1.13 .1 Wiring the events

Event handlers are associated with HTML elements and are responsible to handle or listen to the event taking place on the respective element.

```
<html-element eventHandler="JavaScript code">
```

```
<p onclick="executeMe();">Para says !!! </p>
```

## 1.14 Exception Handling

### Types of Errors

While coding, there can be three types of errors in the code:

**1.Syntax Error:** When a user makes a mistake in the pre-defined syntax of a programming language, a syntax error may appear.

**2.Runtime Error:** When an error occurs during the execution of the program, such an error is known as Runtime error. The codes which create **runtime errors are known as Exceptions**. Thus, exception handlers are used for handling runtime errors.

**3.Logical Error:** An error which occurs when there is any logical mistake in the program that may not produce the desired output and may terminate abnormally. Such an error is known as Logical error.

### Error Object

When a runtime error occurs, it creates and throws an Error object. Such an object can be used as a base for the user-defined exceptions too. An error object has two properties:

**1.name:** This is an object property that sets or returns an error name.

**2.message:** This property returns an error message in the string form.

## **1.14 Exception Handling**

The **try** statement lets you test a block of code for errors.

The **catch** statement lets you handle the error.

The **throw** statement lets you create custom errors.

The **finally** statement lets you execute code, after try and catch, regardless of the result.

## **1.15 Objects**

In any programming language when we need to code real-world entities, we make use of **variables**. For most of the scenarios, we need a variable that can hold data that represents the collection of properties.

For instance, if we must create an online portal for the car industry, we must model Car as an entity that can hold a group of properties.

Such type of variable in JavaScript is called an Object. **An object consists of state and behavior.**

The State of an entity represents properties that can be modeled as **key-value** pairs.

The Behavior of an entity represents the observable effect of an operation performed on it and is modeled using functions.

### **Example:**

A Car is an object in the real world.

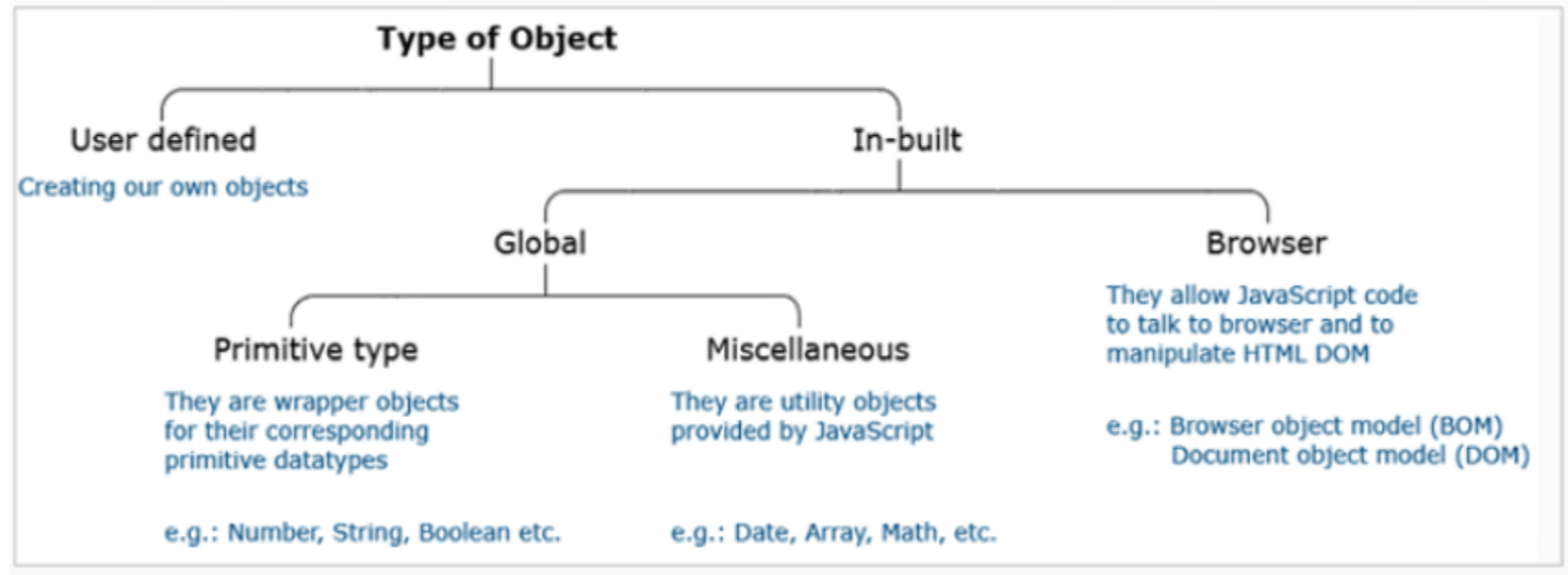
#### **State of Car object:**

- Color=red
- Model = VXI
- Current gear = 3
- Current speed = 45 km / hr
- Number of doors = 4
- Seating Capacity = 5

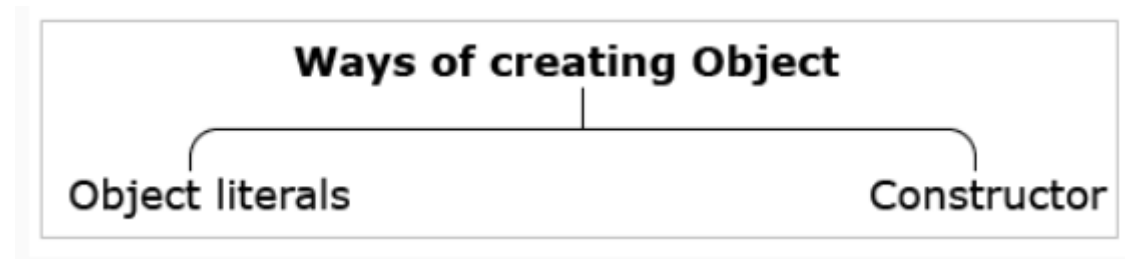
#### **The behavior of Car object:**

- Accelerate
- Change gear
- Brake

## 1.15.1 Types Of Objects



### 1.15.2 Creating Objects



### 1.15.2 Creating Objects Using Object Literals

- Objects can be created using object literal notation.
- Object literal notation is a comma-separated list of **name-value pairs wrapped inside curly braces**.

### 1.15.3 Creating Objects Using Function Constructor

- To construct multiple objects with the same set of properties and methods, function constructor can be used. Function constructor is like regular functions, but it is invoked using a '**new**' keyword.
- '**this**' keyword that we have used here is a JavaScript pointer. It points to an object which owns the code in the current context.
- It does not have any value of its own but is only the substitute for the object reference wherever it is used.



#### 1.15.4 Combining Objects using spread operator

```
let object1Name = {  
  //properties  
};  
let object2Name = {  
  //properties  
};  
let combinedObjectName = {  
  ...object1Name,  
  ...object2Name  
};  
//the combined object will have all the properties of object1 and object2
```

#### 1.15.5 Cloning Objects using spread operator

It is possible to get a copy of an existing object with the help of the spread operator.

```
let copyToBeMade = { ...originalObject };
```

### 1.15.4 Object Properties - Dot and Bracket Operator



### 1.15.5 Object Properties – For In Loop

To work with all the keys of an object, we have a particular form of the loop: for..in. This is a different way from the for() construct.

```
for (key in object) {  
    // executes the body for each key among object properties  
}
```

## **1.16 Global Built In Objects**

The Global object allows us to declare variables and functions that can be accessed anywhere. By default, these are built into the language or the environment.

They are different built-in objects in JavaScript,

- Date
- String
- Math
- RegExp
- JSON

Sr.No.	Method & Description
1	<u>charAt()</u> Returns the character at the specified index.
2	<u>charCodeAt()</u> Returns a number indicating the Unicode value of the character at the given index.
3	<u>concat()</u> Combines the text of two strings and returns a new string.
4	<u>indexOf()</u> Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
5	<u>lastIndexOf()</u> Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
6	<u>localeCompare()</u> Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.
7	<u>length()</u> Returns the length of the string.
8	<u>match()</u> Used to match a regular expression against a string.
9	<u>replace()</u> Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
10	<u>search()</u> Executes the search for a match between a regular expression and a specified string.
11	<u>slice()</u> Extracts a section of a string and returns a new string.
12	<u>split()</u> Splits a String object into an array of strings by separating the string into substrings.
13	<u>substr()</u> Returns the characters in a string beginning at the specified location through the specified number of characters.
14	<u>substring()</u> Returns the characters in a string between two indexes into the string.
15	<u>toLocaleLowerCase()</u> The characters within a string are converted to lower case while respecting the current locale.
16	<u>toLocaleUpperCase()</u> The characters within a string are converted to upper case while respecting the current locale.
17	<u>toLowerCase()</u> Returns the calling string value converted to lower case.
18	<u>toString()</u> Returns a string representing the specified object.
19	<u>toUpperCase()</u> Returns the calling string value converted to uppercase.
20	<u>valueOf()</u> Returns the primitive value of the specified object.

## 1.16 RegExp

A regular expression is a **pattern** of characters.

### Syntax:

/pattern/modifiers

The RegExp object can be constructed using either of the two ways:

- using RegExp constructor
- as a literal value by enclosing within forward-slash (/)

```
let myPattern1 = new RegExp(pattern, modifiers);  
let myPattern2 = /pattern/modifiers
```

### Modifiers:

Modifier	Description
<u>g</u>	Perform a global match (find all matches rather than stopping after the first match)
<u>i</u>	Perform case-insensitive matching
<u>m</u>	Perform multiline matching

# 1.16 RegExp

## Metacharacters

Metacharacter	Description
<code>.</code>	Find a single character, except newline or line terminator
<code>\w</code>	Find a word character
<code>\W</code>	Find a non-word character
<code>\d</code>	Find a digit
<code>\D</code>	Find a non-digit character
<code>\s</code>	Find a whitespace character
<code>\S</code>	Find a non-whitespace character
<code>\b</code>	Find a match at the beginning/end of a word, beginning like this: <code>\bHI</code> , end like this: <code>HI\b</code>
<code>\B</code>	Find a match, but not at the beginning/end of a word
<code>\0</code>	Find a NULL character
<code>\n</code>	Find a new line character
<code>\f</code>	Find a form feed character
<code>\r</code>	Find a carriage return character
<code>\t</code>	Find a tab character
<code>\v</code>	Find a vertical tab character
<code>\xxx</code>	Find the character specified by an octal number xxx
<code>\xdd</code>	Find the character specified by a hexadecimal number dd
<code>\udddd</code>	Find the Unicode character specified by a hexadecimal number dddd

## **1.16 RegExp**

**Brackets** help us define a pattern that enables the search of a given character or a digit in a string or a number.

### **Pattern**

[abc]

[0-9]

(a|b)

[^abc]

[^0-9]

### **Description**

To search in a given string for any of the characters present within the brackets

To search in a given string for any of the digits present within the brackets

To search in a given string for either of the characters separated by '|'

To search in a given string for any of the characters which are not a,b, or c.

To search in a given string for any of the digits which is not between 0-9

## 1.16 RegExp

**Quantifiers** help us define a pattern that enables the search of a set of characters or digits in a string or a number.

### **Pattern**

$n^+$

$n^*$

$n^?$

$?=n$

$n\{x\}$

$n\{x,\}$

$n\{x,y\}$

### **Description**

To check if the given string contains at least one “n”.

To check if the given string contains at least zero or more occurrences of n.

To check if the given string contains at least zero or one occurrence of n.

To match any string that is followed by a specific string n.

To match the given string containing X n’s.

To match the given string contain at least X n’s.

To match the given string containing X to Y n’s.



## 1.16 RegExp

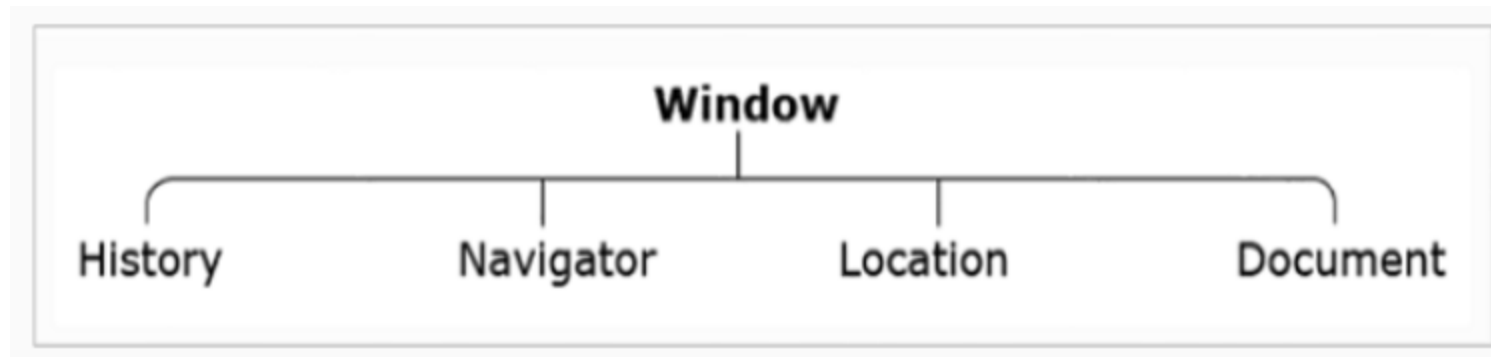
### Regexp Methods

Method	Description
<a href="#"><u>compile()</u></a>	Deprecated in version 1.5. Compiles a regular expression
<a href="#"><u>exec()</u></a>	Tests for a match in a string. Returns the first match
<a href="#"><u>test()</u></a>	Tests for a match in a string. Returns true or false
<a href="#"><u>toString()</u></a>	Returns the string value of the regular expression

## 1.17 Browser Object Model

The dynamic manipulation of an HTML page on the client-side itself is achieved with the help of built-in browser objects. They allow JavaScript code to programmatically control the browser and are collectively known **as Browser Object Model (BOM)**.

For programming purposes, the BOM model virtually splits the browser into different parts and refers to each part as a different type of built-in object. BOM is a hierarchy of multiple objects. 'window' object is the root object and consists of other objects in a hierarchy, namely, 'history' object, 'navigator' object, 'location' object, and 'document' object.



## **1.18 Document**

The HTML web page that gets loaded on the browser is represented using the '**document**' object of the BOM model.

This object considers the web page as a **tree** which is referred to as Document Object Model(DOM). Each node of this tree represents **HTML elements** in the page as '**element**' object and its **attributes** as **properties** of the 'element' object.

With document object , we can

- change all the HTML elements in the page
- change all the HTML attributes in the page
- change all the CSS styles in the page
- remove existing HTML elements and attributes
- add new HTML elements and attributes
- react to all existing HTML events in the page
- create new HTML events in the page

## 1.18 Document

What is the HTML DOM?

The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:

- The HTML elements as **objects**
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements

In other words: **The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

### Finding HTML Elements:

Method	Description
<code>document.getElementById(<i>id</i>)</code>	Find an element by its element id
<code>document.getElementsByTagName(<i>name</i>)</code>	Find elements by tag name
<code>document.getElementsByClassName(<i>name</i>)</code>	Find elements by class name

## Changing HTML Elements

Property	Description
<i>element.innerHTML = new html content</i>	Change the inner HTML of an element
<i>element.attribute = new value</i>	Change the attribute value of an HTML element
<i>element.style.property = new style</i>	Change the style of an HTML element
Method	Description
<i>element.setAttribute(attribute, value)</i>	Change the attribute value of an HTML element

## Adding and deleting Elements

Method	Description
<i>document.createElement(element)</i>	Create an HTML element
<i>document.removeChild(element)</i>	Remove an HTML element
<i>document.appendChild(element)</i>	Add an HTML element
<i>document.replaceChild(new, old)</i>	Replace an HTML element
<i>document.write(text)</i>	Write into the HTML output stream

## Adding Event Handlers

Method	Description
<code>document.getElementById(<i>id</i>).onclick = function(){<i>code</i>}</code>	Adding event handler code to an onclick

# Finding HTML objects

Property	Description	DOM
document.anchors	Returns all <a> elements that have a name attribute	1
document.applets	Deprecated	1
document.baseURI	Returns the absolute base URI of the document	3
document.body	Returns the <body> element	1
document.cookie	Returns the document's cookie	1
document.doctype	Returns the document's doctype	3
document.documentElement	Returns the <html> element	3
document.documentMode	Returns the mode used by the browser	3
document.documentURI	Returns the URI of the document	3
document.domain	Returns the domain name of the document server	1
document.domConfig	Obsolete.	3
document.embeds	Returns all <embed> elements	3
document.forms	Returns all <form> elements	1
document.head	Returns the <head> element	3
document.images	Returns all <img> elements	1
document.implementation	Returns the DOM implementation	3
document.inputEncoding	Returns the document's encoding (character set)	3
document.lastModified	Returns the date and time the document was updated	3
document.links	Returns all <area> and <a> elements that have a href attribute	1
document.readyState	Returns the (loading) status of the document	3
document.referrer	Returns the URI of the referrer (the linking document)	1
document.scripts	Returns all <script> elements	3
document.strictErrorChecking	Returns if error checking is enforced	3
document.title	Returns the <title> element	1
document.URL	Returns the complete URL of the document	1

### **The getElementById Method**

The most common way to access an HTML element is to use the id of the element.

### **The innerHTML Property**

The easiest way to get the content of an element is by using the innerHTML property.

The innerHTML property is useful for getting or replacing the content of HTML elements.

### **document.write()**

In JavaScript, document.write() can be used to write directly to the HTML output stream



## **1.18 Document**

### **getElementById(x)**

Finds element with id 'x' and returns an object of type element

### **getElementsByTagName(x)**

Find element(s) whose tag name is 'x' and return NodeList, which is a list of element objects.

### **getElementsByClassName()**

Find element(s) whose class attribute's values is 'x' and returns NodeList, which is list of element objects

### **querySelectorAll()**

Find element(s) by CSS selectors and return NodeList, which is a list of element objects.

- the **body** returns body element. **Usage:** document.body;
- the **forms** return all form elements. **Usage:** document.forms;
- the **head** returns the head element. **Usage:** document.head;
- the **images** return all image elements. **Usage:** document.images;

To manipulate the content of HTML page we can use the following properties of 'element' object, given by DOM API:

### **innerHTML**

It gives access to the content within HTML elements like div, p, h1, etc. You can set/get a text.

### **attribute**

It is used to set new values to given attributes.

Content and style for a given HTML page can be modified using the BOM model's object 'document'.

## **1.19 Window**

Consider a scenario where you do not want to update the HTML page but only certain properties of the browser window on which it is rendered. Maybe, you want to navigate to a different URL and bring a new web page, or you want to close the web page or you want to store some data related to the web page. Well, to implement this, we would need an object that represents the entire browser window and allows us to access and manipulate the window properties. BOM model provides us 'window' object.

This object resides on top of the BOM hierarchy. Its methods give us access to the toolbars, status bars, menus, and even the HTML web page currently displayed.

### **innerHeight**

This property holds the inner height of the window's content area.

### **innerWidth**

This property holds the inner width of the window's content area.

### **outerHeight**

This property holds the outer height of the window including toolbars and scrollbars.

### **outerWidth**

This property holds the outer width of the window including toolbars and scrollbars.

## **1.19 Window**

### **localStorage**

This property allows access to object that stores data without any expiration date

### **sessionStorage**

This property allows access to objects that store data valid only for the current session.

**open()** method, opens a new window. Usage: window.open("http://www.xyz.com");

**close()** method, closes the current window. Usage: window.close();

**sessionStorage is similar to localStorage** ; the difference is that while data in localStorage doesn't expire, data in sessionStorage is cleared when the page session ends. ... A page session lasts as long as the tab or the browser is open, and survives over page reloads and restores.

## **1.20 History**

If required, BOM also gives us a specific object to target only one of the window properties. For example, if we are only concerned about the list of URLs that have been visited by the user and we do not need any other information about the browser, BOM gives us the 'history' object for this. It provides programmatic navigation to one of the URLs previously visited by the user. The following are the properties or methods that help us do it.

### **Property:**

**length** returns the number of elements in the History list. **Usage:** history.length;

### **Methods:**

**back()** method, loads previous URL from history list. **Usage:** history.back();

**forward()** method, loads next URL from history list. **Usage:** history.forward();

**go()** method, loads previous URL present at the given number from the history list.

## **1.20 Navigation**

It contains information about the client, that is, the browser on which the web page is rendered. The following properties and methods help us get this information.

### **appName**

Returns the name of the client.

### **appVersion**

Returns platform (operating system) and version of the client (browser).

### **Platform**

Returns the name of the user's operating system.

## **1.21 Location**

BOM hierarchy has a 'location' object which contains information about the current URL in the browser window. The information can be accessed or manipulated using the following properties and methods.

### **href**

It contains the entire URL as a string.

### **hostname**

It contains the hostname part of the URL.

### **port**

It contains a port number associated with the URL.

## **1.21 Location**

### **assign()**

Loads new HTML document.

### **reload()**

Reloads current HTML.

## 1.22 Array

Array in JavaScript is an object that allows storing **multiple values in a single variable**.  
An array can store values of **any datatype**.

### Note:

The elements of the array are accessed using an index position that starts from **0** and ends with the value equal to the **length of the array minus 1**.

### 1.22.1 Creating Arrays:

Arrays can be created using the literal notation or array constructor.

#### Array literal notation:

```
let numArr = [1, 2, 3, 4];
```

### 1.22.2 Array Constructor:

Arrays can be created using the Array constructor with a **single parameter which denotes the array length**. If more than one argument is passed to the Array constructor, a new Array with the given elements is created.

```
let myArray = new Array(arrayLength);
```

### 1.22.3 Looping Array

- For loop
- For of

### 1.22.4 Array Property

#### Property

length

#### Description

It is a read-only property. It returns the length of an array, i.e., number of elements in an array

#### Example

```
let myArray = ["Windows", "iOS", "Android"];  
console.log("Length = " + myArray.length);  
//Length = 3
```

## 1.22.5 Array Methods

Methods	Description	Example
push()	Adds new element to the end of an array and return the new length of the array.	<pre>let myArray = ["Android", "iOS", "Windows"]; myArray.push("Linux"); console.log(myArray); // ["Android", "iOS", "Windows", "Linux"]</pre>
pop()	Removes the last element of an array and returns that element.	<pre>let myArray = ["Android", "iOS", "Windows"]; console.log(myArray.pop()); // Windows console.log(myArray); // ["Android", "iOS"]</pre>
shift()	Removes the first element of an array and returns that element.	<pre>let myArray = ["Android", "iOS", "Windows"]; console.log(myArray.shift()); // Android console.log(myArray); // ["iOS", "Windows"]</pre>
unshift()	Adds new element to the beginning of an array and returns the new length	<pre>let myArray = ["Android", "iOS", "Windows"]; myArray.unshift("Linux"); console.log(myArray); // ["Linux", "Android", "iOS", "Windows"]</pre>
splice()	<p>Change the content of an array by inserting, removing, and replacing elements. Returns the array of removed elements.</p> <p>Syntax: array.splice(index,deleteCount,items); index = index for new item deleteCount = number of items to be removed, starting from index next to index of new item items = items to be added</p>	<pre>let myArray = ["Android", "iOS", "Windows"]; //inserts at index 1 myArray.splice(1, 0, "Linux"); console.log(myArray); // ["Android", "Linux", "iOS", "Windows"]</pre>
slice()	<p>Returns a new array object copying to it all items from start to end(exclusive) where start and end represents the index of items in an array. The original array remains unaffected</p> <p>Syntax: array.slice(start,end)</p>	<pre>let myArray=["Android","iOS","Windows"]; console.log(myArray.slice(1,3)); // ["iOS", "Windows"]</pre>
concat()	Joins two or more arrays and returns joined array.	<pre>let myArray1 = ["Android","iOS"]; let myArray2 = ["Samsung", "Apple"]; console.log(myArray1.concat(myArray2)); // ["Android", "iOS", "Samsung", "Apple"]</pre>



## 1.22.5 Array Methods

`indexOf()` Returns the index for the first occurrence of an element in an array and -1 if it is not present

```
let myArray = ["Android","iOS","Windows","Linux"];
console.log(myArray.indexOf("iOS")); // 1
console.log(myArray.indexOf("Samsung"));
// -1
```

Returns the value of the first element in an array that passes a condition specified in the callback function. Else, returns undefined if no element passed the test condition.

Syntax:

`find()` `array.find(callback(item,index,array))`  
callback is a function to execute on each element of the array  
item value represents the current element in the array  
index value indicates index of the current element of the array  
array value represents array on which `find()` is used,  
index and array are optional  
Returns the index of the first element in an array that passes a condition specified in the callback function.  
Returns -1 if no element passes the condition.

```
let myArray = ["Android", "iOS", "Windows", "Linux"];
let result = myArray.find(element => element.length > 5);
console.log(result); //Android
```

Syntax:

`findIndex()` `Array.findIndex(callback(item,index,array));`  
callback is a function to execute on each element of the array  
item value represents current element in the array  
index represents index of the current element of the array  
array represents array on which `findIndex()` is used.  
index and array are optional  
Creates a new array with elements that passes the test provided as a function.

```
let myArray = ["Android", "iOS", "Windows", "Linux"];
let result = myArray.findIndex(element => element.length > 5);
console.log(result) //0
```

Syntax:

`filter()` `array.filter(callback(item,index,array))`  
callback is the Function to test each element of an array  
item value represents the current element of the array  
index value represents Index of current element of the array  
array value indicates array on which `filter()` is used.

```
let myArray = ["Android", "iOS", "Windows", "Linux"];
let result = myArray.filter(element => element.length > 5);
console.log(result)
//[ "Android","Windows"]
```

## 1.22.5 Array Methods

Method	Description	Example
forEach()	<p>Iterates over an array to access each indexed element inside an array.</p> <p>Syntax:</p> <p>array.forEach(callback(item,index,array))</p> <p>callback is a function to be executed on each element of an array</p> <p>item value represents current element of an array</p> <p>index value mentions index of current element of the array</p> <p>array represents the array on which forEach() is called</p>	<pre>let myArray = ["Android", "iOS", "Windows"]; myArray.forEach((element, index) =&gt;   console.log(index + "-" + element)); //0-Android //1-iOS //2-Windows //3-Linux</pre>

## 1.22.5 Array Methods

Methods	Description	Example
map()	<p>Creates a new array from the results of the calling function for every element in the array.</p> <p>Syntax:</p> <pre>array.map(callback(item,index,array))</pre> <p>callback is a function to be run for each element in the array item represents the current element of the array index value represents index of the current element of the array array value represents array on which forEach() is invoked</p>	<pre>let numArr = [2, 4, 6, 8]; let result = numArr.map(num=&gt;num/2); console.log(result); //[ 1, 2, 3, 4 ]</pre> <pre>let myArray = ["Android", "iOS", "Windows"]; console.log(myArray.join()); // Android,iOS,Windows console.log(myArray.join('-')); // Android-iOS-Windows</pre>
join()	<p>Returns a new string by concatenating all the elements of the array, separated by a specified operator such as comma. Default separator is comma</p>	
reduce()	<p>Executes a defined function for each element of passed array and returns a single value</p> <p>Syntax:</p> <pre>array.reduce(callback(accumulator, currentValue, index,array),initialValue)</pre> <p>callback is a function to be executed on every element of the array accumulator is the initialValue or previously returned value from the function. currentValue represents the current element of the passed array index represents index value of the current element of the passed array array represents the array on which this method can be invoked. initialValue represents the Value that can be passed to the function as an initial value. currentValue,index,array and initialValue are optional.</p>	<pre>const numArr = [1, 2, 3, 4]; // 1 + 2 + 3 + 4 console.log(numArr.reduce( (accumulator, currentVal) =&gt; accumulator + currentVal)); // 10 // 5 + 1 + 2 + 3 + 4 console.log(numArr.reduce( (accumulator, currentVal) =&gt; accumulator + currentVal,5)); // 15</pre>