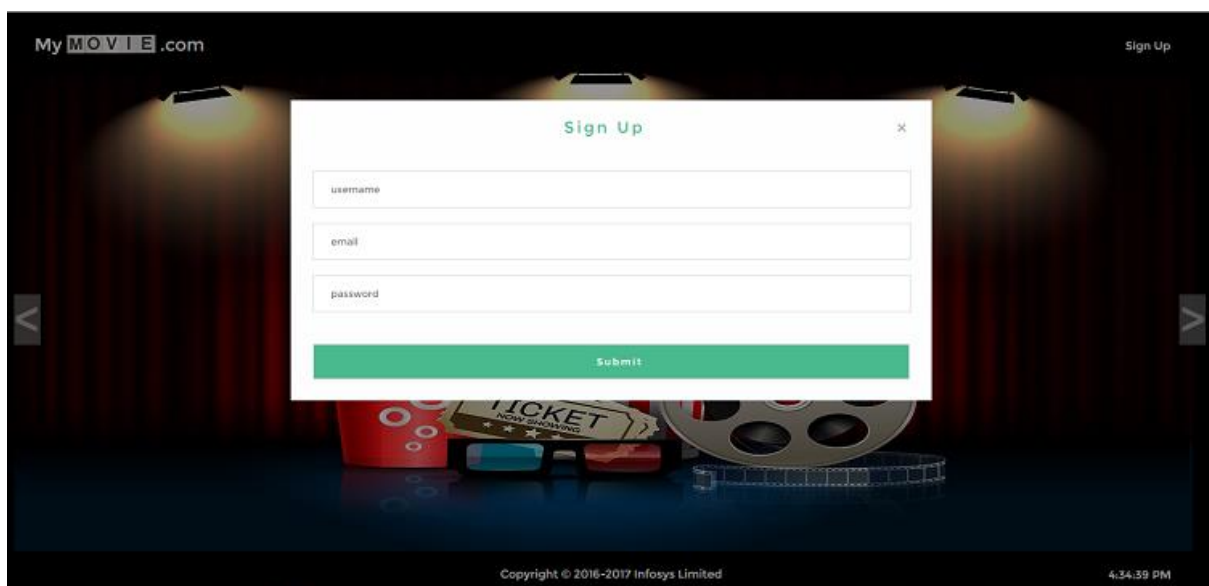


Why we need Javascript?

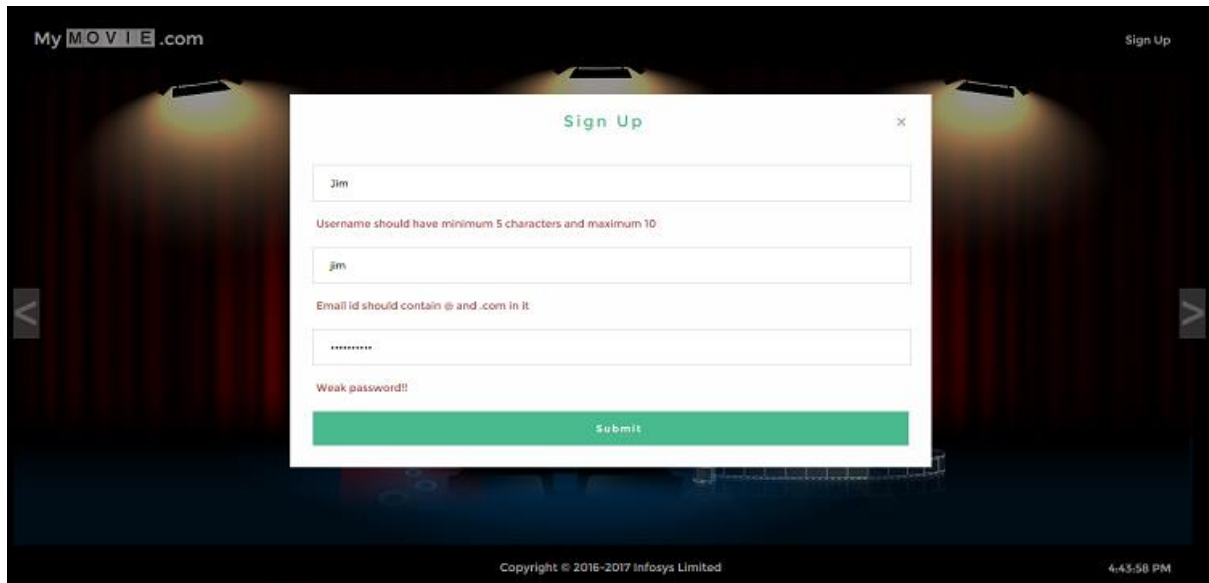
- When an application is loaded on the browser, there is a 'SignUp' link on the top right corner.



- When this link is clicked, the 'SignUp' form is displayed. It contains three fields - 'Username', 'Email', and 'Password' and in some cases a 'Submit' button as well.

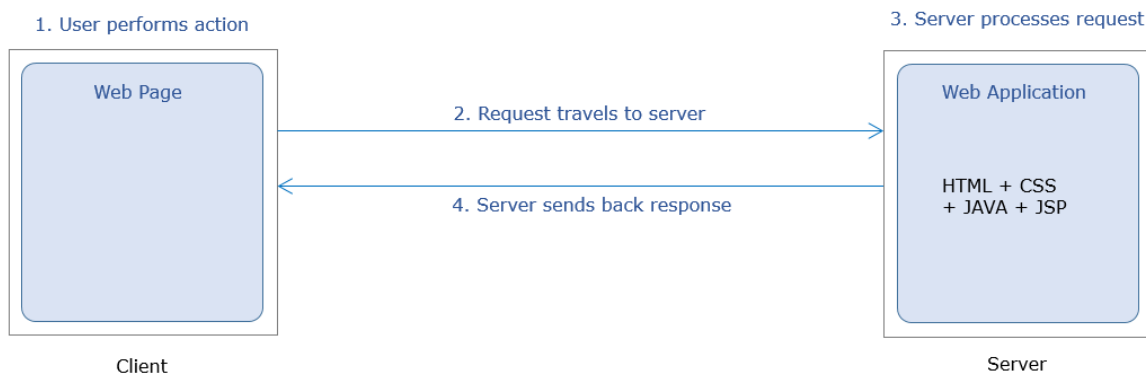


- When data is entered in the fields and the button is clicked, then data entered in the fields will be validated and accordingly, next view page loaded. If data is invalid, an error message is displayed, if valid, the application navigates to homepage



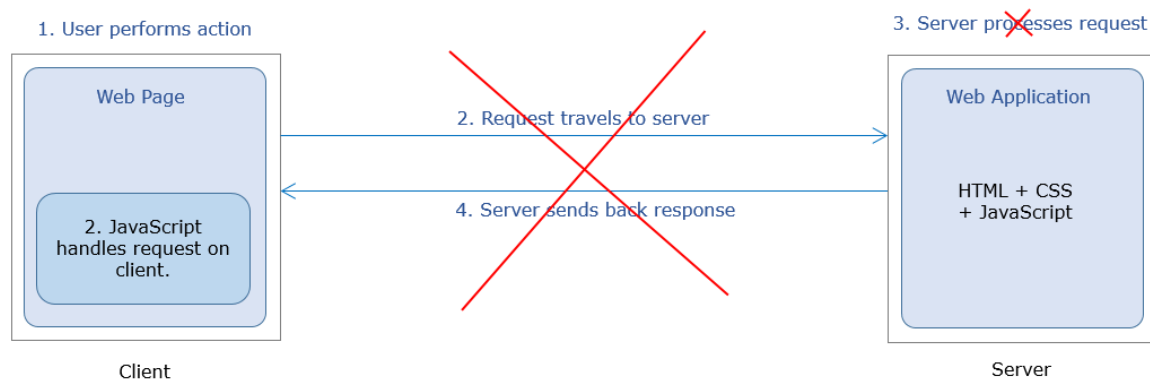
How to handle the user click, validate the user data, and display the corresponding ?

- To implement the requirement of handling user action like a click of a button or link and to respond to these requests by displaying the expected output, server-side languages like Java/JSP can be used as shown in the below diagram.



But server-side languages have certain limitations such as :-

- Multiple request-response cycles to handle multiple user requests
- More network bandwidth consumption
- Increased response time
- If client-side scripting language JavaScript is used then, this can be done without consulting the server as can be seen in the below diagram.



- The home page of MyMovie.com contains the SignUp link. The user performs click action on this link. The user action is handled on the client side itself with the help of the JavaScript code. This code arrives on the client along with the home page of the application.
- When invoked on click of the link, this code executes on the client-side itself to validate the user-entered data and accordingly display the corresponding view.

Following are the advantages of this approach:

- 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.

#### About ES6:

- JavaScript was introduced as a client-side scripting language in 1995.
- ECMAScript established a standard for scripting languages in 1997.
- ES is a parent of many scripting languages like TypeScript, JScript, ActionScript, and JavaScript.
- JavaScript evolved year after year with every new version of ECMAScript introducing new features.
- ES6 also called ES2015.
- ES6 introduces new transformed syntax to extend existing JavaScript constructs to meet the demands of complex applications written in JavaScript.
- Below are some of the features that got introduced which extend the ability of JavaScript to ease the developer's web development
- Block scope variables using let and const keywords
- Template Literals
- Destructuring
- Arrow functions
- Enhanced for loop
- Default and Rest parameters
- Spread operator
- Classes and Objects
- Inheritance
- Native Promises
- Many other built-ins

- The applications which have been implemented in ES6 uses the best practices, new web standards, and cutting-edge features, without using additional frameworks or libraries.
- ES6 is also completely backward compatible. The features like Object Oriented support, New programming constructs, Modules, Templates, support for promises, etc. made ES6 faster.

## What is Javascript?

- JavaScript is the programming language for web users to convert static web pages to dynamic web pages.
- Web page designed using HTML and CSS is static.



- JavaScript combined with HTML and CSS makes it dynamic.



- 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 are 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.
- Below are commonly used browser JavaScript engines.

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

- Even though the latest version of JavaScript has advanced features still developer face challenges in executing the advanced code directly in the browser as:
  - Latest syntax support is still low across browsers & servers (max is less than 70%)
  - The features that are supported differ between browsers (with some overlap)
- None of the IE browsers significantly support the latest features (the new Microsoft Edge browser does)
- So, to overcome these drawbacks, the conversion of JavaScript code written using the latest syntax to browser understandable code takes place using the transpilers such as Babel, Traceur, TypeScript, etc.

Thus, after the code is transpiled, it will be cross-browser compatible.

JavaScript code can be embedded within the HTML page or can be written in an external file.

There are three ways of writing JavaScript depending on the platform :

- Inline Scripting
  - Internal Scripting
  - External Scripting
- When JavaScript code are written within the HTML file itself, it is called internal scripting.
  - Internal scripting, is done with the help of HTML tag : **<script> </script>**
  - This tag can be placed either in the head tag or body tag within the HTML file.

JavaScript code written inside <head> element is as shown below :

```
1. <html>
2. <head>
3.     <script>
4.         //internal script
5.     </script>
6. </head>
7. <body>
8. </body>
9. </html>
10.
```

JavaScript code written inside `<body>` element is as shown below :

```
1.     <html>
2.     <head>
3.     </head>
4.     <body>
5.         <script>
6.             //internal script
7.         </script>
8.     </body>
9. </html>
```

- JavaScript code can be written in an external file also. The file containing JavaScript code is saved with the extension \*.js (e.g. fileName.js)
- To include the external JavaScript file, the script tag is used with attribute 'src' as shown in the below-given code-snippet:

```
1. <html>
2. <head>
3.     <!-- *.js file contain the JavaScript code -->
4.     <script src="*.js"></script>
5. </head>
6. <body>
7. </body>
8. </html>
```

**Example:**  
**Demo.js :-**

```
1. let firstName="Rexha";
2. let lastName ="Bebe";
3. console.log(firstName+" "+lastName);
```

4.

**Demo.html :-**

```
1. <html>
2. <head>
3.     <script src="Demo.js"></script>
4. </head>
5. <body>
6. </body>
7. </html>

8.
```

NOTE: In external file, JavaScript code is not written inside <script> </script> tag.

The below-mentioned points can help you choose between any two ways of writing the script based on some parameters.

|                 | Internal Scripting   | External Scripting   |
|-----------------|--|--|
| Loading time    | Faster, as it is written within the HTML page  | Slower, as it is loaded from server, whenever requested      |
| When to use?    | If number of lines of code is less   | For large amount of code                                     |
| Re-usable       | No, you cannot re-use JavaScript code with any other HTML file   | Yes, Same JavaScript file can be used in multiple HTML files |
| Maintainability | Difficult, as for every change request, each HTML pages containing JavaScript code has to be modified separately | Easy, as only 1 file needs to be modified                    |

Note: External scripting is used throughout this course. But due to the platform built-in feature, HTML code cannot be explicitly linked to external script using <script></script> tag. HTML code that are used to write is automatically linked to JavaScript code written on that page.

## Environment Setup

Following are the options to be used to work with JavaScript:

- Browser (Google Chrome recommended)
- Editor (Visual Studio Code recommended)



As JavaScript works properly on any Browser or OS, you can choose any according to your preference.

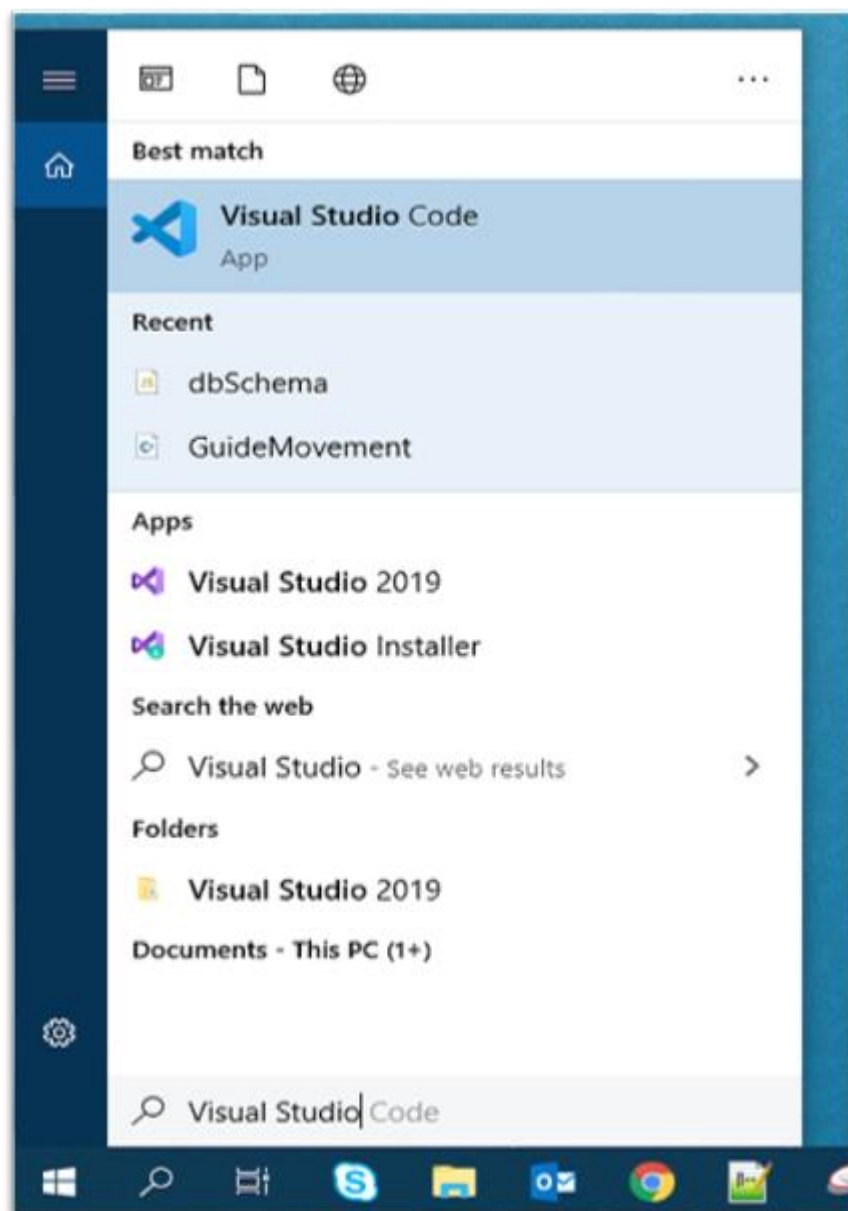
It is also possible to write JavaScript code using Editors. Any simple editors such as notepad or IDE like Visual Studio Code which offers IntelliSense support and syntax error highlighter that makes coding easier can be used.

Go for an IDE that has built-in features.

In Visual Studio Code, extensions can be added as it speeds up the development as well as helps to code in a higher standard by providing linting.

### Steps to Execute the below-mentioned code:

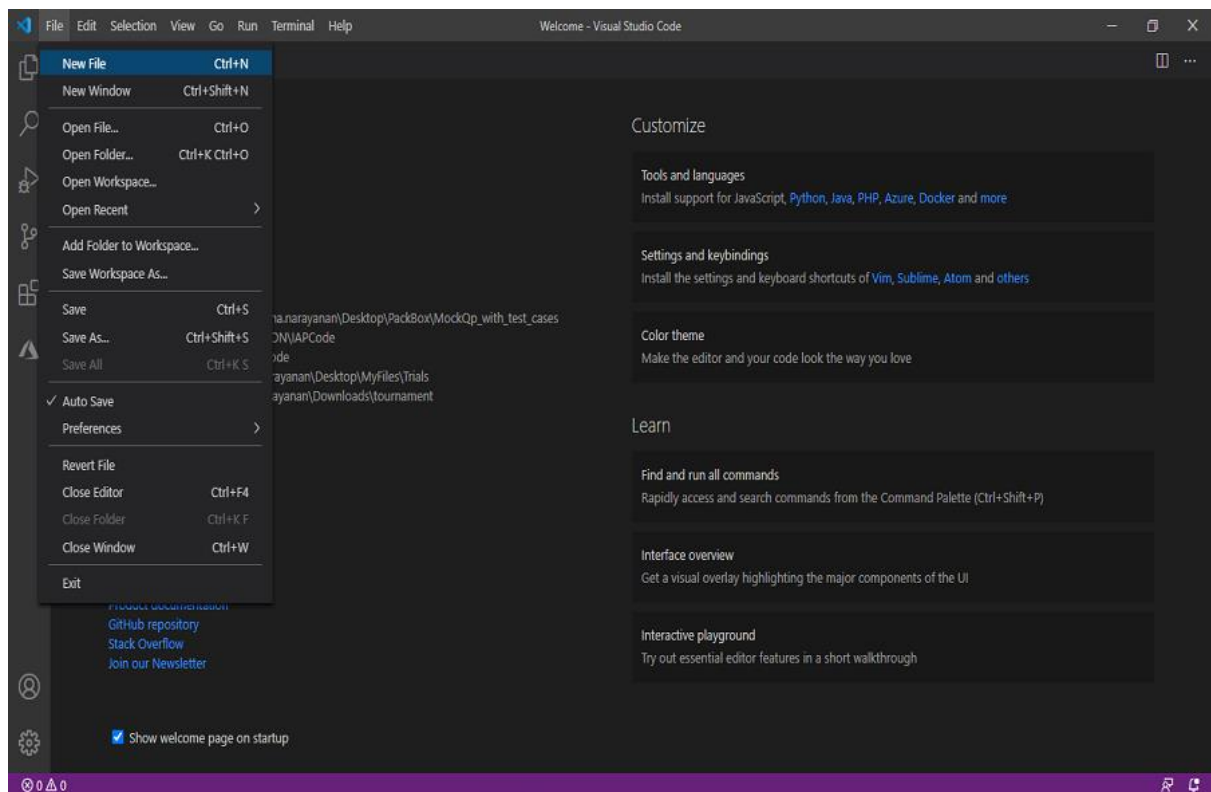
- Open Visual Studio Code from your start menu.







- Once Visual Studio Code is launched, Go to the File menu in the Menu bar, Select the New File option.



- Create the below-mentioned two files (index.js and index.html) and type the below-given code.

## index.js

```
1. console.log("This content is from external JavaScript file");
```

## index.html

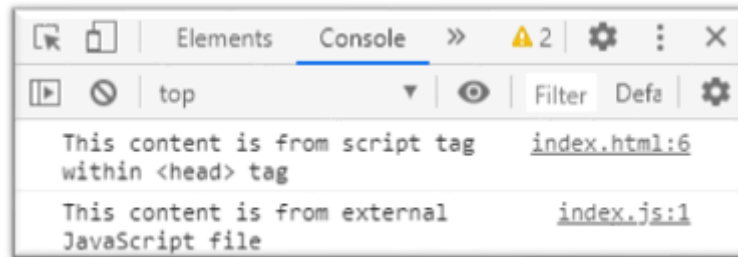
```
1. <html>
2. <head>
3.   <title>JavaScript Introduction</title>
4.   <script>
5.     console.log("This content is from script tag within
6.   <head> tag");
7.   </script>
8.   <script src="./index.js"></script>
9. </head>
10. <body>
11. </body>
```

```
11. </html>
```

Rendering of index.html file will execute the JavaScript code (in index.js) attached to it.

### Output:

Go to the developer tool in the browser, there in the console, the output is as shown below:



In the above output, to render the HTML file, the path of the HTML file is copied into the browser. But in this case, each time any changes are made in the code, the page must be refreshed for the changes to reflect.

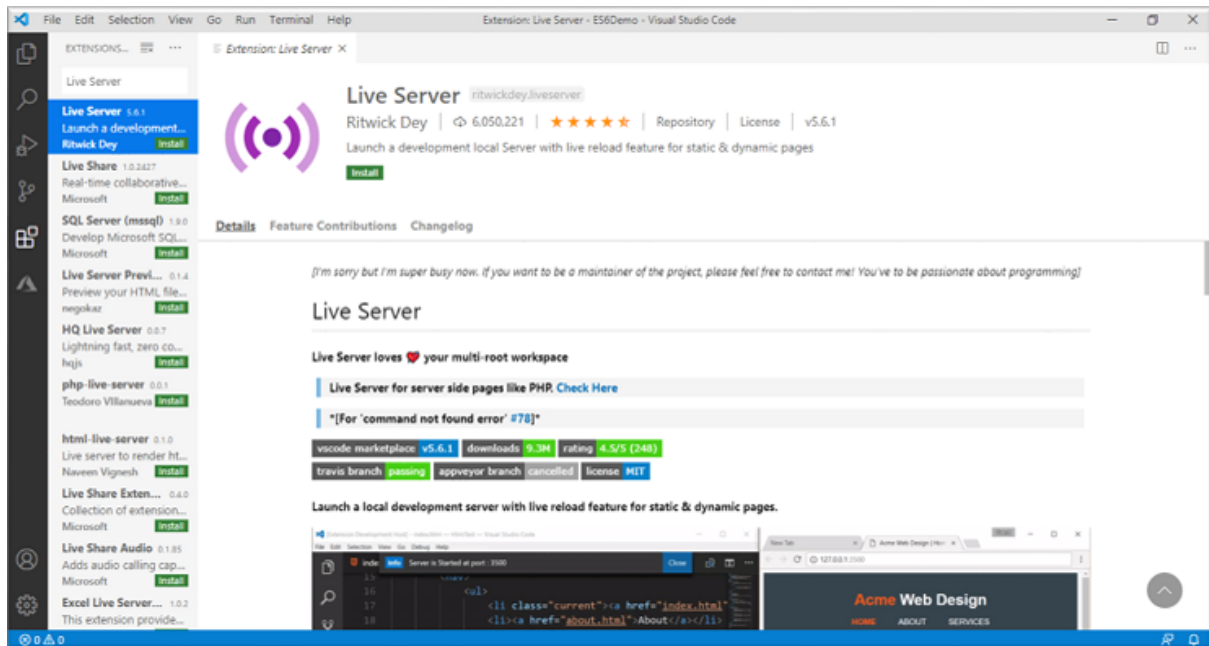
There is a solution to this in the Visual Studio Code. It provides an option to add extensions to render the code in a server.

### Need for Live Server:

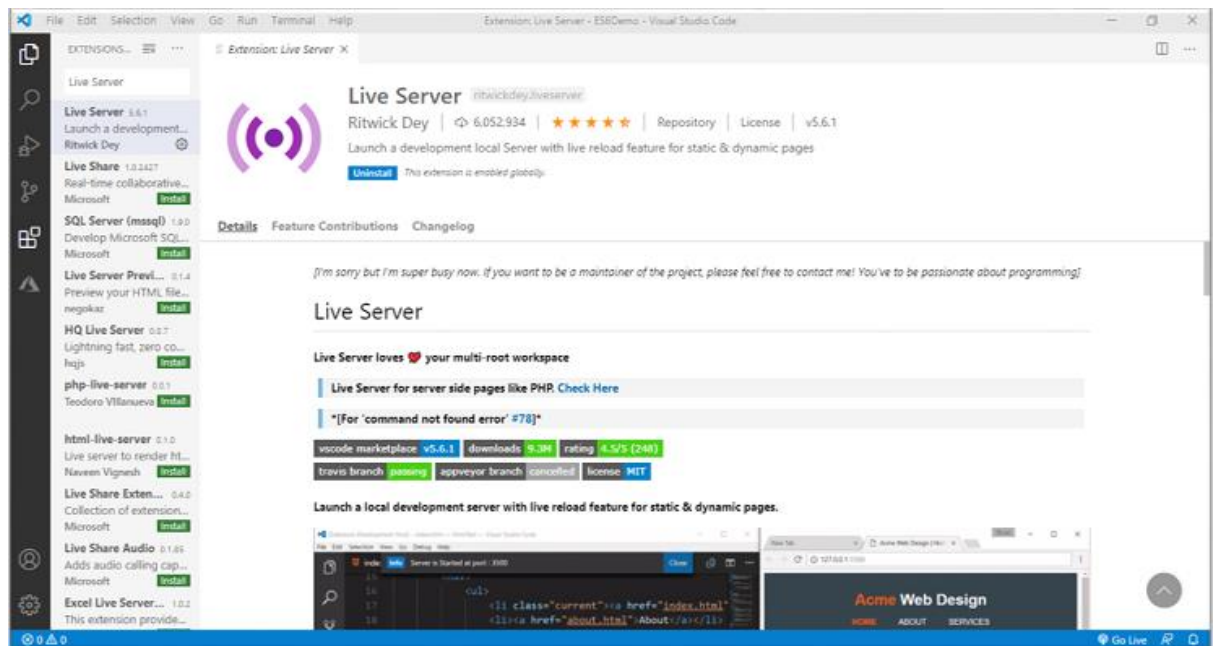
Visual Studio Code provides an extension called Live Server using which HTML page can be rendered and any changes that developers make further will be automatically detected and rendered properly.

### Adding Live Server:

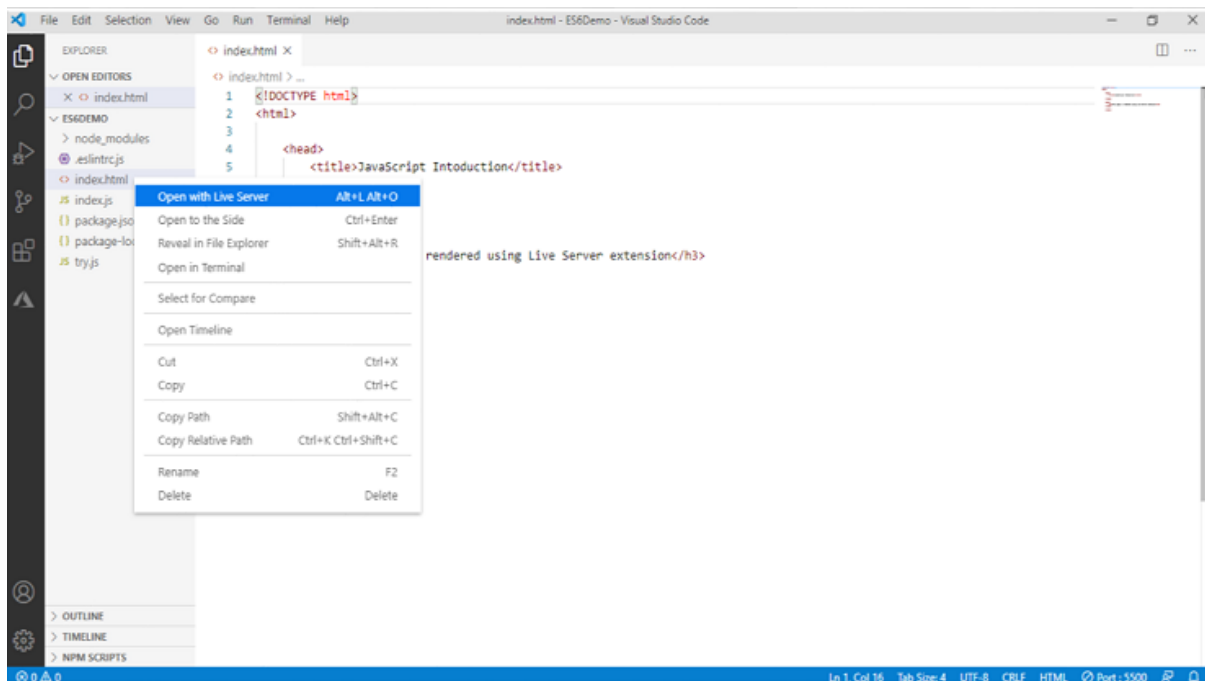
1. Go to the extension tab in Visual Studio Code and search for Live Server and click on the install button that is visible in.



2. Once it gets installed, there will be a screen with an uninstall button as shown below and the Live Server will be ready to render the HTML pages.



3. To render an HTML page, right-click on the intended HTML page in the Explore tab and select the 'Open with Live Server' option.



So far, you have learnt about, how to use Live Server to render the page, now let us proceed on to understand some extension that will help standardize the code written by adding linting.

**Note:** Node.js is a JavaScript runtime environment that executes the JavaScript code outside a web browser. It has a default package manager called Node Package Manager which gets added automatically with the Nodejs environment installation. The package manager has a CLI also called 'npm', using which one can add third-party JavaScript libraries.

Node.js can be installed from the official website of Node.js (<https://nodejs.org/en/>).

### The need for Linting:

As the application develops through multiple stages, code quality becomes very critical. Linting is the process of analyzing the code, notifying the structural and functional errors which helps the developer to improve the code quality.

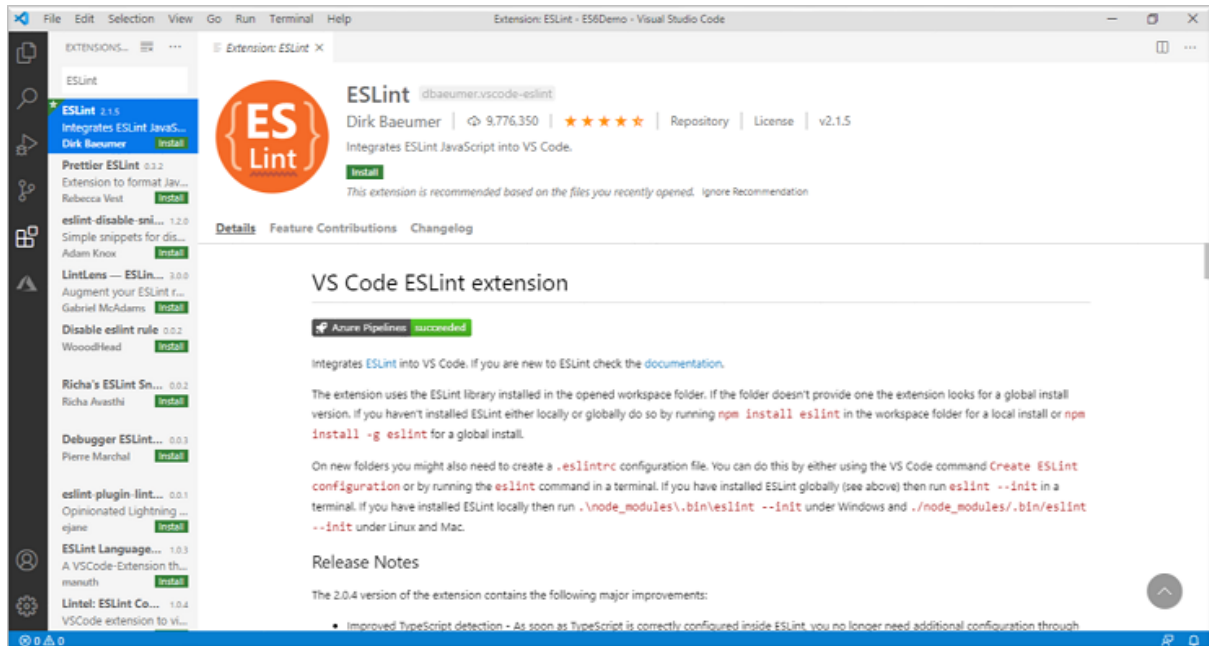
There are various linting tools specific to JavaScript which helps to find and resolve the structural errors. One of the Lint tools which have been mentioned here is ESLint.

Let us have a glance over how to add ESLint to Visual Studio Code Editor.

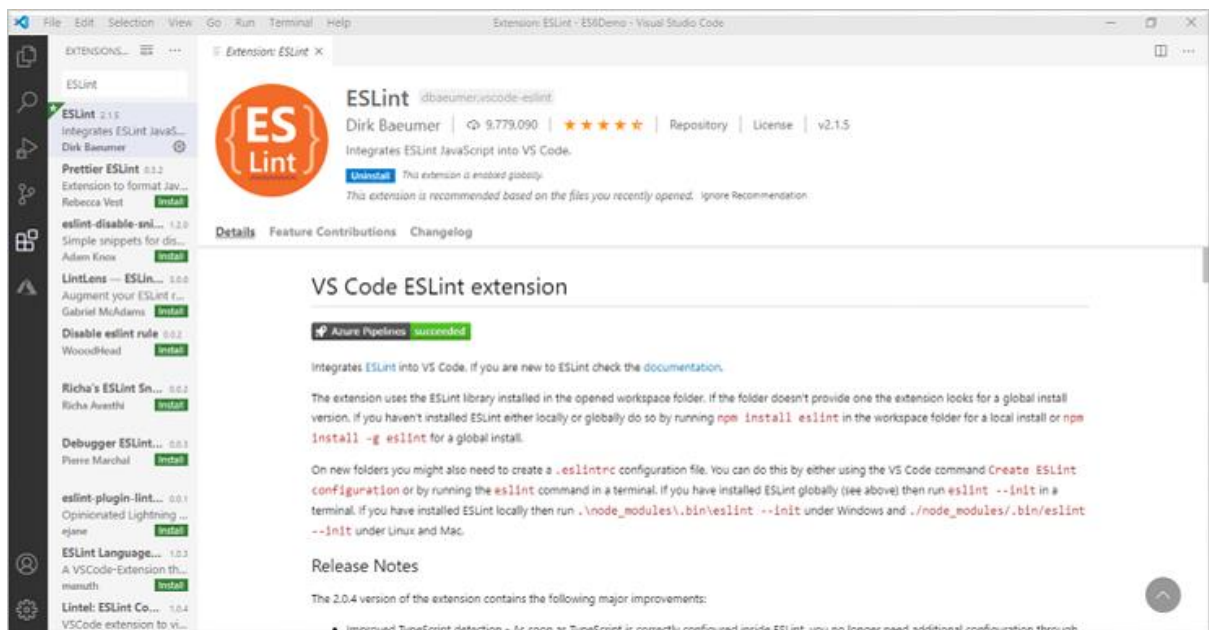
### Adding ESLint:

- The latest Node.js version needs to be installed in the system. If not, configure the latest version from the official website.
- Open the Visual Studio Code, go to the extension, search for "ESLint".

Once the ESLint extension appears, hit the "Install" button as shown below:



After the installation, the below screen is visible:



Following are the steps to configure ESLint with JavaScript.

- Create a JavaScript demo.js file

```
1. let firstName="Rexha"  
2. let lastName ="Bebe";  
3. for( let i=0;i<=5;i--){  
4.     console.log(i);
```

```
5. }
```

```
6.
```

- Goto the Terminal in the VS code IDE, be in project folder (where the demo.js file is present) path.
- Run the below command

```
1. npm init
```

The above command creates a **package.json** file which will have the metadata and package dependencies for the project.

- Now install ESLint using the following command:

```
1. npm install eslint --save-dev
```

- Next, is to create the configuration file: ".eslintrc.js", using one of the below-given command:

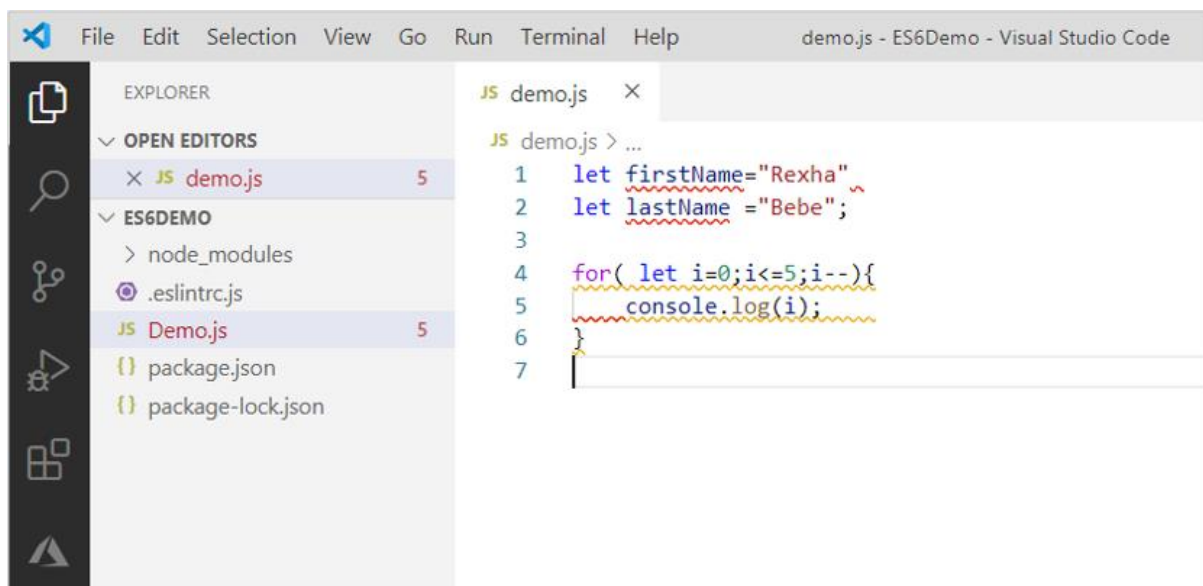
```
1. ./node_modules/.bin/eslint --init  
2. OR  
3. npm init @eslint/config
```

This command will then prompt questions on how ESLint is to be used and completes the creation of a configuration file for ESLint. (as shown in the screenshot below)

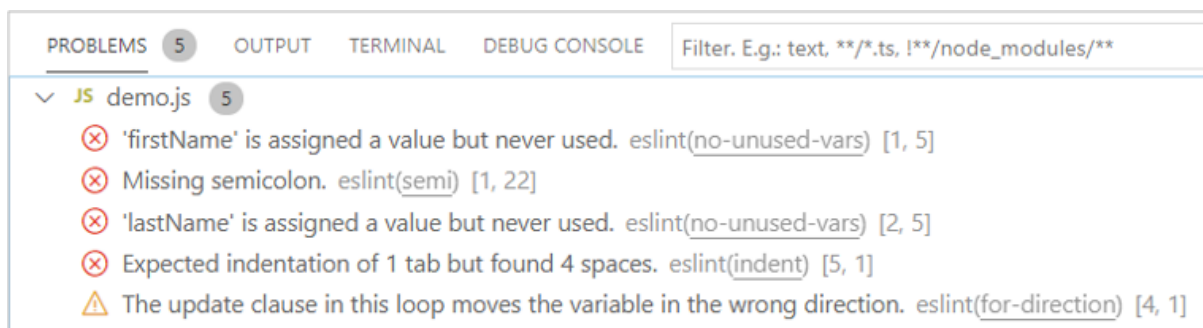


```
How would you like to use ESLint?  
  select: To check the syntax, fix the problems and enforce code style  
What type of modules does your project use?  
  select: JavaScript modules  
Which framework does your project use?  
  select: None of this  
Does your project use TypeScript?  
  select: No  
Where does your code run?  
  select: Node  
What format do you want your config file to be in?  
  select: javascript/json  
What style of indentation do you see?  
  select: tab/space according to your preference
```

If any linting issue occurs in code, all the issues will be highlighted and more details will be provided in the console as shown below.



```
JS demo.js  
JS demo.js > ...  
1 let firstName="Rexha"  
2 let lastName ="Bebe";  
3  
4 for( let i=0;i<=5;i--){  
5   console.log(i);  
6 }  
7
```



PROBLEMS 5 OUTPUT TERMINAL DEBUG CONSOLE Filter: E.g.: text, \*\*/\*.ts, !\*\*/node\_modules/\*\*

- ✗ 'firstName' is assigned a value but never used. eslint(no-unused-vars) [1, 5]
- ✗ Missing semicolon. eslint(semi) [1, 22]
- ✗ 'lastName' is assigned a value but never used. eslint(no-unused-vars) [2, 5]
- ✗ Expected indentation of 1 tab but found 4 spaces. eslint(indent) [5, 1]
- ⚠ The update clause in this loop moves the variable in the wrong direction. eslint(for-direction) [4, 1]

## Working with Identifier

To model the real-world entities, they have to be named to use it in the JavaScript program.



Identifiers are those names that help in naming the elements in JavaScript.

**Example:**

```
1. firstName;  
2. placeOfVisit;
```

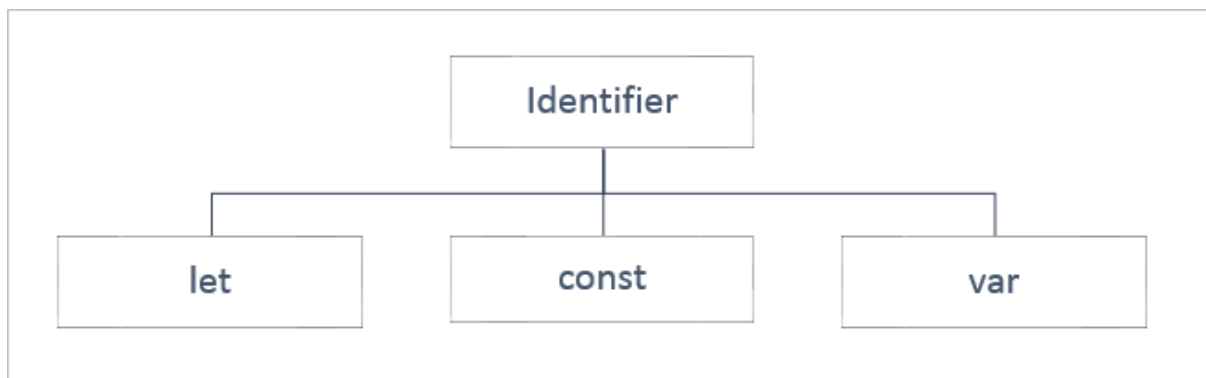
Identifiers should follow below rules:

- 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.

The identifiers in JavaScript can be categorized into three as shown below. They can be declared into specific type based on:

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



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.

All the identifiers known so far vary in their scope and with respect to the data it holds.

**Example:**

```
1. let name="William";  
2. console.log("Welcome to JS course, Mr."+name);  
3.
```

```
4. let name = "Goth"; /* This will throw an error because the
   identifier 'name' has been already declared and we are
   redeclaring the variable, which is not allowed using the 'let'
   keyword. */
5. console.log("Welcome to JS course, Mr."+name);
6.
```

✓ **Note:** As a best practice, use the **let** keyword for identifier declarations that will change their value over time or when the variable need not be accessed outside the code block. For example, in loops, looping variables can be declared using let as they are never used outside the block.

The identifier to hold data that does not vary is called 'Constant' and to declare a constant, 'const' keyword is used, followed by an identifier. The value is initialized during the declaration itself and cannot be altered later.

The identifiers declared using 'const' keyword have block scope i.e., they exist only in the block of code within which they are defined.

### Example:

```
1. const pi = 3.14;
2. console.log("The value of Pi is: "+pi);
3.
4. pi = 3.141592; /* This will throw an error because the assignment
   to a const needs to be done at the time of declaration and it
   cannot be re-initialized. */
5. console.log("The value of Pi is: "+pi);
6.
```

✓ **Note:** As a best practice, the const declaration can be used for string type identifiers or simple number, functions or classes which does not need to be changed or value

For example, if A4 size paper is referred, then the dimension of the paper remains the same, but its colour can vary.

The identifiers declared to hold data that vary are called 'Variables' and to declare a variable, the 'var' keyword is optionally used.

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 is 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.

### Example:

```
1. var name = "William";
```

```
2. console.log("Welcome to JS course, Mr." + name);
3. var name = "Goth"; /* Here, even though we have redeclared the
   same identifier, it will not throw any error.*/
4. console.log("Welcome to JS course, Mr." + name);
5.
```

You will learn in detail about this in the Functions module.

☒ **Note:** As a best practice, use the 'var' keyword for variable declarations for function scope or global scope in the program.

Below table shows the difference between let, const and var.

| Keyword | Scope    | Declaration               | Assignment               |
|---------|----------|---------------------------|--------------------------|
| let     | Block    | Redeclaration not allowed | Re-assigning allowed     |
| const   | Block    | Redeclaration not allowed | Re-assigning not allowed |
| var     | Function | Redeclaration allowed     | Re-assigning allowed     |

## Working with Datatypes

To be able to proceed with the manipulation of the data assigned to the variables, it is mandatory for a programming language to know the type of value or the type of data that the variable holds.

That is because the operations or manipulations that must be applied on a variable will be specific to the type of data that a variable will hold.

For example, the result of add operation on two variables will vary based on the fact whether the value of both the variables is numeric or textual.

To handle such situation, data types are used.

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, JavaScript language is a loosely typed or dynamically typed language.

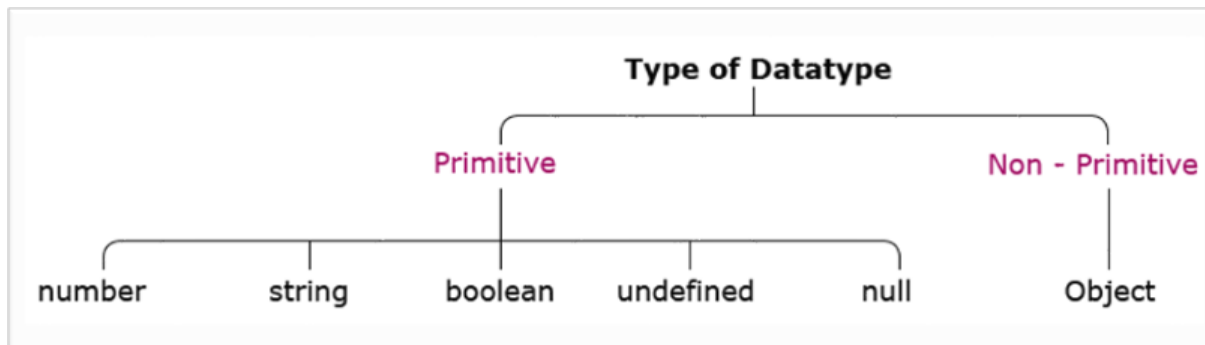
### Example:

```
1. let age = 24; //number
2. let name = "Tom" //string
3. let qualified = true; //boolean
4.
```

Also, there can be same variable of different types in JavaScript code based on the value that is assigned to it.

For example, if let age = 24, the variable 'age' is of type number. But if, let age = "Twenty-Four", variable 'age' is of type string.

JavaScript defines the following data types:



Next, let us understand each data type in detail.

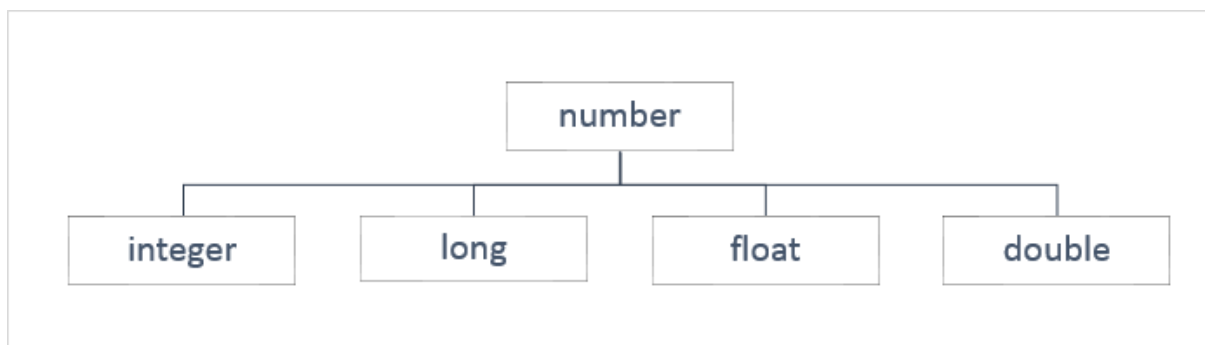
## Primitive Datatypes

The data is said to be primitive if it contains an individual value.

Let us explore each of the primitive data types individually.

### 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:



But in JavaScript, the data type number is assigned to the values of type integer, long, float, and double. For example, the variable with number data type can hold values such as 300, 20.50, 10001, and 13456.89.

Constant of type number can be declared like this:

**Example:**

```
1. const pi = 3.14; // its value is 3.14
2. const smallestNaturalNumber = 0; // its value is 0
3.
```

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).

## Example:

```
1. let result = 0/0; // its value is NaN
2. let result = "Ten" * 5; //its value is NaN
```

## 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.

## Example:

```
1. let personName= "Rexha";    //OR
2. let personName = 'Rexha';    // both will have its value as
    Rexha
```

You can use quotes inside a string but that shouldn't match the quotes surrounding the string. Strings containing single quotes must be enclosed within double quotes and vice versa.

## Example:

```
1. let ownership= "Rexha's";    //OR
2. let ownership = 'Rexha"s';
```

This will be interpreted as Rexha's and Rexha"s respectively. Thus, use opposite quotes inside and outside of JavaScript single and double quotes.

But if you use the same quotes inside a string and to enclose the string:

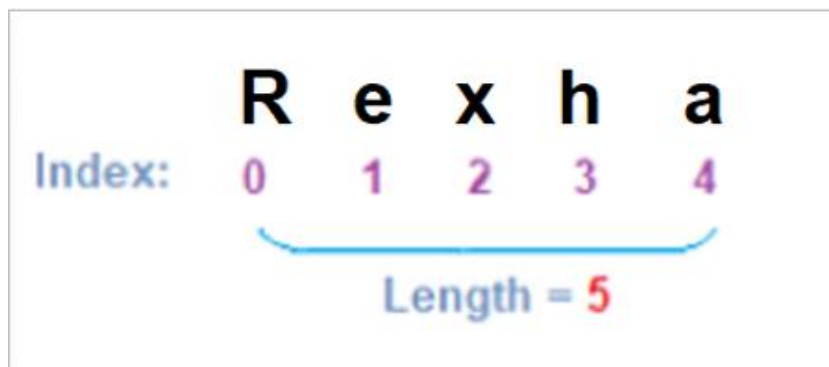
## Example:

```
1. let ownership= "Rexha"s";    //OR
```

```
2. let ownership = 'Rexha's';
```

It is a syntax error.

- Thus, remember, strings containing single quotes must be enclosed within double quotes and strings containing double quotes must be enclosed within single quotes.
- To access any character within the string, it is important to be aware of its position in the string.
- The first character exists at index 0, next at index 1, and so on.



## Literals

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

### Example:

```
1. let firstName="Kevin";
2. let lastName="Patrick";
3. console.log("Name: "+firstName+" "+lastName+"\n
   Email:"+firstName+"_"+lastName+"@abc.com");
4.
5. /*
6. OUTPUT:
7. Name: Kevin Patrick
8. Email:Kevin_Patrick@abc.com
9. */
10.
```

Here, '+' is used for concatenation of identifiers and static content, and '\n' for a new line.

To get the same output, literals can be used as shown below:

```
1. let firstName="Kevin";
2. let lastName="Patrick";
3. console.log(`Name:${firstName} ${lastName}`);
```

```
4. Email: ${firstName}_${lastName}@abc.com`);  
5.  
6. /*  
7. OUTPUT:  
8. Name: Kevin Patrick  
9. Email:Kevin_Patrick@abc.com  
10. */  
  
11.
```

Using template literal, multiple lines can be written in the console.log() in one go.

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

## Boolean

When a variable is used to store a logical value that can always be true or false then, primitive data type Boolean is used. Thus, Boolean is a data type which represents only two values: true and false.

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.

## Undefined

When the variable is used to store "no value", primitive data type undefined is used.

Any variable that has not been assigned a value has the value undefined and such variable is of type undefined. The undefined value represents "no value".

### Example 1:

```
1. let custName; //here value and the data type are undefined
```

The JavaScript variable can be made empty by assigning the value undefined.

### Example 2:

```
1. let custName = "John"; //here value is John and the data type is String  
2. custName = undefined; //here value and the data type are  
  
undefined
```

null



The null value represents "no object".

Null data type is required as JavaScript variable intended to be assigned with the object at a later point in the program can be assigned null during the declaration.

## Example 1:

```
1. let item = null;
2. // variable item is intended to be assigned with object later. Hence
   null is assigned during variable declaration.
```

If required, the JavaScript variable can also be checked if it is pointing to a valid object or null.

## Example 2:

```
1. document.write(item==null);
```



**Note:** 'document' is an object that represents the HTML document rendered on the browser window and write() method helps one to populate HTML expressions or JavaScript code directly to the document.

BigInt is a special numeric type that provides support for integers of random length.

A BigInt is generated by appending `n` to the end of an integer literal or by calling the function `BigInt` that generates BigInt from strings, numbers, etc.

## Example:

```
1. const bigintvar = 67423478234689887894747472389477823647n;
2.
3. OR
4.
5. const bigintvar =
   BigInt("67423478234689887894747472389477823647");
6.
7. const bigintFromNumber = BigInt(10); // same as 10n
8.
```

common math operations can be done on BigInt as regular numbers. But BigInt and regular numbers cannot be mixed in the expression.

## Example:

```
1. alert(3n + 2n); // 5
2.
3. alert(7n / 2n); // 3
4.
5. alert(8n + 2); // Error: Cannot mix BigInt and other types
```

Here the division returns the result rounded towards zero, without the decimal part. Thus, all operations on BigInt return BigInt.

BigInt and regular numbers must be explicitly converted using either `BigInt()` or `Number()`, as shown below:

### Example:

```
1. let bigintvar = 6n;
2. let numvar = 3;
3.
4. // number to bigint
5. alert(bigintvar + BigInt(numvar)); // 9
6.
7. // bigint to number
8. alert(Number(bigintvar) + numvar); // 9
9.
```

In the above example, if the `bigintvar` is too large that it won't fit the number type, then extra bits will be cut off.

Talking about comparison and boolean operations on BigInt, it works fine.

### Example:

```
1. alert( 8n > 2n ); // true
2. alert( 4n > 2 ); // true
```

Even though numbers and BigInts belong to different types, they can be equal `==`, but not strictly equal `===`.

## Example:

```
1. alert( 5 == 5n ); // true
2. alert( 5 === 5n ); // false
```

When inside `if` or other boolean operations, `BigInts` behave like numbers.

## Example:

```
1. if (0n) {
2.   // never executes
3. }
```

`BigInt 0n` is falsy, other values are considered to be truthy.

Boolean operators, such as `||`, `&&` and others also work perfectly with `BigInts` similar to numbers.

## Example:

```
1. alert( 1n || 2 );
2. // 1, here 1n is considered truthy
3.
4. alert( 0n || 2 );
5. // 2, here 0n is considered falsy
```

A "symbol" represents a unique identifier. You can make use of `Symbol()` to generate a value of this type.

## Example:

```
1. // empid is a new symbol
2. let empid = Symbol();
```

Also, a description of the symbol generated can be provided which can be mostly used as a name during debugging.

## Example:

```
1. // empid is a symbol with the description "empno"
```



```
2. let empid = Symbol("empno");
```

Even if various symbols are created with the same description, they are different values. Thus, symbols ensures uniqueness. So the description provided can be considered as just a label.

```
1. let empid1 = Symbol("empno");
2. let empid2 = Symbol("empno");
3.
4. alert(empid1== empid2); // false
```

Here both the symbols have the same description but are never considered equal.

Unlike other values, a symbol value doesn't follow auto-convert.

## Example:

```
1. let empid = Symbol("empno");
2. alert(empid); // TypeError: Cannot convert a Symbol value to a
    string
```

This is a rule because strings and symbols are basically different and should not accidentally get converted to the other one.

But if it is a must to display the Symbol, then the following can be done:

## Example:

```
1. let empid = Symbol("empno");
2. alert(empid.toString()); // Symbol(empno), now it works
3.
4. OR
5.
6. //use description
7. let empid = Symbol("empno");
8. alert(empid.description); // empno
```

## Global symbols

So far you know that symbols remain unique even if they have the same name. But at times, there may be a situation where you may want the symbols with same name to be same entities.



In such a situation, symbols can be created in a global symbol registry and access them later and ensure that repeated accesses by the same name return exactly the same symbol. To read a symbol from the registry, use `Symbol.for(key)` which checks if there's a symbol described as `key`, then returns it, otherwise creates a new symbol `Symbol.for(key)` and stores it in the registry by the given `key`.

## Example:

```
1. // read from the global registry
2. let empid = Symbol.for("empno"); // if the symbol did not exist,
   it is created
3.
4. // read it again (maybe from another part of the code)
5. let empidAgain= Symbol.for("empid");
6.
7. // the same symbol

8. alert( empid === empidAgain ); // false
```

Thus, global symbols help in creating application-wide symbol which is accessible everywhere in the code.

## Non - Primitive Data Types

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 with one of the primitive data types.

There are times a group of values are stored 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.

## Syntax:

```
1. {  
2.   key1 : value1,  
3.   key2 : value2,  
4.   key3 : value3  
  
5. };
```

## Example:

```
1. let mySmartPhone = {  
2.   name: "iPhone",  
3.   brand: "Apple",  
4.   platform: "iOS",  
5.   price: 50000  
  
6. };
```

## Array

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

There are two ways of creating an array:

```
1. let dummyArr = new Array();  
2. //OR  
  
3. let dummyArr = [];
```

Either array can be declared as empty and can be assigned with value later, or can have the value assigned during the declaration.

## Example:

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

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

## Working With Operators

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

Operands: Represents the data.

Operator: Performs certain operations on the operands.

```
1. let sum = 5 + 10;
```

The statement formed using the operator and the operands are called Expression.

In the above example, 5+10 is an expression.

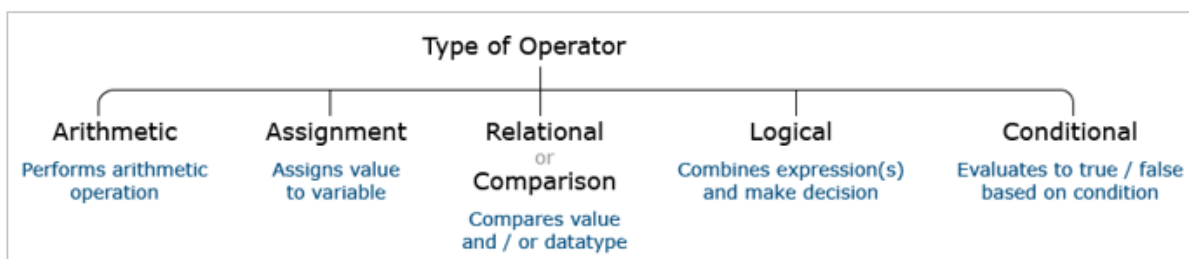
The values are termed as operands.

The symbol '+' is the operator which indicates which operation needs to be performed.

## Operators and Types of Operators

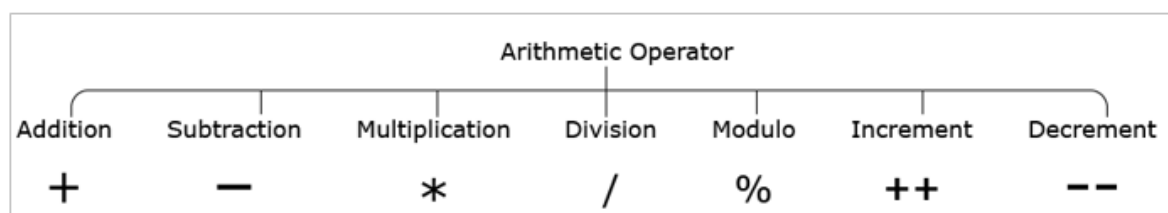
Operators are categorized into unary, binary, and ternary based on the number of operands on which they operate in an expression.

JavaScript supports the following types of operators:



## Arithmetic Operator

Arithmetic operators are used for performing arithmetic operations



```
1. let sum = 5 + 3; // sum=8
2. let difference = 5 - 3; // difference=2
3. let product = 5 * 3; // product=15
4. let division = 5/3; // division=1
5. let mod = 5%3; // mod=2
6. let value = 5;
7. value++; // increment by 1, value=6
8. let value = 10;
```



```
9. value--; // decrement by 1, value=9
```

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

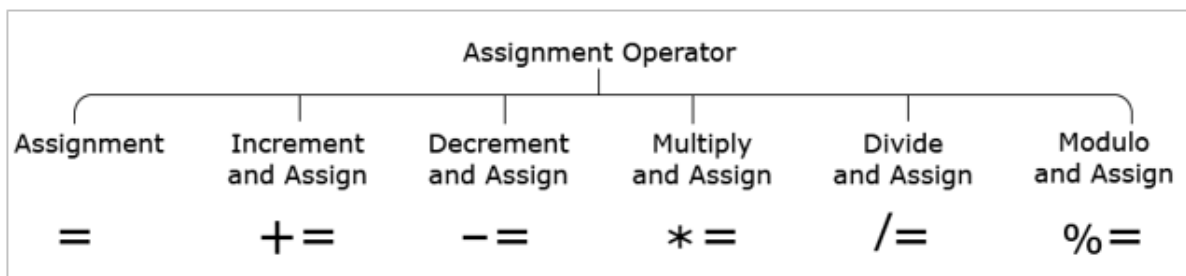
```
1. let firstName = "James";  
2. let lastName = "Roche";  
  
3. let name = firstName + " " + lastName; // name = James Roche
```

Arithmetic operator '+' when used with a string value and a numeric value, it results in a new string value.

```
1. let strValue="James";  
2. let numValue=10;  
3. let newStrValue= strValue + " " + numValue; // newStrValue= James 10
```

## Assignment Operator

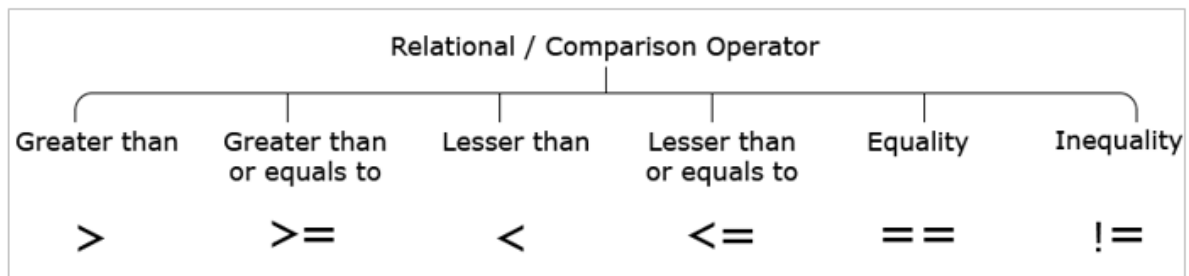
Assignment operators are used for assigning values to the variables.



```
1. let num = 30; // num=30  
2. let num += 10; // num=num+10 => num=40  
3. let num -= 10; // num=num-10 => num=20  
4. let num *= 30; // num=num*30 => num=900  
5. let num /= 10; // num=num/10 => num=3  
6. let num %= 10; // num=num%10 => num=0
```

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. 10 > 10; //false
2. 10 >= 10; //true
3. 10 < 10; //false
4. 10 <= 10; //true
5. 10 == 10; //true
6. 10 != 10; //false

```

Relational operators shown below compares both the values and the value types without any implicit type conversion.

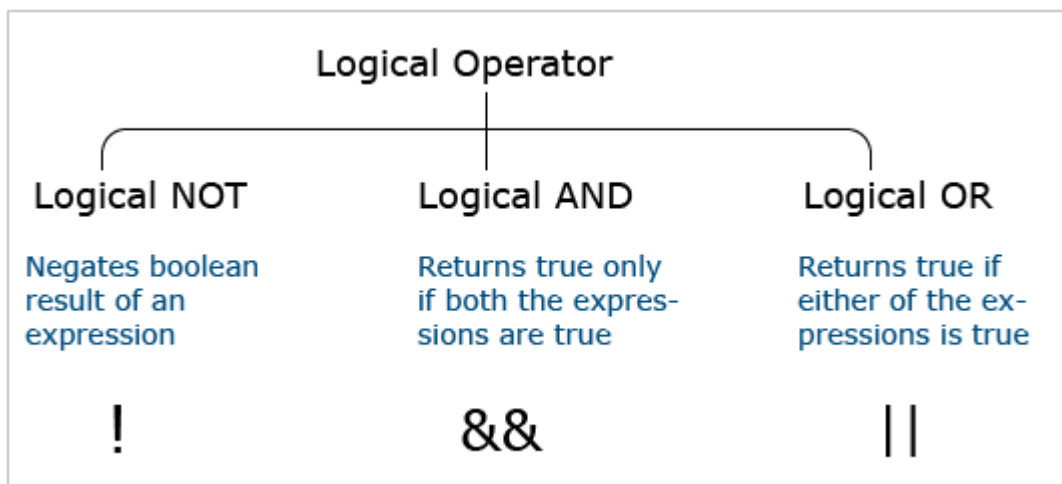
| Strict equality     |   | Strict inequality   |  |
|---------------------|---|---------------------|--|
| <b>Definition:</b>  | Returns true when value and datatype are equal  | <b>Definition:</b>  | Returns true when value or datatype are unequal  |
| <b>Operator:</b>    | ===   | <b>Operator:</b>    | !==  |
| <b>Example:</b>     | 10 === "10"   | <b>Example:</b>     | 10 !== "10"  |
| <b>Result:</b>      | false   | <b>Result:</b>      | true   |
| <b>Explanation:</b> | 10 and "10" have same values but 10 is a number and "10" is a string, hence returns false | <b>Explanation:</b> | 10 and "10" have same values but 10 is a number and "10" is a string, hence returns true |

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 two given values are equal or not.

☑ **Note:** As a best practice, you should use === comparison operator when you want to compare value and type, and the rest of the places for value comparison == operator can be used.

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



```
1. !(10 > 20); //true
2. (10 > 5) && (20 > 20); //false
3. (10 > 5) || (20 > 20); //true
```

"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, the typeof operator can be used to find the data type of a JavaScript variable.

The following are the ways in which it can be used and the corresponding results that it returns.

```
1. typeof "JavaScript World" //string
2. typeof 10.5 // number
```

```
3. typeof 10 > 20 //boolean
4. typeof undefined //undefined
5. typeof null //Object

6. typeof {itemPrice : 500} //Object
```

## Working with Statements and Expressions

Statements are instructions in JavaScript that have to 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.

A Variable declaration is the simplest example of a JavaScript statement.

### Syntax:

```
1. var firstName = "Newton" ;
```

Other types of JavaScript statements include conditions/decision making, loops, etc.

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.

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

Different types of expressions that can be written in JavaScript are:

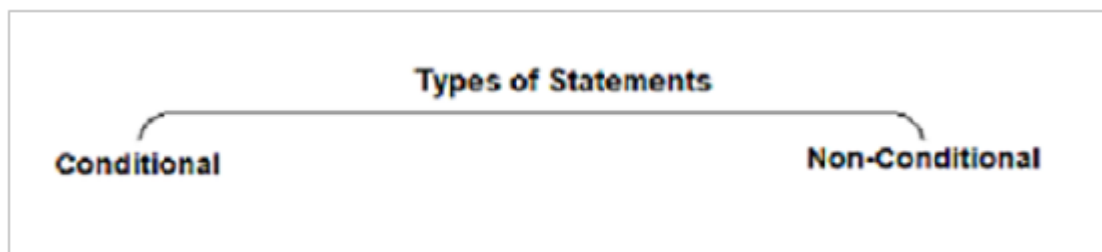
```
1. 10 + 30; //Evaluates to numeric value
2. "Hello" + "World"; //Evaluates to string value
3. itemRating > 5; //Evaluates to boolean value
4. (age > 60) : "Senior citizen" : "Normal citizen";
5. /*
   Evaluates to one string value based on whether condition is true or false
   .
   6. If the condition evaluates to true then the first string value
   "Senior citizen" is assigned otherwise the second string value is
   assigned "Normal citizen" */
```

Example of an expression in a statement:



## Types of Statements

In JavaScript, the statements can be classified into two types.



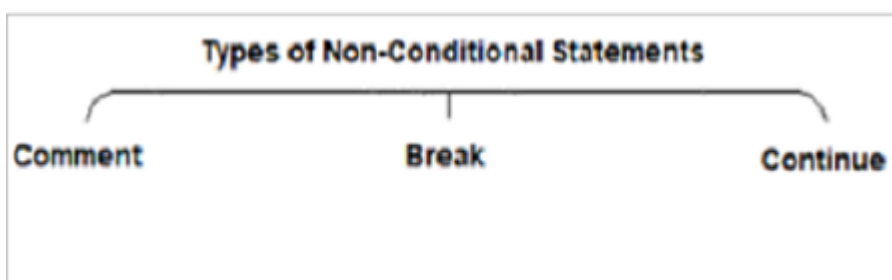
### Conditional Statements:

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:

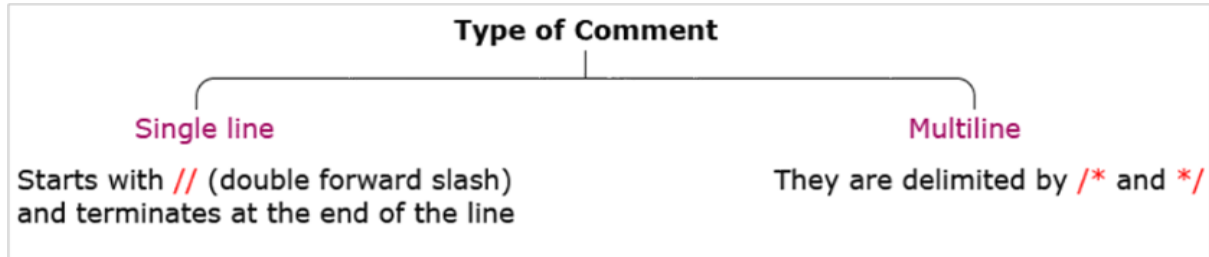
Non-Conditional statements are those statements that do not need any condition to control the program execution flow.



### Comments

Comments in JavaScript can be used to prevent the execution of a certain lines of code and to add information in the code that explains the significance of the line of code being written.

JavaScript supports two kinds of comments.



### Example:

```
1. // Formula to find the area of a circle given its radius
2. var areaOfCircle = 2 * pi * radius;
3.
4. /* Formula to find the area of a circle based on
5.    given its radius value.
6. */
7. var areaOfCircle = 2 * pi * radius;
```

**Note:** As a best practice, it is recommended to use comments for documentation purposes and avoid using it for code commenting.

While iterating over the block of code getting executed within the loop, the loop may be required to be exited if certain condition is met. The 'break' statement is used to terminate the loop and transfer control to the first statement following the loop.

### Syntax:

```
1. break;
```

Below example shows for loop with five iterations which increment variable "counter".

When loop counter = 3, loop terminates.

Also, shown below is the value of the counter and loopVar for every iteration of the loop.

```
1. var counter = 0;
2. for (var loop = 0; loop < 5; loop++) {
3.   if (loop == 3)
4.     break;
5.   counter++;
6. }
```

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

The 'if' statement used in the above example is a conditional / decision-making statement.

There are times when during the iteration of the block of code within the loop, the block execution may be required to be skipped for a specific value and then continue to execute the block for all the other values. JavaScript gives 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.

## Syntax:

```
1. continue;
```

Below example shows for loop with five iterations which increment variable "counter".

When loop counter = 3, the current iteration is skipped and moved to the next iteration.

Also, shown below is the value of the counter and the variable loop for every iteration of the loop.

```
1. var counter = 0;
2. for (var loop = 0; loop < 5; loop++) {
3.   if (loop == 3)
4.     continue;
5.   counter++;
6. }
```



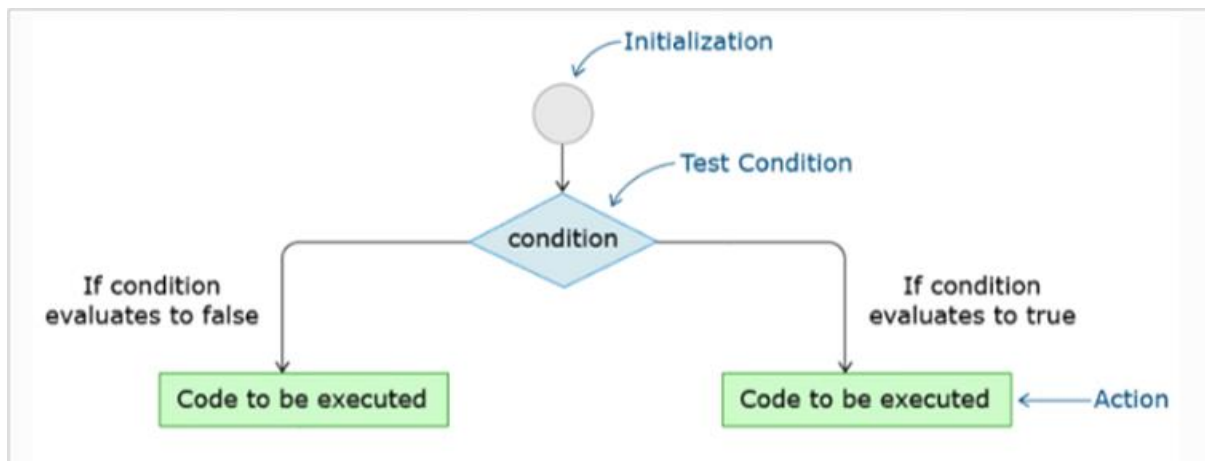
| loopVar | counter   |
|---------|---|
| 0       | 1   |
| 1       | 2   |
| 2       | 3   |
| 3       | Iteration terminated. Hence counter is not incremented. |
| 4       | 4   |

The 'if' statement used in the example is a conditional / decision-making statement.

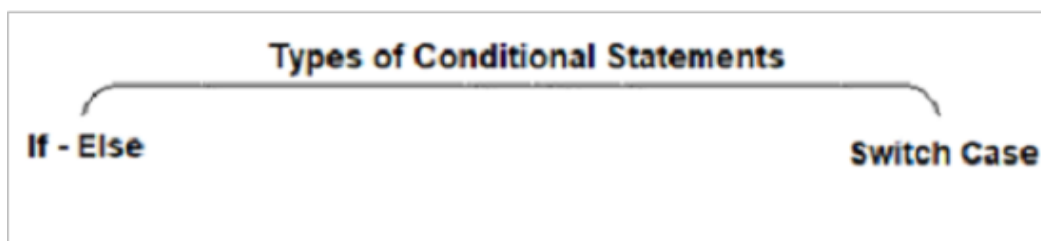
## Types of Conditional Statements

Conditional statements help in performing different actions for different conditions.

It is also termed as decision-making statements.



JavaScript supports two decision-making statements:



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.

**Example:**

```

1. let workingHours = 9.20;
2. let additionalHours;
  
```

```
3. (workingHours > 9.15) ? additionalHours = "You have positive  
   additional hours" : additionalHours = "You have negative additional  
   hours";  
4. console.log(additionalHours);
```

The '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.

There are different flavors of if-else statement:

- Simple 'if' statement
- if -else
- if-else-if ladder

Let us see each of them in detail.

## if statement

The 'if' statement is used to execute a block of code if the given condition evaluates to true.

### Syntax:

```
1. if (condition) {  
2.   //block of code that will be executed, if the condition is true  
  
3. }
```

### Example:

```
1. let num1 = 12;  
2. if (num1 % 2 == 0) {  
3.   console.log("It is an even number!!");  
4. }  
  
5. // OUTPUT: It is an even number!! Because 12%2 evaluates to true
```

The 'else' statement is used to execute a block of code if the given condition evaluates to false.

### Syntax:

```
1. if (condition) {  
2.   //block of code that will be executed, if the condition is true  
3. }  
4. else {
```

```
5. //block of code that will be executed, if the condition is false
6. }

7.
```

## Example:

```
1. let num1 = 1;
2. if (num1 % 2 == 0) {
3.   console.log("It is an even number!!");
4. }
5. else{
6.   console.log("It is an odd number!!");
7. }
8. //OUTPUT: It is an odd number!! Because in if 1%2 evaluates to false
   and moves to else condition
```

if...else ladder is used to check for a new condition when the first condition evaluates to false.

## Syntax:

```
1. if (condition1) {
2.   //block of code that will be executed if condition1 is true
3. }
4. else if (condition2) {
5.   //block of code that will be
   executed if the condition1 is false and condition2 is true
6. }
7. else {
8.   //block of code that will be
   executed if the condition1 is false and condition2 is false
9. }
```

## Example:

```
1. let marks = 76;
2. if (marks >= 75) {
3.   console.log("Very Good");
4. }
5. else if (marks < 85 && marks >= 50) {
6.   console.log("Good");
7. }
8. else {
9.   console.log("Needs Improvement");
10. }
```



```
11. // OUTPUT: Needs Improvement, Because the value of marks is 46
    which doesn't satisfy the first two condition checks.
```

## Switch Statement

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

### Syntax:

```
1. switch (expression) {
2.     case value1: code block;
3.         break;
4.     case value2: code block;
5.         break;
6.     case valueN: code block;
7.         break;
8.     default: code block;
9. }
```

'break' statement is used to come out of the switch and continue execution of statement(s) the following switch.

### Example:

For the given Employee performance rating (between 1 to 5), displays the appropriate performance badge.

```
1. var perfRating = 5;
2.
3. switch (perfRating) {
4.     case 5:
5.         console.log("Very Poor");
6.         break;
7.     case 4:
8.         console.log("Needs Improvement");
9.         break;
10.    case 3:
11.        console.log("Met Expectations");
12.        break;
13.    case 2:
14.        console.log("Commendable");
15.        break;
16.    case 1:
17.        console.log("Outstanding");
18.        break;
19.    default:
20.        console.log("Sorry!! InvalidRating.");
21. }
22.
```

23.

**OUTPUT:** Very Poor

The break statements allow to get control out of the switch once we any match is found.

### Example:

For the given Employee performance rating (between 1 to 5), displays the appropriate performance badge.

```
1. var perfRating = 3;
2.
3. switch (perfRating) {
4.   case 5:
5.     console.log("Very Poor");
6.     break;
7.   case 4:
8.     console.log("Needs Improvement");
9.     break;
10.  case 3:
11.    console.log("Met Expectations");
12.    break;
13.  case 2:
14.    console.log("Commendable");
15.    break;
16.  case 1:
17.    console.log("Outstanding");
18.    break;
19.  default:
20.    console.log("Sorry!! Invalid Rating.");
21. }
22.
23. /*
24.  OUTPUT:
25.
26.  Met Expectation
27.  */
28.
```

The reason for the above output is, first perfRating value is checked against case 5 and it does not match. Next, it is checked against case 4 and it also does not match. Next, when it is checked against case 3. it got a match hence "Met Expectation" is displayed, and the break statement moves the execution control out of the switch statement.

Consider the below code snippet without break statement:

```
1. var perfRating = 5;
2.
3. switch (perfRating) {
4.   case 5:
5.     console.log("Very Poor");
6.
7.   case 4:
8.     console.log("Needs Improvement");
9.
10.  case 3:
11.    console.log("Met Expectations");
12.
13.  case 2:
14.    console.log("Commendable");
15.
16.  case 1:
17.    console.log("Outstanding");
18.
19.  default:
20.    console.log("Sorry!! Invalid Rating.");
21. }
22.
23. /*
24.  OUTPUT:
25.  Very Poor
26.  Needs Improvement
27.  Met Expectations
28.  Commendable
29.  Outstanding
30.  Sorry!! Invalid Rating.
31. */
32.
33.
```

The reason for the above output is, initially perfRating is checked against case 5 and it got matched, hence 'Very Poor' is displayed. But as the break statement is missing, the remaining cases including default got executed.

A scenario in which the default statement gets executed.

```
1. var perfRating = 15;
2.
3. switch (perfRating) {
4.   case 5:
5.     console.log("Very Poor");
6.     break;
7.   case 4:
8.     console.log("Needs Improvement");
9.     break;
```

```
10. case 3:
11.     console.log("Met Expectations");
12.     break;
13. case 2:
14.     console.log("Commendable");
15.     break;
16. case 1:
17.     console.log("Outstanding");
18.     break;
19. default:
20.     console.log("Sorry!! InvalidRating.");
21. }
22.
23. /*
24.     OUTPUT:
25.     Sorry!! InvalidRating.
26. */
27.
```

The reason for the above output is, here perfRating = 15 does not match any case values. Hence, the default statement got executed.

## Working With Loops

In JavaScript code, specific actions may have to be repeated a number of times.

For example, consider a variable counter which has to be incremented five times.

To achieve this, increment statement can be written five times as shown below:

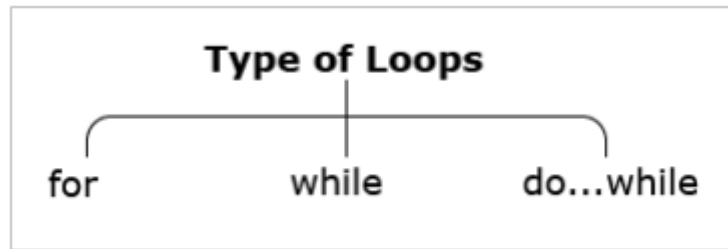
```
1. let counter = 0;
2. /* Same statement repeated 5 times */
3. counter++;
4. counter++;
5. counter++;
6. counter++;
7. counter++;
```

Looping statements in JavaScript helps to execute statement(s) required number of times without repeating code.

☒ **Best Practice:** Avoid heavy nesting inside the loops.

## Types of Loops

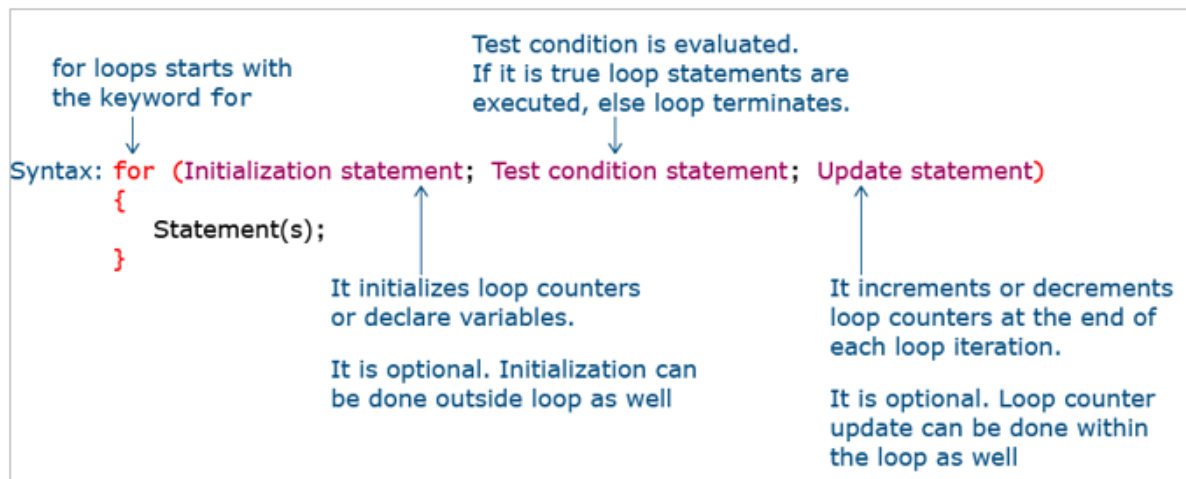
JavaScript supports popular looping statements as shown below:



Let us understand each of them in detail.

## For Loop

'for' loop is used when the block of code is expected to execute for a specific number of times. To implement it, use the following syntax.



**Example:** Below example shows incrementing variable counter five times using 'for' loop:

Also, shown below is output for every iteration of the loop.

```
1. let counter = 0;
2. for (let loopVar = 0; loopVar < 5; loopVar++) {
3.   counter = counter + 1;
4.   console.log(counter);
5. }
```

Here, in the above loop

`let loopVar=0; // Initialization`

`loopVar < 5; // Condition`



loopVar++; // Update

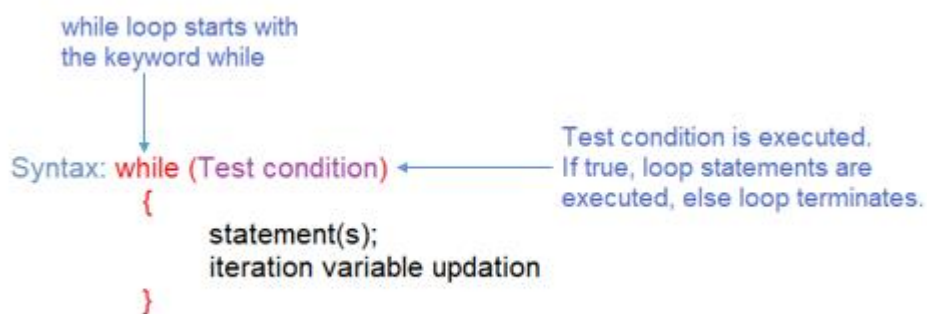
counter = counter + 1; // Action

To understand loops better refer the below table:

| loopVar | counter |
|---------|---------|
| 0       | 1       |
| 1       | 2       |
| 2       | 3       |
| 3       | 4       |
| 4       | 5       |

## While Loop

'while' loop is used when the block of code is to be executed as long as the specified condition is true. To implement the same, the following syntax is used:



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

**Example:** The below example shows an incrementing variable counter five times using a 'while' loop.

Also, shown below is the output for every iteration of the loop.

```
1. let counter = 0;
2. let loopVar = 0;
3. while (loopVar < 5) {
4.   console.log(loopVar);
5.   counter++;
6.   loopVar++;
7.   console.log(counter);
8. }
```

Here, in the above loop



```
let counter=0; // Initialization
```

```
let loopVar=0; // Initialization
```

```
loopVar < 5; // Condition
```

```
loopVar++; // Update
```

```
counter++; // Action
```

To understand loops better refer the below table:

| loopVar | counter |
|---------|---------|
| 0       | 1       |
| 1       | 2       |
| 2       | 3       |
| 3       | 4       |
| 4       | 5       |

## 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.

To implement 'do-while' loop, use the following syntax:

```
Syntax: do
{
    Statement(s);
}while (Test condition)
```

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

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

**Example:** Below example shows incrementing variable counter five times using 'do-while' loop:

Also, shown below is output for every iteration of the loop.

```
1. let counter = 0;
2. let loopVar = 0;
3. do {
4.   console.log(loopVar);
5.   counter++;
6.   loopVar++;
7.   console.log(counter);
8. }

9. while (loopVar < 5);
```

Here, in the above loop

let counter=0; // Initialization

let loopVar=0; // Initialization

loopVar < 5; // Condition

loopVar++; // Update

counter++; // Action

To understand loops better refer the below table:

| loopVar | counter |
|---------|---------|
| 0       | 1       |
| 1       | 2       |
| 2       | 3       |
| 3       | 4       |
| 4       | 5       |

## Functions in Javascript

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 understood so far was running in immediate mode. As soon as the page is loaded 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.

In JavaScript, functions are first-class objects. i.e., functions can be passed as an argument to other functions, it can be a return value of another function or can be assigned as a value to a variable. JavaScript leverages this behavior to extend its capabilities.

## Types of Functions

JavaScript has two types of functions.

### 1. User-defined functions

- JavaScript allows to write 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.

## Function Declaration and Function Invocation

To use a function, it must be defined or declared and then it can be invoked 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 {...}.

### Syntax for Function Declaration:

```
1. function function_name(parameter 1, parameter 2 , ..., parameter n) {  
2.     //statements to be executed  
3. }
```

### Example:

```
1. function multiply(num1, num2) {  
2.     return num1 * num2;  
3. }
```



The code written inside the function body will be executed only when it is invoked or called.

## Syntax for Function Invocation:

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

## Example:

```
1. multiply (5,6);
```

## Arrow Function

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

```
1. let sayHello = function () {  
2.   console.log("Welcome to JavaScript");  
3. };  
  
4. sayHello();
```

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

JavaScript has introduced a new and concise way of writing functions using arrow notation. The arrow function is one of the easiest ways to declare an anonymous function.

## Example:

```
1. let sayHello = () => {  
2.   console.log("Welcome to JavaScript");  
3. };  
  
4. sayHello();
```

There are two parts for the Arrow function syntax:

1. let sayHello = ()

- This declares a variable sayHello and assigns a function to it using () to just say that the variable is a function.

2. => {}

- This declares the body of the function with an arrow and the curly braces.

Below are a few scenarios of arrow functions.

**Syntax 1:** Multi-parameter, multi-line code:

If code is in multiple lines, use {}.

```
1. calculateCost = (ticketPrice, noOfPerson) => {  
2.     noOfPerson = ticketPrice * noOfPerson;  
3.     return noOfPerson;  
4. }  
5. console.log(calculateCost(500, 2));  
6. // 1000  
7.
```

**Syntax 2:** No parameter, single line code:

If the code is single line, {} is not required. The expression is evaluated and automatically returned.

```
1. trip = () => "Let's go to trip."  
2. console.log(trip());  
3. // Let's go to trip.  
4.
```

**Syntax 3:** One parameter, single line code:

If only one parameter, then () is not required.

```
1. trip = place => "Trip to " + place;  
2. console.log(trip("Paris"));  
3. // Trip to Paris  
4.
```

**Syntax 4:** One parameter, single line code:

if only one parameter, use '\_' and do not use a variable name also.

```
1.     trip = _ => "Trip to " + _;  
2.     console.log(trip("Paris"));  
3.     // Trip to Paris
```

Arrow function also adds a great difference with respect to the context object – 'this' reference.

Consider the below code where a regular function is defined within a method:

```
1. const myObject = {  
2.   items: [1],  
3.   myMethod() {  
4.     console.log(this == myObject) // true  
5.     this.items.forEach(function() {  
6.       console.log(this === myObject) // false  
7.       console.log(this === window); // true  
8.     });  
9.   }  
10. };  
11.  
12. myObject.myMethod();
```

A regular function defines its 'this' value based on how the function is invoked.

In the above-mentioned example, the myObject defines 'this' as an instance of itself. So, in line 4, the reference to 'this' points to the myObject itself. The regular function is used within the forEach() method. So, inside of the regular function, 'this' points to the window global object.

If the same logic is re-written using the arrow function as below:

```
1. const myObject = {  
2.   items: [1],  
3.   myMethod() {  
4.     console.log(this == myObject) // => true  
5.     this.items.forEach(() => {  
6.       console.log(this === myObject) // => true  
7.       console.log(this === window); // => false  
8.     });  
9.   }  
10. };  
11.
```

```
12. myObject.myMethod();  
13.
```

Arrow functions do not have their own 'this'. If 'this' is accessed, then its value is taken from the outside of the arrow function. So, in the above-mentioned code, the value of 'this' inside the arrow function equals to the value of 'this' of the outer function, that is, myObject.

## 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. function multiply(num1, num2) {  
2.   if (num2 == undefined) {  
3.     num2 = 1;  
4.   }  
5.   return num1 * num2;  
6. }  
7. console.log(multiply(5, 6)); // 30  
8. console.log(multiply(5)); // 5
```

JavaScript introduces an option to assign default values in functions.

```
1. function multiply(num1, num2 = 1) {  
2.   return num1 * num2;  
3. }  
4. console.log(multiply(5, 5)); //25  
5. console.log(multiply(10)); //10  
  
6. console.log(multiply(10, undefined)); //10
```

In the above example, when the function is invoked with two parameters, the default value of num2 will be overridden and considered when the value is omitted while calling.

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

**Syntax:**



```
1. function(a, ...args) {  
2.     //...  
  
3. }
```

The rest of the parameters can be included in the function definition by using three dots ( ... ) followed by the name of the array that will hold them.

## Example:

```
1. function showNumbers(x, y, ...z) {  
2.     return z;  
3. }  
4. console.log(showNumbers(1, 2, 3, 4, 5)); // [3,4,5]  
5. console.log(showNumbers(3, 4, 5, 6, 7, 8, 9, 10)); //  
  
    [5,6,7,8,9,10]
```

The rest parameter should always be the last parameter in the function definition.

Destructuring gives a syntax which makes it easy to unpack values from arrays, or properties from objects, into different variables.

## Array destructuring in functions

### Example:

```
1. let myArray = ["Andrew", "James", "Chris"];  
2.  
3. function showDetails([arg1, arg2]) {  
4.     console.log(arg1); // Andrew  
5.     console.log(arg2); // James  
6. }  
7.  
  
8. showDetails(myArray);
```

In the above example, the first two array elements 'Andrew' and 'James' have been destructured into individual function parameters arg1 and arg2.

## Object destructuring in functions

### Example:

```
1. let myObject = { name: "Mark", age: 25, country: "India" };  
2.
```

```
3. function showDetails({ name, country }) {  
4.   console.log(name, country); // Mark India  
5. }  
6.  
7. showDetails(myObject);
```

The properties name and country of the object have been destructured and captured as a function parameter.

## Nested Function

In JavaScript, it is perfectly normal to have functions inside functions. The function within another function body is called a nested function.

The nested function is private to the container function and cannot be invoked from outside the container function.

### Example:

```
1. function giveMessage(message) {  
2.   let userMsg = message;  
3.   function toUser(userName) {  
4.     let name = userName;  
5.     let greet = userMsg + " " + name;  
6.     return greet;  
7.   }  
8.   userMsg = toUser("Bob");  
9.   return userMsg;  
10. }  
11.  
12. console.log(giveMessage("The world says hello dear: "));  
13. // The world says hello dear: Bob
```

## Built-In-Functions

JavaScript comes with certain built-in functions. To use them, they need to be invoked.

Below is the table with some of these built-in functions to understand their significance and usage.

| 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   | let decision = confirm("Shall we proceed?"); |

|              |  |  |
|--------------|--|--|
|              | clicked, the function returns "true", else returns "false".  |  |
| 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<br>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<br>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> | <p>parseInt("10"); //10</p> <p>parseInt("10 20 30"); //10, only the integer part is returned</p> <p>parseInt("10 years"); //10</p> <p>parseInt("years 10"); //NaN, the first character stops the parsing</p> |
| 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>   | <p>parseFloat("10.34"); //10.34</p> <p>parseFloat("10 20 30"); //10</p> <p>parseFloat("10.50 years"); //10.50</p>  |
| 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)");   |

JavaScript provides two-timer built-in functions. Let us explore these timer functions.

| Built-in functions | Description  | Example   |
|--------------------|--|---|
| setTimeout()       | It executes a given function after waiting for the specified number of milliseconds.<br><br>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().<br><br>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.<br><br>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().<br><br>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()!") }</pre> |

|  |  |   |
|--|--|---|
|  |  | <pre>setTimeout(stopInterval,5000)  //It executes executeMe() every 2 seconds and after 5 seconds, further calls to executeMe() is stopped.</pre> |
|--|--|---|

## Variable Scope In Functions

Variable declaration in the JavaScript program can be done within the function or outside the function. But the accessibility of the variable to other parts of the same program is decided based on the place of its declaration. This accessibility of a variable is referred to as scope.

JavaScript scopes can be of three types:

- Global scope
- Local scope
- Block scope

## Global Scope

Variables defined outside function have Global Scope and they are accessible anywhere in the program.

### Example:

```
1.      //Global variable
2.      var greet = "Hello JavaScript";
3.
4.      function message() {
5.
6.          //Global variable accessed inside the function
7.          console.log("Message from inside the function: "
+ greet);
8.      }
9.
10.     message();
11.
12.     //Global variable accessed outside the function
13.     console.log("Message from outside the function: " +
greet);
14.
15.     //Message from inside the function: Hello JavaScript
16.     //Message from outside the function: Hello JavaScript
```

## Local Scope

Variables declared inside the function would have local scope. These variables cannot be accessed outside the declared function block.

## Example:

```
1. function message() {
2.   //Local variable
3.   var greet = "Hello JavaScript";
4.   //Local variables are accessible inside the function
5.   console.log("Message from inside the function: " + greet);
6. }
7.
8. message();
9.
10. //Local variable cannot be accessed outside the function
11. console.log("Message from outside the function: " + greet);
12.
13. //Message from inside the function: Hello JavaScript
14. //Uncaught ReferenceError: greet is not defined
```

If a local variable is declared without the use of keyword 'var', it takes a global scope.

## Example:

```
1.   //Global variable
2.   var firstName = "Mark";
3.   function fullName() {
4.     //Variable declared without var has global scope
5.     lastName = "Zuckerberg";
6.     console.log("Full Name from inside the function: " +
7.       firstName + " " + lastName);
8.   }
9.   fullName();
10.  console.log("Full Name from outside the function: " +
11.    firstName + " " + lastName);
12.  //Full Name from inside the function: Mark Zuckerberg
13.  //Full Name from outside the function: Mark Zuckerberg
```

## Block Scope

In 2015, JavaScript introduced two new keywords to declare variables: let and const.

Variables declared with 'var' keyword are function-scoped whereas variables declared with 'let' and 'const' are block-scoped and they exist only in the block in which they are defined.

Consider the below example:

```
1. function testVar() {  
2.   if (10 == 10) {  
3.     var flag = "true";  
4.   }  
5.   console.log(flag); //true  
6. }  
7.  
  
8. testVar();
```

In the above example, the variable flag declared inside 'if' block is accessible outside the block since it has function scope

Modifying the code to use 'let' variable will result in an error:

```
1. function testVar() {  
2.   if (10 == 10) {  
3.     let flag = "true";  
4.   }  
5.   console.log(flag); //Uncaught ReferenceError: flag is not  
   defined  
6. }  
7.  
  
8. testVar();
```

The usage of 'let' in the above code snippet has restricted the variable scope only to 'if' block.

'const' has the same scope as that of 'let' i.e., block scope.

## Hoisting

JavaScript interpreter follows the process called hoisting.

Hoisting means all the variable and function declarations wherever they are present throughout the program, gets lifted and declared to the top of the program. Only the declaration and not the initialization gets hoisted to the top.

If a variable is tried to access without declaration, the Reference Error is thrown.

Let us declare and initialize the variable in the code but after it is accessed.

```
1. console.log("First name: "+firstName); //First name: undefined
```

```
2. var firstName = "Mark";
```

Because of Hoisting, the code is interpreted as below by the interpreter:

```
1. var firstName;  
2. console.log("First name: "+firstName); // First name: undefined  
  
3. firstName ="Mark";
```

Hoisting here helps interpret to find the declaration at the top of the program and thus reference error goes away. But interpreter says that the variable is not defined. This is because hoisting only lifted the variable declaration on the top and not initialization.

Variables declared using 'let' and 'const' are not hoisted to the top of the program.

### Example:

```
1. console.log("First name: "+firstName);  
  
2. let firstName = "Mark";
```

The above code throws an error as "Uncaught ReferenceError: Cannot access 'firstName' before initialization"

## Working With Classes

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 eases the development and work with Classes in the application.
- JavaScript is an object-based language based on prototypes and allows 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.

## Creating and Inheriting Classes

### 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.
- Different methods are used to communicate between various objects, to perform various operations.

## Example:

The below code demonstrates a calculator accepting two numbers to do addition and subtraction operations.

```
1. class Calculator {
2.   constructor(num1, num2){ // Constructor used for initializing
   the class instance
3.
4.     /* Properties initialized in the constructor */
5.     this.num1 = num1;
6.     this.num2 = num2;
7.   }
8.
9.   /* Methods of the class used for performing operations */
10.  add() {
11.    return this.num1 + this.num2;
12.  }
13.
14.  subtract() {
15.    return this.num1 - this.num2;
16.  }
17. }
18.
19. let calculator = new Calculator(300, 100); // Creating
   Calculator class object or instance
20. console.log("Add method returns" + calculator.add()); // Add
   method returns 400.
21. console.log("Subtract method returns" + calculator.subtract());

   // Subtract method returns 200.
```

## Class-Static Method

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

In the below example, display() is a static method and it is accessed using the class name.

```
1. class Calculator {
```

```
2.         constructor(num1, num2) { // Constructor used
           for initializing the class instance
3.           /* Properties initialized in the
           constructor */
4.           this.num1 = num1;
5.           this.num2 = num2;
6.       }
7.
8.           /* static method */
9.       static display() {
10.          console.log("This is a calculator
           app");
11.       }
12.
13.          /* Methods of the class used for
           performing operations */
14.       add() {
15.          return this.num1 + this.num2;
16.       }
17.
18.       subtract() {
19.          return this.num1 - this.num2;
20.       }
21.     }
22.
23.           /*static method display() is invoked using class
           name directly. */
24.       Calculator.display();
```

The output of the above code is :

This is a calculator app

## 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 you have '**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

### Example:

The below code explains the concept of inheritance.

```
1. class Vehicle {
2.   constructor(make, model) {
3.
4.     /* Base class Vehicle with constructor initializing two-
       member attributes */
5.     this.make = make;
6.     this.model = model;
7.   }
8. }
9.
10. class Car extends Vehicle {
11.   constructor(make, model, regNo, fuelType) {
12.     super(make, model); // Sub class calling Base
       class Constructor
13.     this.regNo = regNo;
14.     this.fuelType = fuelType;
15.   }
16.   getDetails() {
17.     /* Template literals used for displaying details
       of Car. */
18.     console.log(`${this.make},${this.model},${this.regNo},${this.fue
       lType}`);
19.   }
20. }
21.
22. let c = new Car("Hundai", "i10", "KA-016447", "Petrol"); //
       Creating a Car object
23. c.getDetails();
```

## Sub Classing Built-ins

### Subclassing Built-ins

The keywords, class and extends, help developers to create classes and implement inheritance in the application where user-defined classes can be created and extended. Similarly, the built-in classes can be subclassed to add more functionality.

#### Example:

To display the array items, the built-in Array class can be extended as mentioned below.

```
1. class MyArray extends Array {
2.   constructor(...args) {
3.     super(...args);
4.   }
5.   display() {
6.     let strItems = "";
7.     for (let val of this) {
8.       strItems += `${val} `;
9.     }
10.    console.log(strItems);
```

```
11.     }  
12. }  
13.  
14. let letters = new MyArray("Sam", "Jack", "Tom");  
  
15. letters.display();
```

Note that display is not the method present in Array built-in class. The MyArray subclasses the Array and adds to it. The output of the above code is given below.

**Sam Jack Tom**

☒ **Best Practice:** Class methods should be either made reference using **this** keyword or it can be made into a static method.

## Working With 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.



When execution of JavaScript code is delayed or deferred till some event occurs, the execution is called deferred mode execution. This makes JavaScript an action-oriented language.

Let us understand how JavaScript executes as a reaction to these events.

## Inbuilt Events and Handlers

Below are some of the built-in event handlers.

| 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. |

## 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.

### Syntax:

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

### Example:

To listen to the click event on the paragraph element, it is done as shown below:

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

When the user clicks on element 'p', event handler 'onclick' listens to the event 'click' and executes the 'executeMe' code written in JavaScript file against the event handler.

### Example:

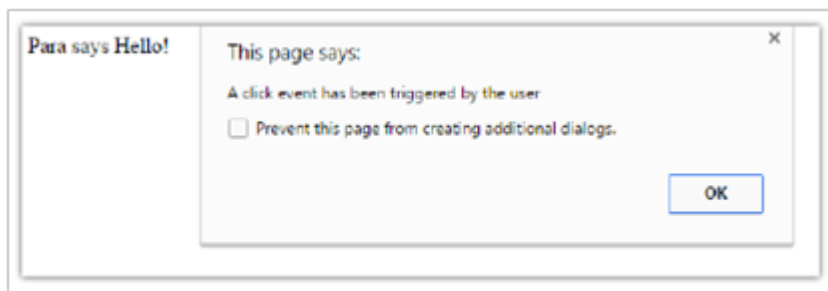
1. Event Handler 'onclick' is associated with the HTML element 'p' to handle the 'click' on this element.

```
1. <html>
2.   <head>
3.     <script src="test.js"></script>
4.   </head>
5.
6.   <body>
7.     <p onclick="executeMe();" >Para says !!! </p>
8.   </body>
9. </html>
```

2. When the user clicks on element 'p', event handler 'onclick' listens to the event 'click' and executes the code written against the event handler. The corresponding code is the function 'executeMe' written in the "test.js" file.

```
1. function executeMe() {  
2.     alert('A click event has been triggered by the user');  
3. }
```

3. The function 'executeMe' will now execute.



As seen in event handling code, event-handler is a piece of JavaScript code put inside the HTML Paragraph element .

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

On the right-hand side of this code instead of invoking a function, lines of code can be directly written as shown below:

```
1. <p onclick="alert('A click event has been triggered by the  
user');" >Para says !!! </p>
```

This is referred to as Inline Scripting where lines of JavaScript code is embedded inline to HTML elements.

However, due to tight coupling with the elements in which the code is written, this approach is not suggested. The alternate and much better approach is to use functions in JavaScript.

☑ **Best Practice:** Event listeners are the most preferred way to handle events in JavaScript. One of the major points to use event listeners is, it does allow us to add multiple event listeners on the same element when compared with the "on" properties like onclick, onmouseover, etc..

## Exception Handling

Exception handling is accomplished with a try...catch statement. When the program encounters an exception, the program will terminate in an unfriendly manner. To protect against this, the code can be placed in a try...catch statement and avoid terminating the program unexpectedly.

- Try statements: It lets the developer validate a block of code whether it will result in some errors or not.
- Catch statements: It lets the developer handle the error without terminating the program in an unfriendly manner.
- Throw statements: It helps the developer to create custom errors.
- Finally statements: It lets the developer execute code, after the try and catch block execution, irrespective of the result.

## Example:

```
1. <html>
2.
3. <head>
4.   <script>
5.     let num1 = 100;
6.     let num2 = 0;
7.     try {
8.       if (isNaN(num1) || isNaN(num2)) {
9.         throw "Not a number"
10.      }
11.      else {
12.        if (num2 == 0) {
13.          throw "Divide by zero error";
14.        }
15.        else {
16.          let result = num1 / num2;
17.        }
18.      }
19.    }
20.    catch (e) {
21.      console.log("Error: " + e);
22.    }
23.    finally {
24.      console.log("Some error occurred");
25.    }
26.  </script>
27. </head>
28.
29. <body>
30. </body>
31.
32. </html>
```

Here, num2 is 0 so when you try to divide num1 by num2 it will throw Divide by zero error. Also, finally() is called whenever an error occurs. So as a result, two statements will be printed: "Divide by zero" and "Some error occurred"

## Working With Objects

In any programming language when real-world entities are to be coded, then variables are used. For most of the scenarios, a variable to hold data that represents the collection of properties is required.

For instance, to create an online portal for the car industry, Car as an entity must be modelled so that it 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:

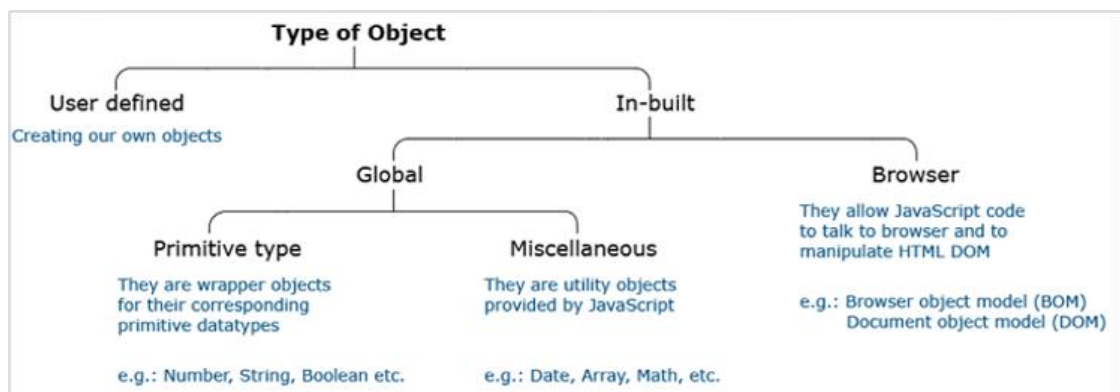
- Color=red
- Model = VXI
- Current gear = 3
- Current speed = 45 km / hr
- Number of doors = 4
- Seating Capacity = 5

The behavior of Car object:

- Accelerate
- Change gear
- Brake

## Types of Objects

JavaScript objects are categorized as follows:



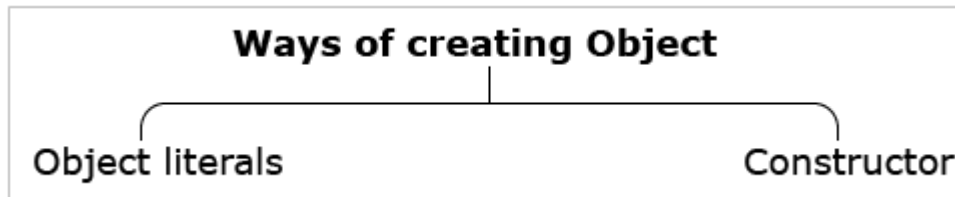


## Creating Objects

In JavaScript objects, the state and behaviour is represented as a collection of properties

Each property is a [key-value] pair where the key is a string and the value can be any JavaScript primitive type value, an object, or even a function.

JavaScript objects can be created using two different approaches.



Objects can be created using object literal notation. Object literal notation is a comma-separated list of name-value pairs wrapped inside curly braces. This promotes the encapsulation of data in a tidy package. This is how the objects in JavaScript are created using the literal notation:

### Syntax:

```
1. objectName = {  
2.     //-----states of the object-----  
3.     key_1: value_1,  
4.     key_2: value_2,  
5.     ...  
6.     key_n: value_n,  
7.     //-----behaviour of the object-----  
8.     key_function_name_1: function (parameter) {  
9.         //we can modify any of the property declared above  
10.    },  
11.    ...  
12.    key_function_name_n: function(parameter) {  
13.        //we can modify any of the property declared above  
14.    }  
15. }
```

### Example:

```
1.     //-----states of the object-----  
2.     let myCar = {  
3.         name: "Fiat",  
4.         model: "VXI",  
5.         color: "red",  
6.         numberOfGears: 5,  
7.         currentGear: 3,
```



```
8.         currentSpeed: 45,  
9.         //-----Behaviour of the object-----  
10.        accelerate: function (speedCounter) {  
11.            this.currentSpeed = this.currentSpeed +  
speedCounter;  
12.            return this.currentSpeed;  
13.        },  
14.  
15.        brake: function (speedCounter) {  
16.            this.currentSpeed = this.currentSpeed -  
speedCounter;  
17.            return this.currentSpeed;  
18.        }  
19.    }
```

Below is the older syntax used to create object literals:

```
1. let name = "Arnold";  
2. let age = 65;  
3. let country = "USA";  
4. let obj = {  
5.     name: name,  
6.     age: age,  
7.     country: country  
8. };
```

Below is the modern way to create objects in a simpler way:

```
1. let name="Arnold";  
2. let age=65;  
3. let country="USA";  
  
4. let obj={name,age,country};
```

The object literal property shorthand is syntactic sugar, which simplifies the syntax when literals are used in function parameters or as return values.

```
1. //Literal property without shorthand  
2. function createCourse(name, status) {  
3.     return {type: "JavaScript", name: name, status: status};  
4. }  
5.  
6. function reviewCourse(name) {  
7.     return {type: "JavaScript", name: name};
```

```
8. }
9.
10.
11. /*Literal property with shorthand
12. when the property and the value identifiers have the same name,
13. the identifier can be omitted to make it implicit*/
14.
15. function createCourse(name, status) {
16.     return {type: "JavaScript", name, status};
17. }
18.
19. function reviewCourse(name) {
20.     return {type: "JavaScript", name};
21. }
```

Earlier in JavaScript to add a dynamic property to an existing object, below syntax was used.

```
1. let personalDetails = {
2.     name: "Stian Kirkeberg",
3.     country: "Norway"
4. };
5.
6. let dynamicProperty = "age";
7. personalDetails[dynamicProperty] = 45;
8. console.log(personalDetails.age); //Output: 45
9.
```

With newer updates in JavaScript after 2015 the dynamic properties can be conveniently added using hash notation and the values are computed to form a key-value pair.

```
1. let dynamicProperty = "age";
2. let personalDetails = {
3.     name: "Stian Kirkeberg",
4.     country: "Norway",
5.     [dynamicProperty]: 45
6. };
7.
8. console.log(personalDetails.age); //Output: 45
```

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.

**Example:**

```
1. function Car(name, model, color, numberOfGears, currentSpeed, current
   Gear) {
2.   //-----States of the object-----
3.   this.name = name;
4.   this.model = model;
5.   this.color = color;
6.   this.numberOfGears = numberOfGears;
7.   this.currentSpeed = currentSpeed;
8.   this.currentGear = currentGear;
9.
10.  //-----Behaviour of the object-----
11.  this.accelerate = function (speedCounter) {
12.    this.currentSpeed = this.currentSpeed + speedCounter;
13.    return this.currentSpeed;
14.  }
15.
16.  this.brake = function (speedCounter) {
17.    this.currentSpeed = this.currentSpeed - speedCounter;
18.    return this.currentSpeed;
19.  }
20. }
```

'this' keyword that is used in this case 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.

### Example:

If used inside an object definition, it points to that object itself. If used inside the function definition, it points to the object that owns the function.

To create objects using function constructor, make use of 'new' keyword, and invoke the function. This initializes a variable of type object. The properties and methods of the object can be invoked using the dot or bracket operator.

Retrieving state using the dot operator:

```
1. myCar.name; //return "Fiat"
2. myCar.currentSpeed; //returns 45
```

Retrieving behavior using the dot operator:

```
1. myCar.accelerate(50); //invokes accelerate() with argument = 50
```

Retrieving state using the bracket operator:

```
1. myCar["name"]; //return "Fiat"
2. myCar["currentSpeed"]; //returns 45
```

Retrieving behavior using the bracket operator:

```
1. myCar["accelerate"](50); //invokes accelerate() with argument = 50
```

## Combining Objects Using Spread Operator

The spread operator is used to combine two or more objects. The newly created object will hold all the properties of the merged objects.

**Syntax:**

```
1. let object1Name = {
2.   //properties
3. };
4. let object2Name = {
5.   //properties
6. };
7. let combinedObjectName = {
8.   ...object1Name,
9.   ...object2Name
10. };
11. //the combined object will have all the properties of object1 and object2
```

**Example:**

```
1. let candidateSelected={
2.   Name:'Rexha Bebe',
3.   Qualification:'Graduation',
4. };
5. let SelectedAs={
```



```
6.   jobTitle:'SystemEngineer',
7.   location:'Bangalore'
8. };
9. let employeeInfo={
10.     ...candidateSelected,
11.     ...SelectedAs
12. };
13. console.log(employeeInfo);
14. /*
15. {
16.     Name: 'Rexha Bebe',
17.     Qualification: 'Graduation',
18.     jobTitle: 'SystemEngineer',
19.     location: 'Bangalore'
20. }
21. */
```