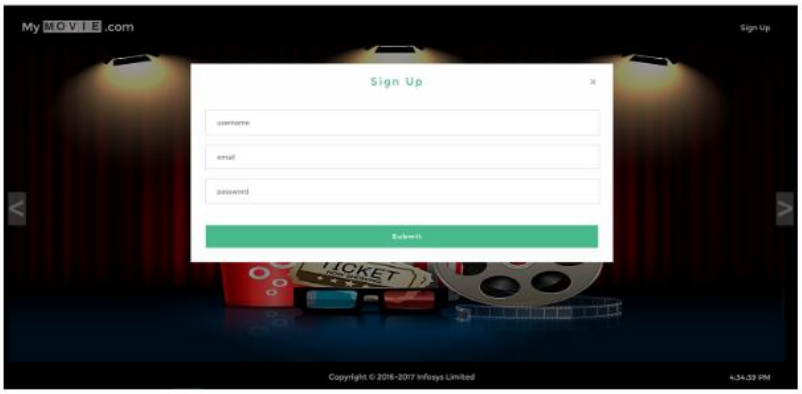
UNIT-2
JavaScript

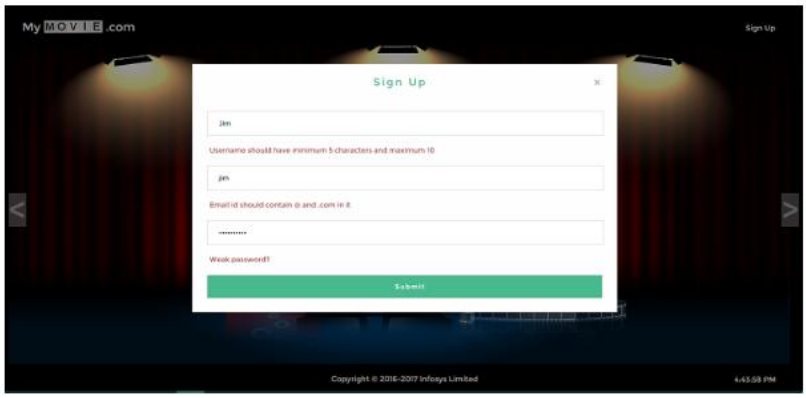
Why we need JavaScript:

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.

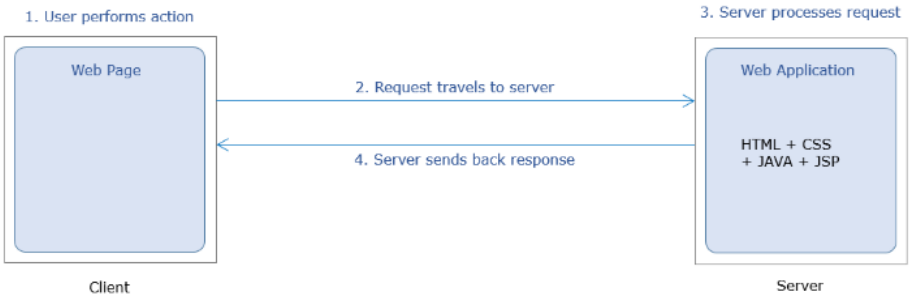


When data is entered in the fields and the button is clicked, then data entered in the fields will be validated and accordingly, next view page loaded. If data is invalid, an error message is displayed, if valid, the application navigates to homepage



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

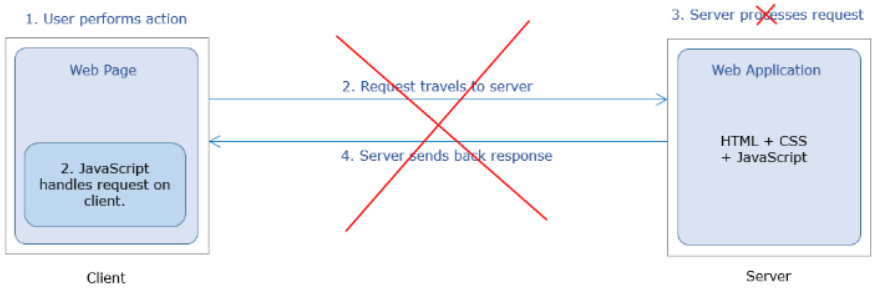
JavaScript To implement the requirement of handling user action like a click of a button or link and to respond to these requests by displaying the expected output, server-side languages like Java/JSP can be used as shown in the below diagram.



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.

Enhancement brought in JavaScript

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.
- 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 Mozilla Firefox
V8	Google Chrome 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.

Where to write JavaScript code?

JavaScript code can be embedded within the HTML page or can be written in an external file.
There are three ways of writting JavaScript depending on the platform :

- Inline Scripting

- Internal Scripting
- External Scripting

Internal 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 :

```
<html>
<head>
   <script>
      //internal script
   </script>
</head>
<body>
</body>
</html>
```

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

```
<html>
<head>
</head>
<body>
   <script>
      //internal script
   </script>
</body>
</html>
```

External Scripting

- 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:

```
<html>
<head>
   <!-- *.js file contain the JavaScript code -->
   <script src="*.js"></script>
</head>
<body>
</body>
</html>
```

Example: Demo.js

```
let firstName="Rexha";
```

```
let lastName ="Bebe";
console.log(firstName+" "+lastName);
```

Demo.html :

```
<html>
<head>
   <script src="Demo.js"></script>
</head>
<body>
</body>
</html>
```

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

Internal v/s External Script

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
Maintainabilty	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

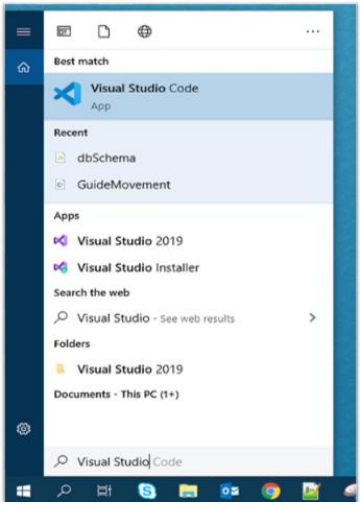
Environmental Setup - Internal

To work with JavaScript, you can use

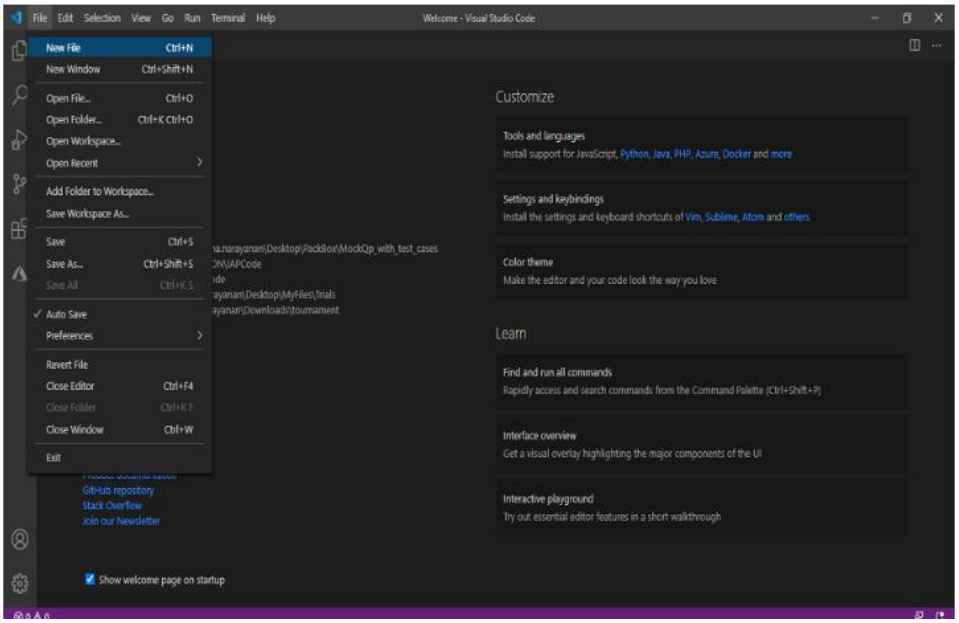
- Editor (Visual Studio Code IDE recommended)
- Browser (Google Chrome recommended)
- As JavaScript works properly on any Browser or OS, you can choose any of them based on your preference.
- It is also possible to write JavaScript code using Editors. You can use simple editors such as notepad or go for an IDE like Visual Studio Code which offers IntelliSense support and syntax error highlighter that makes coding easier.
- To reduce development time, IDE can be used as it has built-in features.
- In Visual Studio Code, extensions can be added that will speed up development and helps code to a higher standard by providing linting.

Steps to execute JavaScript 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 file:

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

index.html

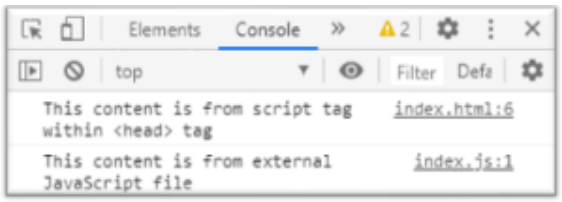
```
<html>
<head>
   <title>JavaScript Intoduction</title>
   <script>
      console.log("This content is from script tag within <head> tag");
   </script>
</head>
<script src="index.js"></script>
<body>
</body>
```

```
</html>
```

Rendering of index.html file will execute the JavaScript code written within the file and also the code in index.js which is included in index.html.

Output:

For the output, copy the index.html file path into the browser. Go to the developer tool in the browser, there in the console option, the output is as shown below:



- In the above example, 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 done 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.

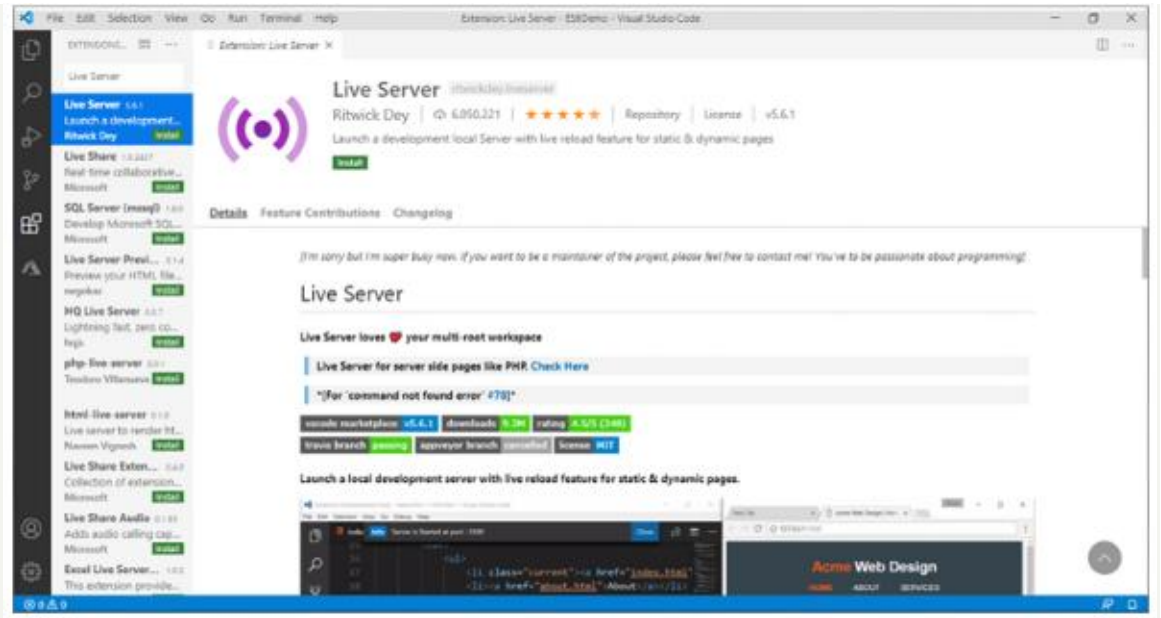
Environmental Setup - Internal - Adding Live Server

Need for Live Server:

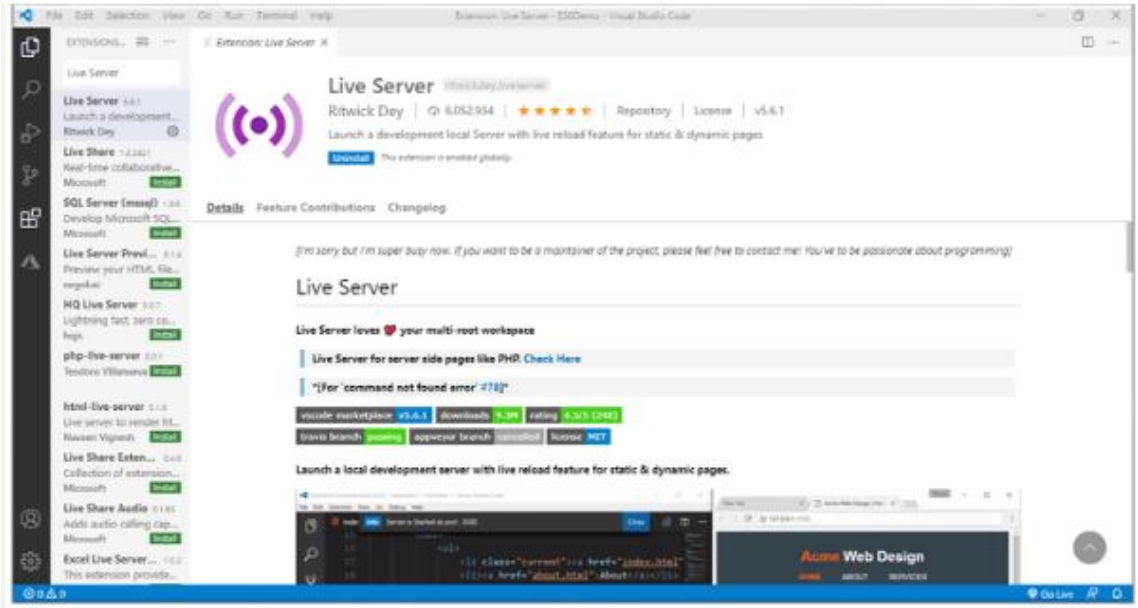
Visual Studio Code provides an extension called Live Server using which HTML page can be rendered and any changes that developers make further will be automatically detected and rendered properly.

Adding Live Server:

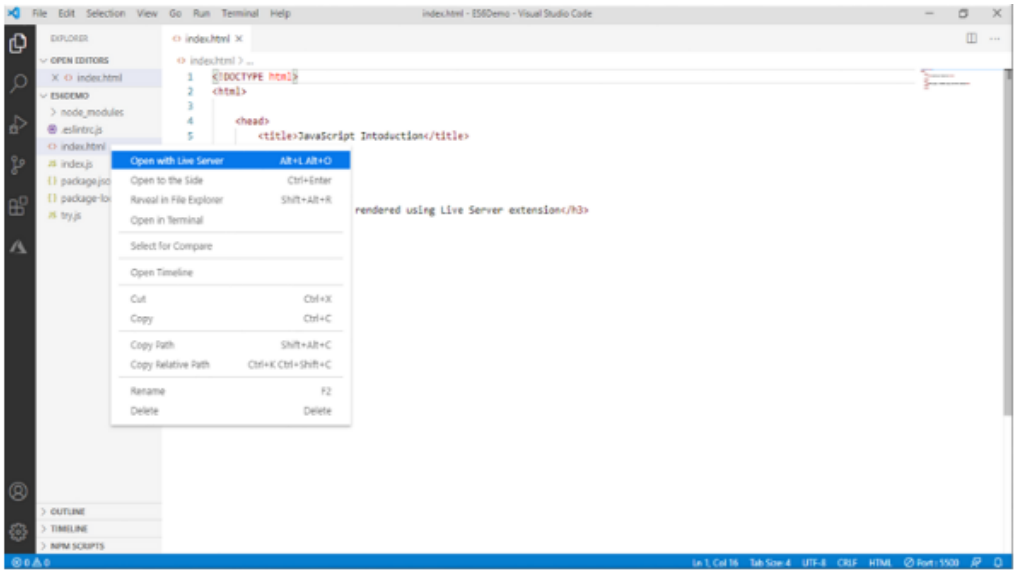
1. Go to the extension tab in Visual Studio Code and search for Live Server and click on the install button that is visible in.



2. Once it gets installed, there will be a screen with an uninstall button option as shown below and the Live Server is 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.



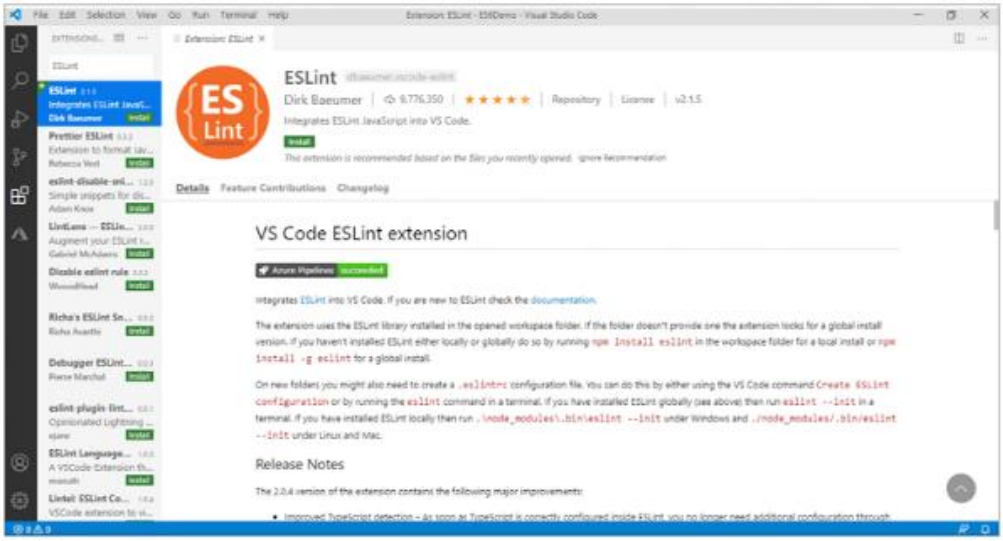
Now that you have understood how to use Live Server to render the page, let us proceed to understand some extensions that will help standardize the code written by adding linting.

Need for Linting:

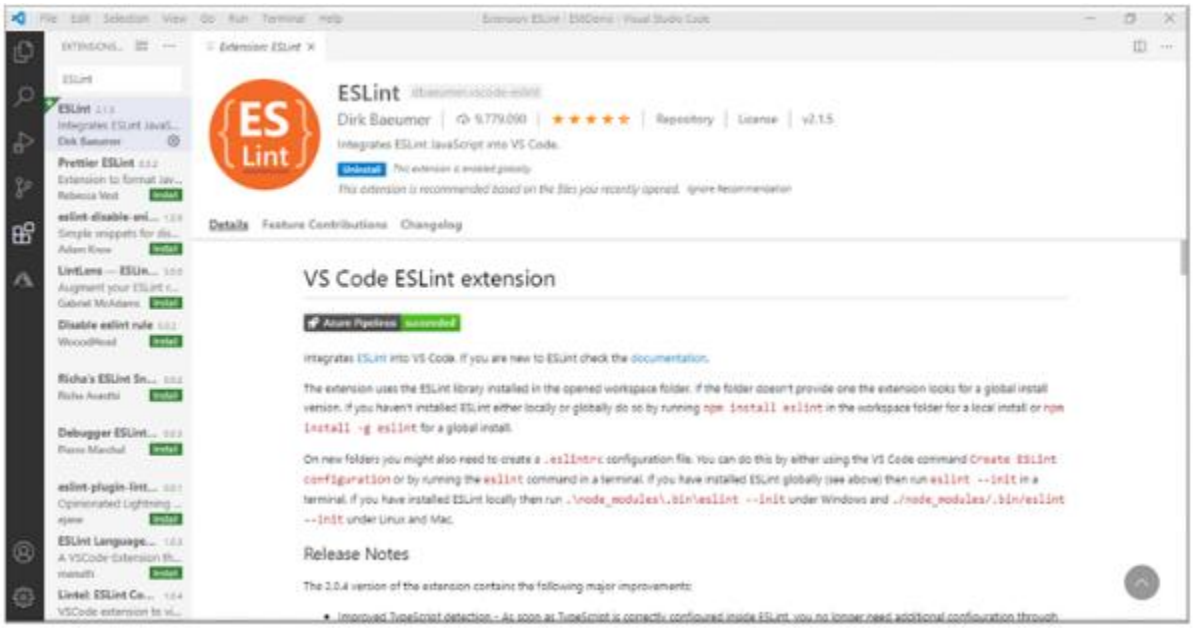
- As the application develops through multiple stages, code quality becomes very critical. Linting is the process of analyzing the code during development stage to notify the issues or errors in the code. This helps developer to quickly fix the issues during development and thereby improving the code quality.
- There are various linting tools specific to JavaScript, which helps to find and resolve the code issues. ESLint is one of the linting tool used for JavaScript.
- So, let us see how to add ESLint to Visual Studio Code Editor and start using it to create a quality code.

Adding ESLint plugin in VS Code IDE:

1. Open Visual Studio Code, go to the extension, search for "ESLint".
2. Once the ESLint extension appears, hit the "Install" button as shown below:



3. After the installation of ESLint, below screen will be visible:



Environmental Setup - Internal - Adding ESLint

- In order to display the code issues dynamically in the console, we need to install eslint node module from npm repository.
- Hence, the latest Node.js software needs to be installed in system in order to use npm. Download the latest version of Node.js from the software center.
- 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 (npm) which gets installed automatically along with Node.js installation.
- npm can be used to install any third-party JavaScript libraries. Here, we are using npm to install eslint for linting purpose.

Following are the steps to configure ESLint with JavaScript.

- Create a JavaScript demo.js file with below sample code

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

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

```
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:

```
npm install eslint --save-dev
```

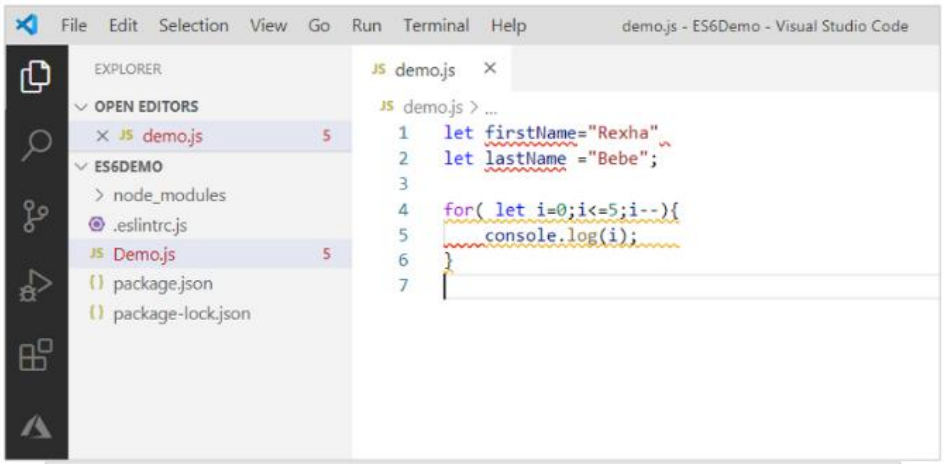
- Next, is to create the configuration file: ".eslintrc.js", using one of the below-given command:

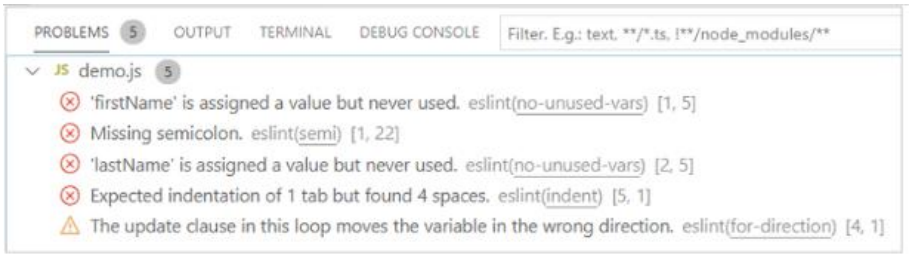
```
./node_modules/.bin/eslint --init
OR
npm init @eslint/config
```

This command will 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 ytpe 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 you r 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.





Working with Identifiers:

- 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:

```
firstName;
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.

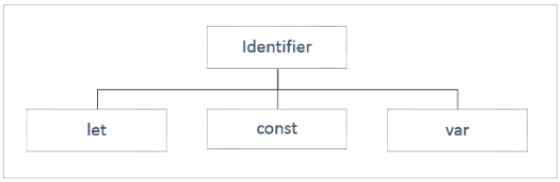
Reserved keywords are part of programming language syntax and cannot be used as identifiers.

Type of Identifiers:

let, const, var:

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



Let:

- An identifier declared using ‘let’ keyword has a block scope i.e., it is available only within the block in which it is defined.
- The value assigned to the identifier can be done either at the time of declaration or later in the code and can also be altered further.
- All the identifiers known so far vary in their scope and with respect to the data it holds.

Example:

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

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. */
console.log("Welcome to JS course, Mr."+name);
```

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.

Const

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:

```
const pi = 3.14;
console.log("The value of Pi is: "+pi);

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. */
console.log("The value of Pi is: "+pi);
```

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

Var:

- 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:

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

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

let vs. const vs. var

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

Data types:

- 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.

What are data types?

- 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.

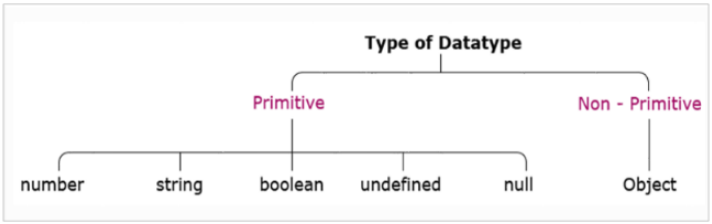
Example:

```
let age = 24; //number
let name = “Tom” //string
let qualified = true; //boolean
```

- 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.

Types of data types

JavaScript defines the following data types:



Primitive Data types:

Primitive data type – Number The data is said to be primitive if it contains an individual value.

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:

```
const pi = 3.14; // its value is 3.14
const smallestNaturalNumber = 0; // its value is 0
```

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:

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

Primitive data type – String

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:

```
let personName= “Rexha”;    //OR
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:

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

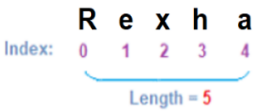
This will be interpreted as Rexha's and Rexha"s respectively. Thus, use opposite quotes inside and outside of JavaScript single and double quotes.

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

Example:

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

- It is a syntax error.
- Thus, remember, strings containing single quotes must be enlosed 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.



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

Example:

```
let firstName="Kevin";
let lastName="Patrick";
console.log("Name: "+firstName+" "+lastName+"\n
Email:"+firstName+"_"+lastName+"@abc.com");
/*
OUTPUT:
Name: Kevin Patrick
Email:Kevin_Patrick@abc.com
*/
```

- 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:

```
let firstName="Kevin";
let lastName="Patrick";
console.log(`Name:${firstName} ${lastName}
Email: ${firstName}_${lastName}@abc.com`);

/*
OUTPUT:
Name: Kevin Patrick
Email:Kevin_Patrick@abc.com
*/
```

- 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.

Primitive data type - 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:

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

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

Example 2:

```
let custName = "John"; //here value is John and the data type is String
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:

```
let item = null;
// 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:

```
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.

Primitive data type – BigInt

- 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:

```
const bigintvar = 67423478234689887894747472389477823647n;
        OR
const bigintvar = BigInt("67423478234689887894747472389477823647");
const bigintFromNumber = BigInt(10); // same as 10n
```

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

Example:

```
alert(3n + 2n); // 5
alert(7n / 2n); // 3
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:

```
let bigintvar = 6n;
let numvar = 3;
// number to bigint
alert(bigintvar + BigInt(numvar)); // 9
// bigint to number
alert(Number(bigintvar ) + numvar); // 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:

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

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

Example:

```
alert( 5 == 5n ); // true
alert( 5 === 5n ); // false
```

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

Example:

```
if (0n) {
```

```
  // never executes
}
```

- 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:

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

Primitive data type – Symbol

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

Example:

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

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

Example:

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

```
let empid1 = Symbol("empno");
let empid2 = Symbol("empno");
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:

```
let empid = Symbol("empno");
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:

```
let empid = Symbol("empno");
alert(empid.toString()); // Symbol(empno), now it works
        OR
//use description
let empid = Symbol("empno");
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:

```
// read from the global registry
let empid = Symbol.for("empno"); // if the symbol did not exist, it is created
// read it again (maybe from another part of the code)
let empidAgain= Symbol.for("empid");
// the same symbol
alert( empid === empidAgain ); // false
```

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

Non-Primitive data types

Objects:

- The data type is said to be non-primitive if it is a collection of multiple values.
- The variables in JavaScript may not always hold only individual values which are 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:

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

Example:

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

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:

```
let dummyArr = new Array();
        //OR
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:

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

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

Operators:

Working With Operators

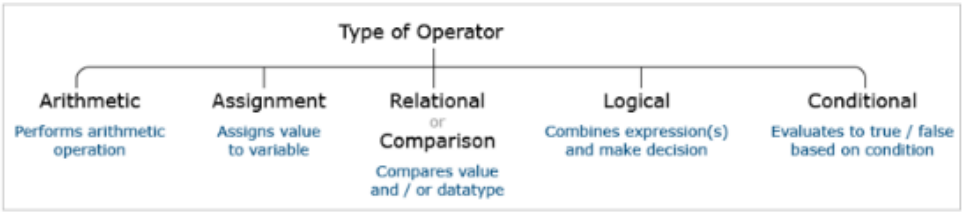
- Operators in a programming language are the symbols used to perform operations on the values.
- Operands: Represents the data.
- Operator: Performs certain operations on the operands.

```
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.

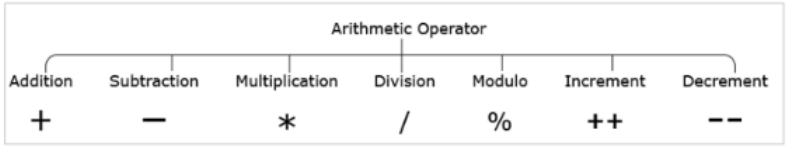
Types of Operators

- 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 Operators

Arithmetic operators are used for performing arithmetic operations




```
let sum = 5 + 3; // sum=8
let difference = 5 – 3; // difference=2
let product = 5 * 3; // product=15
let division = 5/3; // division=1
let mod = 5%3; // mod=2
```

```
let value = 5;
value++; // increment by 1, value=6
let value = 10;
value--; // decrement by 1, value=9
```

Arithmetic Operators - String Type Operands

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

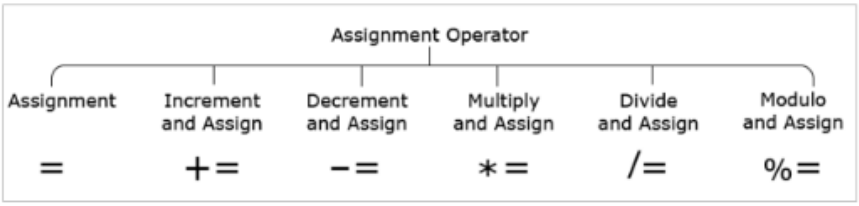
```
let firstName = "James";
let lastName = "Roche";
let name = firstName + " " + lastName;  // name = James Roche
```

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

```
let strValue="James";
let numValue=10;
let newStrValue= strValue + " " + numValue;  // newStrValue= James 10
```

Assignment Operators

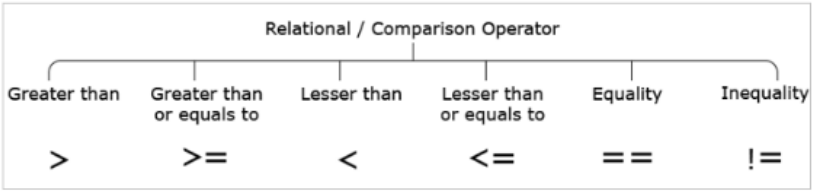
Assignment operators are used for assigning values to the variables.




```
let num = 30; // num=30
let num += 10; // num=num+10 => num=40
let num -= 10; // num=num-10 => num=20
let num *= 30; // num=num*30 => num=900
let num /= 10; // num=num/10 => num=3
let num %= 10; // num=num%10 => num=0
```

Relational or Comparison Operators

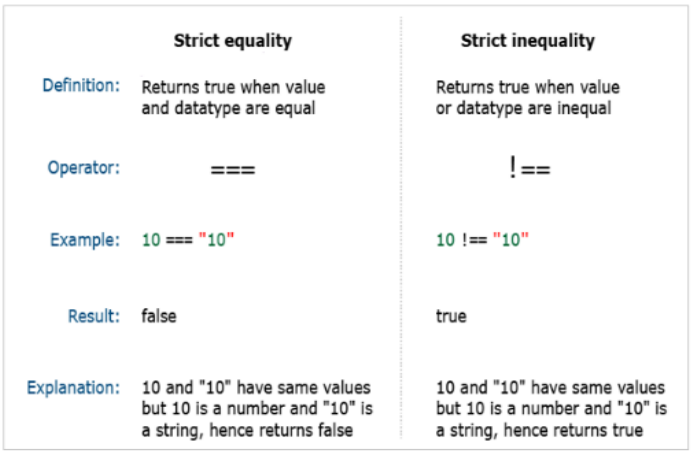
- Relational operators are used for comparing values and the result of comparison is always either true or false.
- Relational operators shown below do implicit data type conversion of one of the operands before comparison.




```
10 > 10; //false
10 >= 10; //true
10 < 10; //false
10 <= 10; //true
```

```
10 == 10; //true
10 != 10; //false
```

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



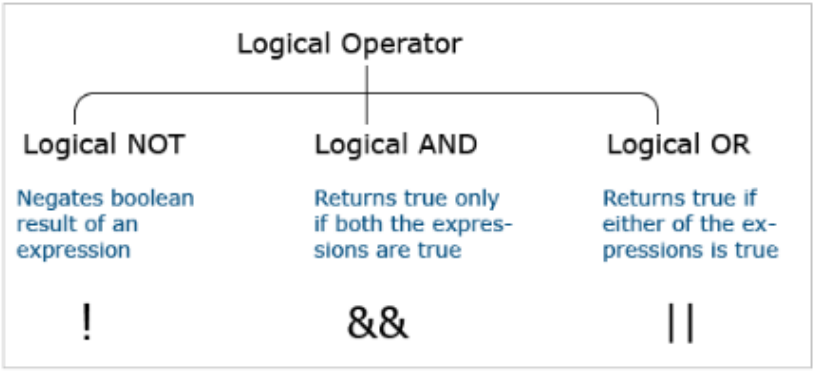
	Strict equality	Strict inequality
Definition:	Returns true when value and datatype are equal	Returns true when value or datatype are inequal
Operator:	===	!==
Example:	10 === "10"	10 !== "10"
Result:	false	true
Explanation:	10 and "10" have same values but 10 is a number and "10" is a string, hence returns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

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.




```
!(10 > 20); //true
(10 > 5) && (20 > 20); //false
(10 > 5) || (20 > 20); //true
```

typeof Operator

- "typeof" is an operator in JavaScript.
- JavaScript is a loosely typed language i.e., the type of variable is decided at runtime based on the data assigned to it. This is also called dynamic data binding.

- As programmers, if required, 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.

```
typeof "JavaScript World" //string
typeof 10.5 // number
typeof 10 > 20 //boolean
typeof undefined //undefined
typeof null //Object
typeof {itemPrice : 500} //Object
```

Statements

- 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:

```
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.

Expressions

- 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:

```
10 + 30; //Evaluates to numeric value
"Hello" + "World"; //Evaluates to string value
itemRating > 5; //Evaluates to boolean value
(age > 60): "Senior citizen": "Normal citizen";
/* Evaluates to one string value based on whether condition is true or false.
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.

Types of Non-Conditional Statements

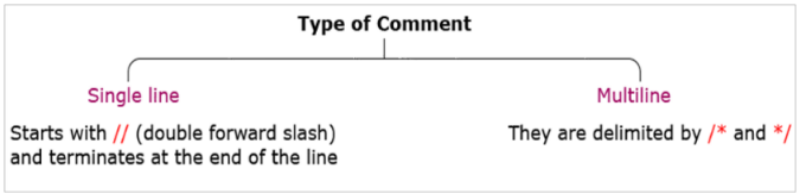
Non-Conditional statements are those statements that do not need any condition to control the program execution flow.

In JavaScript, it can be broadly classified into three categories as follows:

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:

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

Break Statement

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:

```
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.

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

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

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

Continue 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:

```
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.

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

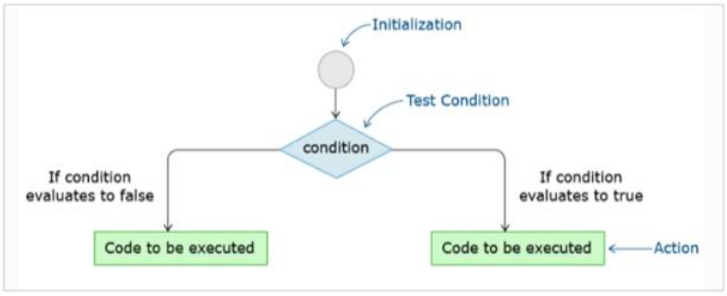
loopVar	counter
0	1
1	2
2	3
3	Iteration terminated. Hence counter is not incremented.
4	4

}

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:

Ternary operator

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

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

Example:

```
let workingHours = 9.20;
let additionalHours;
(workingHours > 9.15) ? additionalHours = "You have positive additional
hours" : additionalHours = "You have negative additional hours";
console.log(additionalHours);
```

If Statements

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:

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

Example:

```
let num1 = 12;
if (num1 % 2 == 0) {
   console.log("It is an even number!!");
}
```

// OUTPUT: It is an even number!! Because 12%2 evaluates to true

If-else

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

Syntax:

```
if (condition) {
 // block of code that will be executed, if the condition is true
}
else {
 // block of code that will be executed, if the condition is false
}
```

Example:

```
let num1 = 1;
if (num1 % 2 == 0) {
   console.log("It is an even number!!");
}
else{
   console.log("It is an odd number!!");
}
```

//OUTPUT: It is an odd number!! Because in if 1%2 evaluates to false and moves to else condition

If-else-if Ladder

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

Syntax:

```
if (condition1) {
   // block of code that will be executed if condition1 is true
}
else if (condition2) {
   // block of code that will be executed if the condition1 is false and condition2 is true
}
else {
   // block of code that will be executed if the condition1 is false and condition2 is false
}
```

Example:

```
let marks = 76;
if (marks >= 75) {
        console.log("Very Good");
}
else if (marks < 85 && marks >= 50) {
        console.log("Good");
}
else {
        console.log("Needs Improvement");
}
```

```
// 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:

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

'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.

```
var perfRating = 5;

switch (perfRating) {
   case 5:
      console.log("Very Poor");
      break;
   case 4:
      console.log("Needs Improvement");
      break;
   case 3:
      console.log("Met Expectations");
      break;
   case 2:
      console.log("Commendable");
      break;
   case 1:
      console.log("Outstanding");
      break;
   default:
      console.log("Sorry!! Invalid Rating.");
}
```

OUTPUT: Very Poor

Switch Statement - with break statement

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.

```
var perfRating = 3;

switch (perfRating) {
  case 5:
    console.log("Very Poor");
     break;
  case 4:
    console.log("Needs Improvement");
     break;
  case 3:
    console.log("Met Expectations");
     break;
  case 2:
    console.log("Commendable");
     break;
  case 1:
    console.log("Outstanding");
     break;
  default:
    console.log("Sorry!! Invalid Rating.");
}
/*
 OUTPUT:

 Met Expectation
*/
```

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.

Switch Statement - without break statement

Consider the below code snippet without break statement:

```
var perfRating = 5;

switch (perfRating) {
  case 5:
    console.log("Very Poor");
      case 4:
    console.log("Needs Improvement");
```

```
  case 3:
    console.log("Met Expectations");

  case 2:
    console.log("Commendable");

  case 1:
    console.log("Outstanding");

  default:
    console.log("Sorry!! Invalid Rating.");
}

/*
 OUTPUT:
 Very  Poor
 Needs Improvement
 Met Expectations
 Commendable
 Outstanding
 Sorry!! Invalid Rating.
*/
```

The reason for the above output is, **i**nitially 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.

Switch Statement - default statement

A scenario in which the default statement gets executed.

```
var perfRating = 15;

switch (perfRating) {
  case 5:
    console.log("Very Poor");
    break;
  case 4:
    console.log("Needs Improvement");
    break;
  case 3:
    console.log("Met Expectations");
    break;
  case 2:
    console.log("Commendable");
    break;
  case 1:
```

```
    console.log("Outstanding");
    break;
  default:
    console.log("Sorry!! Invalid Rating.");
}
 /*  OUTPUT:
  Sorry!! Invalid Rating.
 */
```

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

Loops:

Working With Loops

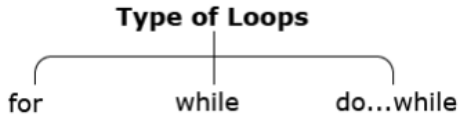
In JavaScript code, specific actions may have to be repeated a number of times.
For example, consider a variable counter which has to be incremented five times.
To achieve this, increment statement can be wiritten five times as shown below:

```
let counter = 0;
/* Same statement repeated 5 times */
counter++;
counter++;
counter++;
counter++;
counter++;
```

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

Types of Loops

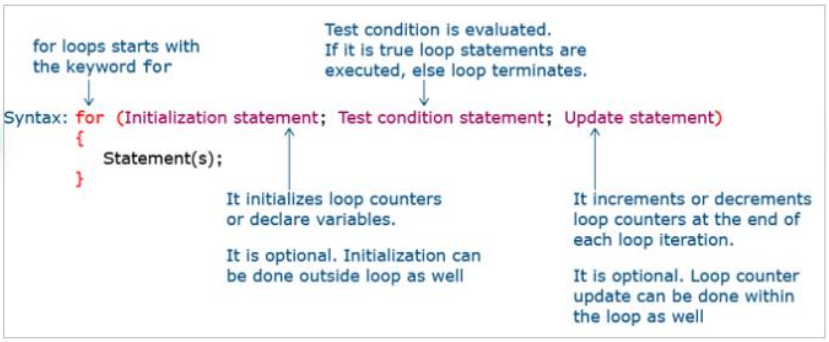
JavaScript supports popular looping statements as shown below:



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.

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

Here, in the above loop

let loopVar=0; // Initialization

loopVar < 5; // Condition

loopVar++; // Update

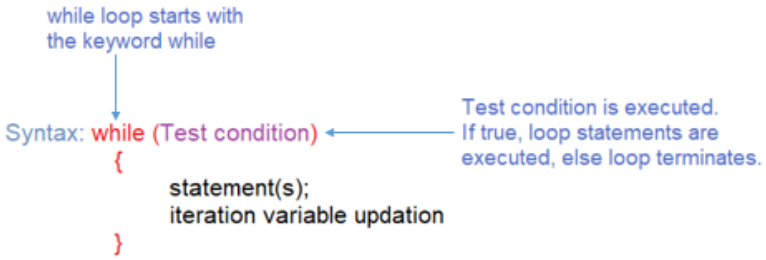
counter = counter + 1; // Action

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.

```
let counter = 0;
let loopVar = 0;
while (loopVar < 5) {
        console.log(loopVar);
        counter++;
        loopVar++;
        console.log(counter);
}
```

Here, in the above loop

let counter=0; // Initialization

let loopVar=0; // Initialization

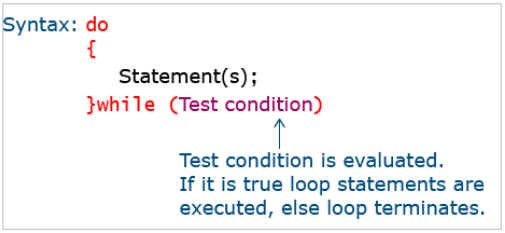
loopVar < 5; // Condition
loopVar++; // Update
counter++; // Action

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:



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.

```
let counter = 0;
let loopVar = 0;
do {
        console.log(loopVar);
        counter++;
        loopVar++;
        console.log(counter);
}
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:

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

```
}
```

Example:

```
function multiply(num1, num2) {
   return num1 * num2;
}
```

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

Syntax for Function Invocation:

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

Example:

```
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,

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

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

```
calculateCost = (ticketPrice, noOfPerson)=>{
   noOfPerson= ticketPrice * noOfPerson;
   return noOfPerson;
}
console.log(calculateCost(500, 2));
// 1000
```

Syntax 2: No parameter, single line code:

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

```
trip = () => "Let's go to trip."
```

```
console.log(trip());
// Let's go to trip.
```

Syntax 3: One parameter, single line code:

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

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

Syntax 4: One parameter, single line code:

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

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

'this' keyword in Arrow function

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:

```
const myObject = {
  items: [1],
  myMethod() {
    console.log(this == myObject) // true
    this.items.forEach(function() {
      console.log(this === myObject) // false
      console.log(this === window); // true
    });
  }
};
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:

```
const myObject = {
  items: [1],
  myMethod() {
    console.log(this == myObject) // => true
    this.items.forEach(() => {
      console.log(this === myObject) // => true
      console.log(this === window); // => false
    });
  }
};
myObject.myMethod();
```

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 matc3h 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.

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

Default Parameters

JavaScript introduces an option to assign default values in functions.

```
function multiply(num1, num2 = 1) {
   return num1 * num2;
}
console.log(multiply(5, 5)); //25
console.log(multiply(10)); //10
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 Parameters

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

Syntax:

```
function(a, …args) {
   //...
}
```

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:

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

```
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 Assignment

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:

```
let myArray = ["Andrew", "James", "Chris"];

function showDetails([arg1, arg2]) {
        console.log(arg1); // Andrew
        console.log(arg2); // James
}
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:

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

function showDetails({ name, country }) {
        console.log(name, country); // Mark India
}
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:

```
function giveMessage(message) {
        let userMsg = message;
        function toUser(userName) {
                let name = userName;
                let greet = userMsg + " " + name;
                return greet;
        }
        userMsg = toUser("Bob");
        return userMsg;
}
console.log(giveMessage("The world says hello dear: "));
```

```
// 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 clicked, the function returns "true", else returns "false".	let decision = confirm("Shall we proceed?");
prompt()	It produces a box where user can enter an input. The user input may be used for some processing later. This function takes parameter of type string which represents the label of the box.	let userInput = prompt("Please enter your name:");

isNaN()	This function checks if the data-type of given parameter is number or not. If number, it returns "false", else it returns "true".	isNaN(30); //false
		isNaN('hello'); //true

isFinite()	It determines if the number given as parameter is a finite number. If the parameter value is NaN, positive infinity, or negative infinity, this method will return false, else will return true.	isFinite(30); //true
		isFinite('hello'); //false

parseInt()	This function parses string and returns an integer number.
	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.
	The method stops parsing when it encounters a non-numerical character and returns the gathered number.
	It returns NaN when the first non-whitespace character cannot be converted to number.
		parseInt("10"); //10
		parseInt("10 20 30"); //10, only the integer part is returned
		parseInt("10 years"); //10
		parseInt("years 10"); //NaN, the first character stops the parsing

parseFloat()	This function parses string and returns a float number.
	The method stops parsing when it encounters a non-numerical character and further characters are ignored.
	It returns NaN when the first non-whitespace character cannot be converted to number.
		parseFloat("10.34"); //10.34
		parseFloat("10 20 30"); //10
		parseFloat("10.50 years"); //10.50

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.
	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.
		function executeMe(){
		console.log("Function says hello!")
		}
		setTimeout(executeMe, 3000);
		//It executes executeMe() after 3 seconds.

clearTimeout()	It cancels a timeout previously established by calling setTimeout().
	It takes the parameter "timeoutID" which is the identifier of the timeout that can be used to cancel the execution of setTimeout(). The ID is returned by the setTimeout().
		function executeMe(){
		console.log("Function says hello!")
		}
		let timerId= setTimeout(executeMe, 3000);
		clearTimeout(timerId);

setInterval()	It executes the given function repetitively.
	It takes 2 parameters, first is the function to be executed and second is the number of milliseconds. The function executes continuously after every given number of milliseconds.
		function executeMe(){
		console.log("Function says hello!");
		}
		setInterval(executeMe,3000);
		//It executes executeMe() every 3 seconds

clearInterval()	It cancels the timed, repeating execution which was previously established by a call to setInterval().
	It takes the parameter “intervalID” which is the identifier of the timeout that can be used to cancel the execution of setInterval(). The ID is returned by the setInterval().
		function executeMe(){
		console.log("Function says hello!");
		}
		let timerId=setInterval(executeMe, 2000);
		function stopInterval(){
		clearInterval(timerId);
		console.log("Function says bye to setInterval()!")
		setTimeout(stopInterval,5000)
		//It executes executeMe() every 2 seconds and after 5 seconds, further calls to executeMe() is stopped.

Variable Scope in Functions:

Scopes

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

```
//Global variable
var greet = "Hello JavaScript";

function message() {

        //Global variable accessed inside the function
        console.log("Message from inside the function: " + greet);
}

message();

//Global variable accessed outside the function
console.log("Message from outside the function: " + greet);

//Message from inside the function: Hello JavaScript
//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:

```
function message() {
        //Local variable
        var greet = "Hello JavaScript";
        //Local variables are accessible inside the function
        console.log("Message from inside the function: " + greet);
}

message();

//Local variable cannot be accessed outside the function
console.log("Message from outside the function: " + greet);

//Message from inside the function: Hello JavaScript
//Uncaught ReferenceError: greet is not defined
```

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

Example:

```
//Global variable
var firstName = "Mark";
function fullName() {
        //Variable declared without var has global scope
        lastName = "Zuckerberg";
        console.log("Full Name from inside the function: " + firstName + " " + lastName);
}
```

```
fullName();
console.log("Full Name from outside the function: " + firstName + " " + lastName);
//Full Name from inside the function: Mark Zuckerberg
//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:

```
function testVar() {
        if (10 == 10) {
                var flag = "true";
        }
        console.log(flag); //true
}
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:

```
function testVar() {
   if (10 == 10) {
      let flag = "true";
   }
   console.log(flag); //Uncaught ReferenceError: flag is not defined
}
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.

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

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

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

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

Creating 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.

```
class Calculator {
        constructor(num1, num2){  // Constructor used for initializing the class instance

                /* Properties initialized in the constructor */
                this.num1 = num1;
                this.num2 = num2;
        }

        /* Methods of the class used for performing operations */
        add() {
                return this.num1 + this.num2;
```

```
        }

        subtract() {
                return this.num1 - this.num2;
        }
}

let calculator = new Calculator(300, 100); // Creating Calculator class object or instance
console.log("Add method returns" + calculator.add()); // Add method returns 400.
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.

```
        class Calculator {
                        constructor(num1, num2) {  // Constructor used for initializing the class
instance
                                /* Properties initialized in the constructor */
                                this.num1 = num1;
                                this.num2 = num2;
                        }

                        /* static method */
                        static display() {
                                console.log("This is a calculator app");
                        }

                        /* Methods of the class used for performing operations */
                        add() {
                                return this.num1 + this.num2;
                        }

                        subtract() {
                                return this.num1 - this.num2;
                        }
                }

                /*static method display() is invoked using class name directly. */
                Calculator.display();
```

The output of the above code is :

This is a calculator app

Inheriting Classes:

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.

```
class Vehicle {
	constructor(make, model) {

		/* Base class Vehicle with constructor initializing two-member attributes */
		this.make = make;
		this.model = model;
	}
}

class Car extends Vehicle {
	constructor(make, model, regNo, fuelType) {
		super(make, model); // Sub class calling Base class Constructor
		this.regNo = regNo;
		this.fuelType = fuelType;
	}
	getDetails() {
		/* Template literals used for displaying details of Car. */
		console.log(`${this.make},${this.model},${this.regNo},${this.fuelType}`);
	}
}

let c = new Car("Hundai", "i10", "KA-016447", "Petrol"); // Creating a Car object
c.getDetails();
```

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.

```
class MyArray extends Array {
	constructor(...args) {
		super(...args);
	}
```

```
        display() {
                let strItems = "";
                for (let val of this) {
                        strItems += `${val} `;
                }
                console.log(strItems);
        }
}

let letters = new MyArray("Sam", "Jack", "Tom");
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.

Summary – Classes

In this module, you have learnt:

- What Classes are
- To create class and inherit them
- To inherit built-in classes

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.

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:

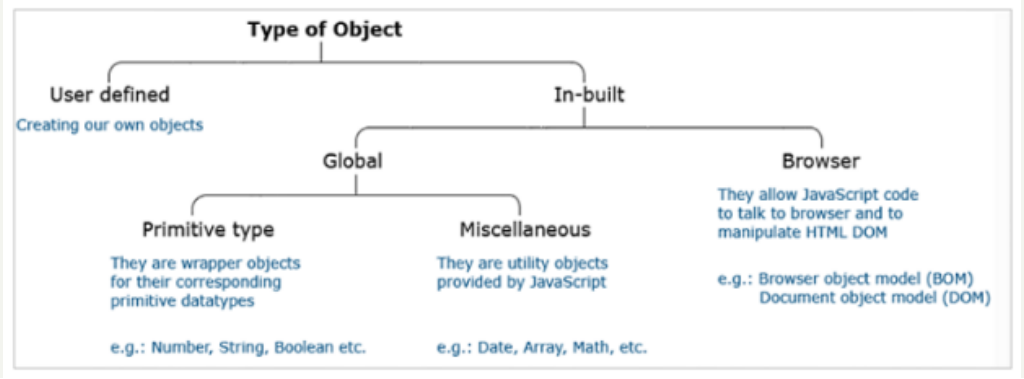
- Color=red
- Model = VXI
- Current gear = 3
- Current speed = 45 km / hr
- Number of doors = 4
- Seating Capacity = 5

The behavior of Car object:

- Accelerate
- Change gear
- Brake

Type of Objects:

JavaScript objects are categorized as follows:

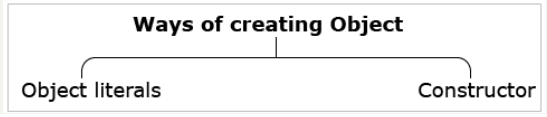


Creating Objects:

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.



Creating Object using Literal notation:

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:

```
objectName = {
   //----------states of the object----------
   key_1: value_1,
   key_2: value_2,
   ...
   key_n: value_n,
   //----------behaviour of the object---------
   key_function_name_1: function (parameter) {
      //we can modify any of the property declared above
   },
   ...
   key_function_name_n: function(parameter) {
      //we can modify any of the property declared above
   }
}
```

Example:

```
//----------states of the object---------
let myCar = {
   name: "Fiat",
   model: "VXI",
   color: "red",
   numberOfGears: 5,
   currentGear: 3,
   currentSpeed: 45,
   //----------Behaviour of the object---------
   accelerate: function (speedCounter) {
      this.currentSpeed = this.currentSpeed + speedCounter;
      return this.currentSpeed;
   },

   brake: function (speedCounter) {
      this.currentSpeed = this.currentSpeed - speedCounter;
      return this.currentSpeed;
   }
}
```

Creating Object using Enhanced Object Literals:

Below is the older syntax used to create object literals:

```
let name = "Arnold";
let age = 65;
```

```
let country = "USA";
let obj = {
   name: name,
   age: age,
   country: country
};
```

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

```
let name="Arnold";
let age=65;
let country="USA";

let obj={name,age,country};
```

Creating Object using Enhanced Object Literals - Property Shorthand

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

```
//Literal property without shorthand
function createCourse(name, status) {
   return {type: "JavaScript", name: name, status: status};
}

function reviewCourse(name) {
   return {type: "JavaScript", name: name};
}


/*Literal property with shorthand
when the property and the value identifiers have the same name,
the identifier can be omitted to make it implicit*/

function createCourse(name, status) {
   return {type: "JavaScript", name, status};
}

function reviewCourse(name) {
   return {type: "JavaScript", name};
}
```

Creating Object using Enhanced Object Literals - Computed Property:

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

```
let personalDetails = {
   name: "Stian Kirkeberg",
   country: "Norway"
};
```

```
let dynamicProperty = "age";
personalDetails[dynamicProperty] = 45;
console.log(personalDetails.age); //Output: 45
```

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.

```
let dynamicProperty = "age";
let personalDetails = {
   name: "Stian Kirkeberg",
   country: "Norway",
   [dynamicProperty]: 45
};
console.log(personalDetails.age);  //Output: 45
```

Creating Object using Function Constructor:

To construct multiple objects with the same set of properties and methods, function constructor can be used. Function constructor is like regular functions but it is invoked using a 'new' keyword.

Example:

```
function Car(name, model, color, numberOfGears, currentSpeed, currentGear) {
  //----------States of the object---------
  this.name = name;
  this.model = model;
  this.color = color;
  this.numberOfGears = numberOfGears;
  this.currentSpeed = currentSpeed;
  this.currentGear = currentGear;

  //----------Behaviour of the object---------
  this.accelerate = function (speedCounter) {
    this.currentSpeed = this.currentSpeed + speedCounter;
    return this.currentSpeed;
  }

  this.brake = function (speedCounter) {
    this.currentSpeed = this.currentSpeed - speedCounter;
    return this.currentSpeed;
  }
}
```

'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:

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

Retrieving behavior using the dot operator:

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

Retrieving state using the bracket operator:

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

Retrieving behavior using the bracket operator:

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

Combining and cloning objects using Spread operator:

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:

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

Example:

```
let candidateSelected={
        Name:'Rexha Bebe',
        Qualification:'Graduation',
};
let SelectedAs={
        jobTitle:'System Engineer',
        location:'Bangalore'
};
let employeeInfo={
        ...candidateSelected,
        ...SelectedAs
};
console.log(employeeInfo);
```

```
/*
{
  Name: 'Rexha Bebe',
  Qualification: 'Graduation',
  jobTitle: 'System Engineer',
  location: 'Bangalore'
}
*/
```

Cloning of Objects using Spread Operator:

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

Syntax:

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

Example:

```
let originalObj = { one: 1, two: 2, three: 3 };
let clonedObj = { ...originalObj };

/*
Here spreading the object into a list of parameters happens
which return the result as a new object
checking whether the objects hold the same contents or not
*/
alert(JSON.stringify(originalObj) === JSON.stringify(clonedObj)); // true

//checking whether both the objects are equal

alert(originalObj === clonedObj); // false (not same reference)

//to show that modifying the original object does not alter the copy made

originalObj.four = 4;
alert(JSON.stringify(originalObj)); // {"one":1,"two":2,"three":3,"four":4}
alert(JSON.stringify(clonedObj)); // {"one":1,"two":2,"three":3}
```

Destructuring objects:

Destructuring gives a syntax that makes it easy to create objects based on variables.

It also helps to extract data from an object. Destructuring works even with the rest and spread operators.

In the below example an object is destructured into individual variables:

```
let myObject = { name: 'Arnold', age: 65, country: 'USA' };
let { name, age:currentAge } = myObject; //alias can be used with :
console.log(name);
console.log(currentAge);

//OUTPUT: Arnold 65
```

An alias currentAge is used for age

Object destructuring in functions

```
let myObject = { name: 'Marty', age: 65, country: 'California' };
function showDetails({ country }) {
  console.log(country);
}
showDetails(myObject);  //invoke the function using the object


//OUTPUT: California
```

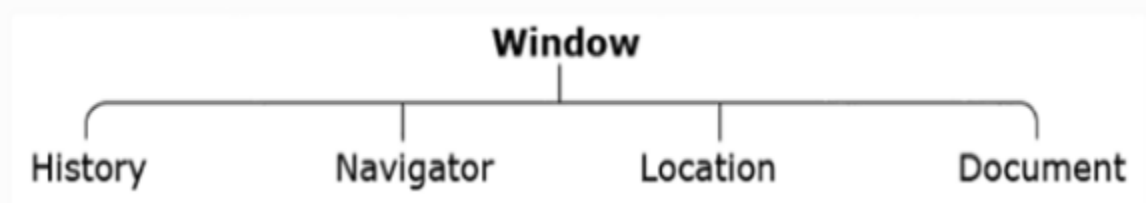
The property 'country' of the object has been destructured and captured as a function parameter.

Browser Object Model:

As you know that, JavaScript is capable of dynamically manipulating the content and style of HTML elements of the web page currently rendered on the browser. The content given for para during HTML creation or the style given for heading during HTML creation can be changed even after the page has arrived on the browser.

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

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



Document Object :

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

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

W3C provides DOM API consisting of properties and methods that help in traversal and manipulation of the HTML page.

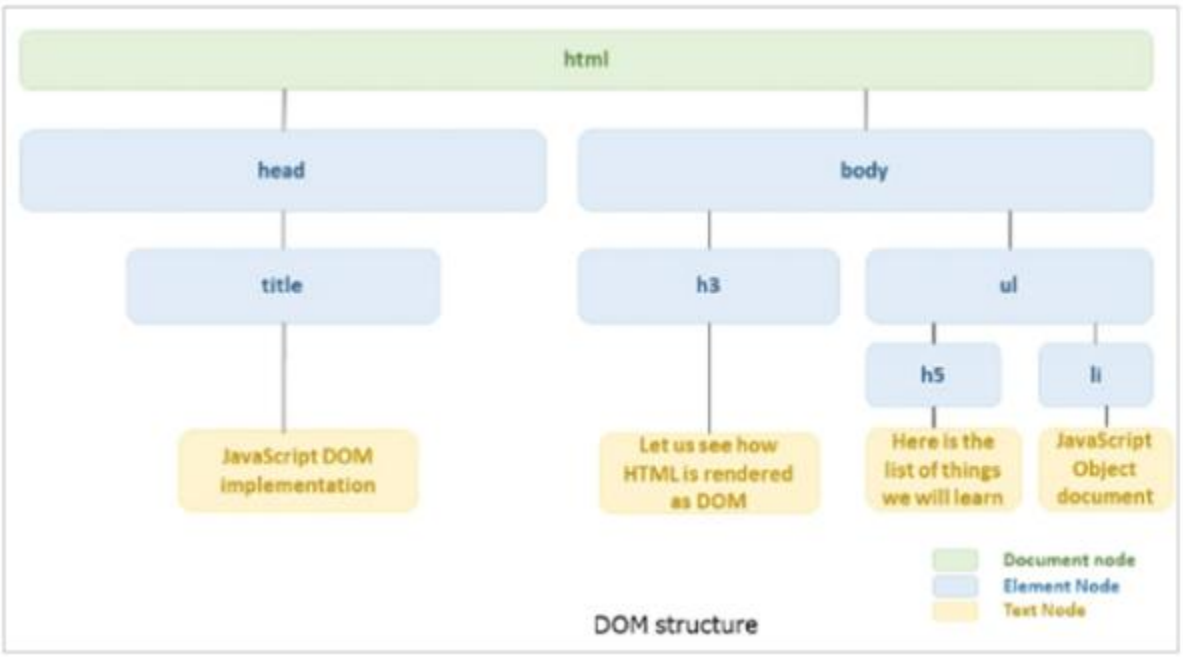
Shown below is the HTML web page and it's corresponding DOM structure that can be accessed using DOM API methods and properties:

Sample HTML Code:

```
<html>
<head>
  <title>JavaScript DOM Implementation</title>
</head>
<body>
  <h3>Let us see how HTML is rendered as DOM</h3>
```

```
  <ul>
    <h5>Here is the list of things we will learn</h5>
    <li>JavaScript Object Document</li>
  </ul>
</body>
</html>
```

DOM Structure:



There are certain methods and properties that allow to traverse the DOM tree and manipulate content or style for the specified node representing the HTML element.

Document Object - Methods:

To access an element in the HTML page, following methods can be used on the 'document' object from DOM.

getElementById(x)

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

Example:

```
<p id="p1"> Paragraph 1</p>
<p> Paragraph 2</p>
<script>
   //Selects paragraph having id 'p1'
   document.getElementById('p1');
</script>
```

getElementsByTagName(x)

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

Example:

```
<p id="p1">Paragraph 1</p>
<p>Paragraph 2</p>
<script>
```

```
    document.getElementsByTagName('p');
</script>
//OUTPUT:
//Paragraph 1

//Paragraph 2
```

getElementsByClassName()

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

Example:

```
<p class="myClass">Paragraph 1</p>
<p>Paragraph 2</p>
<script>
    //Selects paragraph having class = "myClass"
    var x = document.getElementsByClassName('myClass');

</script>
```

querySelectorAll()

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

Example:

```
<p class="blue">Paragraph 1</p>
<script>
    var x = document.querySelectorAll("p.blue");
    x[0].innerHTML;
</script>
```

```
//OUTPUT: Paragraph 1
```

Document Object - Properties:

Some of the other properties of the 'document' object to access the HTML element are:

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

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

innerHTML

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

Example:

```
<div id="div1">
    <h1 id="heading1">Welcome to JavaScript Tutorial</h1>
        <p id="para1" style="color: blue;">Let us learn DOM API</p>
</div>
<script>
    //retieves current content
```

```
    document.getElementById("heading1").innerHTML;
    //sets new content
    document.getElementById.innerHTML = "Heading generated dynamically"
</script>
```

attribute

It is used to set new values to given attributes.

Example:

```
<head>
<title>JavaScript DOM Implementation</title>
  <style>
    .div2 {
      color: yellow;
    }
  </style>
</head>
<body>
  <div id="div1">
    <h1 id="heading1">Welcome to JavaScript Tutorial</h1>
    <p id="para1" style="color: blue;">Let us learn DOM API</p>
  </div>
  <script>
    document.getElementById("div1").attributes[0].value;
    document.getElementById("div1").setAttribute('class', 'div2');
  </script>
</body>
```

Output:

Initially, no color was applied to the heading, later the class ‘div2’ was added using setAttribute.

To manipulate the style of an HTML element, the following property of the 'element' object given by DOM API can be used:

style

It gives access to the style attribute of the HTML element and allows it to manipulate the CSS modifications dynamically.

Example:

```
<div id="div1">
    <h1 id="heading1">Welcome to JavaScript Tutorial</h1>
    <p id="para1" style="color: blue;">Let us learn DOM API</p>
</div>
  <script>
    //resets style property
    document.getElementById("div1").style.color = "red";
```

```
</script>
```

Window Object :

So far, you know how the content and style for a given HTML page can be modified using the BOM model's object 'document'.

Suppose it is not required to update the HTML page but only certain properties of the browser window on which it is rendered. That is to navigate to a different URL and display a new web page, or close the web page or store some data related to the web page. Well, to implement this, an object that represents the entire browser window and allows us to access and manipulate the window properties is required. BOM model provides that 'window' object.

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

Window Object - Properties

innerHeight

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

Example:

```
let inHeight = window.innerHeight;
console.log(" Inner height: " + inHeight);
//Returns Inner height: 402
```

innerWidth

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

Example:

```
let inWidth = window.innerWidth;
console.log(" Inner width: " + inWidth);
//Returns Inner width: 1366
```

outerHeight

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

Example:

```
let outHeight = window.outerHeight;
console.log(" Outer Height: " + outHeight);
//Returns Outer height: 728
```

outerWidth

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

Example:

```
let outWidth = window.outerWidth;
console.log("Outer width of window: " + outWidth);
//Returns Outer width: 1366
```

Window Object - Methods

localStorage

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

Example:

```
localStorage.setItem('username','Bob');
console.log("Item stored in localStorage is" + localStorage.getItem('username'));
```

```
//Returns Item stored in localStorage is Bob
```

sessionStorage

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

Example:

```
sessionStorage.setItem('password', 'Bob@123');
console.log("Item stored in sessionStorage is " + sessionStorage.getItem('password'));
//Returns Item stored in sessionStorage is Bob@123
```

Methods

In addition to these methods, 'window' object gives us a few more methods that are helpful in the following way:

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

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

History Object:

If required, BOM also gives a specific object to target only one of the window properties. For example, if the concern is about the list of URLs that have been visited by the user and there is no need for any other information about the browser, BOM gives the 'history' object for this. It provides programmatic navigation to one of the URLs previously visited by the user. Following are the properties or methods that helps in doing so.

Property:

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

Methods:

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

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

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

Navigation Object:

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

appName

Returns the name of the client.

Example:

```
navigator.appName;
//Browser's name: Netscape
```

appVersion

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

Example:

```
console.log(navigator.appVersion);
//5.0 (Windows NT 10.0; Win64; x64)
//AppleWebKit/537.36 (KHTML, like Gecko)
//Chrome/83.0.4103.106 Safari/537.36
```

Platform

Returns the name of the user's operating system.

Example:

```
console.log(navigator.platform);
```

```
//Browser's platform: Win 32
```

userAgent

Returns string equivalent to HTTP user-agent request header.

Example:

```
console.log(navigator.userAgent);
//Browser's useragent: Mozilla/5.0 5.0 (Windows NT 6.1; WOW64)
//AppleWebKit/537.36 (KHTML, like Gecko)
//Chrome/53.0.2785.116 Safari/537.36
```

The output shown above is for the Chrome browser running on Windows.

Location Object:

So far, you learnt about different objects in the BOM hierarchy for accessing the history of URLs visited by the user or to know the properties of the browser. However, which object should be used to programmatically refresh the current page or navigate to a new page?

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

If this is the URL: http://localhost:8080/JS_Demos/myLocationFile.html, properties have the following interpretation:

href

It contains the entire URL as a string.

Example:

```
console.log(location.href);
//Returns http://localhost:8080/JS_Demos/myLocationFile.html
```

hostname

It contains the hostname part of the URL.

Example:

```
console.log(location.hostname);
//Returns localhost
```

port

It contains a port number associated with the URL.

Example:

```
console.log(location.port)
//Returns 8080
```

pathname

It contains a filename or path specified by the object.

Example:

```
console.log(location.pathname);
//Returns /JS_Demos/myLocationFile.html
```

'location' object gives the following methods to reload the current page or to navigate to a new page:

assign()

Loads new HTML document.

Example:

```
location.assign('http://www.facebook.com');
//Opens facebook page
```

reload()

Reloads current HTML.

Example:

```
location.reload();
//Current document is reloaded
```

Document Object Model:

DOM Nodes:

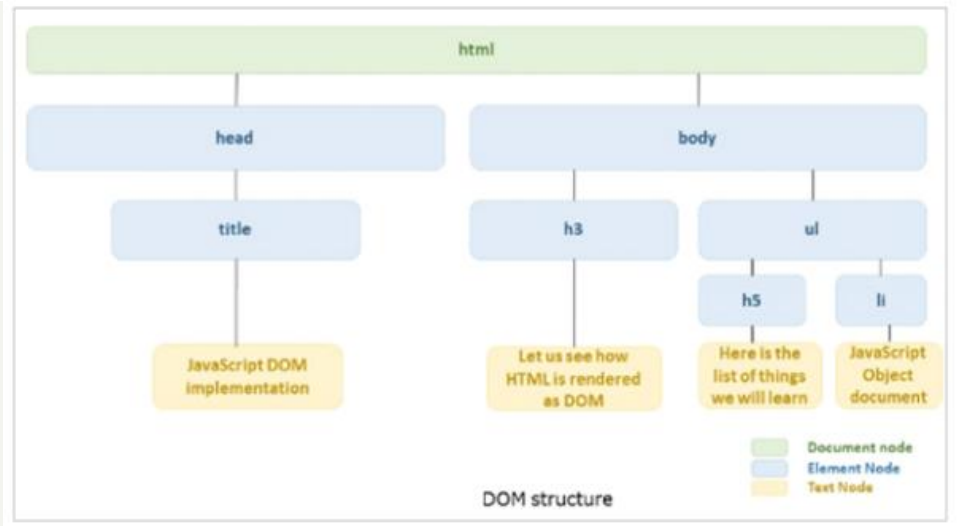
You know how the BOM hierarchy consisting of numerous built-in objects allows to dynamically manipulate the given web page on the client-side. Also, you are aware that the HTML page is considered as the DOM tree by the browser with every HTML element having a hierarchical relationship with each other.

There is one more kind of manipulation that can be achieved on the DOM tree. HTML elements can be dynamically added or removed. Also, the elements can be accessed or modified by referring to the relationship of the target HTML element with the element that can already be accessed.

According to the W3C DOM standard, each HTML element can be referred to as a Node. For example, the entire the HTML document is a 'document node', every other element inside HTML is 'element node'. The content inside these HTML elements is a 'text node'.

```
<html>
<head>
  <title>JavaScript DOM Implementation</title>
</head>
<body>
  <h3>Let us see how HTML is rendered as DOM</h3>
  <ul>
    <h5>Here is the list of things we will learn</h5>
    <li>JavaScript Object Document</li>
  </ul>
</body>
</html>
```

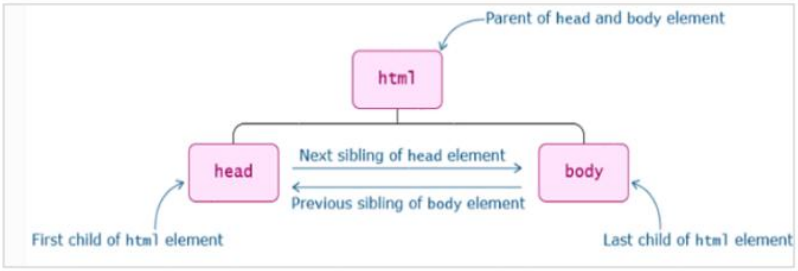
Corresponding DOM structure :



So, how does the Node relationship helps in Node manipulation?

These nodes appear in a hierarchical structure inside the browser. And this hierarchical relationship between the nodes allows us to traverse through the DOM tree.

- The top node is called the root. It does not have any parents.
- Every other node in the tree belongs to one parent.
- Every node may have several children.
- Nodes with the same parent are referred to as siblings.



DOM API Properties:

parentNode

Returns a Node object that is the parent node of the specified node. This property can be retrieved and cannot set it.

Example:

```
<html>
<head></head>
<body>
   <script>
      //Returns Node onject<html>
      document.body.parentNode
   </script>
</body>
</html>
```

childNodes

Returns NodeList object, i.e collection of child nodes of the specified node. Each child can be accessed by an index number that refers to its position inside the parent element. The first position is at index '0'.

Example:

```
<html>
<head></head>
<body>
   <script>
      //Returns NodeList object consisting of nodes: h1 and p
      document.body.childNodes
   </script>
   <h1></h1>
   <p></p>
</body>
</html>
```

firstChild

Returns Node object which is the first child of the specified node. Its is equivalent to childNodes[0].

Example:

```
<script>
   //Returns h1 element
   document.getElementById("div1").firstChild;
</script>
<div id="div1"><h1></h1><p></p><div>
```

Note: Whitespace inside elements is considered as text, and text is considered as nodes.

lastChild

Returns Node object which is the last child of the specified node.

Example:

```
<div id="div1"><h1></h1><p></p></div>
<script>
   document.getElementById("div1").lastChild; //Returns p element
</script>
```

Note: Whitespace inside elements is considered as text, and text is considered as nodes.

nextSibling returns the Node object of the node that immediately follows the specified node at the same tree level.

Example:

```
 <div id="div1">
      <h1 id="heading1"></h1><p id="para1"></p>
  </div>
    <script>
      let elem=document.getElementById("heading1").nextSibling;
      console.log(elem);
   </script>
//Returns p element
```

previousSibling

Returns the Node object of the node that the previous node of the specified node at the same tree level.

Example:

```
<div id="div1">
  <h1 id="heading1"></h1><p id="para1"></p>
  <script>
    //Returns h1 element
      console.log(document.getElementById("para1").previousSibling;);
  </script>
</div>
```

Note: Whitespace inside elements is considered as text, and text is considered as nodes.

Please note:

Similar to all these properties of Node object, you also have properties such as parentElement, firstElementChild, lastElementChild, nextElementSibling and previousElementSibling.

The difference is that element properties return only the Element object whereas node properties return element, text, and attribute nodes with respect to the specified node. Also, whitespaces are considered as '#text' nodes by the node properties.

Node Manipulation:

The node relationship allows to modify the tree of nodes by adding new nodes and removing the existing nodes if required.

For the given HTML page, below methods will do the following:

- Create a new element
- Create new content
- Add new content to the new element
- Add a new element to the existing DOM tree

HTML code:

```
<div id="div1">
  <h1 id="heading1">Hello World</h1>
  <p id="para1">Good luck!!</p>
</div>
<br>
<input type="button" value="Add span" onclick="createNew()">
<input type="button" value="Remove para" onclick="removeOld()">
```

Node Manipulation Methods:

createElement()

Creates a new element.

Example:

```
let newElement = document.createElement('span');
```

createTextNode()

Creates content at runtime. This node then can be appended to any node that can hold content.

Example:

```
let newTextElement = document.createTextNode('The span is added just now');
```

appendChild()

Appends a newly created element to the existing DOM tree at the desired position.

Example:

```
newElement.appendChild(newTextElement);
document.getElementById('div1').appendChild(newElement);
```

removeChild()

Removes the element from the existing DOM tree.

Example:

```
document.getElementById('div1').removeChild(document.getElementById('para1'));
```

DOM Event Handling:

You are aware of event handlers in JavaScript which are invoked using inline scripting. This approach has a limitation of the tight coupling of the script code with the HTML element.

DOM API provides couple of ways to handle events in JavaScript using internal or external scripting. It segregates HTML elements completely from any JavaScript code. There are two different ways of doing this.

Suppose to listen to the click event on the HTML paragraph elements, given are two different ways of doing it.

Example:

HTML Code:

```
<p id="para1"> Para one of my page</p>
<p id="para2"> Para two of my page</p>
```

JS Code:

```
document.getElementById('para1').onclick=function(){
    alert('Para one clicked');
}

//OR

document.getElementById('para2').addEventListener('click', function(){
    alert('Para two clicked');}
,false);
```

DOM Event Objects:

Events in JavaScript are considered as objects.

When events are fired, the 'event' object is generated by the browser. This object encapsulates all data related to that event.

To access or manipulate this object, it can optionally be passed as the first argument to the event handler function.

The properties of this object are as follows:

- target

- type

target Event Property:

Refers to the HTML element that fired the event.

Example:

```
<p id="para1" onclick="executeMe(event)"> Para one of my page</p>
<script>
   function executeMe(event) {
         alert(event.target.nodeName)
      }
</script>
//alert box shows P ID="PARA1"
```

type Event Property:

Tells the type of events that have taken place like click, load, etc.

Example:

```
<p id="para1" onclick="executeMe(event)"> Para one of my page</p>
<script>
  function executeMe(event) {
         alert(event.type)
      }
</script>
//alert box shows click
```

Methods:

preventDefault()

Cancels default action associated with HTML element and adds user-defined action (if required).

For example, an element's default action is to navigate to the given link. That action can be cancelled and some other action can be done instead.

Usage:

```
event.preventDefault();
```

Arrays :

Objects in JavaScript is a collection of properties stored as key-value pair.

Often, there is requirement for an ordered collection, where there are 1st, 2nd, 3rd element, and so on. For example, you need to store a list of students in a class based on their roll numbers

It is not convenient to use an object here, because objects don't store the values in an ordered fashion. Also, a new property or element cannot be inserted "between" the existing ones.

This is why Array is used to store values in order.

Array in JavaScript is an object that allows storing multiple values in a single variable. An array can store values of any datatype. An array's length can change at any time, and data can be stored at non-contiguous locations in the array,

Example:

```
let numArr = [1, 2, 3, 4];
let empArr = ["Johnson", 105678, "Chicago"];
```

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

Example:

```
let numArr = [1, 2, 3, 4];
console.log(numArr[0]); //1
console.log(numArr[3]); //4
```

Creating Arrays:

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

Array Literal Notation:

Arrays are created using literal notation almost all the time.

Syntax:

```
let myArray = [element 1, element2,…, element N];
```

Example:

```
let colors = ["Red", "Orange", "Green"]
```

Array Constructor:

Arrays can be created using the Array constructor with a single parameter which denotes the array length. The parameter should be an integer between 0 and 232-1 (inclusive). This creates empty slots for the array elements. If the argument is any other number, a RangeError exception is thrown.

Syntax:

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

Example:

```
let colors = new Array(2);
console.log(colors.length); //2

//Assign values to an empty array using indexes
colors[0] = "Red";
colors[1] = "Green";
console.log(colors); //['Red','Green']
```

If more than one argument is passed to the Array constructor, a new Array with the given elements is created.

Syntax:

```
let myArray = new Array(element 1, element 2,…,element N);
```

Example:

```
let colors = new Array("Red", "Orange", "Green");
```

Destructuring arrays:

JavaScript introduced the destructuring assignment syntax that makes it possible to unpack values from arrays or objects into distinct variables. So, how does this syntax help to unpack values from an array.

Example:

```
// [RN1] we have an array with the employee name and id
let empArr = ["Shaan", 104567];
// destructuring assignment
// sets empName = empArr[0]
// and empId = empArr[1]
let [empName, empId] = empArr;
console.log(empName); // Shaan
console.log(empId);  // 104567
```

Destructuring assignment syntax is just a shorter way to write:

```
let empName = empArr[0];
let empId = empArr[1];
```

You can also ignore elements of the array using an extra comma.

Example:

```
let [empName, , location] = ["Shaan", 104567, "Bangalore"];

//Here second element of array is skipped and third element is assigned to location variable
console.log(empName); // Shaan
console.log(location);  // Bangalore
```

Rest operator can also be used with destructuring assignment syntax.

Example:

```
let [empName, ...rest] = ["Shaan", 104567, "Bangalore"];
console.log(empName); // Shaan
console.log(rest);  // [104567,'Bangalore']
```

Here, the value of the rest variable is the array of remaining elements and the rest parameter always goes last in the destructuring assignment.

Accessing arrays:

Array elements can be accessed using indexes. The first element of an array is at index 0 and the last element is at the index equal to the number of array elements – 1. Using an invalid index value returns undefined.

Example:

```
let arr = ["first", "second", "third"];
console.log(arr[0]); //first
console.log(arr[1]); //second
console.log(arr[3]); //undefined
```

Loop over an array

You can loop over the array elements using indexes.

Example:

```
let colors = ["Red", "Orange", "Green"];
for (let i = 0; i < colors.length; i++) {
   console.log(colors[i]);
```

```
}
//Red
//Orange
//Green
```

JavaScript also provides for..of statement to iterate over an array.

Example:

```
let colors = ["Red", "Orange", "Green"];

// iterates over array elements
for (let color of colors) {
   console.log(color);
}

//Red
//Orange
//Green
```

Array Methods:

Array Property:

JavaScript arrays consist of several useful methods and properties to modify or access the user-defined array declaration.

Below is the table with property of JavaScript array:

Property	Description	Example

Length	It is a read-only property. It returns the length of an array, i.e., number of elements in an array	let myArray = ["Windows", "iOS", "Android"];
		console.log("Length = " + myArray.length);
		//Length = 3

Array Methods:

Below is the table with methods to add/remove array elements:

Methods	Description	Example

push()	Adds new element to the end of an array and return the new length of the array.	let myArray = ["Android", "iOS", "Windows"];
		myArray.push("Linux");
		console.log(myArray);
		// ["Android","iOS","Windows","Linux"]

pop()	Removes the last element of an array and returns that element.	let myArray = ["Android", "iOS", "Windows"];
		console.log(myArray.pop()); // Windows
		console.log(myArray); // ["Android","iOS"]

shift()	Removes the first element of an array and returns that element.	let myArray = ["Android", "iOS", "Windows"];
		console.log(myArray.shift()); //Android
		console.log(myArray); //["iOS", "Windows"]

unshift()	Adds new element to the beginning of an array and returns the new length	let myArray = ["Android", "iOS", "Windows"]; myArray.unshift("Linux");

		console.log(myArray); //["Linux","Android","iOS","Windows"]

splice()	Change the content of an array by inserting, removing, and replacing elements. Returns the array of removed elements.
	Syntax:
	array.splice(index,deleteCount,items);
	index = index for new item
	deleteCount = number of items to be removed, starting from index next to index of new item
	items = items to be added
		let myArray = ["Android", "iOS", "Windows"];
		//inserts at index 1
		myArray.splice(1, 0, "Linux");
		console.log(myArray);
		// ["Android","Linux", "iOS", "Windows"]

slice()	Returns a new array object copying to it all items from start to end(exclusive) where start and end represents the index of items in an array. The original array remains unaffected
	Syntax:
	array.slice(start,end)
		let myArray=["Android","iOS","Windows"];
		console.log(myArray.slice(1,3));
		// ["iOS", "Windows"]

concat()	Joins two or more arrays and returns joined array.	let myArray1 = ["Android","iOS"];
		let myArray2 = ["Samsung", "Apple"];
		console.log(myArray1.concat(myArray2));
		//["Android", "iOS", "Samsung", "Apple"]

Below is the table with methods to search among array elements:

Methods	**Description**	**Example**

indexOf()	Returns the index for the first occurrence of an element in an array and -1 if it is not present	let myArray = ["Android","iOS","Windows","Linux"];
		console.log(myArray.indexOf("iOS")); // 1
		console.log(myArray.indexOf("Samsung"));
		
		//-1

find()	Returns the value of the first element in an array that passes a condition specified in the callback function.
	Else, returns undefined if no element passed the test condition.
	Syntax:
	array.find(callback(item,index,array))
	callback is a function to execute on each element of the array
	item value represents the current element in the array
	index value indicates index of the current element
		let myArray = ["Android", "iOS", "Windows", "Linux"];
		let result = myArray.find(element => element.length > 5);
		console.log(result); //Android

	of the array array value represents array on which find() is used, index and array are optional	

findIndex()	Returns the index of the first element in an array that passes a condition specified in the callback function. Returns -1 if no element passes the condition.
	Syntax:
	Array.findIndex(callback(item,index,array));
	callback is a function to execute on each element of the array
	item value represents current element in the array
	index represents index of the current element of the array
	array represents array on which findIndex() is used.
	index and array are optional
		let myArray = ["Android", "iOS", "Windows", "Linux"];
		let result = myArray.findIndex(element => element.length > 5);
		console.log(result) //0

filter()	Creates a new array with elements that passes the test provided as a function.
	Syntax:
	array.filter(callback(item,index,array))
	callback is the Function to test each element of an array
	item value represents the current element of the array
	index value represents Index of current element of the array
	array value indicates array on which filter() is used.
		let myArray = ["Android", "iOS", "Windows", "Linux"];
		let result = myArray.filter(element => element.length > 5);
		console.log(result)
		//["Android","Windows"]

Below is the table with methods to iterate over array elements:

Method	**Description**	**Example**

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

	array represents the array on which forEach() is called	

Below is the table with methods to transform an array:

Methods	Description	Example

map()	Creates a new array from the results of the calling function for every element in the array.
	Syntax:
	array.map(callback(item,index,array))
	callback is a function to be run for each element in the array
	item represents the current element of the array
	index value represents index of the current element of the array
	array value represents array on which forEach() is invoked
		let numArr = [2, 4, 6, 8];
		let result = numArr.map(num=>num/2);
		console.log(result);
		//[1, 2, 3, 4]

join()	Returns a new string by concatenating all the elements of the array, separated by a specified operator such as comma. Default separator is comma	let myArray = ["Android", "iOS", "Windows"];
		console.log(myArray.join());
		// Android,iOS,Windows
		console.log(myArray.join('-'));
		// Android-iOS-Windows

reduce()	Executes a defined function for each element of passed array and returns a single value Syntax: array.reduce(callback(accumulator, currentValue, index,array),initialValue) callback is a function to be executed on every element of the array accumulator is the initialValue or previously returned value from the function. currentValue represents the current element of the passed array index represents index value of the current element of the passed array array represents the array on which this method can be invoked. initialValue represents the Value that can be passed to the function as an initial value. currentValue,index,array and initialValue are optional.	const numArr = [1, 2, 3, 4]; // 1 + 2 + 3 + 4 console.log(numArr.reduce((accumulator, currentVal) => accumulator + currentVal)); // 10 // 5 + 1 + 2 + 3 + 4 console.log(numArr.reduce((accumulator, currentVal) => accumulator + currentVal,5)); // 15

Introduction to Asynchronous Programming:

Consider below-given code snippet:

```
console.log("Before For loop execution");
for (var i = 0; i < 2; i++) {
   console.log("setTimeout message");
   func1();
   func2();
}
console.log("After For loop execution");

function func1() {
   console.log("Am in func1");
}
function func2() {
   console.log("Am in func2");
}
```

According to JavaScript sequentially execution nature, the output of the above code snippet would be as shown below:

```
Before For loop execution
setTimeout message
Am in func1
Am in func2
setTimeout message
Am in func1
Am in func2
After For loop execution
```

If previous code is modified by adding setTimeout() method in for loop as shown below, then observe the output once again.

Modified code snippet:

```
for (var i = 0; i < 2; i++) {
   setTimeout(function() {
      console.log("setTimeout message");
      func1();
   }, );
   func2();
}
```

New Output:

```
Before For loop execution
Am in func2
After For loop execution
setTimeout message
Am in func1
setTimeout message
Am in func1
>
```

As observed in the output above, due to usage of setTimeout() method the entire execution of code behavior has been changed, and the code has been executed asynchronously.

Asynchronous Programming Techniques:

Some of the real-time situations where you may need to use the JavaScript Asynchronous code of execution while implementing business logic are:

- To make an HTTP request call.
- To perform any input/output operations.
- To deal with client and server communication.

These executions in JavaScript can also be achieved through many techniques.

Some of the techniques are:

- Callbacks
- Promises
- Async/Await

Callbacks:

A callback function is a function that is passed as an argument to another function. Callbacks make sure that a certain function does not execute until another function has already finished execution.

Callbacks are handy in case if there is a requirement to inform the executing function on what next when the asynchronous task completes. Here the problem is there are bunch of asynchronous tasks, which expect you to define one callback within another callback and so on. This leads to callback hell.

Callback hell, which is also called a Pyramid of Doom, consists of more than one nested callback which makes code hard to read and debug. As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if there are more loops, conditional statements, and so on in the code.

Example:

```
myFun1(function () {
   myFun2(function () {
      myFun3(function () {
         myFun4(function () {
            ....
         });
      });
   });
});
```

In the above example, it is noticed that the "pyramid" of nested calls grows to the right with every asynchronous action. It leads to callback hell. So, this way of coding is not very good practice.

To overcome the disadvantage of callbacks, the concept of Promises was introduced.

Promises:

In JavaScript, promises are a way to handle asynchronous operations. A promise is an object that represents a value that may not be available yet, but will be resolved at some point in the future. Promises are commonly used for network requests, file I/O, and other time-consuming operations.

Promises have three states:

Pending: The initial state, before the promise has resolved or rejected.

Fulfilled: The state when the promise has successfully resolved with a value.

Rejected: The state when the promise has failed to resolve with an error.
Promises can be created using the Promise constructor, which takes a function as its argument. This function takes two parameters: resolve and reject. The resolve function is used to fulfill the promise, and the reject function is used to reject the promise with an error.
Here is an example of creating a promise:

```
const promise = new Promise((resolve, reject) => {
  // Perform some asynchronous operation
  // When it's done, call either `resolve` or `reject`
  if (/* operation was successful */) {
    resolve('result');
  } else {
    reject(new Error('Something went wrong'));
  }
});
```

Once a promise is created, you can attach callbacks using the then and catch methods. The then method is called when the promise is fulfilled, and the catch method is called when the promise is rejected.
Here is an example of using then and catch:

```
promise.then((result) => {
  console.log(result); // output: 'result'
}).catch((error) => {
  console.error(error); // output: 'Error: Something went wrong'
});
```

Promises can also be chained using the then method. This allows you to perform a series of asynchronous operations in a specific order.
Here is an example of chaining promises:

```
promise.then((result) => {
  return doSomethingElse(result);
}).then((result) => {
  console.log(result); // output: 'result from second operation'
}).catch((error) => {
  console.error(error);
});
```

Examples:

```
// Nested callbacks using setTimeout
setTimeout(function() {
  console.log('First timeout executed');
  setTimeout(function() {
    console.log('Second timeout executed');
    setTimeout(function() {
      console.log('Third timeout executed');
    }, 1000);
  }, 1000);
```

```
}, 1000);

// Promise-based solution using setTimeout
function wait(ms) {
  return new Promise(function(resolve) {
    setTimeout(resolve, ms);
  });
}

wait(1000)
  .then(function() {
    console.log('First timeout executed');
    return wait(1000);
  })
  .then(function() {
    console.log('Second timeout executed');
    return wait(1000);
  })
  .then(function() {
    console.log('Third timeout executed');
  });
```

In this example, the first code block uses nested callbacks to execute three timeouts with a delay of one second between them. The second code block uses the wait() function to create a promise that resolves after a specified delay, and then chains three promises together to execute the three timeouts with a delay of one second between them. The then() method is used to chain the promises together, and the final then() method is used to handle the result of the third timeout. By using promises, the code is more readable and easier to maintain, and it avoids the pitfalls of callback hell.

Promises:

A Promise is a holder for a result (or an error) that will become available in the future.

Promise provides a more structured way to write asynchronous calls.

Promises have replaced callback functions as the preferred programming style for handling asynchronous calls.

Built-in support for promises has been introduced as part of JavaScript from 2015.

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

A Promise is a returned object to which you can attach callbacks, instead of passing callbacks into a function.

Promise comes to the rescue when there are chained asynchronous calls that are dependent on each other.

Using Promises :

The constructor of the Promise accepts only one argument, a function with parameters resolve and reject.

```
new Promise(function (resolve, reject) {
   //async code here
```

```
    //resolve if success, reject if error
});
```

A Promise has three states:

- Pending: the result of the async call is not known yet.
- Resolved: async call returned with success.
- Rejected: async call returned with an error.

To structure the async code, the async operation will be wrapped in a Promise object and handled using "then".

```
var myPromise = new Promise(function (resolve, reject) {
        setTimeout(function () {
                resolve("success");
        }, 2000);
});
myPromise.then(
        function (data) {
                console.log(data + " received in 2 seconds");
        },
        function (error) {
                console.log(error);
        }
);
```

Promises have replaced callbacks and have solved the problem of ‘callback hell’. The sample code has been shown below to understand how developers handled multiple asynchronous calls without Promises in traditional JavaScript applications.

```
doSomething(function(result){
        doSomethingElse(result,function(newResult){
                doThirfThing(newResult,function(finalResult){
                        console.log('Print the final result ' +finalResult);
                }, failureCallback);
        }, failurCallback);
}, failureCallback);
```

The ’Callback hell’, is now resolved using ‘Chaining’ which creates readable code and is an eminent feature of Promise. Here, the asynchronous code can be chained using multiple then statements.

```
doSomething().then(function (result) {
        return doSomethingElse(result);
})
        .then(function (newResult) {
                return doThirdThing(newResult);
        })
        .then(function (finalResult) {
                console.log("Print the final result " + finalResult)
        })
        .catch(failureCallBack);
```

Async/Await:

"async/await" was introduced to implement asynchronous code with promises that resemble synchronous code. "async/await" is simple, easy, readable and understandable than the promises.

Async/Await vs Promises

	Async/Await	**Promises**
Scope	The entire wrapper function is asynchronous.	Only the promise chain itself is asynchronous.
Logic	• Synchronous work needs to be moved out of the callback. • Multiple promises can be handled with simple variables.	• Synchronous work can be handled in the same callback. • Multiple promises use Promise.all().
Error Handling	You can use try, catch and finally.	You can use then, catch and finally.

Using Async/Await:

Async Function

An async function is declared with an async keyword. It always returns a promise and if the value returned is not a promise, the JavaScript will automatically wrap the value in a resolved promise.

Example:

```
async function hello() {
   //Value will be wrapped in a resolved promise and returned
   return "Hello Async";
}
hello().then(val => console.log(val)); // Hello Async
async function hello() {
   //Promise can be returned explicitly as well
   return Promise.resolve("Hello Async");
}
hello().then(val => console.log(val)); // Hello Async
```

Await:

Await keyword makes JavaScript wait until the promise returns a result. It works only inside async functions. JavaScript throws Syntax error if await is used inside regular functions. Await keyword pauses only the async function execution and resumes when the Promise is settled.

Example:

```
function sayAfter2Seconds(x) {
        return new Promise(resolve => {
                setTimeout(() => {
                        resolve(x);
                }, 2000);
        });
}
async function hello() {
```

```
        //wait until the promise returns a value
        var x = await sayAfter2Seconds("Hello Async/Await");
        console.log(x); //Hello Async/Await
}
hello();
```

Executing Network Requests using Fetch API:

Fetch API:

JavaScript plays an important role in communication with the server. This can be achieved by sending a request to the server and obtaining the information sent by the server. For example:

1. Submit an order,
2. Load user information,
3. Receive latest information updates from the server

All the above works without reloading the current page!

There are many ways to send a request and get a response from the server. The fetch() is a modern and versatile method available in JavaScript.

Fetch provides a generic definition of Request and Response objects. The fetch() method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves to Response if the fetch operation is successful or throws an error that can be caught in case the fetch fails. You can also optionally pass in an init options object as the second argument.

Syntax:

```
PromiseReturned = fetch(urlOfTheSite, [options])
```

- urlOfTheSite – The URL to be accessed.
- options – optional parameters: method, headers, etc.

Without options, this is a simple/default GET request which downloads the contents from the URL. The fetch() returns a promise which needs to be resolved to obtain the response from the server or for handling the error.

fetch() Method:

Getting a response from a fetch() is a two-step process.

1. The promise object returned by fetch() needs to be resolved to an object after the server sends a response.

- Here, HTTP status needs to be checked to see it is successful or not.
- The promise will be rejected if the fetch is unable to make a successful HTTP-request to the server e.g. may be due to network issues, or if the URL mentioned in fetch does not exist.
- HTTP-status can be seen in response properties easily by doing console.log
    - status – HTTP status code returned from a response, e.g., 200.
    - ok – Boolean, true if the HTTP status code returned from a response, is 200-299.

2. Get the response body using additional methods.

- Response object can be converted into various formats by using multiple methods to access the body/data from response object:
    - response.text() –read body/data from response object as a text.
    - response.json() – parse body/data from response object as JSON.
    - response.formData() – return body/data from response object as FormData.

- response.blob() – return body/data from response object as Blob (binary data with its type).

```
//pass any url that you wish to access to fetch()
let response = await fetch(url);
if (response.ok) { // if HTTP-status is 200-299
   // get the response body
   let json = await response.json();
   console.log(json)
}
else {
   console.log("HTTP-Error: " + response.status);
}
```

Modular Programming:

Modules are one of the most important features of any programming language.

In 2015 modules were introduced in JavaScript officially and they are considered to be first-class citizens while coding the application.

Modules help in state and global namespace isolation and enable reusability and better maintainability.

We need modules in order to effectively reuse, maintain, separate, and encapsulate internal behavior from external behavior.

Each module is a JavaScript file.

Modules are always by default in strict-mode code. That is the scope of the members (functions, variables, etc.) which reside inside a module is always local.

The functions or variables defined in a module are not visible outside unless they are explicitly exported.

The developer can create a module and export only those values which are required to be accessed by other parts of the application.

Modules are declarative in nature:

- The keyword "export" is used to export any variable/method/object from a module.
- The keyword "import" is used to consume the exported variables in a different module.

Creating Modules:

The export keyword is used to export some selected entities such as functions, objects, classes, or primitive values from the module so that they can be used by other modules using import statements.

There are two types of exports:

- Named Exports (More exports per module)
- Default Exports (One export per module)

Named Exports:

Named exports are recognized by their names. You can include any number of named exports in a module.

There are two ways to export entities from a module.

1. Export individual features

Syntax:

```
export let name1, name2, …, nameN; // also var, const
export let name1 = …, name2 = …, …, nameN;
export function functionName(){...}
export class ClassName {...}
```

Example:

```
export let var1,var2;
export function myFunction() { ... };
```

2. Export List

Syntax:

```
export { name1, name2, …, nameN };
```

Example:

```
export { myFunction, var1, var2 };
```

Default Exports:

The most common and highly used entity is exported as default. You can use only one default export in a single file.

Syntax:

```
export default entityname;
```

where entities may be any of the JavaScript entities like classes, functions, variables, etc.

Example:

```
export default function () { ... }
export default class { .. }
```

You may have both default and named exports in a single module.

Consuming Modules:

How to import Named Exports:

If you want to utilize an exported member of a module, use the import keyword. You can use many numbers of import statements.

Syntax:

```
//import multiple exports from module
import {entity1, entity 2... entity N} from modulename;
//import an entire module's contents
import * as variablename from modulename;
//import an export with more convenient alias
import {oldentityname as newentityname } from modulename;
```

Example:

```
import {var1,var2} from './mymodule.js';
import * as myModule from './mymodule.js';
import {myFunction as func} from './mymodule.js';
```

How to import Default Exports:

You can import a default export with any name.

Syntax:

```
import variablename from modulename;
```

Example:

```
import myDefault from './mymodule.js';
```