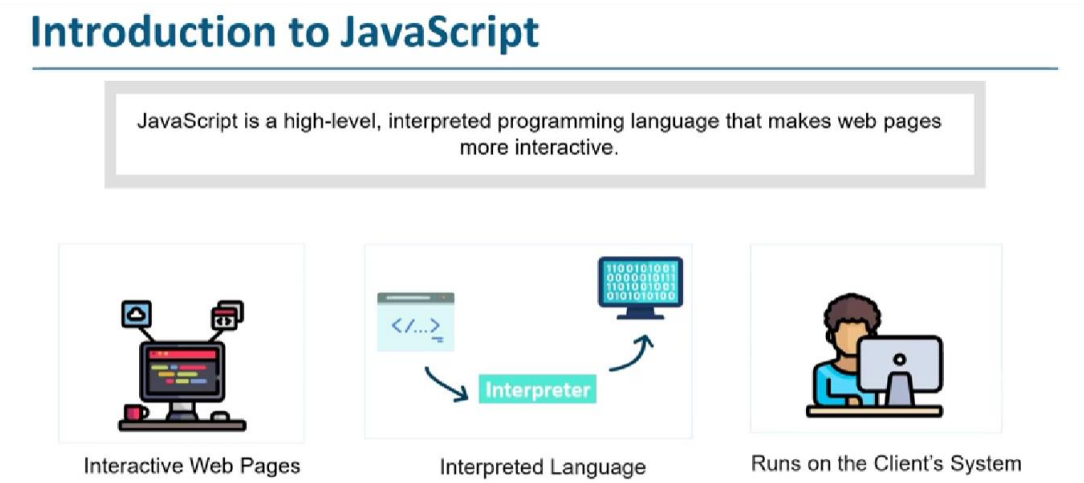
# JavaScript

- JavaScript provides functionality to our web application.
- JavaScript mainly work on events that perform by clients.
- Worlds 98% of web application uses JavaScript.
- JavaScript help us to develop mobile app development, web development, desktop app development and game development etc.




## How many ways to add JavaScript in our application


- Best way of implementing a JavaScript in a down of body section because it is a light weighted scripting language so firstly need to load the html and css then it will otherwise it will load first and html and css not.

.html

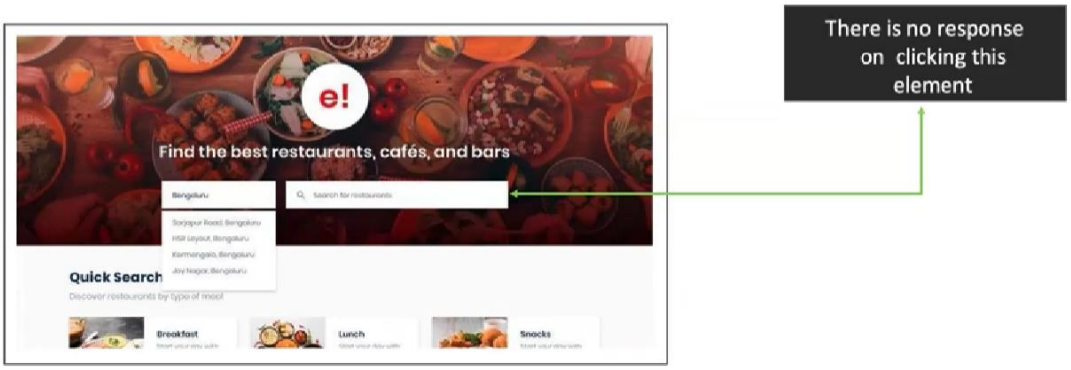
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="script.js"></script>
</head>
<body>
  <input type="button" value="click" onclick="help();">
</body>
</html>
```

.js

```js
function help(){
    alert('this is home page');
}
```

## Life without JavaScript

- Until now, you have made a static webpage with only HTML elements. The images, dropdowns, and buttons are unresponsive, and no action is performed on clicking or hovering over the different elements.



## Life without JavaScript (Contd.)

**What are the drawbacks/ restrictions of the webpage created till now?**

- It is a static unresponsive HTML with images, text, buttons, and other UI features.
- There is no use of such web pages from the perspective of a business or the customers.

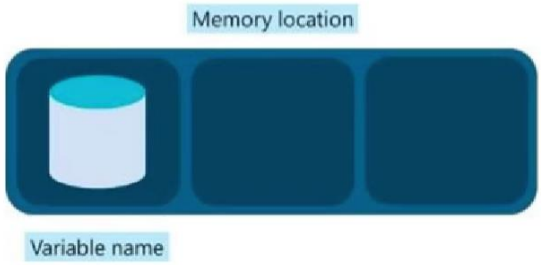
**This is where JavaScript comes into the picture. It provides the following advantages:**

- It is very fast because any code can run immediately instead of having to contact the server.
- It allows you to create highly responsive interfaces to improve the user experience.
- JavaScript has no compilation step. Instead, an interpreter in the browser reads over the. JavaScript code interprets each line and runs it.
- It provides dynamic functionality without having to wait for the server to react and show another page.

# Variable

## Variables

A variable is a name given to a memory location that acts as a container for storing data temporarily. They are reserved memory locations to store values.




Syntax

```
<script type = "text/javascript">
    var num = 2;
    var company = "Edureka";
</script>
```

Example – In our colony campus there are 200 homes and every homes have their different home number just like in ram there are bytes of space so for assigning any variable it can - make bytes of space in memory.

- Every variable is unique in ram.
- If name of variable is same it should either override or give error.

# Rules for declaring a variable-

a. Variable does not start with number.
b. Space does not exist between a variable.
c. Variable contain number in between.
d. Underscore can we use either in start or in between.

# Scope of variables

## Scope of a variable in JavaScript

Scope of a variable refers to its visibility in the variable i.e., in which parts of the program the variable can be seen or used.

JavaScript variables have only two scopes:

- Global Variables – A global variable has a global scope, which means it is not defined locally (i.e., within any function or block of code) and can be accessed anywhere in your JavaScript code.
- Local Variables – A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.
- Lexical Scoping: Visibility of a global variable in the local scope is called lexical scoping.

```
<body>
  <script>
  var text = "Blockcept";
  console.log(text);

  </script>
</body>
```

Output- blockcept

```
<body>
  <script>
  var text = "Blockcept";
  console.log(text);
  function a(){
    console.log(text);
  }
   a();
  </script>
</body>
```

Output- blockcept, blockcept, lexical variable

```
  var text = "Blockcept";
  console.log(text);
  function a(){
    var text ="edureka";
    console.log(text);
  }
  a();
  </script>
```

Output-blockcept,edureka

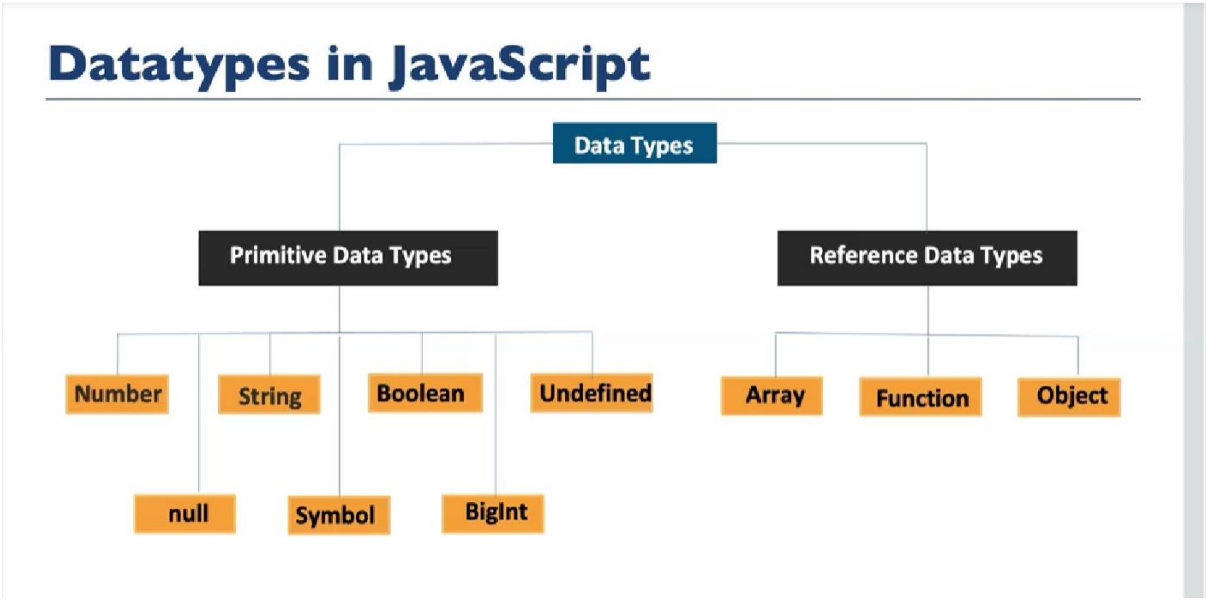
```
<script>
  var text = "Blockcept";
  console.log(text);
  function a(){
    var text2 ="edureka";
  }
  a();

  console.log(text2);
  </script>
```

Output – blockcept,text2 is not defined due to local variable.

# Data types in JavaScript

Data type define which type of value the variable has it should be either number, string etc.

## Primitive data type

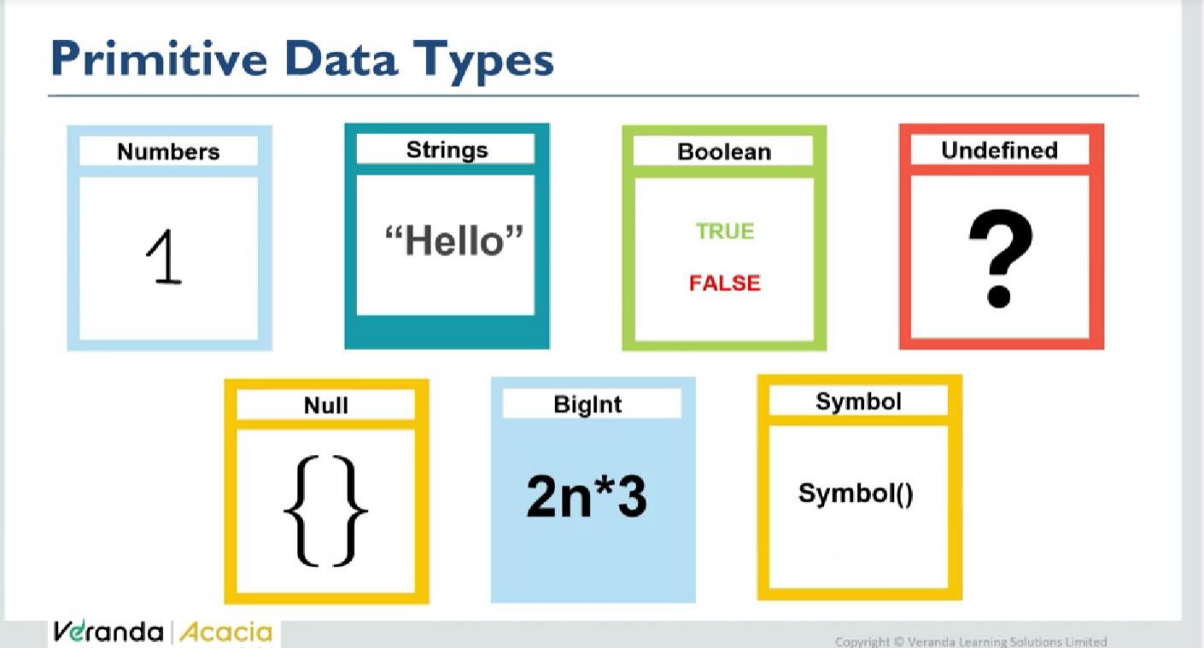



Reference data type are made of primitive data type.

Null defined as empty or unknown value.

Undefined can assign but not declare a value.

New keyword is used to make instance or object.




```
 <script>

var a='deepak';
var b= new String("again deepak");

console.log(a);
console.log(b);
console.log(b[0])
console.log(a[5])
</script>
```

```
var a = 23;
    var b = new Number(23);
    console.log(a);
    console.log(b);
```

```
var a=true;
var b= new Boolean(false);
console.log(a);
console.log(b);
```

# Undefined

- It is a primitive data type in JavaScript.
- The undefined property indicates that a variable has not been assigned a value or not declared at all.
- It indicates there is no value in the variable, or you can assume a variable has nothing inside it.

Variable with value – (Defined)

Variable without value – (Undefined)

```
var a=undefined;
   console.log(a);
```

```
  var a=null;
   console.log(a);
```

# BigInt

- BigInt is a numeric primitive data type in javascript. It is also used to store numbers.
- The name of data types itself tells that it can store large numbers. It stores a large number which is not possible for Number data type to store.
- We must add n at the end of the number while assigning value to a BigInt variable.

**Syntax:**

```
Keyword variablename = value;
                       OR
Keyword variablename = new BigInt(value);
```

The general way for creating a variable of type number

Creation of number using **new** operator –using Constructor

```
var a=236547899547889 85n;
console.log(a);
var b=BigInt(586974587987);
console.log(b);
```

```
var a = Symbol(23);
   console.log(a);
```

# Reference data type

## Reference Data Types

- Reference data types are formed with the help of primitive data types.
- Following are the reference data types in JavaScript:
  - Arrays
  - Objects
  - Function

## Objects in JavaScript

- JavaScript object is a non-primitive data type that allows you to store multiple collections of data.
- An object can be created with figure brackets {…} with an optional list of properties. A property is a key: value pair, where the key is a string (also called a property name), and value can be anything.
- There are different ways to create new objects :
  - Using an object literal
  - Using new keyword

1. Object is made up of key value pair.
2. Key with unique variable name.
3. Value should be either string, number, char, Boolean, function, array etc.

```
4.  <body>
5.    <script>
6.      var text;
7.      var arr;
8.      var a = {
9.        a: "deepak",
10.       b: "patel",
11.       c: "teacher of edureka",
12.       d: function (num) {
13.         console.log("hii");
14.         console.log("this is the example of writing function under js
    object")
15.         console.log(num);
16.       },
17.       e:["Sea","ocean","river"],
```

```
    };
    console.log(a.a);
    console.log(a.b);
    console.log(a.c);
    console.log(a.e[0]);
    a.d(20);
  </script>
</body>

</html>
```

e. Array is collection of heterogeneous element in JavaScript not in all programming language.

# Var, let and const

a. 3 ways of declare a JavaScript variable var, let and const.
b. The var keyword is used in all JavaScript code from 1995 to 2015.
c. The let and const keyword were added to JavaScript in 2015.

## var vs let vs const

The differences between var, let, and const keywords are mentioned below.

| Behavior | var | let | const |
|---|---|---|---|
| **hoisting** | exhibits hoisting | does not exhibit hoisting | does not exhibit hoisting |
| **scope** | functional scope | block scope | Block scope |
| **mutability** | mutable | mutable | immutable |

# Var

a. Var have global scope.
b. It is mutable.

```
c. <script>
d.     var text="deepk";
e.     text="patel"
```

```
f.      console.log(text);
g.
h.    </script>
```

# Let

a. Let have block scope
b. It is mutable

```
c. <script>
d.      function de(){
e.      let text="deepk";
f.      text="patel";
g.      console.log(text);
h.      }
i.
j.
k.      de();
l.
m.    </script>
```

Output –patel (due to override),code runs.

```
<script>
    function de(){
   let text="deepak";
   text="patel";
   console.log(text);
   }
   console.log(text);

   de();

 </script>
```

output-text is not defined(due to block scope of text variable)

# Const

a. It is immutable.
b. It has block scope.
c. Once we declare a variable with const it cannot reassign.

```
d.   <script>
e.       function de(){
```

```
    const text="deepak";
    text="patel";
    console.log(text);
    }
    console.log(text);

    de();

  </script>
```

# Hoisting

a. Hoisting in JavaScript only work on var declared variable.
b. Hoisting can all initialization variable should be at the top of code internally.
c. It cannot depend on declarization.

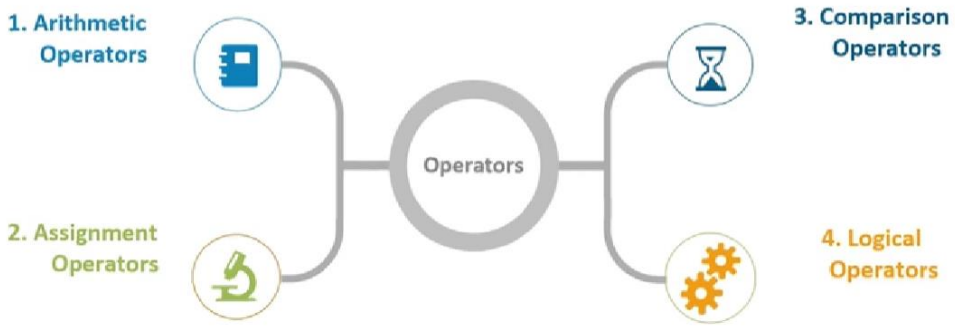
```
<script>
  function a(){
    console.log(a);
    var a;
    a=10;
    console.log(a)
}

a();
</script>
```

d. Let, const, function expression, class expression does not follow the hoisting.
e.

# Operators in JavaScript

## JavaScript Operators

- In JavaScript, operators are used for performing several operations.
- Operators can be used to assign values, compare values and perform arithmetic operations.
- The commonly used operators of JavaScript are classified into four categories.




---

## Arithmetic Operators

- Arithmetic operators are used to perform arithmetic operations.
- In JavaScript, the operators that are used to perform such tasks are as follows:

| Operator Name | Description |
|---|---|
| + | It is used to perform the addition of two or more variables |
| - | It is used to perform subtraction of two or more variables |
| * | It is used to perform multiplication of two or more variables |
| / | It is used to perform the division of two variables |
| % | It is used to find modulus (or remainder) of the division |
| ++ | It is used to perform the increment on the number |
| -- | It is used to perform the decrement on the number |

---

Post and Pre increment and decrement should be explain in detail.

# Assignment Operators

- Arithmetic operators are used to assign values to the variables.
- In JavaScript, the operators that are used to perform such tasks are:

| Operator Name | Description |
|---|---|
| = | It is used to assign any value directly to the variable e.g., x = y |
| += | It is used to increment the value and assign it to the variable e.g., x=x+1 |
| -= | It is used to decrement the value and assign it to the variable e.g., x=x-1 |
| *= | It is used to multiply the value and assign it to the variable e.g., x=x*1 |
| /= | It is used to divide the two number and assign it to the variable e.g., x=x/1 |
| %= | It is used to calculate the modulus of two numbers and assign it to the variable, e.g., x=x%1 |

```
var a = 23;
var b = a;
console.log(b);

a += 12;
console.log(a);

a -=16;
console.log(a);

a *= 12;
console.log(a);

a /=12;
console.log(a);

a%=10
console.log(a);
```

# Comparison Operators

- Comparison operators are used to compare the two values.
- These operators will always return a Boolean value.
- In JavaScript, the operators that are used to perform tasks such as:

| Operator Name | Description |
|---|---|
| == | It is used to compare whether the two values are equal |
| === | It is used to compare whether the two values and their data types are equal |
| != | It is used to compare whether the two values are not equal |
| !== | It is used to compare whether the two values and their data types are not equal |
| > | It is used to check whether 1 value is greater than other |
| < | It is used to check whether 1 value is lesser than other |
| >= | It is used to check whether 1 value is equal or greater than other |
| <= | It is used to check whether 1 value is equal or less than other |

```
if(10==10){
    console.log("We are in")
}
else{
    console.log("We are out")
}

if(10===10){
    console.log("We are in")
}
else{
    console.log("We are out")
}

if(10!=10){
    console.log("We are in")
}
else{
    console.log("We are out")
}

if(6<10){
    console.log("We are in")
}
else{
    console.log("We are out")
}
```

# Logical Operators

- A logical operator is used to connect two or more expressions into a more complex expression.
- These operators are mainly used to control the flow of the program.
- These operators always return a Boolean value.
- In JavaScript, the operators that are used to perform tasks such as :

| Operator Name | Description |
|---|---|
| && | It returns true if all the conditions are true, otherwise it will return false |
| \|\| | It returns true if one of the conditions are true, otherwise it will return false |
| ! | Its opposites the result of logical operations |

- && is a And operator.
- Both left and right hand condition should be true for achieving the and operator.
- || or operator.
- ! Not operator.

```
-     var a=10;
- var b=11;
- var c=12;
- var d=13;
-
- if((a<11)&&(c<d)){
-     console.log("We are in")
- }
- else{
-     console.log("We are out")
- }
-
- if((a<11)||(c<d)){
-     console.log("We are in")
- }
- else{
-     console.log("We are out")
- }
-
- if((a<b)||(c>d)){
-     console.log("We are in")
- }
- else{
-     console.log("We are out")
- }
-
```

```
if((a>b)||(c>d)){
    console.log("We are in")
}
else{
    console.log("We are out")
}

var h=true;
if(!h){
    console.log("We are in")
}
else{
    console.log("We are out")
}
```

# Ternary Operator

The ternary operator is a conditional operator used as a shortcut for the if-else statement. It takes three operands: a condition followed by a question mark(?), then a statement to execute if the condition is met, and a statement to execute if the condition is false.

**Code:**

```
<script>
      function getFee(isMember){
         return (isMember ? `$2.00` : `$10.00`);
      }
      document.write(getFee(true) + "<br>");
      //expected output : "$2.00"
      document.write(getFee(false) + "<br>");
      //expected output : "$10.00"
      document.write(getFee(null) + "<br>");
      //expected output : "$10.00"
</script>
```

```
(10<12)?(console.log('hii')):(console.log('bye'));
(10>12)?(console.log('hii')):(console.log('bye'));
```

## Typeof() Operator

In JavaScript, the typeof() operator returns the data type of its operand in the form of a string. The operand can be any object, function, or variable.

**Code:**

```
<script>
    var x = 10;
    var z = "Edureka";
    var studObje = { rno : 101, name : "Alexa"};
    document.writeln(typeof(x));
    document.writeln(typeof (z));
    document.writeln(typeof(studObje));
</script>
```

console.log(typeof(a));

# JavaScript conditional statement

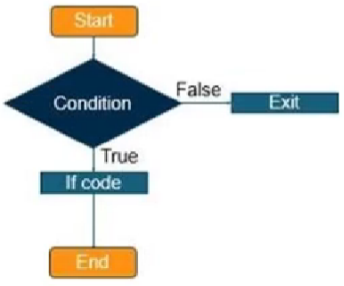
## If Statement

The conditional statement is a set of rules performed if a certain condition is met. It is similar to the if-then statement (if a condition is met, then an action is performed).

**Syntax:**

```
if(condition) {
   Statement;
   ...
 }
```

**Data Flow Diagram:**



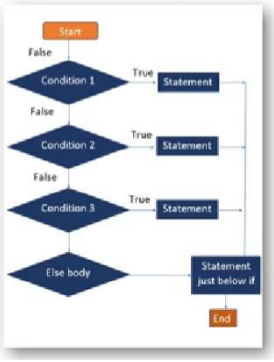
# else if Statement

In programming, the else-if statement is also known as the if-else-if ladder. It is used when more than two possible actions are based on different conditions.

**Syntax:**

```
if(condition)
{
  Statement 1
}
else if
{
   Statement 2
}
else
{
   Statement 3
}
```

**Data Flow Diagram:**



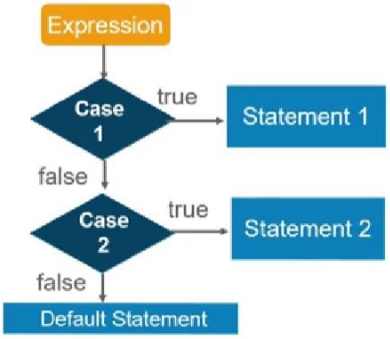

# Switch Statement

Here JavaScript expression is evaluated. The result of the expression is compared with the values of each case. If there is a match, the associated block of code is executed. Otherwise, the default code block is executed.

**Syntax:**

```
switch(expression) {
   case 1:
     //statement
     break;
  case 2:
      //statement
      break;
   default:
     //statement
}
```

**Data Flow Diagram:**




```
switch(1) {
 case 1:
     console.log("we in one");
   // code block
   break;
 case 2:
   console.log("we in two");
   break;
 default:
   console.log("hii");
}
```

# Looping statement

## Looping Statements

When you want to execute a statement/block of statements multiple times, you can use looping statements to avoid writing the same statements repeatedly.

JavaScript supports 5 types of looping statements:

- for loop
- while loop
- do..while
- for..in
- for..of

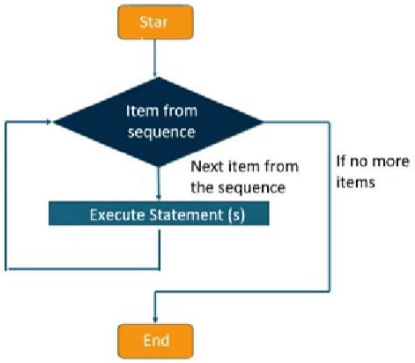

## for Loop

The for loop repeatedly executes the code until a given condition is TRUE. It tests the condition before executing the loop body. This loop is used when you know how many times the loop should get executed at compile time.

**Syntax:**

```
for(begin; end; step)
{
   Loop body;
   ...
}
```

**Data Flow Diagram:**




```
for(let a=0;a<10;a++){
    console.log(a);
}
```

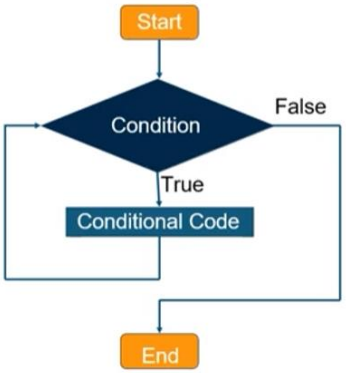
# while Loop

while loop executes the given block of code as long as the given condition is true. A while loop is normally used in a scenario where you don't know how many times a loop will get executed at runtime.

**Syntax:**

```
while(condition)
{
    Loop body;
    ...
}
```

**Data Flow Diagram:**




```
var a=1;
while ( a < 10){
    console.log(a);
    a++;
}
```

```
let a=1;
while ( a < 10){
    console.log(a);
    a++;
}
```

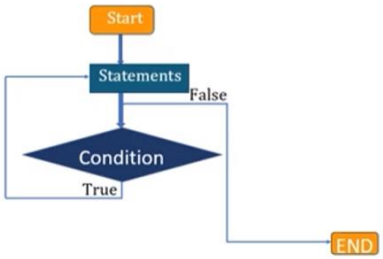
# do..while Loop

do..while loop always executes the statements once before checking the condition. Once after executing the body, it checks the condition and If the condition is true, the code within the loop is executed again. At the end of every execution, the condition is checked. When the condition is false, execution stops, and control passes to the next statement.

**Syntax:**

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

**Data Flow Diagram:**




```
var a=100;
do{
    console.log(a);
    a++;
}while(a<10)
```

# for..in loop

It allows you to iterate over all property keys of an object. It takes three parameters. The first parameter is a variable that iterates over the properties of an object, the second parameter is an operator(in) and the third parameter will specify the object that will be iterated.

**Syntax:**

```
for (key in object)
{
    // body of for...in
    ...
}
```

## for..of Loop

It allows you to iterate through the value of iterable objects. The for..of statement creates a loop iterating over iterable objects (including Array, Map, Set, and so on). It starts the looping from the first element of the array & for each iteration the next value of the array is assigned to a variable.

**Syntax:**

```
for (variable of object)
{
    // body of for...of
    ...
}
```

```
var a={a:"deepak",b:"patel",c:"edureka",d:"we are web developer"};
var b=["deepak","bhopal",23,9399431102];
var x;
for( x in a){
    console.log(a[x]);
}

for(x of b){
    console.log(x);
}
```

# JavaScript Output Statement

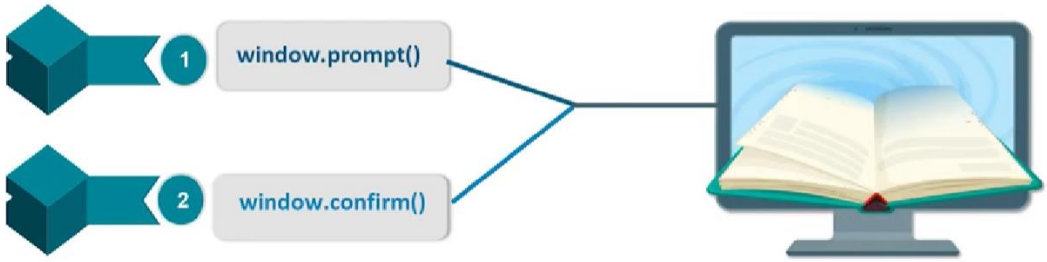
## Understanding JavaScript Output Statements

- In JavaScript, there are different ways to display the data.
- Some of the commonly used ways to display output are:

| Statement | Description |
|---|---|
| innerHTML | It writes a data into a certain HTML element |
| document.write() | It displays a data in the HTML output |
| window.alert() | It displays the data in the alert box |
| console.log() | It displays the data in the browser console |

# Understanding JavaScript Input Statements

- It is important to take input from the user to display the data as per their need.
- Apart from forms, JavaScript provides two statements to accept input from the user as mentioned below:




```
<body>
  <div id="deep">
    <input type="submit" value="click me" onclick="sub();">
  </div>
  <script>
      document.write("this is document write function");
      window.alert("this is a window fuction");
      var d = window.prompt("do you interested in that");
      document.write(d);
      window.confirm("yes or no");
      function sub(){
        var a=document.getElementById('deep');
        a.style.backgroundColor="green";
      }
  </script>
```

# JavaScript String methods

## The String Object

- String is a primitive datatype.
- It is used to represent text.
- It contains a sequence of UTF-16 units.
- Anything inside the double quotation(" ) or single quotation(') will be considered as a string.

**Syntax:**

```
var stringName = "This is string";
```

## The String Object (Contd.).

- The String object lets you work with a series of characters.

**Syntax:**

```
var str = new String("string");
```

- Below is a list of the properties of String object and their description:

| Property | Description |
|---|---|
| **Constructor** | Returns a reference to the String function that created the object |
| **Length** | Returns the length of the String |
| **Prototype** | Allows you to add properties and methods to an object |

# String Methods

- The following methods can be used for primitive type string or a string object.

| Method | Description |
|---|---|
| **length** | Returns the length of the string |
| **slice()** | Returns the elements between the two indexes of a string |
| **concat()** | Combines the text of two strings and returns a new string |
| **substring()** | Returns the elements between the two indexes of a string |
| **replace()** | Used to find a matching text and replaces the text with string |
| **toUpperCase()** | Converts all the alphabets of a string into uppercase |
| **toLowerCase()** | Converts all the alphabets of a string into lowercase |
| **trim()** | Removes the leading and trailing white space and line terminator characters |

# Example of length method

- length method returns the length of String Object.

**Code:**

```
<script>
    var stringExample = "Hello, it is good to talk to you";
    document.write("Length is : " + stringExample.length);
</script>
```

length method gives output in a number

**Output:**


# Example of slice Method

- slice method returns a section of String Object.
- Syntax: array.slice(start, end);
- The start index is required while the end index is an optional parameter. It accepts negative parameters as well.

**Code:**

```
<script>
   var stringExample = "Hello, it is  good to talk to you";
   document.write("Slicing without giving parameters : " +
stringExample.slice(),"</br>");
   document.write("slicing with only start index:  " + stringExample.slice(2),"</br>");
   document.write("Slicing with start & end index:" +
stringExample.slice(2,10),"</br>");
   document.write("Slicing with negative start index: " + stringExample.slice(-
5),"</br>");
   document.write("Slicing with negative end index: " + stringExample.slice(2,-
5),"</br>");
</script>
```

```
<body>
  <script>
    var str= new String("this is one of the best thing to do the web
development course");
    var str1="this is one of the best thing to do the web development course";
    document.write(str1.length+"<br/>");
    document.write(str1.slice(6)+"<br/>");
    document.write(str.slice(6)+"<br/>");
    document.write(str.slice(6,12)+"<br/>");
    document.write(str.slice(-6)+"<br/>");
    document.write(str.slice(2,-6)+"<br/>");
    document.write(str.slice(-6,-2)+"<br/>");
  </script>
</body>
```

# Example of substr Method

- substr method takes two parameters, start index & length.
- Start index denotes the index of string from where it begins slicing.
- length denotes the number of characters to extract.
- Some of the cases of substr method
  - If the start index is omitted, substr() returns the complete string
  - If the start index is negative, substr() counts from the end of the string

**Code:**

```
<script>
    var string_1 = "Full Stack Web Development";
    document.write("<h3>Without start index:" +string_1.substr() + "</h3>");
    document.write("<h3>With start index:" + string_1.substr(4) , "</h3>");
    document.write("<h3>With both the values:" + string_1.substr(8,3) + "</h3>");
    document.write("<h3>With equal values:" + string_1.substr(2,2)} + "</h3>");
    document.write("<h3>With negative start value:" string_1.substr(-2,) + "</h3>");
</script
```

```
 var str= new String("this is one of the best thing to do the web development course");

    var str1="this is one of the best thing to do the web development course";
    document.write(str1.length+"<br/>");
    document.write(str1.slice(-12,6)+"<br/>");
    document.write(str1.substring(-12,6)+"<br/>");
    document.write(str1.substr(6,12)+"<br/>");
    document.write(str1.substr(6,)+"<br/>");
    document.write(str1.replace("development","deepak")+"<br/>");
    document.write(str1.replace("/development/g,"deepak")+"<br/>");
    document.write(str1.toUpperCase() +"<br/>");
    document.write(str1.toLowerCase() +"<br/>");
    document.write(str1.concat(" ",str)+"<br/>");
    document.write(str1.concat(" this is one of the best way of doing thing")+"<br/>");
```

# Example of replace Method

- replace() method replaces text in a string, using a regular expression or search string.
- Some regular expression modifiers can be used with replace method of string:
  - I – it's used to perform case-insensitive matching.
  - g – it's used to perform global matches, i.e., it will find all the matches.
  - m – it's used to perform multiline matching.

**Code:**

```
<script>
    var string_1 = "Welcome to Full Stack Web Development Program of Google";
    document.write(string_1.replace("Google","Edureka"));
</script>
```

**Output:**

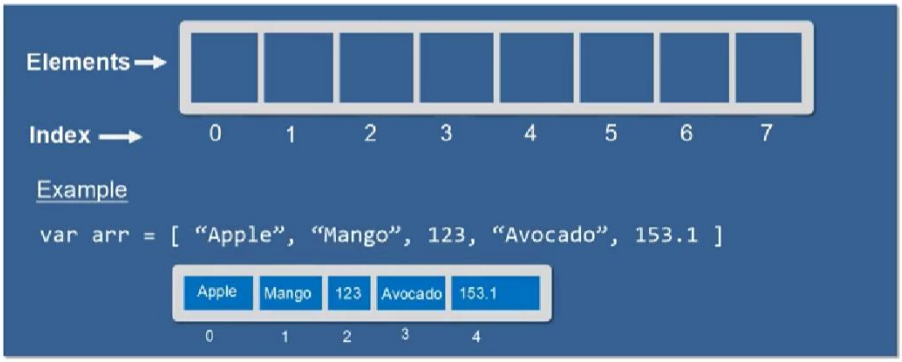
Welcome to Full Stack Web Development Program of Edureka

- Slice(start, end)
- Substring( start, end)
- Substr( start, length)
- The difference is that start and end values less than 0 are treated as 0 in substring.
- The difference is that the second parameter specifies the length of the extract part.

# Array

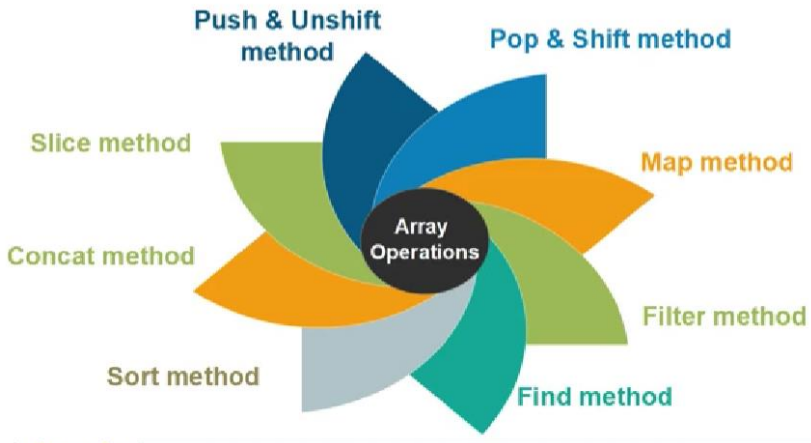
## Arrays in JavaScript

- Array is defined as a collection of heterogeneous elements.
- It will helps to store multiple values of a different or same data type in a single variable.




## JavaScript Array Operations

- The Array operations are used to perform element-by-element operations on matrices.
- Some of the commonly used array methods are given below:



# Push & Unshift Method

- push() method is used to add an element at the end of the array.
- unshift() method can be used to add an element at the beginning of an array.
- These methods will increase the length of the array by 1 and return the updated array.

**Syntax :**

```
array.unshift(item1, item2, ..., itemX);
```

Adds the element at the end of the array

```
array.push(item1, item2, ..., itemX);
```

Adds the element at the starting of the array

**Example :**

```
array.push(10, 20, 30);
array.unshift(1, 2, 3);
```

```
var arr=["deepak","patel",56,56,87,45,23];
arr.shift();
document.write(arr+"<br/>");

arr.unshift("deepak",98,89);
document.write(arr+"<br/>");

arr.push("deepak",98,89);
document.write(arr+"<br/>");

arr.pop();
document.write(arr+"<br/>");
```

# Map Method

- The map() method creates a new array with the results of the given function.
- This method executes the function on the array elements which has proper values.
- The size of the resultant array will always be the same as the original array.
- This method doesn't make any change in the original array.

**Syntax :** `array.map(function_name);`

**Example :** `arr1.map(iterate);`

```
<script>
    var arr=["deepak","patel",56,56,87,45,23];
    function fun(num){
     return num*10;
    }
    const arr1=arr.map(fun);
    document.write(arr1);
</script>
```

```
<script>
    var arr=[56,56,87,45,23];
    function fun(num){
     return num*10;
    }
    const arr1=arr.map(fun);
    document.write(arr1);
  </script>
```

```
<script>
    var arr=[56,56,87,45,23];
    function fun(num){
     return num**2;
    }
    const arr1=arr.map(fun);
    document.write(arr1);
  </script>
```

- Fun is a callback function.

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

## Foreach Method

- The foreach method allows you to loop through arrays.
- The foreach method passes a callback function for each element of an array together with the following parameters:
  - Current Value (required) - The value of the current array element.
  - Index (optional) - The current element's index number.
  - Array (optional) - The array object to which the current element belongs.

**Syntax :**

```
array.forEach(function(currentValue, index, array_name));
```

**Example :**

```
array.forEach((i, index, arr) => { //statements});
```

```
const arr=[56,56,87,45,23];
     function fun(num){
       console.log(num**8);
     }
    arr.forEach(fun);
```

- the map() method returns a new array, whereas the forEach() method does not return a new array.
- The map() method is used to transform the elements of an array, whereas the forEach() method is used to loop through the elements of an array.

# Filter Method

- The filter method creates a new array from the elements of the given array that pass a certain test.
- This method executes the function on the array elements which has proper values.
- The size of the resultant array may or may not be the same as the original array.
- This method doesn't make any change in the original array.

**Syntax :** `array.filter(function_name);` — Function will check the criteria

**Example :** `array.filter(iterate);`

```
const arr=[56,56,87,45,23];
    function fun(num){
     return num<65;
    }
   const arr1=arr.filter(fun);
   document.write(arr1);
```

# Find Method

- The find method is used to find the element in the array.
- This method returns the value of the first element in an array that passes a test.
- This method executes the function for each element of the array.
- If it finds an element where the function returns a true value, it will not check the remaining values and return the first value.
- If no value is found, it will return undefined.

**Syntax :** `array.find(function(currentValue, index, arr), thisValue)`

```
const arr=[56,56,65,45,23];
    function fun(num){
     return num<65;
    }
   const arr1=arr.find(fun);
   document.write(arr1);
```

## reduce Method

- The `arr.reduce()` method in JavaScript is used to reduce the array to a single value.
- It takes two parameters:
  - **Reducer/Callback function:** It denotes what action we will perform to get to one value.
  - **Accumulator/Initial value:** It is the value that we end with.

**Syntax :**

```
array_name.reduce(callback,accumulator);
```

```
const arr=[56,56,65,45,23];
    function fun(num,total){
     return total=total+num;
    }
   const arr1=arr.reduce(fun);
   document.write(arr1);
```

## sort Method

- The sort method is used to sort the items of an array.
- This method can sort the elements (alphabetic/numeric) in either ascending or descending order.
- By default, it sorts the numeric elements in ascending and string in alphabetical order.
- This method can provide incorrect results when working with numbers.
- That's why a compare function is used to solve the problem.
- This method makes the change in the original array.

**Syntax:**

```
array.sort(compareFunction)
```

```
<script>
    const arr=[56,56,65,45,23];
   //  function fun(num){
   //   document.write(arr);
   //  }
   const arr1 =arr.sort();
   document.write(arr1);
  </script>
```

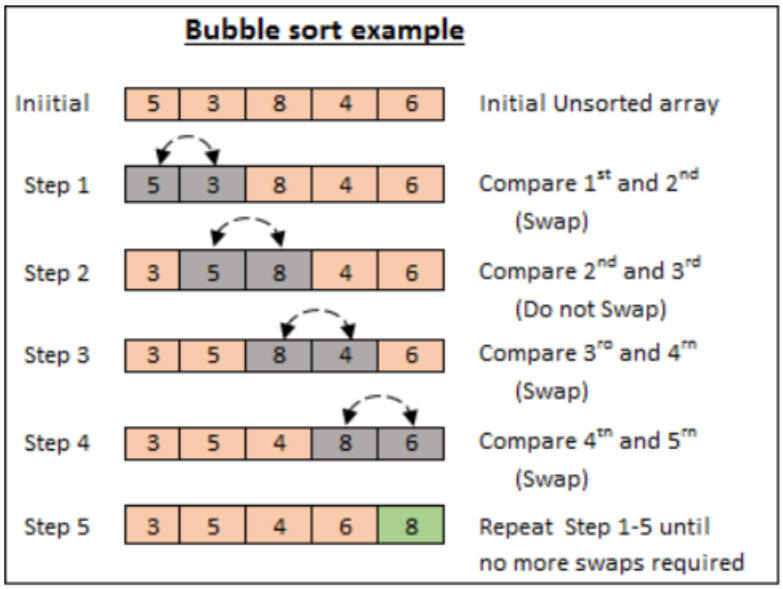
```
const arr=[56,56,65,45,23];
```

```
    function fun(num,num1){
     return num1 -num ;
    }
   const arr1 =arr.sort(fun);
   document.write(arr1);
```

//for ascending and descending order

//JavaScript use insertion sorting by default.

## Image representation




```
<script>
    const arr=[56,1,65,45,23];

    function fun(num,num1){
     document.write(num+"<br/>");
     document.write(num1+"<br/><br/>");
     return num-num1;
    }
   const arr1 =arr.sort(fun);
   document.write(arr1);
  </script>
```

# indexOf Method

- The indexOf method returns the first occurrence of the specified value.
- It takes two parameters, the first is "value to search" & the second is "where to start the search".
- Some cases of indexOf method:
  - If the method returns, a negative value it means the value is not found.
  - To start the search from the end of an array, give a negative value to start index.

**Syntax:**

```
array.indexOf("item", startindex)
```

```
const arr=[56,56,65,45,23];

    //  function fun(num,num1){
    //   return num1 -num ;
    //  }
    const arr1 =arr.indexOf(65);
    document.write(arr1);
```

# concat Method

- It allows us to perform concatenation operations on two arrays or more than two arrays.
- It returns a new array that contains the elements of joined arrays.
- Concatenation doesn't affect the original array.

**Syntax :**

```
array.concat(array1,...,arrayn);
```

## Array.isArray Method

- IsArray method checks whether the passed parameter is an array or not.
- It returns true if the passed object is an array, otherwise it returns false.

**Code :**

```
<script>
    const array1 = [10, 20, 30, 60,70];
    document.write("<b>Array:</b>" +Array.isArray(array1));
</script>
```

**Output :**

Array:true

## slice Method

- Slice method cuts the array into small parts using given parameters.
- This method doesn't make any change in the original array.
- Method takes two parameters:
  - **Start index** – Optional Parameter. It will start the slicing by including the start index. A negative value will start the slicing from the end of the array.
  - **End index** – Optional Parameter. It will slice the array until the end index but exclude it.

**Syntax :**

```
array.slice(startindex,endindex);
```

## splice Method

- Splice method allows you to add/remove elements from an array from the middle.
- Changes made by this method will affect the original array.
- Method takes three parameters:
  - **Start index:** It's a required parameter that denotes from which position we have to add or remove elements from an array. A negative value will add items from the end of the array.
  - **Number:** It's an optional parameter that denotes how many items are to be removed.
  - **End index:** It's an optional parameter that denotes the list of elements to be added.

**Syntax :**

```
array.splice(startindex, number, item1, ....., itemN);
```

```
const arr=[56,1,65,45,23];
```

```
//  function fun(num,num1){
//   document.write(num+"<br/>");
//   document.write(num1+"<br/><br/>");
//   return num-num1;
//  }
arr.splice(1,2,90);
document.write(arr);
```

# Length Method

Length method returns the number of elements in the array.

**Code :**

```
<script>
    const array1 = [10, 20, 30, 60,70];
   document.write("Length of array1 is :" +array1.length);
</script>
```

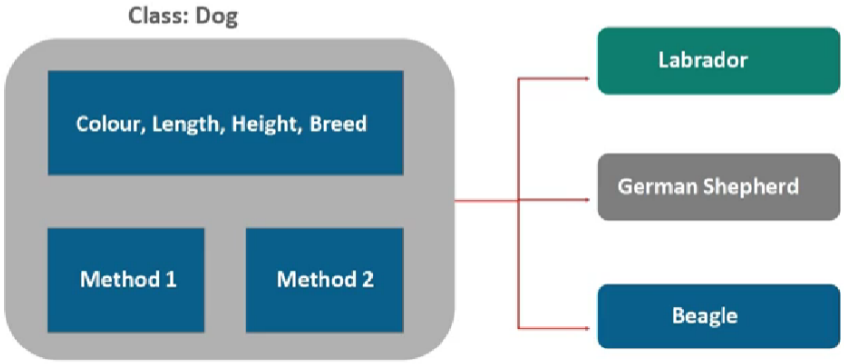
**Output :**

Length of array1 is :5

# Objects and Classes

## Classes in JavaScript

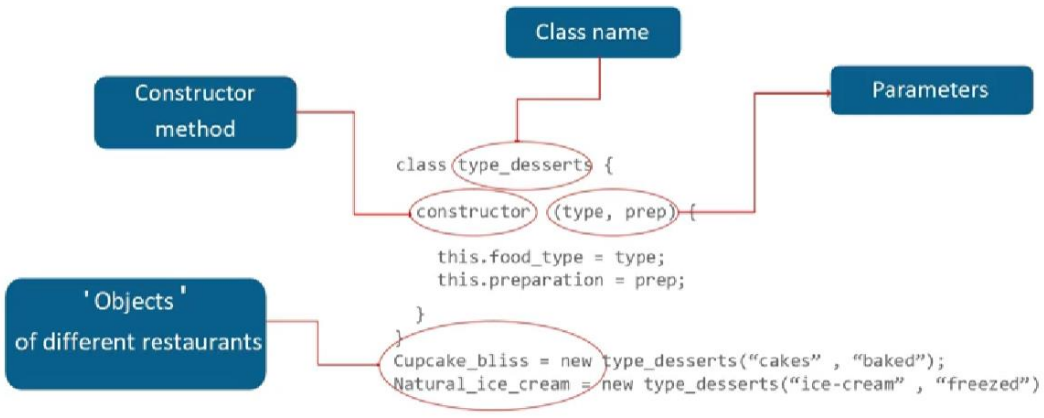
- In object-oriented programming, a class is a blueprint for creating objects.
- It provides initial values for the state (member variables or attributes) and the implementations of behavior (member functions or methods).




- Take example of cars ;

Car is a class and tata, Hyundai, Honda etc are the instance of car class.

## Classes in JavaScript (Contd.)




**Explaining Structure of Class**

//constructor call when we make an objects of that class.

# JavaScript Object

- An object is an instance of a class.
- It has states and behaviors.
- For example, a dog can have the states as color, name, breed.
- A dog's behavior can be wagging the tail, braking and eating.

**Code:**

```
var dog= new Object();
dog.color = "orange";
dog.name = "Tony";
dog.breed ="spitz";
```

# Accessing Objects in JavaScript

The values of objects can be accessed in two ways:

- Using dot(.) operator
- By using square brackets

**Code :**

```
<script>
  var object = {
    name: "Avacado",
    price: 125,
  };
  document.write("<b>" + object.name + "</b> <br>");
  document.write("<b>" + object["price"] + "</b>");
</script>
```

## Constructor In Class

- The constructor method is a special method used to initialize properties. It is called automatically when an object of the class is instantiated, and it has to have the exact name constructor.
- If you do not have a constructor method, JavaScript will add an invisible and empty constructor method.

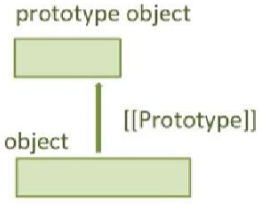
```
Syntax:

class class_name
{
  constructor(brand)
  {
    this.carname = brand;
  }
}
```

```
class a{
      constructor(name,sirname){
        this.hii=name;
        this.sirname=sirname;
      };
    }

     var d=new a("deepak","patel");
     document.write(d.hii);
```

# Inheritance

## Prototypal Inheritance

- Prototypal inheritance is a type of inheritance that adds new properties and methods to an existing object. This inheritance makes use of prototype objects i.e., object.prototype.
- In JavaScript, objects have a special hidden property [[Prototype]] (as named in the specification) that is either null or references another object. That object is called a prototype.



# Prototypal Inheritance (Contd.)

- When we read a property from an object, and it's missing, JavaScript automatically takes it from the prototype. In programming, this is called "prototypal inheritance".
- The property [[Prototype]] is internal and hidden, but there are many ways to set it.

# Prototypal Inheritance Example

**Code :**

```
<script>
    let Friend = {
        have: "iPhone"
    };
    let Person = {
        owns: "OnePlus"
    };

    Person.__proto__ = Friend; // sets Person.[[Prototype]] = Friend
    document.write(Person.have);
</script>
```

After using __proto__ Person is able to access Friend object's properties

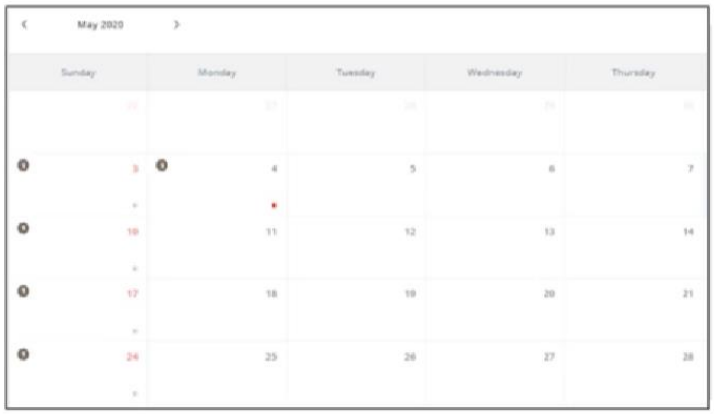
Person doesn't have access to have property of Friend class

```
const a={name:"deeepak"};
     const b={sirname:"patel"};

    a.__proto__=b;
     document.write(a.sirname);
```

# Date

## Understanding Date Object

- The Date object is a data type built into the JavaScript language.
- It is used to deal with date-time manipulations.
- Example: To fetch the system time to track the last time the user logged out.



## Creating Date Object

- There are four ways to create the Date object:
  - **New Date():** Creates a new date object with the current date and time.

```
var d = new Date();
```

  - **New Date(year, month, day, hours, minutes, seconds, milliseconds):** Creates a new date object with a specified date and time. JavaScript counts months from 0 to 11, where January is 0 and December is 11.

```
var d = new Date(2022, 1, 24, 10, 33, 30, 0);
```

```
var a = new Date(2023,3,44,10,23,45,23);
   var b = new Date();
   document.write(a+"<br/>");
   document.write(b);
```

# Creating Date Object (Cont..)

- **New Date(date string):** Creates a new date object from a date String.

```
var d = new Date("October 13, 2014, 11:13:00");
```

- **New Date(milliseconds):** Creates a new date object as zero time plus milliseconds.

  Example: 01, 1970 00:00:00 UTC.

```
var d = new Date(100000000000);
```

# Methods of Date Object

- Below is a description of some commonly used Date methods:

| Method | Description |
|---|---|
| **Date()** | Returns today's date and time. |
| **getDate()** | Returns the day of the month for the specified date according to local time.. |
| **getDay()** | Returns the day of the week for the specified date according to local time. |
| **getFullYear()** | Returns the year of the specified date according to local time. |
| **getHours()** | Returns the hour in the specified date according to local time. |
| **getMilliseconds()** | Returns the milliseconds in the specified date according to local time. |
| **getMinutes()** | Returns the minutes in the specified date according to local time. |
| **getMonth()** | Returns the month in the specified date according to local time. |
| **getSeconds()** | Returns the seconds in the specified date according to local time. |

# Math Object

- In JavaScript, Math object allows you to perform mathematical tasks.
- It provides properties and methods to perform these tasks.
- All the properties and methods of Math objects are static in nature.
- All the properties and methods of a Math object can be called by using the keyword Math as an object.

**Syntax:**

```
Math.propertyname
```

Or

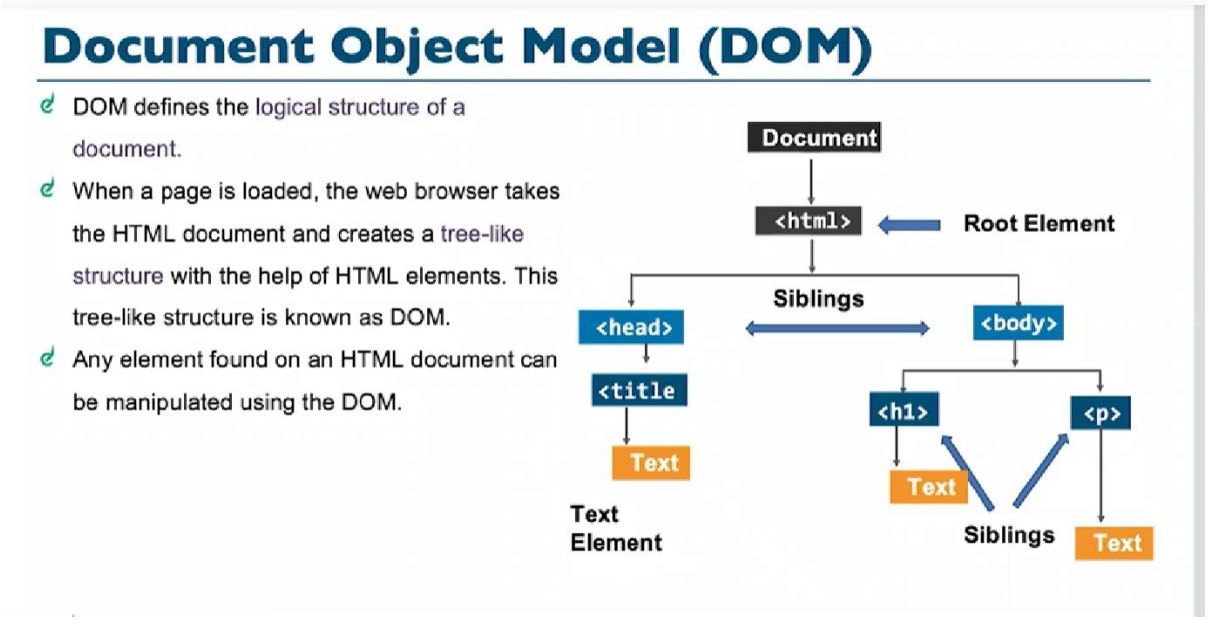
**Syntax:**

```
Math.Methodname()
```

```
const a = new Date(2023,3,44,10,23,45,23);
```

```
    document.write(a+"<br/>");
    document.write(a.getDate());//method
```

# DOM



## Selecting Elements

Various methods in DOM are used to select the elements based on their class, id, tag &, etc.

| Method | Description |
|---|---|
| **getElementById()** | Selects an element by id |
| **getElementsByName()** | Selects all the elements having same name that is given |
| **getElementsByTagName()** | Selects an element by tag name |
| **getElementsByClassName()** | Selects an element by class name |
| **querySelector()** | Selects an element by CSS Selectors |

- For single element access we can use id.
- For multiple element access we can use class.

```
<h2 class="example">A heading</h2>

<p class="example">A paragraph.</p>


<script>

document.querySelector("p.example").style.backgroundColo
r = "red";

</script>
```

# Traversing Elements

There are various methods in DOM which are used to get the information about a node parents, child & siblings elements.

| Method | Description |
|---|---|
| **firstChild** | To get the first child element of a specified element |
| **firstElementChild** | To get the first child with the Element node only |
| **lastChild** | To get the last child element of a specified element |
| **lastElementChild** | To get the last child with the Element node only |
| **children** | To get the child element with only the element node type |
| **parentNode** | To get the parent node of a specified node in the DOM tree |

# Traversing Elements (Contd.)

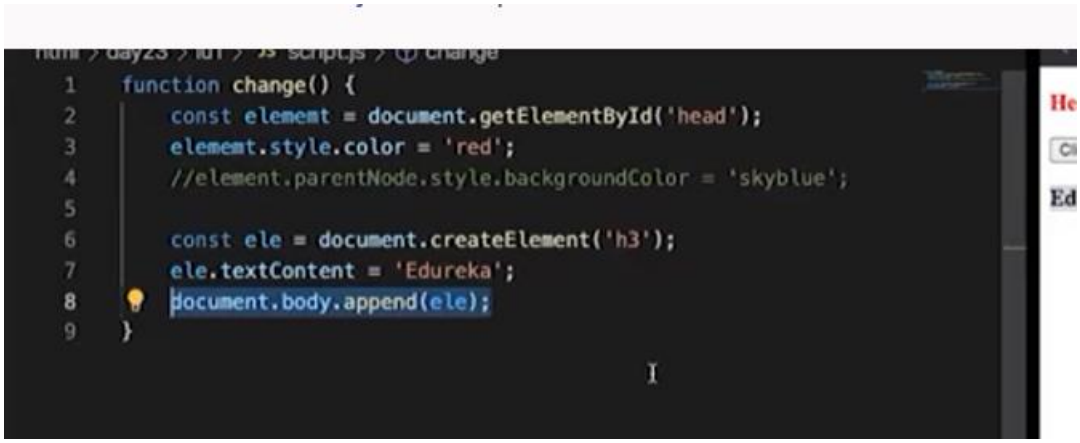
There are various methods in DOM which are used to get the information about a node parents, child & siblings elements.

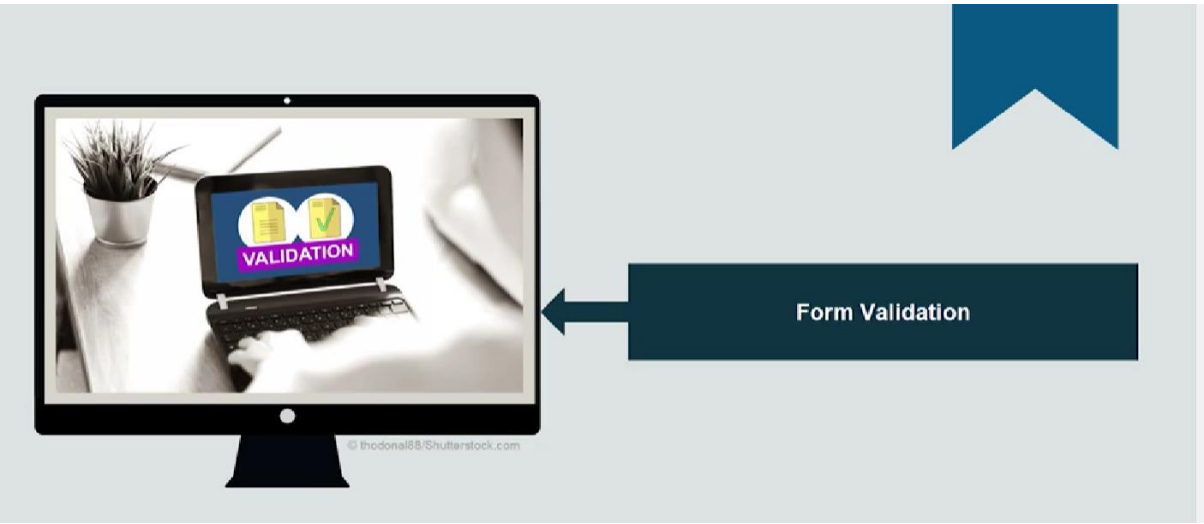
| Method | Description |
|---|---|
| **childNodes** | To get a live NodeList of child elements of a specified element |
| **nextElementSibling** | To get the next sibling of an element |
| **previousElementSibling** | To get the previous sibling of an element |

# Manipulating Elements

There are various methods in DOM which are used to manipulate the HTML element like creation, deletion, updating &, etc.

| Method | Description |
|---|---|
| **createElement()** | Create a new node |
| **appendChild()** | append a node to a list of child nodes of a specified parent node. |
| **textContent** | get and set the text content of a node. |
| **innerHTML** | get and set the HTML content of an element. |
| **append()** | insert a node after the last child node of a parent node. |
| **prepend()** | insert a node before the last child node of a parent node. |
| **removeChild()** | remove child elements of a node. |






# Form Validation

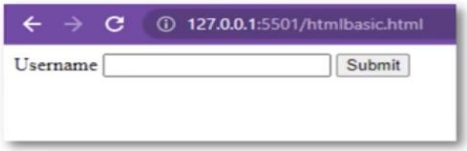
- Normally, Form validation occurs at the server side to verify the details given by the user.
- It is used to make sure that the entered data is correct and clean.
- JavaScript allows you to verify the data before submitting it to the server.
- In JavaScript, it occurs when the user pressed the submit button.
- Some of the commonly used validation tasks are:
  - To ensure all the required fields are filled.
  - To ensure that the user has entered a valid date
  - To ensure that the user has entered numbers in the numeric field.
  - To ensure that the user has entered a valid email.

# HTML form validation- Example

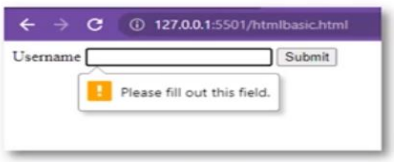
The attribute of HTML input tags will allow us to perform basic form validations.

```
<form>
  <label for="username">Username</label>
  <input type="text" id="username" required />
  <input type="submit" />
</form>
```

The required attribute of the HTML input tag makes the input field mandatory




Before clicking on submit button




After clicking on submit button without entering details of the user

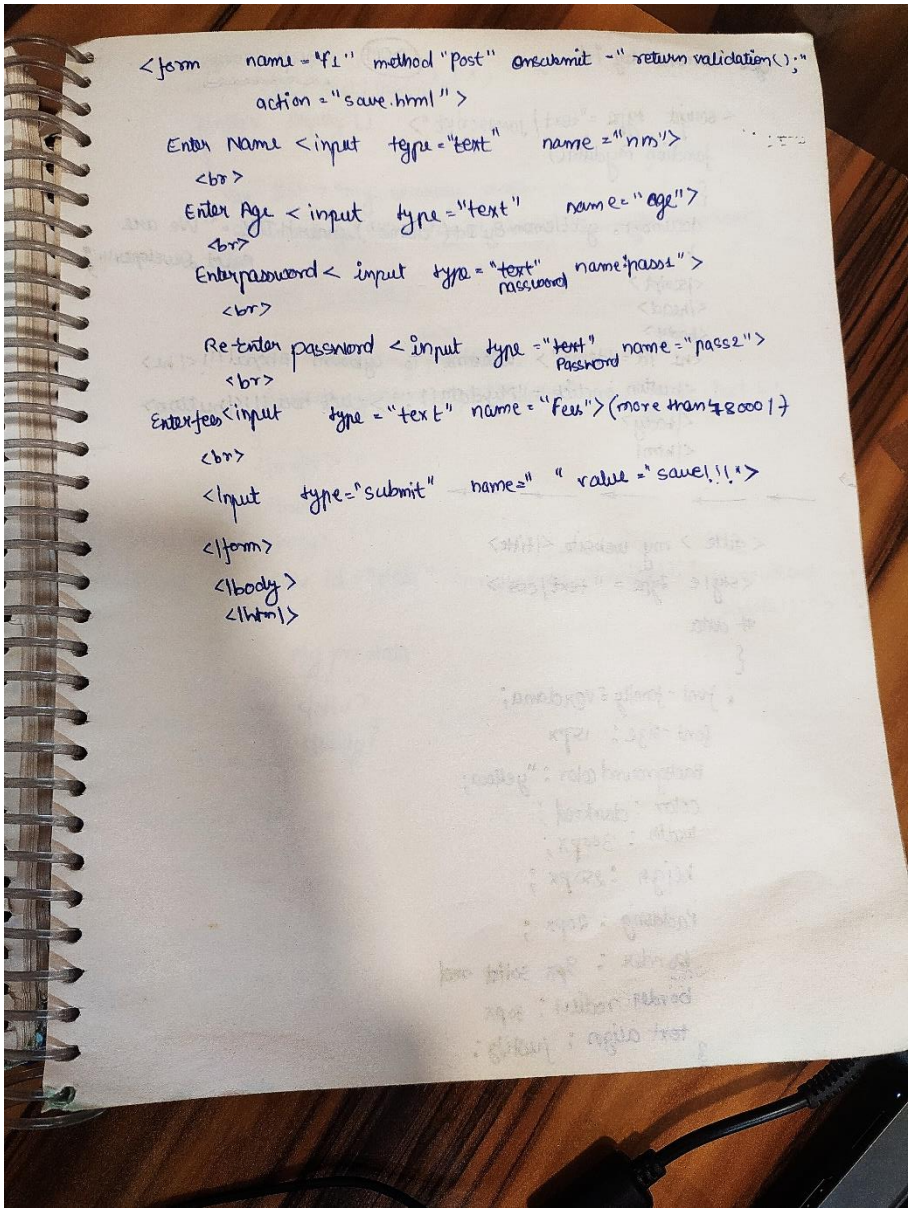
```
document.f1.age.value = " ";
return false;
}
if(pass1 == "")
{
alert("Please enter password
if (isNaN(age1))
{
alert("Please enter numeric age!!!");
document.f1.age.focus();
document.f1.age.value = " ";
return false;
}
if (pass1 == " ")
{
alert("Please enter password!!!");
document.f1.pass1.focus();
return false;
}
if (Pass2 == " ")
{
alert("Re-Enter Your password!!!");
document.f1.pass2.value = "";
document.focus();
return false;
}
if (pass1 != pass2)
{
```

Auto switch

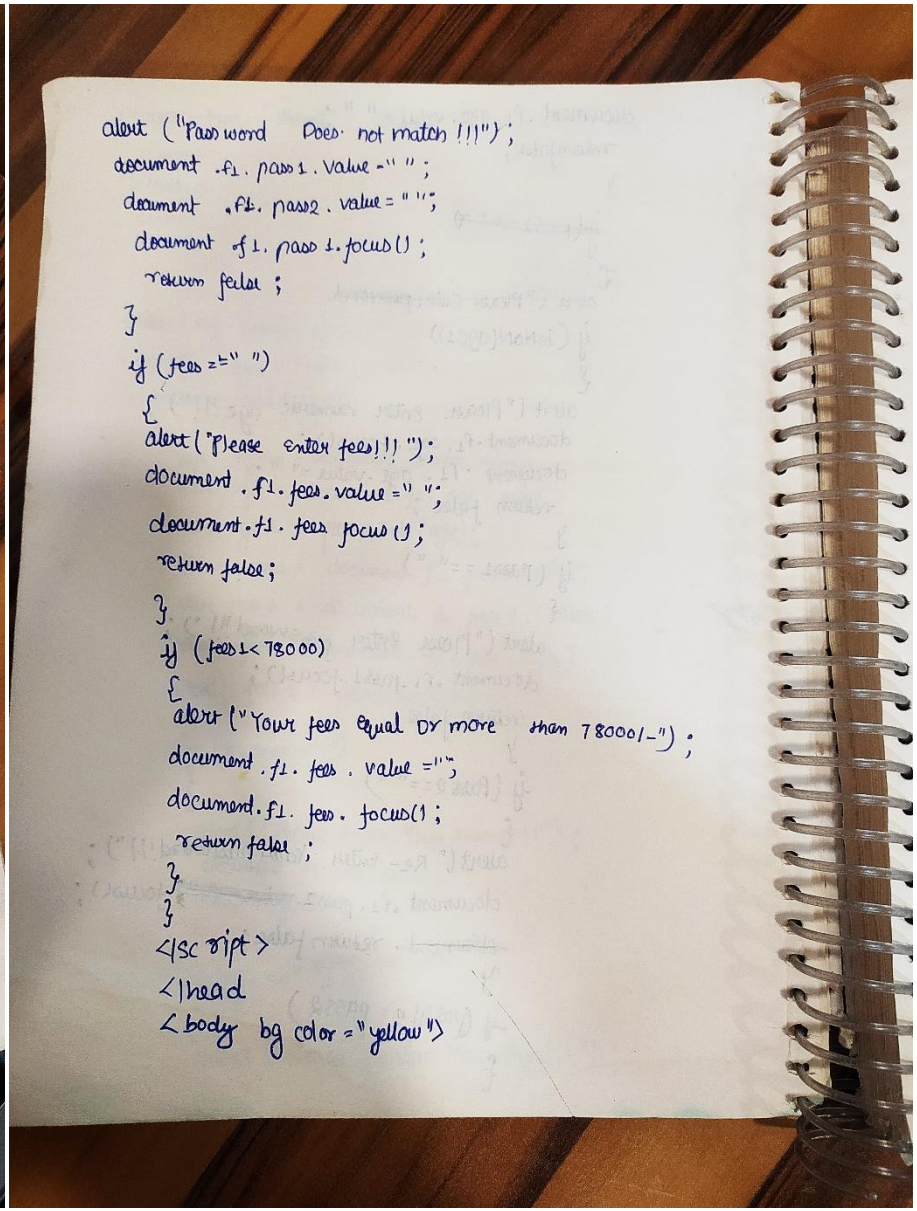
```
<input type="submit" name=" " value="Data save!!!">
</form>
</body>
</html>
```

28/07/22

```
<title>my website</title>
<script type="text/javascript">
function validation()
{
var mynm = document.f1.nm.value;
var age = document.f1.age.value;
var age1 = parseInt(age);
var pass1 = document.f1.pass1.value;
var pass2 = document.f1.pass2.value;
var fees = document.f1.fees.value;
var fees1 = parseInt(fees);
if (mynm == "")
{
alert("Please Enter Your name!!!");
document.f1.nm.focus();
return false;
}
if (age == "")
{
alert("Please enter Your Age!!!");
```



```
<form  name="f1" method "Post" onsubmit ="return validation();"
       action = "save.html" >
Enter Name <input  type="text"  name ="nm">
<br>
Enter Age <input  type="text"  name="age">
<br>
Enterpassword < input  type = "text"  name="pass1" >
                      password
<br>
Re-enter password < input  type ="text"  name ="pass2">
                          Password
<br>
Enterfees<input  type = "text" name = "fees">(more than 78000/-)
<br>
<Input  type="submit"  name=" " value ="save!!!">
</form>
</body>
</html>
```



```
alert ("Password  Does not match !!!");
document .f1. pass1. value ="";
document .f1. pass2. value = "";
document of1. pass1. focus();
return false ;
}
if (fees =="")
{
alert("Please enter fees!!!");
document . f1. fees. value ="";
document.f1. fees focus();
return false;
}
if (fees1< 78000)
{
alert ("Your fees Equal Or more than 78000/-");
document . f1. fees . value ="";
document. f1. fees. focus();
return false ;
}
}
</script>
</head
<body bg color = "yellow">
```

# ES5 and ES6

## Difference between ES5 and ES6

Both ES5 and ES6 are trademark scripting language specifications defined by ECMA(European Computer Manufacturers Association) international. We can find the major differences in the below table:

| Based on | ES5 | ES6 |
|---|---|---|
| **Edition** | Fifth edition released in 2009 | Sixth edition released in 2016 |
| **Datatypes** | Supports primitive data types( strings, Number, Boolean, null, and undefined) | In addition to the ES5 primitive datatypes, it supports new datatype(symbol) |
| **Variable Definition** | In ES5, we define the variable using only var keyword | In ES6, there are three ways to define a variable with var, let and const |
| **Arrow functions** | Not supported in ES5 | Supported in ES6 |
| **Default params** | Not supported | supported |
| **Template strings** | Not supported | supported |
| **Object destructing** | Not supported | supported |

## JavaScript ES6

The JavaScript ES6 (also called ECMAScript 2015) defines a new set of rules governing the syntax of JavaScript code.

| Feature | Description |
|---|---|
| **Let and const keywords** | These are used to create block-level variables |
| **Arrow Functions** | It reduced the length of the code for creating a function |
| **Classes** | These are the templates to create objects |
| **Default parameter values** | It allows to pass the argument through a function by default |
| **find() and findIndex method** | It helps you to find the element or index of an array |
| **Exponentiation (**)** | It raises the first operand to the power of the second operand |

## var vs let vs const

Following are the difference between var, let, and const:

| behavior | var | let | const |
|---|---|---|---|
| **hoisting** | Exhibits hoisting | Does not exhibits hoisting | Does not exhibits hoisting |
| **Scope** | Functional scope | Block scope | Block scope |
| **mutability** | mutable | mutable | immutable |

Arrow function

## Arrow Functions

- Arrow functions will reduce the length of the syntax.
- There is no binding of the this keyword in these functions.
- In arrow functions, this keyword always represents the object that defined the arrow function.
- If there is only one line of code, you can ignore the brackets and return the keyword of the function.

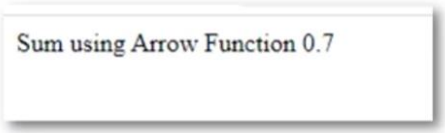
**Syntax:**


## Example of Arrow Function

**Code :**

```
<script>
    const div = (a, b) => a / b;
    document.write("Div using Arrow Function " +div(7, 10));
</script>
```

**Example :**




```
const a=(c,d)=>{
    document.write("hii");
    document.write(c + d);
  }

  a(10,20);

  const arr=(z,f)=>z/f;
  document.write(arr(23,12));
```

# Use Strict Mode

## JavaScript Use Strict

- use strict Directive is a literal expression in JavaScript.
- The sole purpose of using strict is to indicate that code should be executed in strict-mode.
- The term strict-mode says that you can not use non-declared variables, non-writeable properties, non-existing properties in JavaScript.
- Strict mode is declared by adding use strict; to the beginning of a script or a function.
- Strict mode encounters errors when you are writing wrong codes in JavaScript.

**Syntax :**

```
"use strict";
/*
    //Line of codes….
*/
```

```
"use strict";
va=12;
document.write(va);
```

# Default Params (or default parameter)

## Default Params

- Default parameter allows you to give a default value to your formal parameter of a function if no value/undefined is passed to the function.
- In JavaScript, function parameters are set to undefine as their default values. However, it's often helpful to set a different default value. This is where default parameters can help.

**Syntax**

```
function fnName(param1 = defaultValue1, ..., paramN = defaultValueN) {
    /*
    ...
     */
}
```

```
<script>

 const fun=(a,b,c)=>a+b+c;
 console.log(fun());
 console.log(fun(12));
 console.log(fun(12,15));
```

```
 console.log(fun(12,15,78));


 const sun=(a=0,b=0,c=0)=>a+b+c;
 console.log(sun());
 console.log(sun(12));
 console.log(sun(12,15));
 console.log(sun(12,15,78));


</script>
```

# Template string

## Template String

- They are a new kind of string literals introduced in ES6.
- Contains features like multi-line strings and string interpolation.
- They allow embedded expressions.
- Strings are enclosed within backtick (`) instead of single or double-quotes.

**example**

```
`string text`

`string text line 1
string text line 2`

`string text ${expression} string text`
```

```
<script>
 let str1="We are     ";
 let str2="full stack developer";
 const str4=`Hii ${str1},
 ${str2}`;
 document.write(str4);
</script>
```

# Spread Operator

## Spread Operator

- The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.
- The spread operator is often used in combination with destructuring.
- Application of Spread Operator
  - Copying Array or Object
  - Cloning Array or Object

**Syntax:**

```
[...iterableObj, items, ..,itemN];   //For array literals
[...iterableObj, "items",...,"itemN"];   //For string literals
function(...iterableObj);   //passing iterable object to function
{…obj}; //For array literals
```

```
<script>
    const arr1=[23,78,8,97,54];
    const arr2=[36,8,8,"deepak","sir"];
    const arr3=[...arr1,...arr2];
    document.write(arr3);
</script>
```

```
    let obj1={name:"deepak",
    sirname:"patel"};
    let obj2={
      age:23,
    course:"B.Tech"
};

    let obj3={...obj1,
      ...obj2,
      s:"bk"};
    document.write(obj3);
```

Rest Operator

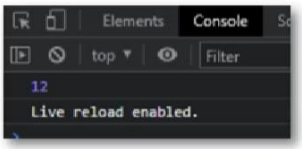
## Rest Operator

- The JavaScript spread operator (...) allows us to pass number N of the argument to a function
- 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 contain them. The dots mean to gather the remaining parameters into an array.

**Syntax :**

```
<script>
  var sum = (a, ...b) => {
    let sum = 0;
    for (let value of b) {
      sum = sum + value;
    }
    return sum + a;
  };
  let res = sum(7, 4, 1);
  console.log(res);
</script>
```

**Output :**




The rest operator will take an infinite number of values and store them into an array

# Object Destructuring

## Object Destructuring

- Object destructuring allows you to extract properties from objects and bind them to variables.
- There are two ways of object destructuring:
  - Using dot(.) operator.
  - By assigning properties of an object to variables.

**Syntax :**

```
Keyword variableName = objectName.property;

Keyword {propertyName,…,propertyN} =
                    objectName;
```

**Example:**

```
var rno = studObj.rollNo;

let {rollNo, name} = studObj;
```

```
let { firstName, isActive, isOnline } = obj;
console.log(firstName);
console.log(isActive);
console.log(isOnline);
```

```
let obj1={name:"deepak",
  sirname:"patel"};
```

```
    let obj2={
      age:23,
    course:"B.Tech"
};

 var {age,course} = obj2;
 console.log(age);
```

# Asynchronous JavaScript programing

a. For handling asynchronous programming-
    A. Callback function
    B. Promises
    C. Async and await
b. Asynchronous programming where action is not in sequence.
c. Function as argument in a function then it is known as callback function.
d. Function in which function is a higher order function.

## Call-back functions

- Callback is a function passed as an argument to another function.
- These functions are primarily used in JavaScript to create asynchronous code.
- JavaScript executes the code in sequential top-down order, but in some cases, we need the code to run in a specific order. Callback functions are used to implement this, known as asynchronous programming.
- The function that accepts other functions as arguments is known as a higher-order function, and it contains the logic for when the callback function gets executed.

```
<script>
  const sum=(a,b,callback)=>{
    setTimeout(()=>{
      printans();
      return console.log("hii");
    },1000);
    return console.log(a+b);
```

```
  }
const printans=()=>{
  console.log("bye");
}
sum(12,13,printans);

</script>
```

```
<script>
 const sum= (deepak)=>{ //deepak is a callback function
  setTimeout(()=>{
    deepak();
  },1000)
 };
 const deepak=()=>{
  return (console.log("done"));
 }
 sum(deepak);
</script>
```

# Promises

- Promises help to handle asynchronous operations in JavaScript.
- Promises makes it easy to manage and handle multiple asynchronous operations whereas callbacks can create callback hell resulting in unmanageable code.
- A promise is an object which can be returned synchronously from an asynchronous function.

**Code**

```
var promise = new Promise(function(resolve, reject) {
   // do a thing, possibly async, then…
    if (/* everything turned out fine */) {
        resolve("Stuff worked!");
    }
    else {
        reject(Error("It has broken"));
    }
});
```

# Promise (Contd.)

It will be in one of 3 possible states:


**Fulfilled:** When **resolve()** is called.


**Rejected:** When **reject()** is called.


**Pending:** Waiting for the asynchronous operation.

```
// callback function
const printSum = (s) => {
    console.log(s);
}

//sum(10, 20, printSum);

const prom = new Promise(
    (resolve, reject) => {
        setTimeout(() => {
            const s = 30 + 40;
            //resolve(s);
            //reject(false);
        }, 500);
    }
);

prom.then((res) => {
    console.log(res);
}).catch((err) => {
    console.log(err);
}).finally(() => {
    console.log('Over');
});
```

```
<script>
 const sum= new Promise((resolve,reject)=>{ //deepak is a callback function
  setTimeout(()=>{
```

```
    console.log(2+3);
    const s=30+40;
    resolve(s);
    reject(false);
  },1000)
 });
 const deepak=(s)=>{
  console.log(s);
 }
 sum.then((res)=>{
  console.log(res);
 }).catch((err)=>{
  console.log(err);
 }).finally(()=>{
  console.log('over');
 });
</script>
```

# Async and Await

Async/Await is a special syntax to work with promises more comfortably. It is easy to understand and use.

**Async Functions:** The async keyword can be placed before a function, like this:

**Code :**

```
<script>
    async function f()
    {
        return 1;
    }
    f().then(alert);
</script>
```

# Async/ Await in JavaScript (Contd.)

The word async before a function means that the function will always return a promise. Other values are wrapped in a resolved promise automatically. For instance, in the example given below, the function returns a resolved promise with the result of 1.

**Code :**

```
<script>
    async function f()
    {
        return 1;
    }
    f().then(alert);
</script>
```

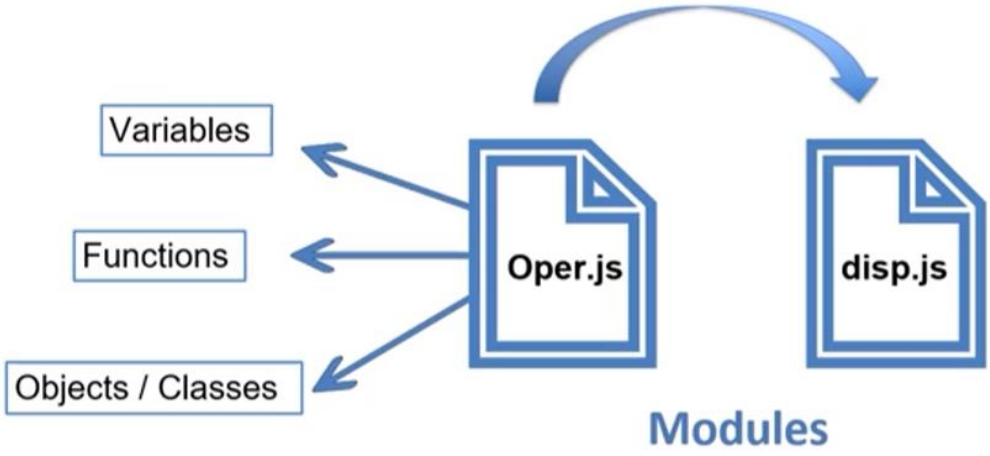
**Example :**

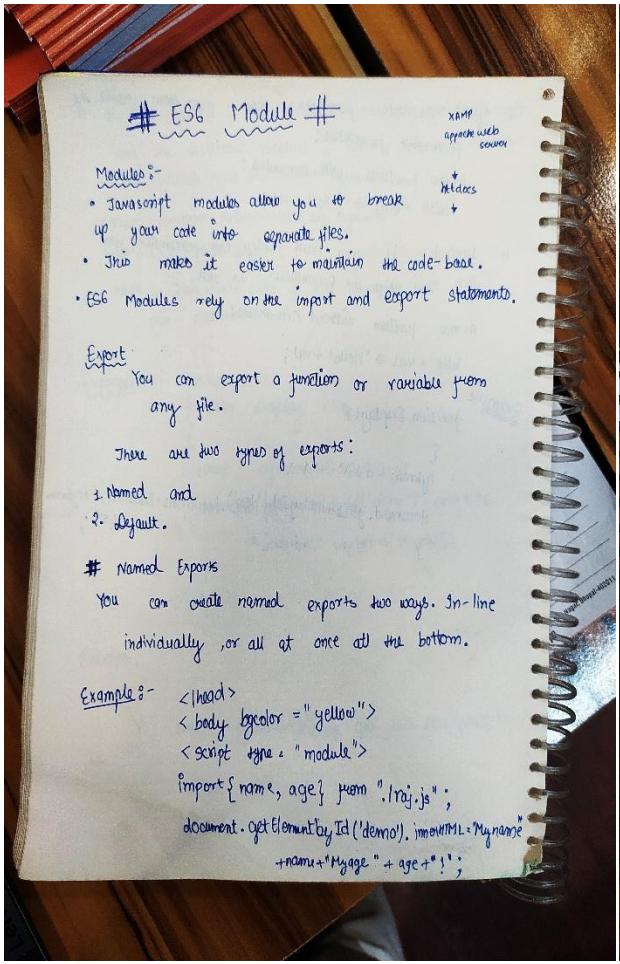

```
async function abc(){
   const sum= new Promise((resolve,reject)=>{ //deepak is a callback function
  setTimeout(()=>{
    console.log(2+3);
    const s=30+40;
    resolve(s);
    reject(false);
  },1000)
 });
  const response= await sum;
  console.log(response)
  return response;
};
 abc().then((res)=>{
  console.log(res);
 }).catch((err)=>{
  console.log(err);
 }).finally(()=>{
  console.log('over');
 });
```

# MODULES (import, export)

## Modules in JavaScript

- Modules are a piece of code in an independent file.
- In other words, we can say that a module is a function or group of similar functions grouped in a single file to accomplish a specific task.
- Sometimes, it becomes a very tedious task to manage chunks of code in an extensive program. Instead of writing them in one file, we can divide them into modules as per the functionality. Now, these modules are independent of each other & can be manageable.
- To use these modules in any other files/modules, two directives will be used:
  - **export** - used to make a function, variable, class, etc., accessible to other modules. It will make our code accessible to all other modules/files.
  - **import** - used to include any public code from other modules.
- * (asterisk) - used to import every public function from another module.





# ES6 Module #

XAMP
Apache web server
+
htdocs
+

Modules :-

- Javascript modules allow you to break up your code into separate files.
- This makes it easier to maintain the code-base.
- ES6 Modules rely on the import and export statements.

Export

You can export a function or variable from any file.

There are two types of exports:

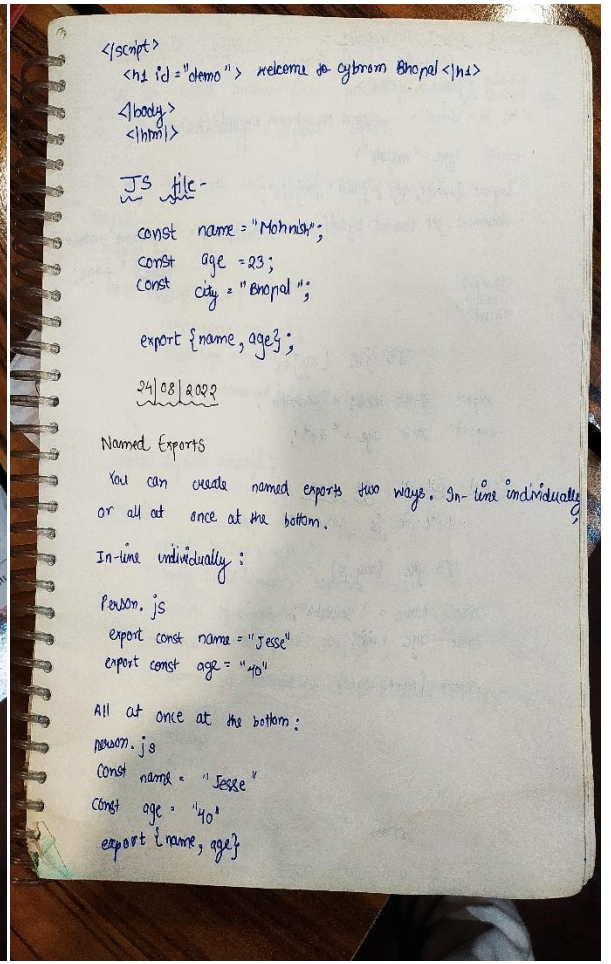
1. Named and
2. Default.

# Named Exports

You can create named exports two ways. In-line individually, or all at once at the bottom.

Example :-

```
<!head>
< body bgcolor = "yellow">
< script type = "module">
import { name, age} from "./raj.js";
document.getElementById('demo').innerHTML="My name"
+name+"My age " + age + "!";
```



```
</script>
  <h1 id = "demo"> welcome to cybrom Bhopal </h1>
</body>
</html>
```

JS file -

```
const name = "Mohnish";
const age = 23;
const city = "Bhopal";

export {name, age};
```

24/08/2022

Named Exports

You can create named exports two ways. In-line individually or all at once at the bottom.

In-line individually :

```
Person. js
export const name = "Jesse"
export const age = "40"
```

All at once at the bottom :

```
person. js
Const name = "Jesse"
Const age = "40"
export {name, age}
```

# Dejault Export :-

Let us create another files, named message.js, and use it for demonstrating dejault export.

You can only have one dejault export in a file.

Example

message.js

```
const message = () => {
    const name = "Jesse";
    const age = "40";
    return name + " is " + age + " years old.";
};

export dejault message;
```

Example :-

HTML file

```
<body bgcolor = "yellow">
<body h1 id = "demo"> Welcome </h1>
<script type="module">
  import name from "./raj.js";
  document.get elementbyId ('demo').innerHTML = "My name:" + name;
</script>
</body>
</html>
```

name or anything like MyNm

JS file

```
const name = "Monish";
const city = "Bhopal";
const age = 25;
export dejault name;
```

named export individually -

HTML file (index.HTML)

```
<body bgcolor = "yellow">
<h1 id = "demo"> welcome to cybrom Bhopal </h1>

<script type = "module">
  import {name, age} from "./raj.js";
  document.get element byId('demo').innerHTML = "My name" + name +
                                                "Mage:" + age;

</script>
</body>
</html>
```

JS file (raj.js)

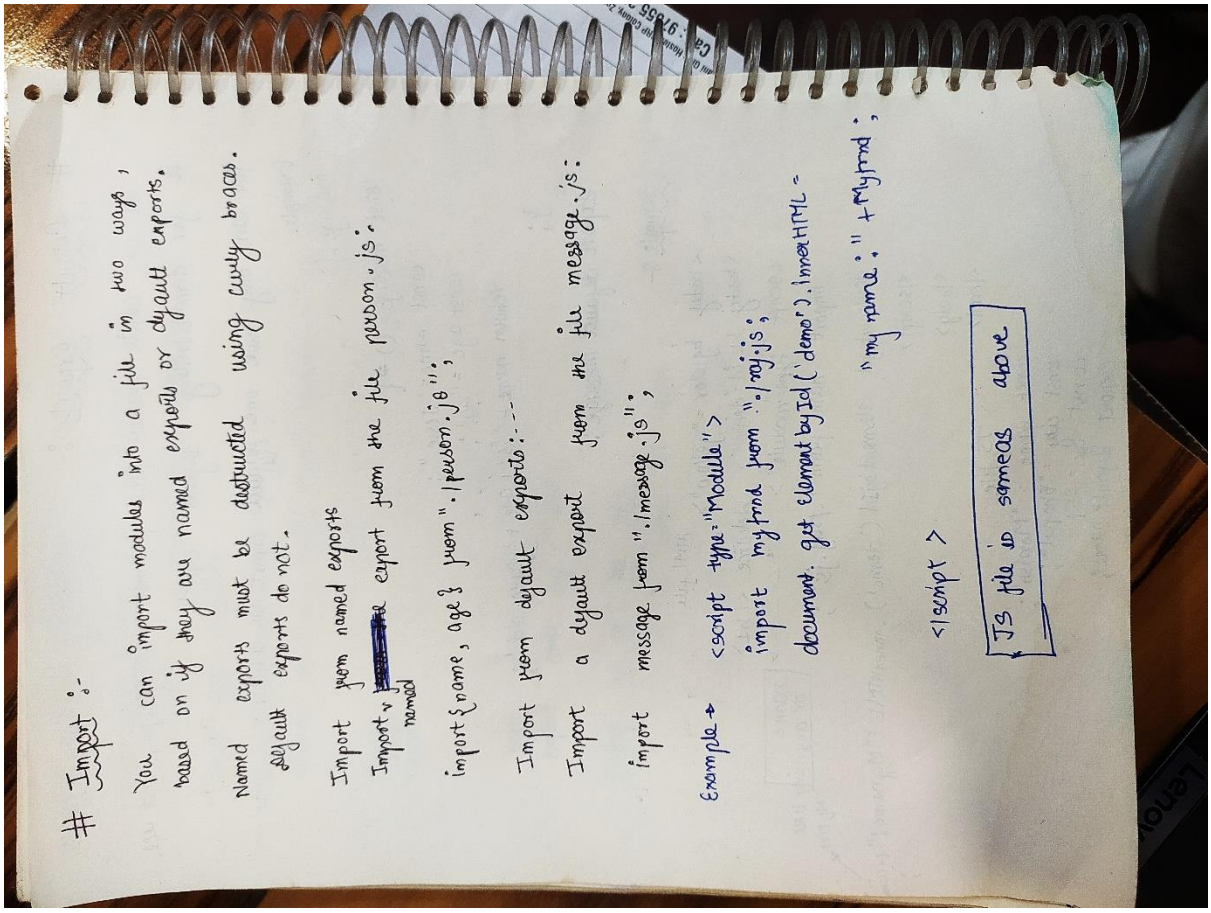
```
export const name = "sourabh";
export const age = "24";
```

Export all at the one.

HTML file is same as upper

JS file (raj.js)

```
const name = "sourabh";
const age = 22;

export {name, age};
```




# This keyword

a. A reference to the object that we are working with.
b. Every time you call a function javascript set the this keyword.
c. This is a dynamic so on runtime its value should be decided.

```
d. <script>
e.     const student={
f.      name:"deepak",
g.      lname:"patel",
h.      fullname:function(){
i.        console.log(this.name+" "+this.lname);
j.
k.      },
l.      inobj:{
m.        age:23,
n.        course:"full stack web development",
o.        address:function(){
p.          console.log(this.age+" "+this.course);
q.          console.log(this.name+" "+this.lname);//this is due to because
   this refers that only that scope
```

```
r.        }
s.      }
t.     }
u.
v.    student.fullname();
w.    console.log(student.inobj);
x.    student.inobj.address();
y.
z.
aa.</script>
```

# Cookies, Session and Local Storage


## Introduction to Cookies

- Cookies are a small chunk of data.
- It consists of a name and a value, stored by the user's web browser
- They can be used for authentication, session tracking by storing specific information such as name, password, last visited date, etc.

**Syntax**:

```
function WriteCookie() {
       cookievalue = “customer_name”
       document.cookie = "name=" + cookievalue;
       document.write ("Setting Cookies : " + "name=" + cookievalue );
    }
```

## Introduction to Cookies (Contd.)

- **Storing cookies:** You can create a cookie by assigning a string value to the document.cookie object.

```
document.cookie = "key1 = value1; key2 = value2;
```

- **Reading cookies:** The value of the document. cookie object returns a list of (key, value) pairs that can be stored in a variable. You can then use the split() function to break the string into keys and values.

```
var cookieList = document.cookie;
var cookieArray = cookieList.split(‘;’);
```

# Advantages of Cookies

- Cookies are stored on the user's computer, so no extra burden on the server.
- You can configure the cookies to expire when the browser session ends (**session cookies**), or they can exist for a specified length of time on the client's computer (**persistent cookies**).
- The cookies are stored on the client's hard disk, so if the server crashes, the cookies are still available.

a. Alert(document.cookie)
b. Document.cookie
c. Document.cookie=”name=Deepak”
d. See in console of storage section.
e. Session storage we see in upi password.
f. Local storage we see in Facebook, Instagram.

# Local Storage and Session Storage

- Local storage and session storage are web storage objects that allow the browser to save key/value pairs.
- They help data to survive a complete browser restart (local storage) and a page refresh (session storage).
- Web storage objects are not sent to the server with each request that enables them to store much more data. Some of the commonly used methods and properties of web storage objects are:

| Methods | Properties |
|---|---|
| setItem(key, value) | stores key/value pair |
| getItem(key) | get the value by key |
| removeItem(key) | remove the key with its value |
| clear() | delete everything |
| key(index) | get the key on a given position |
| length | the number of stored items |

# Local Storage vs Session Storage vs Cookies

| Local Storage | Session Storage | Cookies |
|---|---|---|
| Maximum 5 MB or 10 MB storage | Maximum 5 MB storage | Maximum 4 KB storage |
| Only client side reading is supported | Only client side reading is supported | Supports both client side and server side reading |
| The data stored is permanent, does not expire and remains stored on the user's computer until a user/ webapp asks the browser to delete it | Has the same lifetime as the browser tab in which the data got stored. When the tab is closed, any data stored is deleted | We can set the expiration time for a cookie by setting the '**expires**' attribute to a date and time |
| Stores data as JSON | Stores data as JSON | Stores data as String values |

# Advanced Function

## Immediately Invoked Function Expression

### IIFE Functions

- IIFE functions stands for Immediately Invoked Function Expression.
- These functions are used to create a function that can execute automatically after the function definition.
- These only run once when the interpreter runs into it.
- IIFE function can be assigned to a variable to store its return value, not the function definition.

**Syntax:**

```
(function (){
// Function Logic Here.
//return value
})();
```

Block of code

```
(function (){
  console.log("hii");
})();
```

### Concise Functions

- Concise functions are extended forms of arrow functions.
- Concise functions are a concise way of writing arrow functions, brief in terms of body & parameters.
- It does not have a function keyword & function body. Instead of it, there is only => between the parameters & body of a function & in the case of a single parameter it can be concise.

**Syntax:**

```
(param1, param2, . . .) => //operation on params
                OR
        param => operations
```

multiple params

single params

```
  const a =(a,b)=>a+b;
   console.log(a(12,23))
```

# Scheduling Functions

When you do not want a function to execute immediately on a function call, but after a predefined time interval on a function call, it is called "Scheduling a function call".

There are 2 methods/ ways to schedule a function call:

- **setTimeOut( ):** It runs a function once after the interval of time specified.
- **setInterval( ):** It runs a function repeatedly, starting after the interval of time, then repeating continuously at that interval.

---

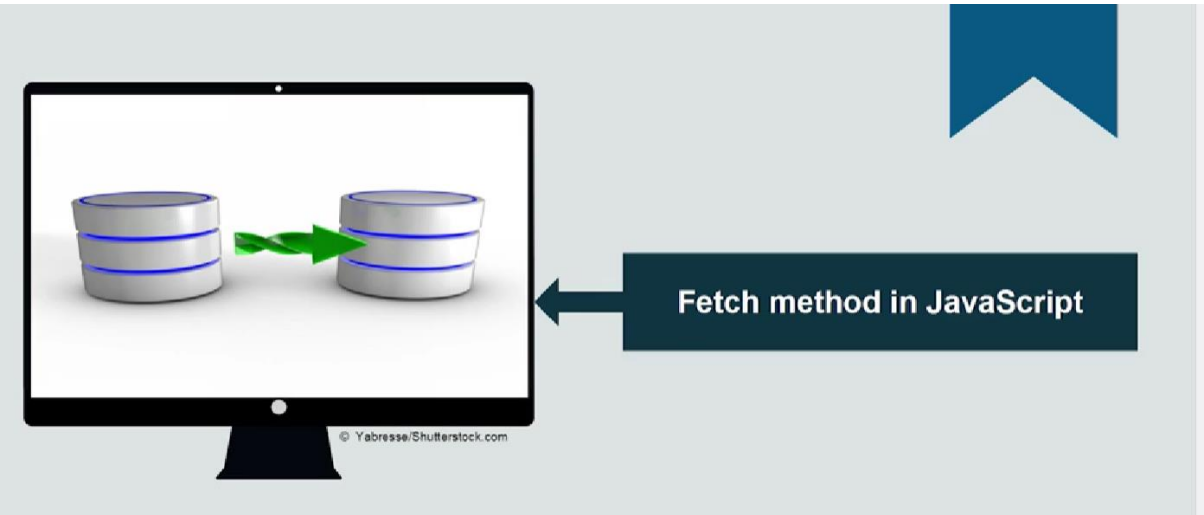
## settimeout

```
function sum(a,b){
    const s=23+24;
    console.log(s);
    console.log(a,"",b)
  }
  setTimeout(sum,2000,"this","we");
```

## setinterval

```
function sum(a,b){
    const s=23+24;
    console.log(s);
    console.log(a,"",b)
  }
  setTimeout(sum,2000);
  setInterval(sum,4000,"this","we");
```

# Fetch




a. The fetch() method starts the process of fetching a resource from a server. The fetch() method returns a promise that resolves to a response object.
b. we can fetch text file, apis, image etc.

## Fetch() Method

- fetch() method allows you to make HTTP requests to the server & those requests will be in form of APIs.
- fetch() method will return a promise.
- This method takes two parameters :
  - **URL:** It is a URL from where it fetches the data.
  - **{}:** It is an optional parameter used to perform what kind of operation it preforms.

**Syntax :**

```
fetch(url, {});
```

```
const a="https://api.publicapis.org/entries";


const user=fetch(a,{method:"GET"});
user.then(res=>res.json()).then(d=>{console.log(d);}).catch(err=>console.log(e
rr));
</script>
```

# Debugging JS Code in Browser

- In any programming language, Debugging is the process of examining the program, finding the errors, and fixing them.
- In JavaScript, we have different ways to debugging code :
  - Using console.log()
  - Using Debuggers
  - Using breakpoints

---

# Using Debuggers

- Debuggers are used to stop the execution of a piece of code.
- In JavaScript, the debugger keyword is used to stop the execution & it will invoke the debugging function.
- The debugger is available in almost all JavaScript engines.

**Example:**

```
<script>
    var a = 20,b = 30;
    const p = a > b ? "A is greater" : "B is greater";
     debugger;
    console.log(p);
</script>
```

---

```
<script>


const a="https://api.publicapis.org/entries";


const user=fetch(a,{method:"GET"});
user
.then(res=>res.json())
.then(d=>{
  debugger
  console.log(d);})
.catch(err=>console.log(err));
</script>
```

## Settings Breakpoints

- Breakpoints are used to examine the working of functionality in our code.
- JavaScript allows you to set multiple breakpoints.
- JavaScript will stop executing at each breakpoint and let you examine the values. Then, you can resume the execution of the code.

**Example:**

```
<script>
    var a = 20,b = 30;
    const p = a > b ? "A is greater" : "B is greater";
    console.log(p);
</script>
```

---

It is totally on practical approach

## Window Object (contd.)

Some of the commonly used window object properties are as shown below:

| Method | Description |
|---|---|
| innerHeight | Returns inner height of the window including scrollbars |
| innerWidth | Returns inner width of the window including scrollbars |
| closed | Returns a Boolean true if a window is closed |
| history | Returns the history object for window |
| location | Returns the location object for window |
| navigator | Returns the navigator object for window |
| screen | Returns the screen object for window |

---

Write on console-

window.location, window.innerHeight etc.

# Function Currying and function borrowing

## Function Currying

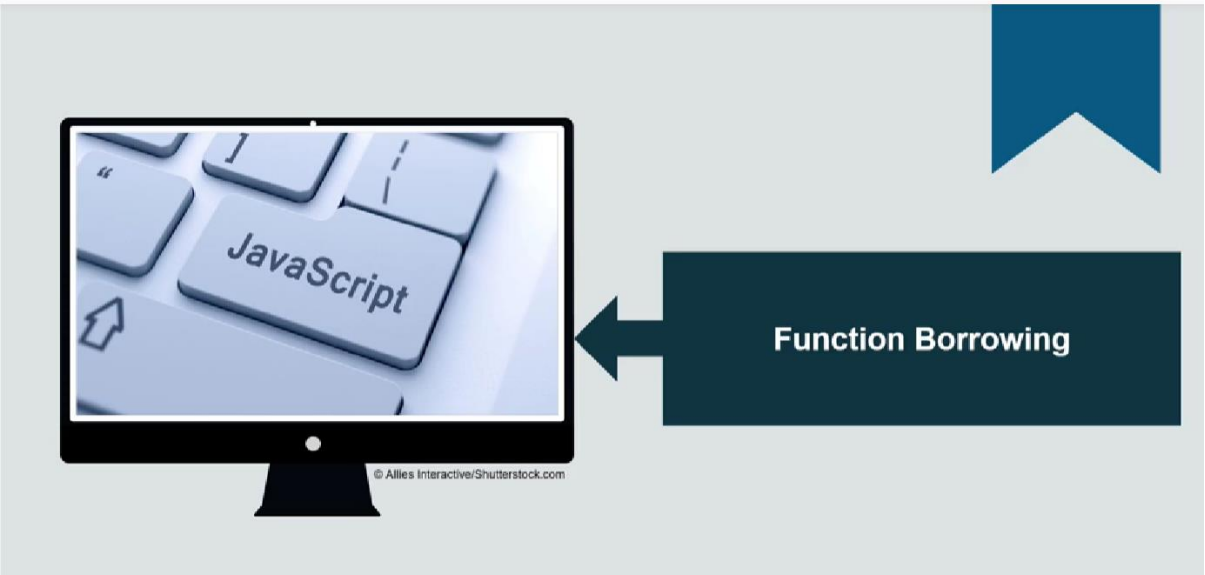
- Function currying is an advanced approach to work with functions.
- Currying is a process in which we can transform multiple argument functions into a nesting of sequential functions.
- In currying, a function call such as `function name (a,b,c,d)` is transformed into different calling format `function name (a)(b)(c)(d)`.
- In simpler terms, a function body is wrapped into another function's body. The inner function will return some value to the outer function, and the inner function's variable will have access to the outer function's variable; however, vice-versa is not true.

## Function Currying (Code Example)

**Code example (script)**

```
<script>
    function sum(a)
    {  console.log
        return function (b)
        {
            return function (c)
            {
                console.log(`a: ${a} b: ${b} c:${c}`)
                console.log(a+b+c);
            }
        }
    }
    sum(10)(20)(30)
</script>
```

```
function calculateVolume(length) {
     return function (breadth) {
         return function (height) {
             return length * breadth * height;
         }
     }
 }
 console.log(calculateVolume(4)(5)(6));
```




a. Function borrowing is also a method borrowing

```
b. <script>
c. const obj1={
d.   age:23,
e.   course:"B.Tech",
f.   name:"Deepak Patel",
g.   information:function(){
h.     console.log(this.age +" c c  "+this.name);
i.   }
j. }
k. const obj2={
l.   age:22,
m.   course:"B.Tech",
n.   name:"Ravendra Patel"
o. }
p.
q. obj1.information();
r. obj1.information.call(obj2);
s. </script>
```

```
<script>
const obj1={
  age:23,
  course:"B.Tech",
  name:"Deepak Patel",
  information:function(a){
    console.log(this.age +" c c  "+this.name+"    "+a);
  }
}
const obj2={
  age:22,
```

```
  course:"B.Tech",
  name:"Ravendra Patel"
}

obj1.information();
obj1.information.call(obj2,"a");
</script>
```

# apply() Method in JavaScript

`apply()` is a predefined JavaScript method that allows an object to use the method of another object.

**Code example (script)**

```
<script>
      let student = {
        firstName: "Tony",
        lastName: "Stark",
        printStudent: function (gender, age) {
          console.log(this.firstName, this.lastName, gender, age);
        },
      };
      let student1 = {
        firstName: "Akshay",
        lastName: "kumar",
      };
      student.printStudent("male", 22);
      student.printStudent.apply(student1, ["male", 18]);
    </script>
```

`student1` can use student function. Here, `apply()` function allows us to pass multiple parameters in array format

```
const obj1={
  age:23,
  course:"B.Tech",
  name:"Deepak Patel",
  information:function(a,sno){
    console.log(this.age +" c c  "+this.name+"   "+a +"   "+sno);
  }
}
const obj2={
  age:22,
  course:"B.Tech",
  name:"Ravendra Patel"
}

obj1.information();
obj1.information.apply(obj2,["a",90]);
</script>
```

# bind() Method in JavaScript

bind() is a predefined JavaScript method that allows an object to use the method of another object.

**Code example (script)**

```
<script>
    let student = {
      firstName: "Tony",
      lastName: "Stark",
      printStudent: function (gender, age) {
        console.log(this.firstName, this.lastName, gender, age);
      },
    };
    let student1 = {
      firstName: "Akshay",
      lastName: "kumar",
    };
    student.printStudent("male", 22);
    let CallLater = student.printStudent.bind(student1, "male", 22);
    CallLater();
</script>
```

**Code output**

```
Tony Stark male 22
Akshay kumar male 22
```

bind() method allows to call the function later

```
<script>
const obj1={
  age:23,
  course:"B.Tech",
  name:"Deepak Patel",
  information:function(a,sno){
    console.log(this.age +" c c  "+this.name+"   "+a +"   "+sno);
  }
}
const obj2={
  age:22,
  course:"B.Tech",
  name:"Ravendra Patel"
}

obj1.information();
const show=obj1.information.bind(obj2,"a",90);
show();
</script>
```

a. Difference between call and apply is in call pass argument and in apply pass array.
b. Difference between call and bind is we can store value of called function in variable and use according to their use by bind method.

# JavaScript Event Handling

## Event Handling

- In JavaScript, events are recognized as the change in a state of an object.
- There are various events that can be triggered either by the user or by the browser automatically based on the code.
- Process of responding to these events is called Event Handling .
- Therefore, JavaScript handles HTML events via Event Handlers.
- Some examples of HTML events are:
  - Action performed by the browser such as page loading finished, page loading started, etc.
  - Action performed by the mouse such as click, double click, etc.
  - Action performed on the form such as submit, change, etc.

## Different Event Handlers

Some of the commonly used for event handlers are as shown below:

| Event handler | Description |
|---|---|
| **onclick** | when a user clicks the mouse button on a certain HTML element |
| **ondblclick** | when a user double clicks the mouse button on a certain HTML element |
| **onmouseover** | when a user moves the cursor over a certain element |
| **onmouseout** | when a user leaves the cursor from an element |
| **onmousedown** | when a user presses a mouse button over an element |
| **onmouseup** | when a user releases a mouse button over an element |
| **onmousemove** | when a user moves the mouse |
| **onKeyPress** | when a user press an alphabetic, numeric or punctuation key |

## Different Event Handlers (Contd.)

Some of the commonly used for event handlers are as shown below:

| Event handler | Description |
|---|---|
| **onkeydown & onkeyup** | when a user has pressed down or released the key. Even if that key does not produce a character value. |
| **onfocus** | when a user focuses on certain element |
| **onsubmit** | when a user submits the form |
| **onchange** | when a user modifies the value of a form element |
| **onload** | when the browser finished the loading of a page |
| **onunload** | when the browser unloads the page |

# Event Bubbling

- In JavaScript, Event bubbling is a method of event propagation in the HTML DOM API.
- It works on a principle that whenever an event happens on an element, it first runs the handlers on it, then on its parent, or up to the containing elements in the hierarchy.
- Suppose if element 'X' consists of element 'Y,' and element 'X' is clicked.
- Then click event will trigger element 'X,' and it will bubble up and trigger it for element 'Y' as well.

```
addEventListener(EventType, Action, userCapture)
```

- **EventType:** It is a type of event which will trigger the action.
- **Action:** It is a task to be performed when the event is triggered.
- **userCapture:** It is a Boolean value that indicates the phase of an event. By default, its value is set to false, which represents it is in the bubbling phase.

# Event Trickle Down

- In JavaScript, Event Trickle down is a method of event propagation in the HTML DOM API.
- It works on a principle that whenever an event happens on an element, it first runs the handlers on its parents or ancestors, then goes to its child or down to the elements in the hierarchy.
- Suppose, if element 'X' consists of element 'Y', and element X is clicked.
- Then click event will trigger element 'Y' and it will trickle down and trigger it for element 'X' as well.

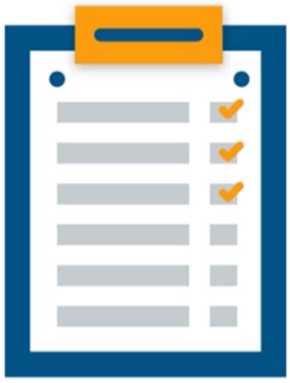
```
addEventListener(EventType, Action, userCapture)
```

- **EventType:** It is a type of event which will trigger the action.
- **Action:** It is a task to be performed when it event is triggered.
- **userCapture:** It is a Boolean value that indicates the phase of an event. Its value can be set to true which represents trickle down phase.

# OOPs Concept

## Contents





## OOPS in JavaScript

- Object is an entity that represents any real-life object.
- It is an instance of a class.
- It contains property and methods.
- JavaScript is full of objects because almost every element here is an Object, whether it is a function, array, or string.

**Code:**

```
class dog{
      color;
      breed;
      run(){
            console.log("Running")
        }
       bark(){
              console.log("Barking")
          }
        }

const D1=new dog()
```

Property

Method

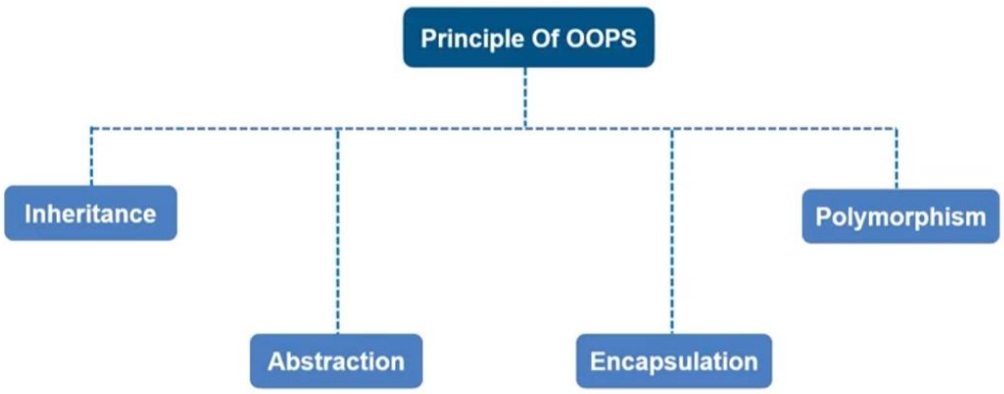
# Constructor

- Constructor is a special method of a class that's called automatically whenever its instance is created.
- It enables to provide any custom initialization that must be done before any other methods can be called.

**Syntax:**

```
class dog {
     constructor() {
     --------
            }
     }
const D1=new dog()
```

# Principle of OOPS



# Inheritance

- In JavaScript, inheritance is the mechanism in which one class inherits the attributes and methods of another class.
- The class whose properties and methods are inherited is known as the parent class.
- The class that inherits the properties from the parent class is the child class.
- extends keyword is used to represent parent class.
- In these cases, an object will call the constructor of its own class first, then it will go for its parent.

Syntax:

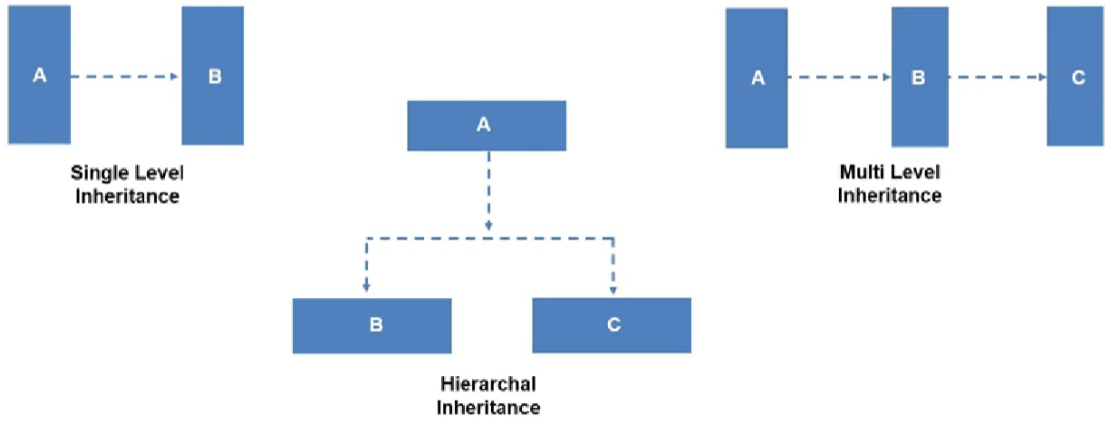
```
class Animal {
              ------
              ------
}

class Wild extends
Animal {
         -----
}
```

Parent class

Child class

# Types of Inheritance

Following are three types of inheritance supported by JavaScript:



# Prototypal Inheritance

Prototypal inheritance is an important concept of inheritance that adds new properties and methods to an existing object. This inheritance makes use of prototype objects.

```
class Dog {
  constructor(color, speed) {
    this._speed = speed;
    this._color = color;
  }
}

class Husky extends Dog {
  constructor(color, speed, owner){
    super(color, speed);
    this._owner = owner;
  }
}
```

```
  showInfo() {
   console.log("color:“ + this._color + "owner:” +
           this._owner +"speed:"+ this._speed);
  }
}
let Dog = {
  color: "white"
};
let Husky = {
  owner:'ABC'
};
Husky.__proto__ = Dog;
console.log( Husky.owner+" is a owner of "+
                Dog.color+" Husky");
```

**Code**

# Abstraction

- An abstraction is a mechanism to hide the implementation details by showing only the functionality to the users.
- In simpler terms, it disregards unnecessary details and shows only the required one.
- It helps to define the real-life components in different complex data types.

**Code:**

```
class Demo {

     setter(a, b) {
           this.x = a;
           this.y = b;
           this.z = this.x + this.y
     }
     print() {
     console.log("Addition: " + this.z);
  }
}
const obj = new Demo()
obj.setter(18, 10)
obj.display()
```

**Output:**

```
Addition: 28
```

# Encapsulation

- In JavaScript, Encapsulation is a process of binding properties with methods.
- It provides better control and validation of data.
- To achieve an encapsulation:
  - Properties of the object should be private.
  - Objects should have setter and getter methods to set and get the data respectively.

# Example of Encapsulation

**Code**

```
class Add {
    #a
    #b
    #c
    set(x,y) {
        this.#a = x
        this.#b = y
        this.#c = this.#a+ this.#b
    }
    display() {
        console.log("Addition: " + this.#c)
    }
}
const obj = new Add()
obj.set(9,4)
obj.display()
```

**Output**

```
Addition: 13
```

# Polymorphism

- The word “poly” means many and the word “morphism” means forms.
- In JavaScript, polymorphism is a method that helps to perform a single action in different forms.
- In simpler terms, it’s an ability to call the same method on different objects.
- It takes advantage of inheritance to design the objects in such a way that they can override any behaviors of its parent class.
- For example a man can have distinct characteristics at the same time like he is a father, a husband, a son, and an employee at the same time.

# Method Overriding

In JavaScript, method overriding is a technique that replaces the implementation of the parent class with its own implementation by providing a method that has the same name.

**Code:**

```
class Animal {

      print() {
            console.log("This is Animal")
      }
}

class Dog extends Animal {

      print() {
            console.log("This is Dog")
      }
}
let obj = new Dog()
obj.print()
```

# Debounce

# Understanding Debounce in JavaScript

- Debounce is a method of programming to improve the browser’s performance.
- If a browser has time-consuming functions and if those functions are frequently/repeatedly getting invoked, then it will affect the browser’s performance.
- Debouncing is a concept used to ensure that the time-consuming tasks do not fire so often, which helps improve the efficiency of the webpage performance.
- In other words, debounce is a way of programming that limits the rate at which a time-consuming function gets called.

#take practical example by going into the console and network section opening any website like amazon.

# #throatlling