JAVASCRIPT OPERATORS

zero fill right shift

It is same as a bitwise right shift the only difference is that overflowing bits are discarded.

```
console.log(6 >>> 2) ;  // 6 : 110 2: 010 Ans --> 1
console.log(6 >> 2) ;
```

COMMA OPERATOR

When a comma operator is placed in an expression, it executes each expression and returns the rightmost expression.

```
function Func1() {
  console.log('one');
  return 'one';
}
function Func2() {
  console.log('two');
  return 'two';
}
function Func3() {
  console.log('three');
  return 'three';
}
// returns the rightmost returned value
let x = (Func1(), Func2(), Func3());
console.log(x); // 'three'
```

UNARY OPERATOR (typeof)

It returns the operand type, The possible types that exist in javascript are undefined, Object, boolean, number, string, symbol, and function.

```
let a = 17;
let b = "GeeksforGeeks";
let c = "";
let d = null;
console.log("Type of a = " + (typeof a)); // number
console.log("Type of b = " + (typeof b)); // string
console.log("Type of c = " + (typeof c)); // string
console.log("Type of d = " + (typeof d)); // object
console.log("Type of e = " + (typeof e)); // undefined
```

DELETE OPERATOR

```
let emp = {
        firstName: "Raj",
        lastName: "Kumar",
        salary: 40000
}
console.log(delete emp.salary);
console.log(emp);
```

JAVASCRIPT RELATIONAL OPERATOR

JavaScript Relational operators are used to compare its operands and determine the relationship between them.

They return a Boolean value (true or false) based on the comparison result.

'IN' OPERATOR

```
let car = {
   make: 'Toyota',
   model: 'Camry',
   year: 2022
};
console.log("make" in car) // true
// 'INSTANCEOF' OPERATOR
function Car(make, model, year) {
   this.make = make;
   this.model = model;
   this.year = year;
} ;
let mycar = new Car('Toyota', 'Camry', 2022)
console.log(mycar instanceof Car )
```

Nullish Coalescing Assignment (??=) Operator in JavaScript

This is a new operator introduced by javascript. This operator is represented by x ??= y and it is called Logical nullish assignment operator. Only if the value of x is nullish then the value of y will be assigned to x that means if the value of x is null or undefined then the value of y will be assigned to x.

```
let x = 12 ; let y = null ;
y ??= 13
console.log("x = ", x, " :  y : ", y)
let arr = [1, 2, "apple", null, undefined, []] ;
console.log(arr) ;
arr.forEach((item, index)=> {
  arr[index] ??= "GFG" ;
})
console.log(arr) ; ----------
```

```
[ 1, 2, 'apple', null, undefined, [] ]
[ 1, 2, 'apple', 'GFG', 'GFG', [] ]
```

JavaScript yield Operator

JavaScript yield operator is used to delegate control of a generator function to another generator function or iterable object. It allows you to yield the values of an inner generator or iterable object from within an outer generator function. This operator is useful in a variety of scenarios, such as when working with iterators, asynchronously processing data, or implementing coroutines.

```
function* innerGenerator() {
   yield 1;
   yield 2;
   yield 3;
}
function* outerGenerator() {
   yield* innerGenerator();
}
const generator = outerGenerator();
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
```

PIPELINE OPERATOR

The JavaScript Pipeline Operator (|>) is used to pipe the value of an expression into a function. This operator makes chained functions more readable. This function is called the (|>) operator and whatever value is used on the pipeline operator is passed as an argument to the function. The functions
are placed in the order in which they operate on the argument.

```
function add(x) {
  return x + 10 ;
}
function subtract(x) {
  return x - 5 ;
}
// Without pipeline operator
let val1 = add(subtract(add(subtract(10))));
console.log(val1) ;
let val2 = 10 |> subtract |> add |> subtract |> add;
console.log(val2) ;
```

LOOP

FOR-IN LOOP

JS for-in loop is used to iterate over the properties of an object. The for-in loop iterates only over those keys of an object which have their enumerable property set to “true”.

```
let myObj = { x: 1, y: 2, z: 3 };
```

```
for(let key in myObj) {
   console.log(key, ":", myObj[key])
}
```

FOR-OF LOOP

JS for-of loop is used to iterate the iterable objects for example – array, object, set and map.
It directly iterate the value of the given iterable object and has more concise syntax than for loop.

```
const iterable = new Map([
   ["a", 1],
   ["b", 2],
   ["c", 3],
]);
for (const entry of iterable) {
   console.log(entry);
}  ------------------------
['a', 1]
 ['b', 2]
['c', 3]
for (const [key, value] of iterable) {
   console.log(value);
}   ------------------------
1
2
3
```

FOR-EACH LOOP

A forEach loop is a method on arrays that executes a function for each element in the array.

```
const arr = [1, 2, 3];
arr.forEach(val => val*2);
console.log(arr)
```

MAP LOOP

A map loop is a method on arrays that creates a new array by executing a function on each element in the array.

```
const arr = [1, 2, 3];
arr1 = arr.map(val => val*2);
console.log(arr) ;
console.log(arr1) ;
str = String("What the fuck is going on here!")
str.replace('fuck', 'duck')
console.log(str)
console.log(str.includes("What"))
```

Introduction to 'OBJECT ORIENTED PROGRAMMING' in JavaScript

There are certain features or mechanisms which make a Language Object-Oriented like:
Object Classes Encapsulation Abstraction Inheritance Polymorphism
The object can be created in two ways in JavaScript:
1. Object Literal
2. Object Constructor

Example: Using an Object Literal.

```
let person = {
first_name: 'Mukul',
last_name: 'Latiyan',
//method
getFunction: function () {
        return (`The name of the person is
                ${person.first_name} ${person.last_name}`)
},
//object within object
phone_number: {
                mobile: '12345',
                landline: '6789'
}
}
console.log(person.getFunction()) ;
console.log(person.phone_number.landline) ;
```

Example: Using an Object Constructor.

```
function person(first_name, last_name) {
  this.first_name = first_name;
  this.last_name = last_name;
}
// Creating new instances of person object
let person1 = new person('Mukul', 'Latiyan');
let person2 = new person('Rahul', 'Avasthi');

console.log(person1.first_name);
console.log(`${person2.first_name} ${person2.last_name}`);
```

Object.create() Method

Note: The JavaScript Object.create() Method creates a new object, using an existing object as the prototype
of the newly created object.

```
const coder = {
   isStudying: false,
   printIntroduction: function () {
```

```
        console.log(`My name is ${this.name}. Am I
            studying?: ${this.isStudying}.`)
    }
}
const me = Object.create(coder);
me.name = 'Mukul';
me.isStudying = true;
me.printIntroduction();
```

`JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax is not introducing a new object-oriented inheritance
model to JavaScript. JavaScript classes provide a much simpler and clearer syntax to create objects and deal with inheritance. `

```
class Vehicle {
    constructor(name, maker, engine) {
        this.name = name;
        this.maker = maker;
        this.engine = engine;
    }
    getDetails() {
        return (`The name of the bike is ${this.name}.`)
    }
}
// Making object with the help of the constructor
let bike1 = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
let bike2 = new Vehicle('Ninja', 'Kawasaki', '998cc');

console.log(bike1.name);    // Hayabusa
console.log(bike2.maker);   // Kawasaki
console.log(bike1.getDetails());
```


Abstraction

Abstraction means displaying only essential information and hiding the details.
Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Encapsulation

The process of wrapping properties and functions within a single unit is known as encapsulation.

ENCAPSULATION example

```
class person {
```

```
    constructor(name, id) {
        this.name = name;
        this.id = id;
    }
    add_Address(add) {
        this.add = add;
    }
    getDetails() {
        console.log(`Name is ${this.name},
        Address is: ${this.add}`);
    }
}
let person1 = new person('Mukul', 21);
person1.add_Address('Delhi');
person.sex = "M" ;
person1.getDetails();
```

Sometimes encapsulation refers to the hiding of data or data Abstraction which means representing essential features hiding the background detail. Most of the OOP languages provide access modifiers to restrict the scope of a variable, but there are no such access modifiers in JavaScript, there are certain ways by which we can restrict the scope of variables within the Class/Object.

ABSTRACTION example

```
function person(fname, lname) {
        let firstname = fname;
        let lastname = lname;

        let getDetails_noaccess = function () {
                return (`First name is: ${firstname} Last
                        name is: ${lastname}`);
        }

        this.getDetails_access = function () {
                return (`First name is: ${firstname}, Last
                        name is: ${lastname}`);
        }
}
let person1 = new person('Mukul', 'Latiyan');
console.log(person1.firstname);
console.log(person1.getDetails_noaccess); // undefined
console.log(person1.getDetails_access()); // undefined
```

INHERITANCE

It is a concept in which some properties and methods of an Object are being used by another Object. Unlike most of the OOP languages where classes inherit classes, JavaScript Objects inherit Objects i.e. certain features (property and methods) of one object can be reused by other Objects.

INHERITANCE example

```
class person {
   constructor(name) {
      this.name = name;
   }
   // method to return the string
   toString() {
      return (`Name of person: ${this.name}`);
   }
}
class student extends person {
   constructor(name, id) {
      // super keyword for calling the above
      // class constructor
      super(name);
      this.id = id;
   }
   toString() {
      return (`${super.toString()},
      Student ID: ${this.id}`);
   }
}
let student1 = new student('Mukul', 22);
console.log(student1.toString()) ;
```


Note: The Person and Student objects both have the same method (i.e toString()), this is called Method Overriding. Method Overriding allows a method in a child class to have the same name(polymorphism) and method
signature as that of a parent class.

POLYMORPHISM:

Polymorphism is one of the core concepts of object-oriented programming languages. Polymorphism means the same function with different signatures is called many times. In real life, for example, a boy at the same time may be a student, a class monitor, etc. So a boy can perform different operations at the same time. Polymorphism can be achieved by method overriding and method overloading

JavaScript JSON Objects

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. In JavaScript, JSON is often used to represent data in the form of objects.

```
let myOrder, i;
// Object is created with name myOrder
myOrder = {
   "name_of_the_product": "Earbuds",
   "cost": "799",
   "warranty": "1 year "
};
// Accessing for particular detail
// from object myOrder
i = myOrder.name_of_the_product;
// It prints the detail of name
// of the product
console.log(typeof myOrder);
console.log(i);
```

1. JSON.parse()parses a JSON string, constructing the JavaScript value or object described by the string.
2. In the example, jsonString is a JSON-formatted string.
3. JSON.parse(jsonString) converts the JSON string into a JavaScript object, which is stored in the variable jsonObject.
4. You can access properties of the JavaScript object jsonObject just like any other JavaScript object.

JavaScript Function Call

JavaScript function calls involve invoking a function to execute its code. You can call a function by using its name followed by parentheses (), optionally passing arguments inside the parentheses.

```
function product(a, b) {
   return a * b;
}
Calling product() function
let result = product.call(this, 20, 5);
console.log(result);
```

Different ways of writing functions in JavaScript

Below are the ways of writing functions in JavaScript:
1. Function Declaration
2. Function Expression
3. Arrow Functions

Function Declaration

1. Function Declaration is the traditional way to define a function.

function fun() { }

Function Expression

2. Function Expression is another way to define a function in JavaScript. Here we define a function using a

variable and store the returned value in that variable.

const fun = function ( ) { }

3. Arrow Functions

Arrow functions are been introduced in the ES6 version of JavaScript. It is used to shorten the code. Here we do not use the “function” keyword and use the arrow symbol.

let fun = () => { }

This function could be an anonymous function if the name is not assigned and wrapped in parenthesis interpreted as an expression.

JAVASCRIPT INBUILT FUNCTIONS

1. .apply() It is different from the function call() because it takes arguments as an array.

```
let student = {
   details: function () {
      return this.name + "\n" + this.class;
   }
}
let stud1 = {
   name: "Dinesh",
   class: "11th",
}
let stud2 = {
   name: "Vaibhav",
   class: "11th",
}
let x = student.details.apply(stud2);
console.log(x);
```

isFinite isNaN

Number() Convert the data type to a number.

```
// .map() The calling function for each and every array element in an array.
let webname = ["welcome", "to", "GeeksforGeeeks"];
webname.map((web, index) => console.log(web, index));
```

ARRAY

Remove elements using methods like pop(), shift(), or splice().

```
let courses = ["HTML", "CSS", "Javascript", "React", "Node.js"];
let lastElement = courses.pop();
let firstElement = courses.shift();
courses.splice(1, 2);
```

array length arr.length = 12 ; <empty space for extra space>

JavaScript Array Methods

Below is the JavaScript Array Methods list, covering all important array methods and properties in JavaScript with examples.

```
JavaScript Array length                 // ---> array.length
JavaScript Array toString() Method      // ---> array.toString()
JavaScript Array join() Method          // ---> array.join("-")  '-' separator
JavaScript Array delete Method          // ---> array.length
JavaScript Array concat() Method        // ---> array.concat(arr2, arr3, ...)
JavaScript Array flat() Method          // ---> array.flat([depth])
Javascript Array.push() Method          // ---> array.push(70, 80, 90);
Javascript Array.unshift() Method       // ---> The unshift() method is used to add elements to the front of an Array.
JavaScript Array.pop() Method           // --->
JavaScript Array.shift() Method
JavaScript Array.splice() Method        // ---> Array.splice (start, deleteCount, item1, item2….) remove(start dC) and add item1, item2, ..
JavaScript Array.slice() Method         // ---> Array.slice (startIndex , endIndex)
JavaScript Array some() Method          // ---> let value = array.some(isGreaterThan5) [at least one]
```

The reduce() method is used to reduce the array to a single value and executes a provided function for each value of the array (from left to right) and the return value of the function is stored in an accumulator.

```
JavaScript Array reduce() Method    // --->
JavaScript Array map() Method       // --->
console.log(typeof ("2344"/3))      // ----> number
```

Number.MAX_VALUE: It's the maximum possible value a javascript number can have (2^53 – 1) OR 1.7976931348623157e+308.

Number.MIN_VALUE: It's the maximum possible value a javascript number can have -(2^53 – 1) or 5e-324.

Number.NAN: It returns "undefined".

Number.NEGATIVE_INFINITY: It's a particular value that represents negative infinity. it has a lower value than **MIN_VALUE.**

Number.POSITIVE_INFINITY: It's a particular value that represents positive infinity. it has a greater value than **MAX_VALUE**

JavaScript MAP

JavaScript map is a collection of elements where each element is stored as a Key, value pair.
Map objects can hold both objects and primitive values as either key or value.

On iterating a map object returns the key, and value pair in the same order as inserted.
Map() constructor is used to create Map in JavaScript.

```
let map1 = new Map([
    [1 , 10], [2 , 20] ,
    [3, 30],[4, 40]
]);
console.log(map1);
```

Map.set() | You can add elements to a Map with the set() method.

```
const prices = new Map([
    ["Laptop", 1000],
    ["Smartphone", 800],
    ["Tablet", 400]
]);
prices.set('Laptop', 1000);
prices.set('Smartphone', 800);
console.log(prices) ;
console.log(prices.size)
prices.clear() ;
console.log(prices) ;
console.log(prices.size)

my_map.delete(key) ;
```

JavaScript Map entries() Method

JavaScript Map.entries() method is used for returning an iterator object which contains all the [key, value] pairs of each element of the map. It returns the [key, value] pairs of all the elements of a map in the orderof their insertion. The Map.entries() method does not require any argument to be passed and returns an iterator object of the map.

```
let myMap = new Map();
myMap.set(0, 'geeksforgeeks');
myMap.set(1, 'is an online portal');
myMap.set(2, 'for geeks');
// creating an iterator object using Map.entries() method
let iterator_obj = myMap.entries();
// displaying the [key, value] pairs of all the elements of the map
console.log(iterator_obj.next().value);
```

```
console.log(iterator_obj.next().value);
console.log(iterator_obj.next().value);
```

JavaScript Map forEach() Method

JavaScript Map.forEach method is used to loop over the map with the given function and executes the given function over each key-value pair.

Parameters: This method accepts four parameters as mentioned above and described below:

1. callback: This is the function that executes on each function call.
2. value: This is the value for each iteration.
3. key: This is the key to reach iteration.

Logging map object to console

```
prices.forEach((values, keys) => {
  console.log(values, keys)
}) ;
console.log(prices.get("Laptop")) ;
mapObj.has(key)
myMap.values()
```

SETS in JavaScript

```
let set1 = new Set(["sumit","sumit","amit","anil","anish"]);
let set2 = new Set("fooooooood"); // it contains 'f', 'o', 'd'
let set3 = new Set([10, 20, 30, 30, 40, 40]); // it contains [10, 20, 30, 40]
let set4 = new Set();  // it is an  empty set
Set.add(val)      // It adds the new element with a specified value at the end of the Set object.
Set.delete(val)  // It deletes an element with the specified value from the Set object.
Set.clear()         // It removes all the element from the set.
Set.entries()     // It returns an iterator object which contains an array having the entries of the set, in the insertion order.
let set1 = new Set();
set1.add(50);
set1.add(30);
set1.add(40);
set1.add(20);
set1.add(10);
// using entries to get iterator
let getEntriesArry = set1.entries();
// each iterator is array of [value, value]
console.log(getEntriesArry.next().value) ;
console.log(getEntriesArry.next().value) ;
console.log(getEntriesArry.next().value) ;
Set.has()       // It returns true if the specified value is present in the Set object.
Set.values()    // It returns all the values from the Set in the same insertion order.
```

Set.forEach() // It executes the given function once for every element in the Set, in the insertion order.

JavaScript Promise

JavaScript Promises to simplify managing multiple asynchronous operations, preventing callback hell and unmanageable code. They represent future values, associating handlers with eventual success or failure, resembling synchronous methods by postponing value delivery until later.

Syntax:

```
let promise = new Promise(function(resolve, reject){
    //do something
});
```

Parameters

1. The promise constructor takes only one argument which is a callback function
2. The callback function takes two arguments, resolve and reject
    1. Perform operations inside the callback function and if everything went well then call resolve.
    2. If desired operations do not go well then call reject.

A Promise has four states:

1. fulfilled: Action related to the promise succeeded
2. rejected: Action related to the promise failed
3. pending: Promise is still pending i.e. not fulfilled or rejected yet
4. settled: Promise has been fulfilled or rejected

```
let promise = new Promise((resolve, reject) => {
   const x = 12 ;
   const y = 12 ;
   if(x === y) {
      resolve() ;
   } else {
      reject() ;
   }
}) ;
promise
   .then(() => {
      console.log("Yes they are equal!") ;
   }).catch(() => {
      console.log("No they are not equal!") ;
   })
```

1. Promise then() Method:

Promise method is invoked when a promise is either resolved or rejected. It may also be defined as a carrier that takes data from promise and further executes it successfully.

Parameters: It takes two functions as parameters.

The first function is executed if the promise is resolved and a result is received.
The second function is executed if the promise is rejected and an error is received.
(It is optional and there is a better way to handle error using .catch() method

```
.then(function(result){
    //handle success
}, function(error){
    //handle error
})
```

Example:

This example shows how the then method handles when a promise is resolved

```
let promise = new Promise(function (resolve, reject) {
   resolve('Geeks For Geeks');
}) ;

promise
   .then(function (successMessage) {
      //success handler function is invoked
      console.log("success: ", successMessage);
   }, function (errorMessage) {
      console.log("failure: ", errorMessage);
   });
```

Example :

This example shows the condition when a rejected promise is handled by second function of then method

```
let promise = new Promise(function (resolve, reject) {
   reject('Promise Rejected')
})
promise
   .then(function (successMessage) {
      console.log("success: ", successMessage);
   }, function (errorMessage) {
      //error handler function is invoked
      console.log("failure: ", errorMessage);
   });
```

2. Promise catch() Method:

Promise catch() Method is invoked when a promise is either rejected or some error has occurred in execution. It is used as an Error Handler whenever at any step there is a chance of getting an error.
Parameters: It takes one function as a parameter.
Function to handle errors or promise rejections.(.catch() method internally calls .then(null, errorHandler),
i.e. .catch() is just a shorthand for .then(null, errorHandler))

```
let promise = new Promise(function (resolve, reject) {
   reject('Promise Rejected')
})

promise
   .then(function (successMessage) {
      console.log("success: ", successMessage);
   }, function(errorMessage) {
      //error handler function is invoked
      console.log("failure1: ", errorMessage);
   })
   .catch(function (errorMessage) {
      //error handler function is invoked when second function of .then is absent
      console.log("failure2: ", errorMessage);
   });
```

Async and Await in JavaScript

Async/await is a feature in JavaScript that allows you to work with asynchronous code in a more synchronous-like manner, making it easier to write and understand asynchronous code.
Async Functions always return a promise. Await Keyword is used only in Async Functions to wait for promise.
JS Async/Await is the extension of promises that we get as support in the language. In this article, we will be learning about async and await in JavaScript with examples.

Async Function

1. The Async function simply allows us to write promises-based code as if it were synchronous and it checks that we are not breaking the execution thread.
2. Async functions will always return a value. It makes sure that a promise is returned and if it is not returned then JavaScript automatically wraps it in a promise which is resolved with its value.

```
const getData = async () => {
   let data = "Hahahahahhaah" ;
   return data ;
}
getData().then(data => console.log(data))
```

Await Keyword

Await is used to wait for the promise. It could be used within the async block only.
It makes the code wait until the promise returns a result.

Await Syntax

```
let value = await promise ;
```

```
const getData = async () => {
   let y = await "Hello World" ;
   console.log(y) ;
}
console.log(1) ;
getData() ;
console.log(2) ;
```

```
------------------
1
2
Hello World
```

Example:

```
function asynchronous_operational_method() {
   let first_promise =
      new Promise((resolve, reject) => resolve("Hello"));
   let second_promise =
      new Promise((resolve, reject) => {
      setTimeout(() => {
         resolve(" GeeksforGeeks..");
      }, 1000);
   });
   let combined_promise =
      Promise.all([first_promise, second_promise]);
   return combined_promise;
}

async function display() {
   let data = await asynchronous_operational_method();
   console.log(data);
}
display();
```