Javascript Interview Questions

Developed by Brendan Eich in 1995, JavaScript is one of the most popular languages for web development.
It was initially created to develop dynamic web pages. Every JS program is called a script, which can be attached to any web page's HTML. These scripts run automatically when the page loads.
A language which was initially used to create dynamic web pages, can now be executed on the server and practically on any device consisting of the JavaScript Engine.

We are going to learn JavaScript, by answering the most frequently asked javascript interview questions:

Basic js interview questions:

1. What are the different data types present in javascript?
2. Explain Hoisting in javascript.
3. Difference between "==" and "===" operators.
4. Explain Implicit Type Coercion in javascript.
5. Is javascript a statically typed or a dynamically typed language?
6. What is NaN property in JavaScript?
7. Explain passed by value and passed by reference.
8. What is an Immediately Invoked Function in javascript?
9. Explain Higher Order Functions in javascript.
10. Explain "this" keyword.
11. Explain call(), apply() and, bind() methods.
12. What is Currying in javascript?
13. Explain Scope and Scope Chain in javascript.
14. Explain Closures in JavaScript.
15. What are object prototypes?

16. What are callbacks?
17. What is memoization?
18. What is recursion in a programming language?
19. What is the use of a constructor function in javascript?
20. What is DOM?

Advanced js interview questions:

21. What are arrow functions?
22. Differences between declaring variables using var, let and const.
23. What is the rest parameter and spread operator?
24. What is the use of promises in javascript?
25. What are classes in javascript?
26. What are generator functions?
27. Explain WeakSet in javascript.
28. Explain WeakMap in javascript.
29. What is Object Destructuring?
30. What is a Temporal Dead Zone?

Javscript Coding Problems:

Basic Javascript Interview Questions:

1. What are the different data types present in javascript?

To know the type of a JavaScript variable, we can use the **typeof** operator.

Primitive types

- **String** - It represents a series of characters and is written with quotes. A string can be represented using a single or a double quote.

Example :

```
var str = "Vivek Singh Bisht"; //using double quotes
var str2 = 'John Doe'; //using single quotes
```

- **Number** - It represents a number and can be written with or without decimals.

Example :

```
var x = 3; //without decimal
var y = 3.6; //with decimal
```

- **BigInt** - This data type is used to store numbers which are above the limitation of the Number data type. It can store large integers and is represented by adding “n” to an integer literal.

Example :

```
var bigInteger = 234567890123456789012345678901234567890;
```

- **Boolean** - It represents a logical entity and can have only two values : true or false. Booleans are generally used for conditional testing.

Example :

```
var a = 2;
var b = 3;
var c = 2;
(a == b) // returns false
(a == c) //returns true
```

- **Undefined** - When a variable is declared but not assigned, it has the value of undefined and it’s type is also undefined.

Example :

```
var x; // value of x is undefined
var y = undefined; // we can also set the value of a variable as undefined
```

- **Null** - It represents a non-existent or a invalid value.

Example :

```
var z = null;
```

- **Symbol** - It is a new data type introduced in the ES6 version of javascript. It is used to store an anonymous and unique value.

Example :

```
var symbol1 = Symbol('symbol');
```

- `typeof` **of primitive types** :

```
typeof "John Doe" // Returns "string"
typeof 3.14 // Returns "number"
typeof true // Returns "boolean"
typeof 2345678901234567890123456789012345678901n // Returns bigint
typeof undefined // Returns "undefined"
typeof null // Returns "object" (kind of a bug in JavaScript)
typeof Symbol('symbol') // Returns Symbol
```

Non-primitive types

Primitive data types can store only a single value. To store multiple and complex values, non-primitive data types are used.

Object - Used to store collection of data.

Example:

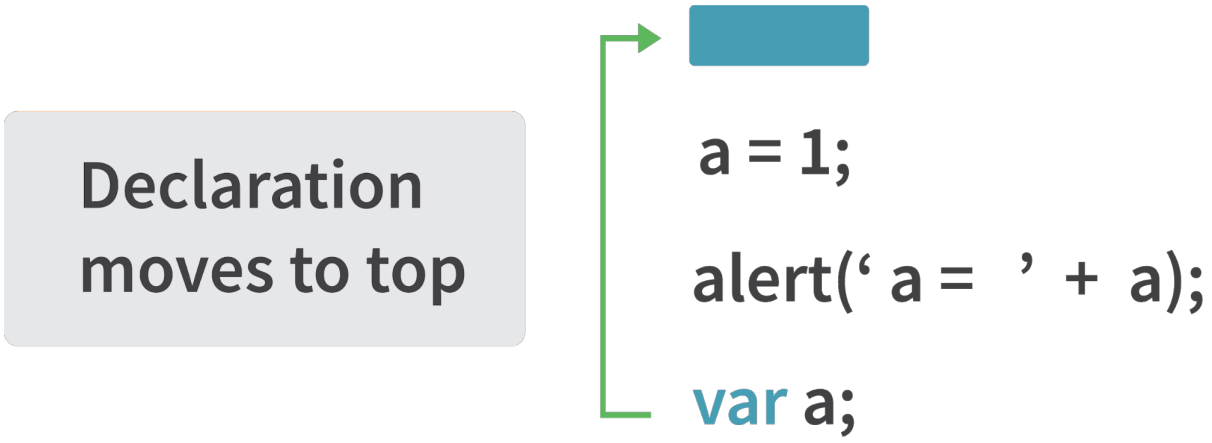
```
// Collection of data in key-value pairs

var obj1 = {
   x: 43,
   y: "Hello world!",
   z: function(){
      return this.x;
   }
}

// Collection of data as an ordered list

var array1 = [5, "Hello", true, 4.1];
```

*Note- It is important to remember that any data type that is not primitive data type, is of Object type in javascript.

2. Explain Hoisting in javascript.

Hoisting is a default behaviour of javascript where all the variable and function declarations are

moved on top.



This means that irrespective of where the variables and functions are declared, they are moved on top of the scope. The scope can be both local and global.

Example 1:

```
hoistedVariable = 3;
console.log(hoistedVariable); // outputs 3 even when the variable is declared after it is initiali
var hoistedVariable;
```

Example 2:

```
hoistedFunction();  // Outputs " Hello world! " even when the function is declared after calling

function hoistedFunction(){
  console.log(" Hello world! ");
}
```

Example 3:

```
// Hoisting takes place in the local scope as well
function doSomething(){
  x = 33;
  console.log(x);
  var x;
}
```

doSomething(); // Outputs 33 since the local variable “x” is hoisted inside the local scope

****Note - Variable initializations are not hoisted, only variable declarations are hoisted:**

```
var x;
console.log(x); // Outputs "undefined" since the initialization of "x" is not hoisted
x = 23;
```

****Note - To avoid hoisting, you can run javascript in strict mode by using “use strict” on top of the code:**

```
"use strict";
x = 23; // Gives an error since 'x' is not declared
var x;
```

3. Difference between “ == “ and “ === “ operators.

Both are comparison operators. The difference between both the operators is that,“==” is used to compare values whereas, “ === “ is used to compare both value and types.

Example:

```
var x = 2;
var y = "2";
(x == y)  // Returns true since the value of both x and y is the same

(x === y) // Returns false since the typeof x is "number" and typeof y is "string"
```

4. Explain Implicit Type Coercion in javascript.

Implicit type coercion in javascript is automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

String coercion

String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type.

Example 1:

```
var x = 3;
var y = "3";
x + y // Returns "33"
```

Example 2:

```
var x = 24;
var y = "Hello";
x + y    // Returns "24Hello";
```

**Note - ‘ + ‘ operator when used to add two numbers, outputs a number. The same ‘ + ‘ operator when used to add two strings, outputs the concatenated string:

```
var name = "Vivek";
var surname = " Bisht";

name + surname      // Returns "Vivek Bisht"
```

Let’s understand both the examples where we have added a number to a string,

When JavaScript sees that the operands of the expression x + y are of different types (one being a number type and the other being a string type) , it converts the number type to the string type and then performs the operation. Since after conversion, both the variables are of string type, the ‘ + ‘ operator outputs the concatenated string “33” in the first example and “24Hello” in the second example.

**Note - Type coercion also takes place when using the ‘ - ‘ operator, but the difference while using ‘ - ‘ operator is that, a string is converted to a number and then subtraction takes place.

```
var x = 3;
Var y = "3";
x - y     //Returns 0 since the variable y (string type) is converted to a number type
```

Boolean Coercion

Boolean coercion takes place when using logical operators, ternary operators, if statements and loop checks. To understand boolean coercion in if statements and operators, we need to understand truthy and falsy values.

Truthy values are those which will be converted (coerced) to **true** . Falsy values are those which will be converted to **false** .

All values except **0, 0n, -0, “”, null, undefined and NaN** are truthy values.

If statements:

Example:

```
var x = 0;
var y = 23;

if(x) { console.log(x) }    // The code inside this block will not run since the value of x is 0(Fa

if(y) { console.log(y) }     // The code inside this block will run since the value of y is 23 (Tru
```

Logical operators:

Logical operators in javascript, unlike operators in other programming languages, **do not return true or false. They always return one of the operands.**

OR (| |) operator - If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.

AND (&&) operator - If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.

Example:

```
var x = 220;
var y = "Hello";
var z = undefined;

x || y      // Returns 220 since the first value is truthy

x || z    // Returns 220 since the first value is truthy

x && y      // Returns "Hello" since both the values are truthy

y && z    // Returns undefined since the second value is falsy

if( x && y ){
  console.log("Code runs" ); // This block runs because x && y returns "Hello" (Truthy)
}

if( x || z ){
  console.log("Code runs"); // This block runs because x || y returns 220(Truthy)
}
```

Equality Coercion

Equality coercion takes place when using ‘ == ‘ operator. As we have stated before

The ‘ == ‘ operator compares values and not types.

While the above statement is a simple way to explain == operator, it’s not completely true

The reality is that while using the ‘==’ operator, coercion takes place.

The ‘==’ operator, converts both the operands to the same type and then compares them.

Example:

```
var a = 12;
var b = "12";
a == b // Returns true because both 'a' and 'b' are converted to the same type and then compa
```

Coercion does not take place when using the ‘===’ operator. Both operands are not converted to the same type in the case of ‘===’ operator.

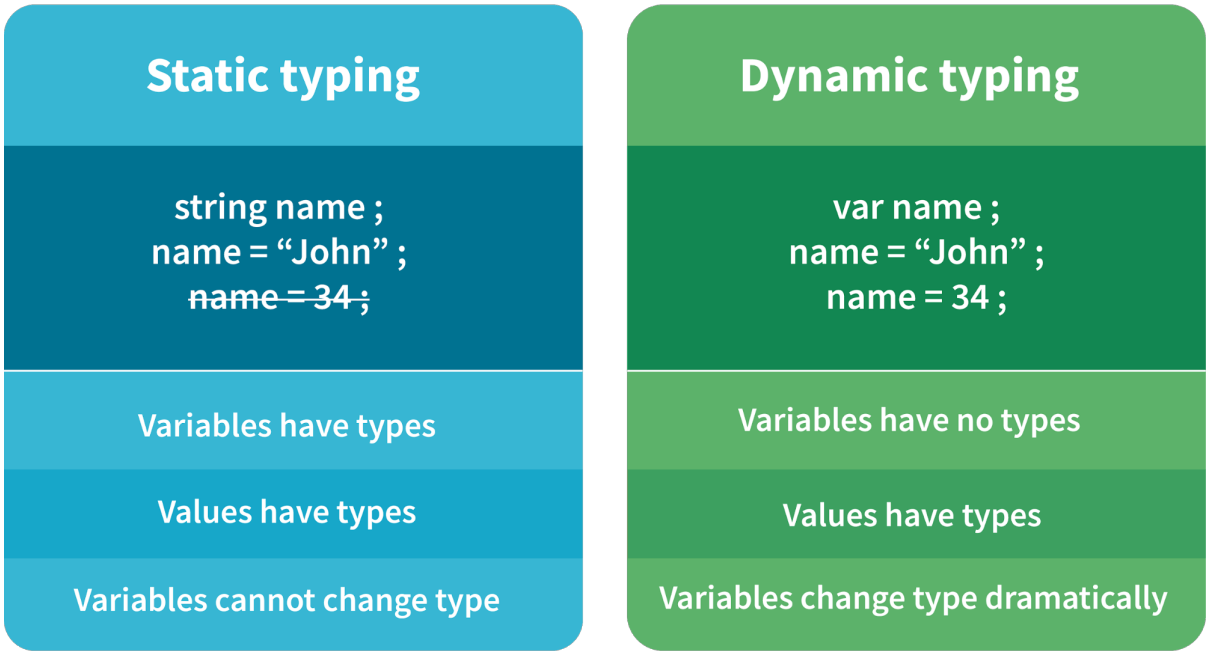
Example:

```
var a = 226;
var b = "226";

a === b // Returns false because coercion does not take place and the  operands are of differ
```

5. Is javascript a statically typed or a dynamically typed language?

JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during **run-time** in contrast to statically typed language, where the type of a variable is checked during **compile-time.**

Static vs Dynamic Typing



Since javascript is a loosely(dynamically) typed language, variables in JS are not associated with any type. A variable can hold the value of any data type.

For example, a variable which is assigned a number type can be converted to a string type:

```
var a = 23;
var a = "Hello World!";
```

6. What is NaN property in JavaScript?

NaN property represents **“Not-a-Number”** value. It indicates a value which is not a legal number.

typeof of a NaN will return a **Number** .

To check if a value is NaN, we use the **isNaN()** function,

**Note- isNaN() function converts the given value to a Number type, and then equates to NaN.

```
isNaN("Hello")   // Returns true
isNaN(345)    // Returns false
isNaN('1')   // Returns false, since '1' is converted to Number type which results in 0 ( a numbe
isNaN(true)  // Returns false, since true converted to Number type results in 1 ( a number)
isNaN(false)  // Returns false
isNaN(undefined)  // Returns true
```

7. Explain passed by value and passed by reference.

In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.

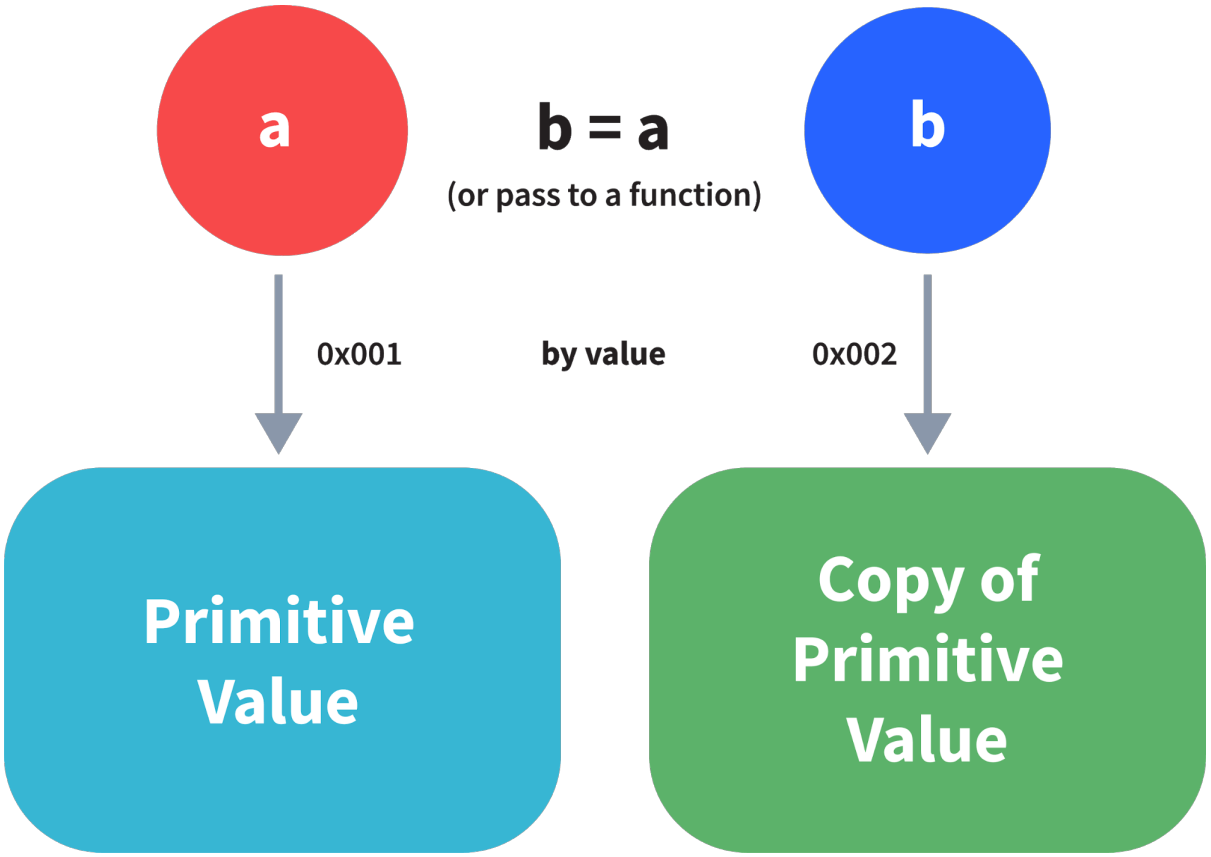
For understanding passed by value and passed by reference, we need to understand what happens when we create a variable and assign a value to it,

```
var x = 2;
```

In the above example, we created a variable x and assigned it a value “2”. In the background, the “=” (assign operator) allocates some space in the memory, stores the value “2” and returns the location of the allocated memory space. Therefore, the variable x in the above code points to the location of the memory space instead of pointing to the value 2 directly.

Assign operator behaves differently when dealing with primitive and non primitive data types,

Assign operator dealing with primitive types:



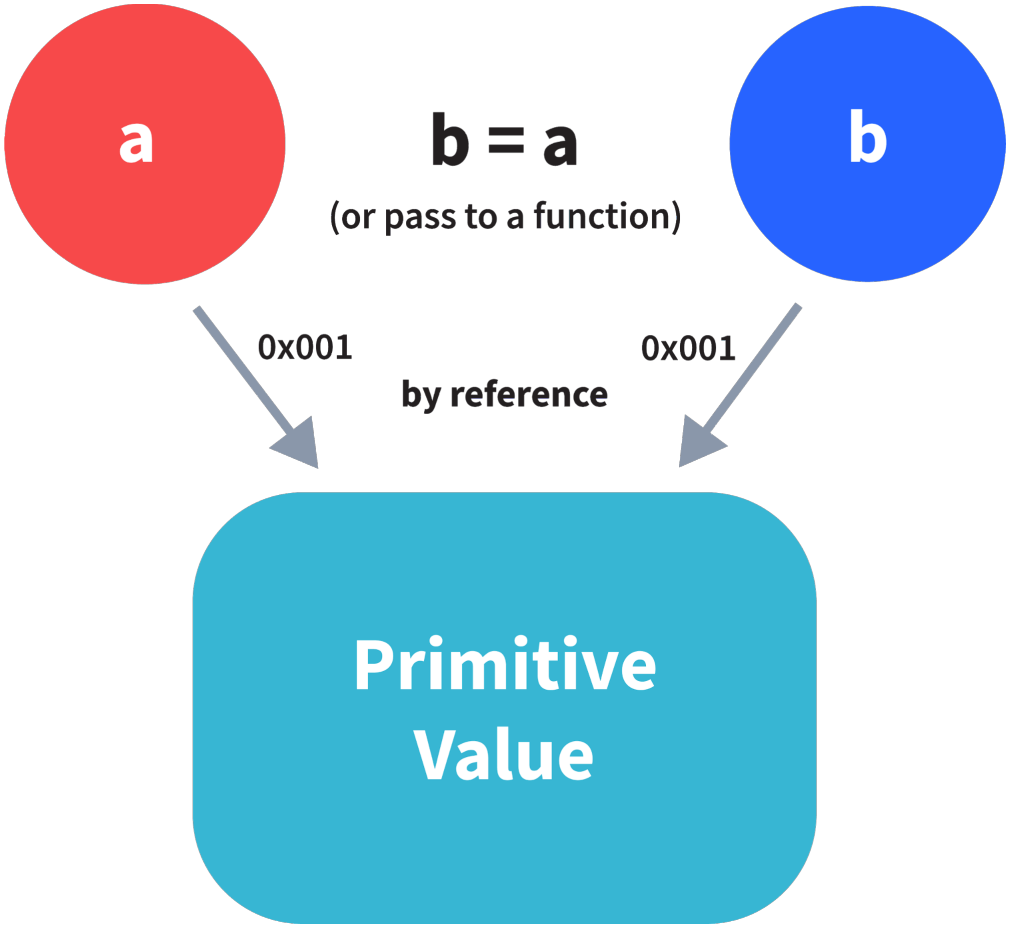

```
var y = 234;
var z = y;
```

In the above example, assign operator knows that the value assigned to y is a primitive type (number type in this case), so when the second line code executes, where the value of y is assigned to z, the assign operator takes the value of y (234) and allocates a new space in the memory and returns the address. Therefore, variable z is not pointing to the location of variable y, instead it is pointing to a new location in the memory.

```
var y = #8454; // y pointing to address of the value 234

var z = y;

var z = #5411; // z pointing to a completely new address of the value 234

// Changing the value of y
y = 23;
console.log(z);  // Returns 234, since z points to a new address in the memory so changes ir
```

From the above example, we can see that primitive data types when passed to another variable, are passed by value. Instead of just assigning the same address to another variable, the value is passed and new space of memory is created.

Assign operator dealing with non-primitive types:



```
var obj = { name: "Vivek", surname: "Bisht" };

var obj2 = obj;
```

In the above example, the assign operator, directly passes the location of the variable obj to the variable obj2. In other words, the reference of the variable obj is passed to the variable obj2.

```
var obj = #8711;   // obj pointing to address of { name: "Vivek", surname: "Bisht" }

var obj2 = obj;

var obj2 = #8711; // obj2 pointing to the same address


// changing the value of obj1

obj1.name = "Akki";

console.log(obj2);

// Returns {name:"Akki", surname:"Bisht"} since both the variables are pointing to the same adc
```

From the above example, we can see that while passing non-primitive data types, the assign operator directly passes the address (reference).

Therefore, non-primitive data types are always **passed by reference.**

8. What is an Immediately Invoked Function in JavaScript?

An Immediately Invoked Function (known as IIFE and pronounced as IIFY) is a function that runs as soon as it is defined.

Syntax of IIFE :

```
(function(){
  // Do something;
})();
```

To understand IIFE, we need to understand the two sets of parentheses which are added while creating an IIFE :

First set of parenthesis:

```
(function (){
    //Do something;
})
```

While executing javascript code, whenever the compiler sees the word “function”, it assumes that we are declaring a function in the code. Therefore, if we do not use the first set of parentheses, the compiler throws an error because it thinks we are declaring a function, and by the syntax of declaring a function, a function should always have a name.

```
function() {
   //Do something;
}
// Compiler gives an error since the syntax of declaring a function is wrong in the code above.
```

To remove this error, we add the first set of parenthesis that tells the compiler that the function is not a function declaration, instead, it’s a function expression.

Second set of parenthesis:

```
(function (){
   //Do something;
})();
```

From the definition of an IIFE, we know that our code should run as soon as it is defined. A function runs only when it is invoked. If we do not invoke the function, the function declaration is returned:

```
(function (){
   // Do something;
})

// Returns the function declaration
```

Therefore to invoke the function, we use the second set of parenthesis. .

9. Explain Higher Order Functions in javascript.

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

Higher order functions are a result of functions being **first-class citizens** in javascript.

Examples of higher order functions:

```
function higherOrder(fn) {
  fn();
}

higherOrder(function() { console.log("Hello world") });
```

```
function higherOrder2() {
  return function() {
    return "Do something";
  }
}

var x = higherOrder2();
x()    // Returns "Do something"
```

10. Explain “this” keyword.

The “this” keyword refers to the object that the function is a property of.

The value of “this” keyword will always depend on the object that is invoking the function.

Confused? Let’s understand the above statements by examples:

```
function doSomething() {
  console.log(this);
}

doSomething();
```

What do you think the output of the above code will be?

**Note - Observe the line where we are invoking the function.

Check the definition again:

The “this” keyword refers to the object that the function is a property of.

In the above code, function is a property of which object?

Since the function is invoked in the global context, **the function is a property of the global object.**

Therefore, the output of the above code will be **the global object.** Since we ran the above code inside the browser, the global object is **the window object.**

Example 2:

```
var obj = {
    name: "vivek",
    getName: function(){
    console.log(this.name);
  }
}

obj.getName();
```

In the above code, at the time of invocation, the getName function is a property of the object **obj** , therefore, the **this** keyword will refer to the object **obj** , and hence the output will be “vivek”.

Example 3:

```
 var obj = {
    name: "vivek",
    getName: function(){
    console.log(this.name);
  }

}

var getName = obj.getName;

var obj2 = {name:"akshay", getName };
obj2.getName();
```

Can you guess the output here?

The output will be “akshay”.

Although the getName function is declared inside the object **obj** , at the time of invocation, getName() is a property of **obj2** , therefore the “this” keyword will refer to **obj2** .

The silly way to understanding the **this** keyword is, whenever the function is invoked, check the object before the **dot** . The value of **this** . keyword will always be the object before the **dot** .

If there is no object before the dot like in example1, the value of this keyword will be the global object.

Example 4:

```
var obj1 = {
    address : "Mumbai,India",
    getAddress: function(){
    console.log(this.address);
  }
}

var getAddress = obj1.getAddress;
var obj2 = {name:"akshay"};
obj2.getAddress();
```

Can you guess the output?

The output will be an error.

Although in the code above, the this keyword refers to the object **obj2** , obj2 does not have the property “address”‘, hence the getAddress function throws an error.

11. Explain call(), apply() and, bind() methods.

call()

It’s a predefined method in javascript.

This method invokes a method (function) by specifying the owner object.

Example 1:

```

```

```
function sayHello(){
  return "Hello " + this.name;
}

var obj = {name: "Sandy"};

sayHello.call(obj);

// Returns "Hello Sandy"
```

call() method allows an object to use the method (function) of another object.

Example 2:

```
var person = {
  age: 23,
  getAge: function(){
   return this.age;
  }
}

var person2 = {age: 54};
person.getAge.call(person2);

// Returns 54
```

call() accepts arguments:

```
function saySomething(message){
  return this.name + " is " + message;
}

var person4 = {name: "John"};

saySomething.call(person4, "awesome");
// Returns "John is awesome"
```

apply()

The apply method is similar to the call() method. The only difference is that,

call() method takes arguments separately whereas, apply() method takes arguments as an array.

```

```

```
function saySomething(message){
  return this.name + " is " + message;
}

var person4 = {name: "John"};

saySomething.apply(person4, ["awesome"]);
```

bind()

This method returns a new function, where the value of **“this”** keyword will be bound to the owner object, which is provided as a parameter.

Example with arguments:

```
var bikeDetails = {
    displayDetails: function(registrationNumber,brandName){
    return this.name+ " , "+ "bike details: "+ registrationNumber + " , " + brandName;
  }
}

var person1 = {name: "Vivek"};

var detailsOfPerson1 = bikeDetails.displayDetails.bind(person1, "TS0122", "Bullet");

// Binds the displayDetails function to the person1 object


detailsOfPerson1();
// Returns Vivek, bike details: TS0452, Thunderbird
```

12. What is currying in JavaScript?

Currying is an advanced technique to transform a function of arguments n, to n functions of one or less arguments.

Example of a curried function:

```
function add (a) {
  return function(b){
   return a + b;
  }
}

add(3)(4)
```

For Example, if we have a function **f(a,b)** , then the function after currying, will be transformed to **f(a)(b).**

By using the currying technique, we do not change the functionality of a function, we just change the way it is invoked.

Let's see currying in action:

```
function multiply(a,b){
  return a*b;
}

function currying(fn){
  return function(a){
    return function(b){
      return fn(a,b);
    }
  }
}

var curriedMultiply = currying(multiply);

multiply(4, 3); // Returns 12

curriedMultiply(4)(3); // Also returns 12
```

As one can see in the code above, we have transformed the function **multiply(a,b)** to a function **curriedMultiply** , which takes in one parameter at a time.

13. Explain Scope and Scope Chain in javascript.

Scope in JS, determines the accessibility of variables and functions at various parts in one's code.

In general terms, the scope will let us know at a given part of code, what are the variables and functions that we can or cannot access.

There are three types of scopes in JS:

- Global Scope
- Local or Function Scope
- Block Scope

Global Scope

Variables or functions declared in the global namespace have global scope, which means all the variables and functions having global scope can be accessed from anywhere inside the code

```
var globalVariable = "Hello world";

function sendMessage(){
  return globalVariable; // can access globalVariable since it's written in global space
}

function sendMessage2(){
  return sendMessage(); // Can access sendMessage function since it's written in global space
}

sendMessage2();  // Returns “Hello world”
```

Function Scope

Any variables or functions declared inside a function have local/function scope, which means that all the variables and functions declared inside a function, can be accessed from within the function and not outside of it.

```
function awesomeFunction(){
  var a = 2;

  var multiplyBy2 = function(){
    console.log(a*2); // Can access variable "a" since a and multiplyBy2 both are written insi
  }
}
console.log(a); // Throws reference error since a is written in local scope and cannot be acce

multiplyBy2(); // Throws reference error since multiplyBy2 is written in local scope
```

Block Scope

Block scope is related to the variables declared using let and const. Variables declared with var do not have block scope.

Block scope tells us that any variable declared inside a block { }, can be accessed only inside that block and cannot be accessed outside of it.

```

```

```
{
  let x = 45;
}

console.log(x); // Gives reference error since x cannot be accessed outside of the block

for(let i=0; i<2; i++){
  // do something
}

console.log(i); // Gives reference error since i cannot be accessed outside of the for loop blo
```

Scope Chain

JavaScript engine also uses Scope to find variables.

Let's understand that using an example:

```
var y = 24;

function favFunction(){
  var x = 667;
  var anotherFavFunction = function(){
    console.log(x); // Does not find x inside anotherFavFunction, so looks for variable inside
  }

  var yetAnotherFavFunction = function(){
    console.log(y); // Does not find y inside yetAnotherFavFunction, so looks for variable insi
  }

  anotherFavFunction();
  yetAnotherFavFunction();
}


favFunction();
```

As you can see in the code above, if the javascript engine does not find the variable in local scope, it tries to check for the variable in the outer scope. If the variable does not exist in the outer scope, it tries to find the variable in the global scope.

If the variable is not found in the global space as well, reference error is thrown.

14. Explain Closures in JavaScript.

Closures is an ability of a function to remember the variables and functions that are declared in its outer scope.

```
var Person = function(pName){
  var name = pName;

  this.getName = function(){
    return name;
  }
}

var person = new Person("Neelesh");
console.log(person.getName());
```

Let’s understand closures by example:

```
function randomFunc(){
  var obj1 = {name:"Vivian", age:45};

  return function(){
    console.log(obj1.name + " is "+ "awesome"); // Has access to obj1 even when the rando

  }
}

var initialiseClosure = randomFunc(); // Returns a function

initialiseClosure();
```

Let’s understand the code above,

The function randomFunc() gets executed and returns a function when we assign it to a variable:

```
var initialiseClosure = randomFunc();
```

The returned function is then executed when we invoke initialiseClosure:

```
initialiseClosure();
```

The line of code above outputs “Vivian is awesome” and this is possible because of closure.

When the function randomFunc() runs, it sees that the returning function is using the variable obj1 inside it:

```
console.log(obj1.name + " is "+ "awesome");
```

Therefore randomFunc(), instead of destroying the value of obj1 after execution, **saves the value in the memory for further reference.** This is the reason why the returning function is able to use the variable declared in the outer scope even after the function is already executed.

This ability of a function to store a variable for further reference even after it is executed, is called Closure.

15. What are object prototypes?

All javascript objects inherit properties from a prototype.

For example,

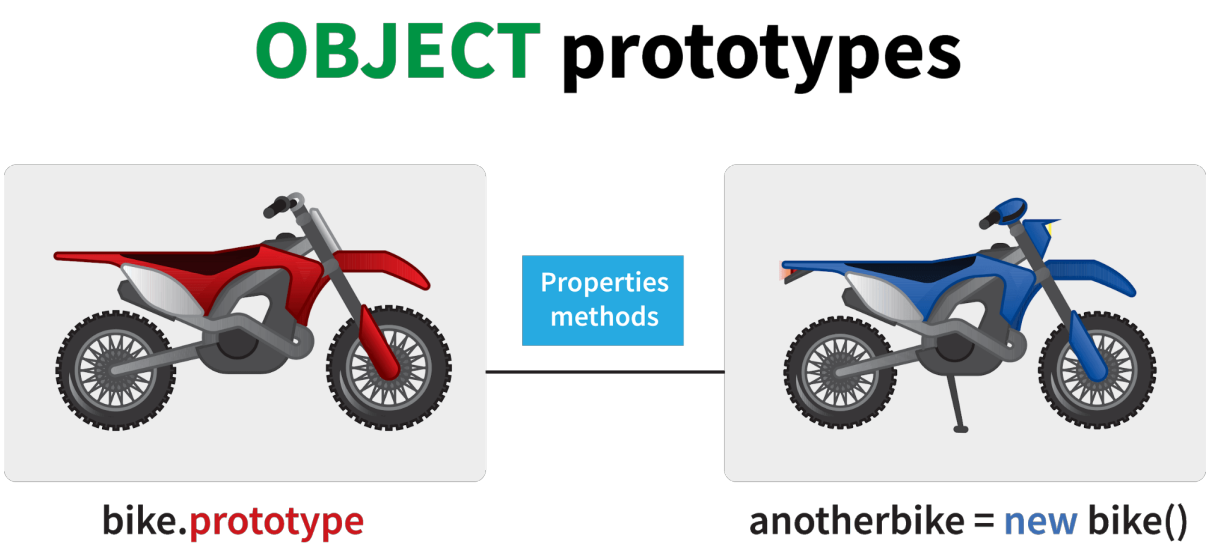
Date objects inherit properties from the Date prototype

Math objects inherit properties from the Math prototype

Array objects inherit properties from the Array prototype.

On top of the chain is **Object.prototype.** Every prototype inherits properties and methods from the Object.prototype.

A prototype is a blueprint of an object. Prototype allows us to use properties and methods on an object even if the properties and methods do not exist on the current object.



Let's see prototypes help us use methods and properties:

```
var arr = [];
arr.push(2);

console.log(arr); // Outputs [2]
```

In the code above, as one can see, we have not defined any property or method called push on the array "arr" but the javascript engine does not throw an error.

The reason being the use of prototypes. As we discussed before, Array objects inherit properties from the Array prototype.

The javascript engine sees that the method push does not exist on the current array object and therefore, looks for the method push inside the Array prototype and it finds the method.

Whenever the property or method is not found on the current object, the javascript engine will always try to look in its prototype and if it still does not exist, it looks inside the prototype's prototype and so on.

16. What are callbacks?

A callback is a function that will be executed after another function gets executed.

In javascript, functions are treated as first-class citizens, they can be used as an argument of another function, can be returned by another function and can be used as a property of an object.

Functions that are used as an argument to another function are called callback functions.

Example:

```
function divideByHalf(sum){
  console.log(Math.floor(sum / 2));
}

function multiplyBy2(sum){
  console.log(sum * 2);
}

function operationOnSum(num1,num2,operation){
  var sum = num1 + num2;
  operation(sum);
}

operationOnSum(3, 3, divideByHalf); // Outputs 3

operationOnSum(5, 5, multiplyBy2); // Outputs 20
```

In the code above, we are performing mathematical operations on the sum of two numbers.

The operationOnSum function takes 3 arguments, first number, second number, and the operation that is to be performed on their sum (callback) .

Both divideByHalf and multiplyBy2 functions are used as callback functions in the code above.

These callback functions will be executed only after the function operationOnSum is executed.

Therefore, callback is a function that will be executed after another function gets executed.

17.What is memoization?

Memoization is a form of caching where the return value of a function is cached based on its parameters. If the parameter of that function is not changed, the cached version of the function is returned.

Let’s understand memoization, by converting a simple function to a memoized function:

****Note- Memoization is used for expensive function calls but in the following example, we are considering a simple function for understanding the concept of memoization better.**

Consider the following function:

```
function addTo256(num){
  return num + 256;
}

addTo256(20); // Returns 276
addTo256(40); // Returns 296
addTo256(20); // Returns 276
```

In the code above, we have written a function that adds the parameter to 256 and returns it.

When we are calling the function addTo256 again with the same parameter (“20” in the case above), we are computing the result again for the same parameter.

Computing the result with the same parameter again and again is not a big deal in the above case, but imagine if the function does some heavy duty work, then, computing the result again and again with the same parameter will lead to wastage of time.

This is where memoization comes in, by using memoization we can store(cache) the computed results based on the parameters. If the same parameter is used again while invoking the function, instead of computing the result, we directly return the stored (cached) value.

Let’s convert the above function addTo256, to a memoized function:

```

```

```
function memoizedAddTo256(){
  var cache = {};

  return function(num){
    if(num in cache){
      console.log("cached value");
      return cache[num]

    }
    else{
      cache[num] = num + 256;
      return cache[num];
    }
  }
}

var memoizedFunc = memoizedAddTo256();

memoizedFunc(20); // Normal return
memoizedFunc(20); // Cached return
```

In the code above, if we run memoizedFunc function with the same parameter, instead of computing the result again, it returns the cached result.

****Note- Although using memoization saves time, it results in larger consumption of memory since we are storing all the computed results.**

18. What is recursion in a programming language?

Recursion is a technique to iterate over an operation by having a function call itself repeatedly until it arrives at a result.

```
function add(number) {
  if (number <= 0) {
    return 0;
  } else {
    return number + add(number - 1);
  }
}

add(3) => 3 + add(2)
          3 + 2 + add(1)
          3 + 2 + 1 + add(0)
          3 + 2 + 1 + 0 = 6
```

Example of a recursive function:

The following function calculates the sum of all the elements in an array by using recursion:

```
function computeSum(arr){
  if(arr.length === 1){
    return arr[0];
  }
  else{
    return arr.pop() + computeSum(arr);
  }
}

computeSum([7, 8, 9, 99]); // Returns 123
```

19. What is the use of a constructor function in javascript?

Constructor functions are used to create objects in javascript.

When do we use constructor functions?

If we want to create multiple objects having similar properties and methods, constructor functions are used.

**Note- Name of a constructor function should always be written in Pascal Notation: every word should start with a capital letter.

Example:

```
function Person(name,age,gender){
  this.name = name;
  this.age = age;
  this.gender = gender;
}


var person1 = new Person("Vivek", 76, "male");
console.log(person1);

var person2 = new Person("Courtney", 34, "female");
console.log(person2);
```

In the code above, we have created a constructor function named Person.

Whenever we want to create a new object of the type Person,

We need to create it using the new keyword:

```
var person3 = new Person("Lilly", 17, "female");
```

The above line of code will create a new object of the type Person.

Constructor functions allow us to group similar objects.

20. What is DOM?

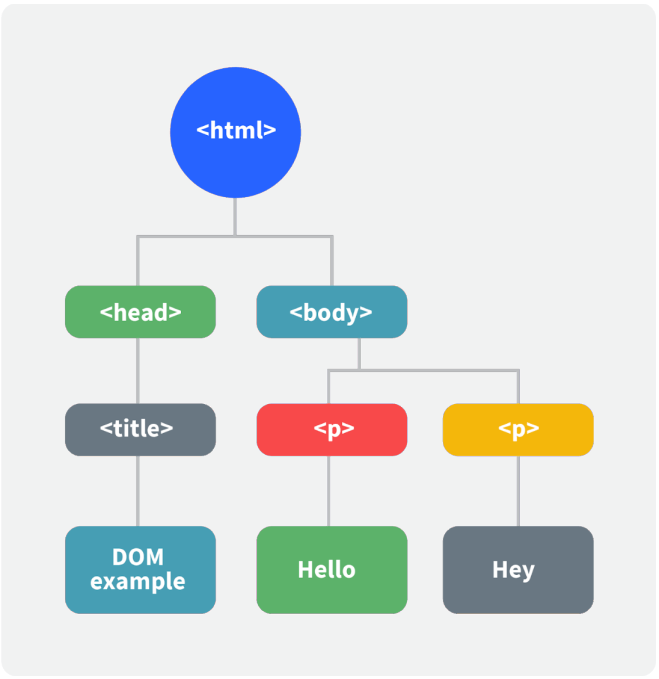
DOM stands for Document Object Model.

DOM is a programming interface for HTML and XML documents.

When the browser tries to render a HTML document, it creates an object based on the HTML document called DOM. Using this DOM, we can manipulate or change various elements inside the HTML document.

Example of how HTML code gets converted to DOM:

```
<html>
<head>
<title>
DOM example
</title>
</head>
<body>
<p id = “para1”>Hello</p>
<p id = “para2”>Hey</p>
</body>
</html>
```



Advanced Javascript Interview

Questions:

21. What are arrow functions?

Arrow functions were introduced in the ES6 version of javascript.

They provide us with a new and shorter syntax for declaring functions.

Arrow functions can only be used as a function expression.

Let's compare the normal function declaration and the arrow function declaration in detail:

```
// Traditional Function Expression
var add = function(a,b){
  return a + b;
}

// Arrow Function Expression
var arrowAdd = (a,b) => a + b;
```

Arrow functions are declared without the function keyword. If there is only one returning expression then we don't need to use the return keyword as well in an arrow function as shown in the example above. Also, for functions having just one line of code, curly braces { } can be omitted.

```
// Traditional function expression
var multiplyBy2 = function(num){
  return num * 2;
}
// Arrow function expression
var arrowMultiplyBy2 = num => num * 2;
```

If the function takes in only one argument, then the parenthesis () around the parameter can be omitted as shown in the code above.

```

```

```
var obj1 = {
  valueOfThis: function(){
    return this;
  }
}
var obj2 = {
  valueOfThis: ()=>{
    return this;
  }
}

obj1.valueOfThis(); // Will return the object obj1
obj2.valueOfThis(); // Will return window/global object
```

The biggest difference between the traditional function expression and the arrow function, is the handling of the **this** keyword.

By general definition, the **this** keyword always refers to the object that is calling the function.

As you can see in the code above, **obj1.valueOfThis()** returns obj1, since **this** keyword refers to the object calling the function.

In the arrow functions, there is no binding of the **this** keyword.

The **this** keyword inside an arrow function, does not refer to the object calling it. It rather inherits its value from the parent scope which is the window object in this case.

Therefore, in the code above, **obj2.valueOfThis()** returns the window object.

22. Differences between declaring variables using var, let and const.

Before the ES6 version of javascript, only the keyword var was used to declare variables.

With the ES6 Version, keywords let and const were introduced to declare variables.

keyword	const	let	var
global scope	no	no	yes
function scope	yes	yes	yes
block scope	yes	yes	no
can be reassigned	no	yes	yes

Let’s understand the differences with examples:

Let’s understand the differences with examples:

```
var variable1 = 23;

let variable2 = 89;

function catchValues(){
  console.log(variable1);
  console.log(variable2);

// Both the variables can be accessed anywhere since they are declared in the global scope
}

window.variable1; // Returns the value 23

window.variable2; // Returns undefined
```

The variables declared with the let keyword in the global scope behave just like the variable declared with the var keyword in the global scope.

Variables declared in the global scope with var and let keywords can be accessed from anywhere in the code.

But, there is one difference !

Variables that are declared with the var keyword in the global scope are added to the window/global object. Therefore, they can be accessed using window.variableName.

Whereas, the variables declared with the let keyword are not added to the global object, therefore, trying to access such variables using window.variableName results in an error.

var vs let in functional scope

```
function varVsLetFunction(){
  let awesomeCar1 = "Audi";
  var awesomeCar2 = "Mercedes";
}

console.log(awesomeCar1); // Throws an error
console.log(awesomeCar2); // Throws an error
```

Variables declared in a functional/local scope using **var** and **let** keywords behave exactly the same, meaning , they cannot be accessed from outside of the scope.

```
{
  var variable3 = [1, 2, 3, 4];
}

console.log(variable3); // Outputs [1,2,3,4]

{
  let variable4 = [6, 55, -1, 2];
}

console.log(variable4); // Throws error

for(let i = 0; i < 2; i++){
  //Do something
}

console.log(i); // Throws error

for(var j = 0; j < 2; i++){
  // Do something
}

console.log(j) // Outputs 2
```

In javascript, a block means the code written inside the curly braces **{}** .

Variables declared with **var** keyword do not have block scope. It means a variable declared in block scope **{}** with the **var** keyword is the same as declaring the variable in the global scope.

Variables declared with **let** keyword inside the block scope cannot be accessed from outside of the block.

Const keyword

Variables with the **const** keyword behave exactly like a variable declared with the let keyword with only one difference, **any variable declared with the const keyword cannot be reassigned.**

Example:

```
```

```
const x = {name:"Vivek"};
x = {address: "India"}; // Throws an error
x.name = "Nikhil"; // No error is thrown
const y = 23;
y = 44; // Throws an error
```

In the code above, although we can change the value of a property inside the variable declared with **const** keyword, we cannot completely reassign the variable itself.

23. What is the rest parameter and spread operator?

Both rest parameter and spread operator were introduced in the ES6 version of javascript.

Rest parameter (…)

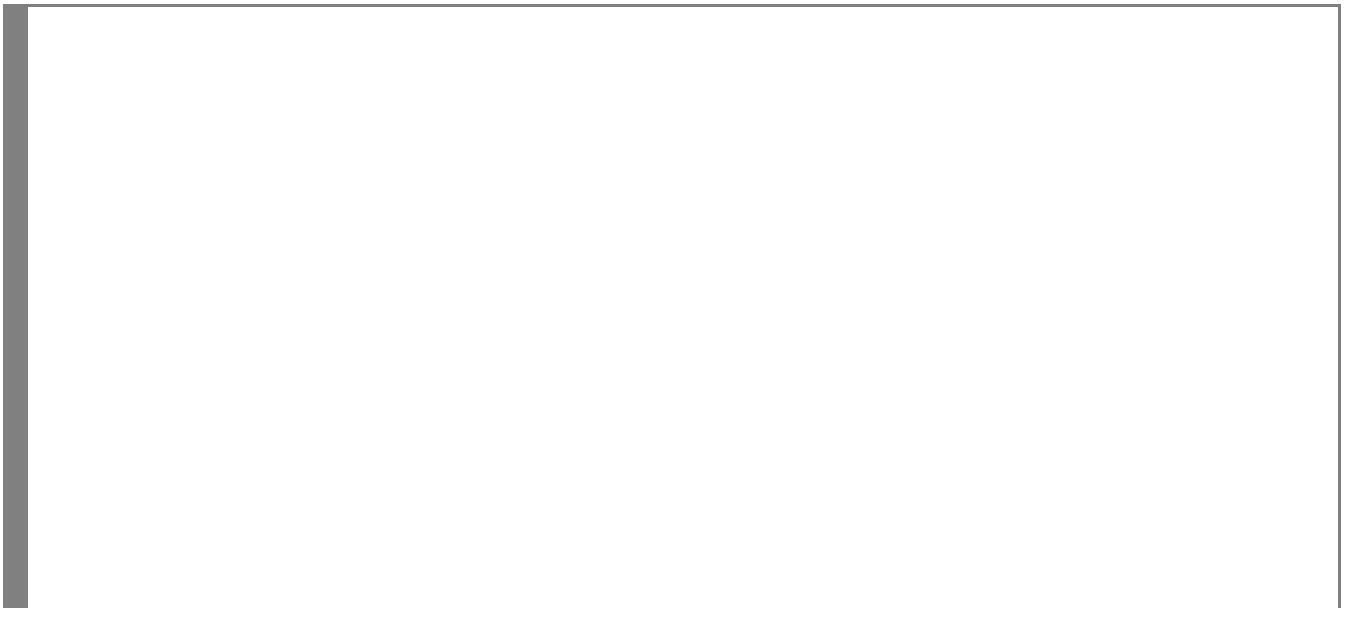
It provides an improved way of handling parameters of a function.

Using the rest parameter syntax, we can create functions that can take a variable number of arguments.

Any number of arguments will be converted into an array using the rest parameter.

It also helps in extracting all or some parts of the arguments.

Rest parameter can be used by applying three dots (...) before the parameters.



```
function extractingArgs(...args){
  return args[1];
}

// extractingArgs(8,9,1); // Returns 9

function addAllArgs(...args){
  let sumOfArgs = 0;
  let i = 0;
  while(i < args.length){
    sumOfArgs += args[i];
    i++;
  }
  return sumOfArgs;
}

addAllArgs(6, 5, 7, 99); // Returns 117
addAllArgs(1, 3, 4); // Returns 8
```

****Note- Rest parameter should always be used at the last parameter of a function:**

```
// Incorrect way to use rest parameter
function randomFunc(a,...args,c){
//Do something
}

// Correct way to use rest parameter
function randomFunc2(a,b,...args){
//Do something
}
```

Spread operator (…)

Although the syntax of spread operator is exactly the same as the rest parameter, spread operator is used to spread an array, and object literals. We also use spread operators where one or more arguments are expected in a function call.

```

```

```
function addFourNumbers(num1,num2,num3,num4){
  return num1 + num2 + num3 + num4;
}

let fourNumbers = [5, 6, 7, 8];


addFourNumbers(...fourNumbers);
// Spreads [5,6,7,8] as 5,6,7,8

let array1 = [3, 4, 5, 6];
let clonedArray1 = [...array1];
// Spreads the array into 3,4,5,6
console.log(clonedArray1); // Outputs [3,4,5,6]


let obj1 = {x:'Hello', y:'Bye'};
let clonedObj1 = {...obj1}; // Spreads and clones obj1
console.log(obj1);

let obj2 = {z:'Yes', a:'No'};
let mergedObj = {...obj1, ...obj2}; // Spreads both the objects and merges it
console.log(mergedObj);
// Outputs {x:'Hello', y:'Bye',z:'Yes',a:'No'};
```

*****Note- Key differences between rest parameter and spread operator:**

- **Rest parameter is used to take a variable number of arguments and turns into an array while the spread operator takes an array or an object and spreads it**
- **Rest parameter is used in function declaration whereas the spread operator is used in function calls.**

24. What is the use of promises in javascript?

Promises are used to handle asynchronous operations in javascript.

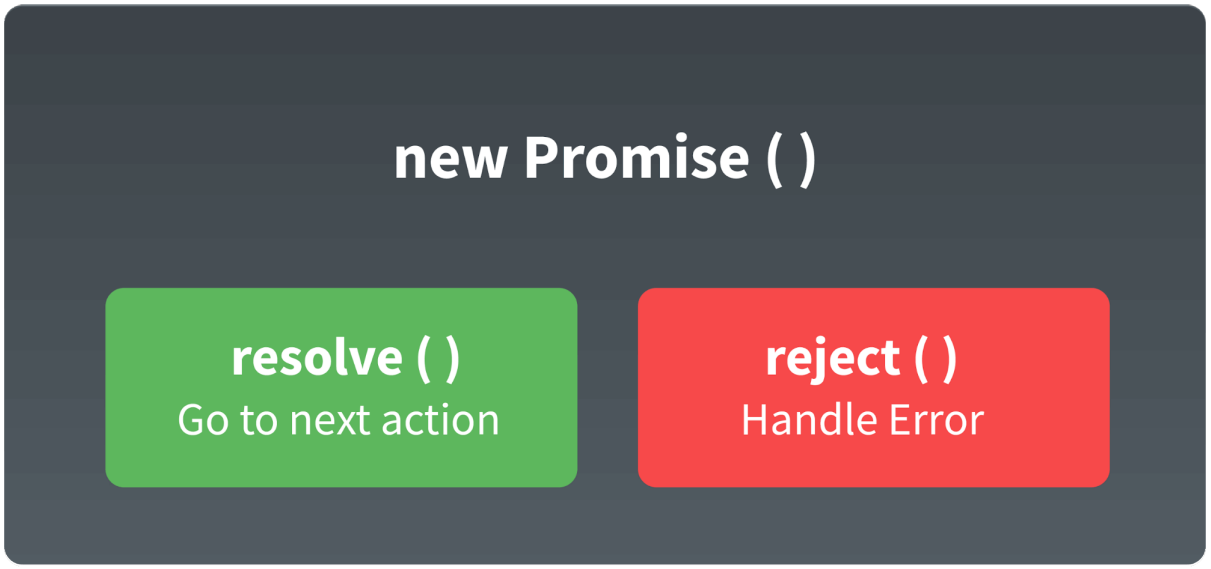
Before promises, callbacks were used to handle asynchronous operations. But due to limited functionality of callback, using multiple callbacks to handle asynchronous code can lead to unmanageable code.

Promise object has four states -

- Pending - Initial state of promise. This state represents that the promise has neither been fulfilled nor been rejected, it is in the pending state.
- Fulfilled - This state represents that the promise has been fulfilled, meaning the async operation is completed.

- Rejected - This state represents that the promise has been rejected for some reason, meaning the async operation has failed.
- Settled - This state represents that the promise has been either rejected or fulfilled.

A promise is created using the **Promise** constructor which takes in a callback function with two parameters, **resolve** and **reject** respectively.



resolve is a function that will be called, when the async operation has been successfully completed.

reject is a function that will be called, when the async operation fails or if some error occurs.

Example of a promise:

Promises are used to handle asynchronous operations like server requests, for the ease of understanding, we are using an operation to calculate the sum of three elements.

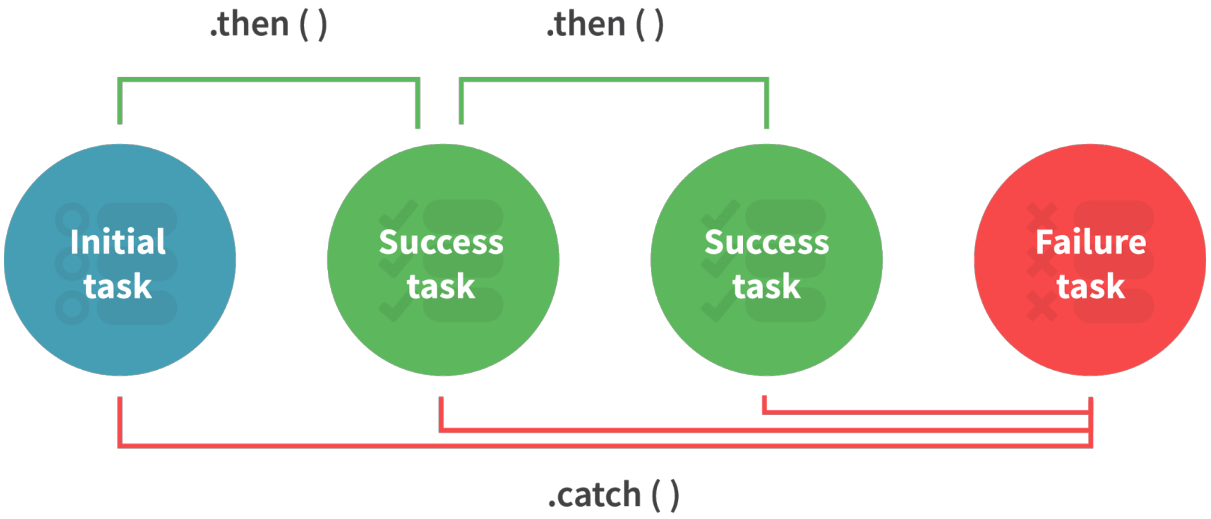
In the function below, we are returning a promise inside a function:

```
```

```
function sumOfThreeElements(...elements){
  return new Promise((resolve,reject)=>{
    if(elements.length > 3 ){
      reject("Only three elements or less are allowed");
    }
    else{
      let sum = 0;
      let i = 0;
      while(i < elements.length){
        sum += elements[i];
        i++;
      }
      resolve("Sum has been calculated: "+sum);
    }
  })
}
```

In the code above, we are calculating the sum of three elements, if the length of elements array is more than 3, promise is rejected, else the promise is resolved and the sum is returned.

We can consume any promise by attaching then() and catch() methods to the consumer.



then() method is used to access the result when the promise is fulfilled.

catch() method is used to access the result/error when the promise is rejected.

In the code below, we are consuming the promise:

```
```

```
sumOfThreeElements(4, 5, 6)
.then(result=> console.log(result))
.catch(error=> console.log(error));
// In the code above, the promise is fulfilled so the then() method gets executed

sumOfThreeElements(7, 0, 33, 41)
.then(result => console.log(result))
.catch(error=> console.log(error));
// In the code above, the promise is rejected hence the catch() method gets executed
```

25.What are classes in javascript?

Introduced in the ES6 version, classes are nothing but syntactic sugars for constructor functions.

They provide a new way of declaring constructor functions in javascript.

Below are the examples of how classes are declared and used:

```
// Before ES6 version, using constructor functions
function Student(name,rollNumber,grade,section){
  this.name = name;
  this.rollNumber = rollNumber;
  this.grade = grade;
  this.section = section;
}

// Way to add methods to a constructor function
Student.prototype.getDetails = function(){
  return 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade: ${this.grade}, Section:${this.
}


let student1 = new Student("Vivek", 354, "6th", "A");
student1.getDetails();
// Returns Name: Vivek, Roll no:354, Grade: 6th, Section:A

// ES6 version classes
class Student{
  constructor(name,rollNumber,grade,section){
    this.name = name;
    this.rollNumber = rollNumber;
    this.grade = grade;
    this.section = section;
  }

  // Methods can be directly added inside the class
  getDetails(){
    return 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade:${this.grade}, Section:${thi
  }
}

let student2 = new Student("Garry", 673, "7th", "C");
student2.getDetails();
// Returns Name: Garry, Roll no:673, Grade: 7th, Section:C
```

Key points to remember about classes:

- Unlike functions, classes are not hoisted. A class cannot be used before it is declared.
- A class can inherit properties and methods from other classes by using the extend keyword.
- All the syntaxes inside the class must follow the strict mode(‘use strict’) of javascript. Error will be thrown if the strict mode rules are not followed.

26. What are generator functions?

Introduced in ES6 version, generator functions are a special class of functions.

They can be stopped midway and then continue from where it had stopped.

Generator functions are declared with the **function*** keyword instead of the normal **function** keyword:

```
function* genFunc(){
  // Perform operation
}
```

In normal functions, we use the **return** keyword to return a value and as soon as the return statement gets executed, the function execution stops:

```
function normalFunc(){
  return 22;
  console.log(2); // This line of code does not get executed
}
```

In the case of generator functions, when called, they do not execute the code, instead they return a **generator object** . This generator object handles the execution.

```
function* genFunc(){
  yield 3;
  yield 4;
}
genFunc(); // Returns Object [Generator] {}
```

The generator object consists of a method called **next()** , this method when called, executes the code until the nearest **yield** statement, and returns the yield value.

For example if we run the next() method on the above code:

```
genFunc().next(); // Returns {value: 3, done:false}
```

As one can see the next method returns an object consisting of **value** and **done** properties.
Value property represents the yielded value.
Done property tells us whether the function code is finished or not. (Returns true if finished)
Generator functions are used to return iterators. Let’s see an example where an iterator is returned:

```

```

```
function* iteratorFunc() {
  let count = 0;
  for (let i = 0; i < 2; i++) {
      count++;
      yield i;
  }
  return count;
}

let iterator = iteratorFunc();
console.log(iterator.next()); // {value:0,done:false}
console.log(iterator.next()); // {value:1,done:false}
console.log(iterator.next()); // {value:2,done:true}
```

As you can see in the code above, the last line returns **done:true** , since the code reaches the return statement.

27. Explain WeakSet in javascript.

In javascript, Set is a collection of unique and ordered elements.

Just like Set, WeakSet is also a collection of unique and ordered elements with some key differences:

- Weakset contains only objects and no other type.
- An object inside the weakset is referenced weakly. This means, if the object inside the weakset does not have a reference, it will be garbage collected.
- Unlike Set, WeakSet only has three methods, **add()** , **delete()** and **has()** .

```
const newSet = new Set([4, 5, 6, 7]);
console.log(newSet);// Outputs Set {4,5,6,7}

const newSet2 = new WeakSet([3, 4, 5]); //Throws an error


let obj1 = {message:"Hello world"};
const newSet3 = new WeakSet([obj1]);
console.log(newSet3.has(obj1)); // true
```

28. Explain WeakMap in javascript.

In javascript, Map is used to store key-value pairs. The key-value pairs can be of both primitive and non-primitive types.

WeakMap is similar to Map with key differences:

- The keys and values in weakmap should always be an object.
- If there are no references to the object, the object will be garbage collected.

```
const map1 = new Map();
map1.set('Value', 1);

const map2 = new WeakMap();
map2.set('Value', 2.3); // Throws an error

let obj = {name:"Vivek"};
const map3 = new WeakMap();
map3.set(obj, {age:23});
```

29. What is Object Destructuring?

Object destructuring is a new way to extract elements from an object or an array.

Object destructuring:

Before ES6 version:

```
const classDetails = {
  strength: 78,
  benches: 39,
  blackBoard:1
}

const classStrength = classDetails.strength;
const classBenches = classDetails.benches;
const classBlackBoard = classDetails.blackBoard;
```

The same example using object destructuring:

```
```

```
const classDetails = {
  strength: 78,
  benches: 39,
  blackBoard:1
}

const {strength:classStrength, benches:classBenches,blackBoard:classBlackBoard} = classD

console.log(classStrength); // Outputs 78
console.log(classBenches); // Outputs 39
console.log(classBlackBoard); // Outputs 1
```

As one can see, using object destructuring we have extracted all the elements inside an object in one line of code.

If we want our new variable to have the same name as the property of an object we can remove the colon:

```
const {strength:strength} = classDetails;
// The above line of code can be written as:
const {strength} = classDetails;
```

Array destructuring:

Before ES6 version:

```
const arr = [1, 2, 3, 4];
const first = arr[0];
const second = arr[1];
const third = arr[2];
const fourth = arr[3];
```

The same example using object destructuring:

```
const arr = [1, 2, 3, 4];
const [first,second,third,fourth] = arr;

console.log(first); // Outputs 1
console.log(second); // Outputs 2
console.log(third); // Outputs 3
console.log(fourth); // Outputs 4
```

30. What is a Temporal Dead Zone?

Temporal Dead Zone is a behaviour that occurs with variables declared using **let** and **const** keywords.

It is a behaviour where we try to access a variable before it is initialized.

Examples of temporal dead zone:

```
x = 23; // Gives reference error

let x;


function anotherRandomFunc(){
  message = "Hello"; // Throws a reference error

  let message;
}
anotherRandomFunc();
```

In the code above, both in global scope and functional scope, we are trying to access variables which have not been declared yet. This is called the **Temporal Dead Zone** .

Coding problems:

31. Guess the outputs of the following codes:

```

```

```
// Code 1:

function func1(){
  setTimeout(()=>{
    console.log(x);
    console.log(y);
  },3000);

  var x = 2;
  let y = 12;
}

func1();

// Code 2:

function func2(){
  for(var i = 0; i < 3; i++){
    setTimeout(()=> console.log(i),2000);
}

}

func2();


// Code 3:


(function(){
  setTimeout(()=> console.log(1),2000);
  console.log(2);
  setTimeout(()=> console.log(3),0);
  console.log(4);
})();
```

Answers:

Code 1 - Outputs **2** and **12** . Since, even though **let** variables are not hoisted, due to async nature of javascript, the complete function code runs before the setTimeout function. Therefore, it has access to both x and y.

Code 2 - Outputs **3** , three times since variable declared with **var** keyword does not have block scope. Also, inside the for loop, the variable i is incremented first and then checked.

Code 3 - Output in the following order:

```
2
4
3
1 // After two seconds
```

Even though the second timeout function has a waiting time of zero seconds, the javascript engine always evaluates the setTimeout function using the Web API and therefore, the complete function executes before the setTimeout function can execute.

32.Guess the outputs of the following code:

```
// Code 1:

let x= {}, y = {name:"Ronny"},z = {name:"John"};

x[y] = {name:"Vivek"};
x[z] = {name:"Akki"};

console.log(x[y]);



// Code 2:

function runFunc(){
  console.log("1" + 1);
  console.log("A" - 1);
  console.log(2 + "-2" + "2");
  console.log("Hello" - "World" + 78);
  console.log("Hello"+ "78");
}

runFunc();


// Code 3:

let a = 0;
let b = false;
console.log((a == b));
console.log((a === b));
```

Answers:

Code 1 - Output will be **{name: “Akki”}.**

Adding objects as properties of another object should be done carefully.

Writing **x[y] = {name:”Vivek”}** , is same as writing **x[‘object Object’] = {name:”Vivek”}** ,

While setting a property of an object, **javascript coerces the parameter into a string.**

Therefore, since **y** is an object, it will be converted to **‘object Object’.**

Both x[y] and x[z] are referencing the same property.

Code 2 - Outputs in the following order:

```
11
Nan
2-22
NaN
Hello78
```

Code 3 - Output in the following order due to equality coercion:

```
true
false
```

33. Guess the output of the following code:

```
var x = 23;

(function(){
  var x = 43;
  (function random(){
    x++;
    console.log(x);
    var x = 21;
  })();
})();
```

Answer:

Output is **NaN** .

random() function has functional scope, since x is declared and hoisted in the functional scope.

Rewriting the random function will give a better idea about the output:

```
function random(){
  var x; // x is hoisted
  x++; // x is not a number since it is not initialized yet
  console.log(x); // Outputs NaN
  x = 21; // Initialization of x
}
```

34. Guess the outputs of the following code:

```
// Code 1

let hero = {
  powerLevel: 99,
  getPower(){
    return this.powerLevel;
  }
}

let getPower = hero.getPower;

let hero2 = {powerLevel:42};
console.log(getPower());
console.log(getPower.apply(hero2));



// Code 2

const a = function(){
  console.log(this);

  const b = {
    func1: function(){
      console.log(this);
    }
  }

  const c = {
    func2: ()=>{
      console.log(this);
    }
  }

  b.func1();
  c.func2();
}

a();



// Code 3

const b = {
  name:"Vivek",
  f: function(){
    var self = this;
    console.log(this.name);
    (function(){
      console.log(this.name);
      console.log(self.name);
    })();
  }
}

b.f();
```

Answers:

Code 1 - Output in the following order:

```
undefined
42
```

Reason - The first output is **undefined** since when the function is invoked, it is invoked referencing the global object:

```
window.getPower() = getPower();
```

Code 2 - Outputs in the following order:

```
global/window object
object "b"
global/window object
```

Since we are using arrow function inside **func2, this** keyword refers to the global object.

Code 3 - Outputs in the following order:

```
"Vivek"
undefined
"Vivek"
```

Only in the IIFE inside the function **f** , the **this** keyword refers to the global/window object.

35. Guess the outputs of the following code:

**Note - Code 2 and Code 3 require you to modify the code, instead of guessing the output.

```
```

```
// Code 1

(function(a){
  return (function(){
    console.log(a);
    a = 23;
  })()
})(45);


// Code 2

// Each time bigFunc is called, an array of size 700 is being created,
// Modify the code so that we don't create the same array again and again

function bigFunc(element){
  let newArray = new Array(700).fill('♥');
  return newArray[element];
}

console.log(bigFunc(599)); // Array is created
console.log(bigFunc(670)); // Array is created again



// Code 3

// The following code outputs 2 and 2 after waiting for one second
// Modify the code to output 0 and 1 after one second.

function randomFunc(){
  for(var i = 0; i < 2; i++){
    setTimeout(()=> console.log(i),1000);
  }
}

randomFunc();
```

Answers -

Code 1 - Outputs **45** .

Even though a is defined in the outer function, due to closure the inner functions have access to it.

Code 2 - This code can be modified by using closures,

```
function bigFunc(){
  let newArray = new Array(700).fill('♥');
  return (element) => newArray[element];
}

let getElement = bigFunc(); // Array is created only once
getElement(599);
getElement(670);
```

Code 3 - Can be modified in two ways:

Using **let** keyword:

```
function randomFunc(){
  for(let i = 0; i < 2; i++){
    setTimeout(()=> console.log(i),1000);
  }
}

randomFunc();
```

Using closure:

```
function randomFunc(){
  for(var i = 0; i < 2; i++){
  (function(i){
      setTimeout(()=>console.log(i),1000);
    })(i);
  }
}

randomFunc();
```

36. Write a function that performs binary search on a sorted array.

Answer:

```

```

```
function binarySearch(arr,value,startPos,endPos){
  if(startPos > endPos) return -1;

  let middleIndex = Math.floor(startPos+endPos)/2;

  if(arr[middleIndex] === value) return middleIndex;

  elsif(arr[middleIndex > value]){
    return binarySearch(arr,value,startPos,middleIndex-1);
  }
  else{
    return binarySearch(arr,value,middleIndex+1,endPos);
  }
}
```

37. Implement a function that returns an updated array with `r` right rotations on an array of integers `a`.

Example:

Given the following array:

[2,3,4,5,7]

Perform **3** right rotations:

First rotation : [7,2,3,4,5] , Second rotation : [5,7,2,3,4] and, Third rotation: [4,5,7,2,3]

return **[4,5,7,2,3]**

Answer:

```
function rotateRight(arr,rotations){
  if(rotations == 0) return arr;
  for(let i = 0; i < rotations;i++){
    let element = arr.pop();
    arr.unshift(element);
  }

  return arr;
}


rotateRight([2, 3, 4, 5, 7], 3); // Return [4,5,7,2,3]
rotateRight([44, 1, 22, 111], 5); // Returns [111,44,1,22]
```