

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

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

```
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.

```
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.

```
var bigInteger = 234567890123456789012345678901234567890n;
```

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

```
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.

```
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.

```
var z = null;
```

Object - Used to store collection of data.

// Collection of data in key-value pairs

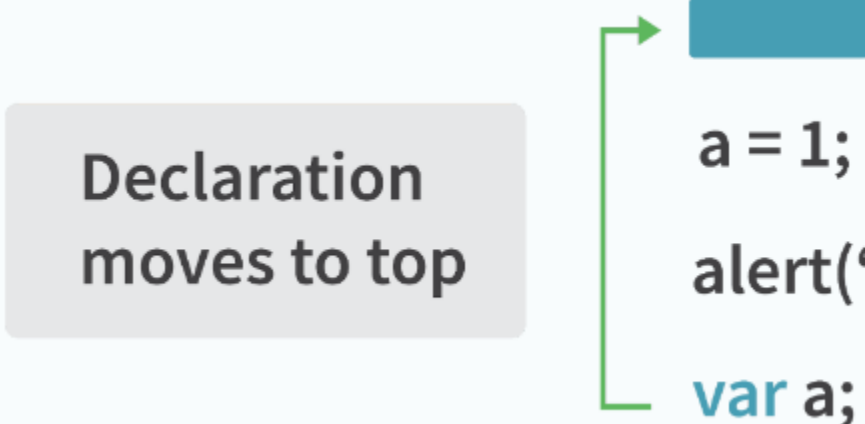
```
var obj1 = {  
  x: 43,  
  y: "Hello world!",
```

```
z: function(){  
  return this.x;  
}  
}  
// Collection of data as an ordered list  
var array1 = [5, "Hello", true, 4.1];
```

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.



Declaration
moves to top

```
a = 1;  
alert(''  
var a;
```

Example 1:

```
hoistedVariable = 3;  
console.log(hoistedVariable); // outputs 3 even when the variable is  
declared after it is initialized  
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 false, 0, 0n, -0, "", null, undefined and NaN are truthy values.

If statements:

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

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.

```
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 compared. Hence the operands are equal.

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

```
var a = 226;
```

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

a === b // Returns false because coercion does not take place and the operands are of different types. Hence they are not equal.

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.

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!";
```

Static vs Dynamic Typing

Static typing	Dynamic typing
<pre>string name ; name = "John" ; name = 34 ;</pre>	<pre>var name ; name = "John" ; name = 34 ;</pre>
Variables have types	Variables have no types
Values have types	Values have types
Variables cannot change type	Variables change type dramatically

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 1 (a number)

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.

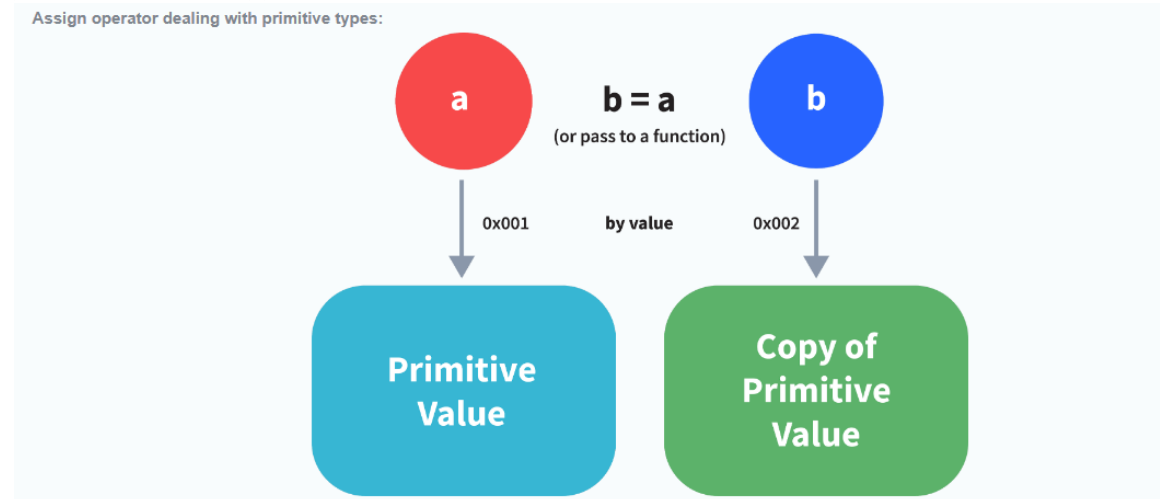
In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.

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,



```
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;
```

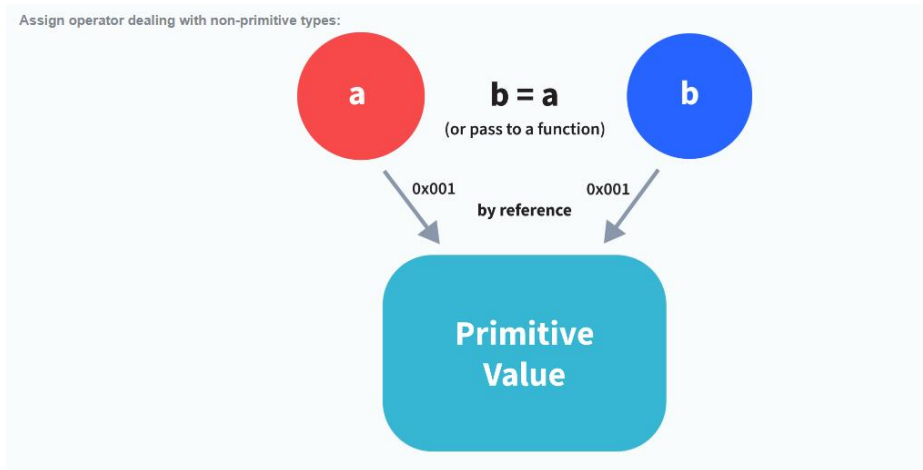
```
var z = #5411; // z pointing to a completely new address of the value 234
```

```
// Changing the value of y
```

```
y = 23;
```

```
console.log(z); // Returns 234, since z points to a new address in the memory so changes in y will not effect z
```

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.



```
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 address.
```

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 inside the same function
    }
}
console.log(a); // Throws reference error since a is written in local scope
and cannot be accessed outside
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 block
```

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 favFunction, outputs 667
  }
  var yetAnotherFavFunction = function(){
    console.log(y); // Does not find y inside yetAnotherFavFunction, so looks
for variable inside favFunction and does not find it, so looks for variable in
global scope, finds it and outputs 24
  }
  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 randomFunc function is executed
}
}
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 Is the Prototype in JavaScript?

The prototype is a fundamental concept of JavaScript and its must to known JavaScript developers.

All the JavaScript objects have an object and its property called prototype and it's used to add and the custom functions and property.

The example without prototype as given below -

```

var employee = function () {
  //This is a constructor function.
}

```

```

//Crate the instance of above constructor function and assign in a variable
var empInstance = new employee();

```

```

emplInstance.deportment = "IT";
console.log(emplInstance.deportment);//The output of above is IT.
//The example with prototype as given below-
var employee = function () { //This is a constructor function.}
employee.prototype.deportment = "IT";//Now, for every instance
employee will have a department.
//Crate the instance of above constructor functions and assign in a variable
var emplInstance = new employee();
emplInstance.deportment = "HR";
console.log(emplInstance.deportment);//The output of above is IT not HR.

```

16. What are Callbacks, Promises and Async/Await?

Callbacks: A callback is a function that is passed to another function. When the first function is done, it will run the second function.

```

function printString(string, callback){
  setTimeout( () => {
    console.log(string)
    callback()
  }, Math.floor(Math.random() * 100) + 1
)
}

```

You can see that is super easy to modify the original function to work with callbacks.

Again, let's try to print the letters A, B, C in that order:

```

function printAll(){
  printString("A", () => {
    printString("B", () => {
      printString("C", () => {})
    })
  })
}
printAll()

```

Well, the code is a lot uglier now, but at least it works! Each time you call printAll, you get the same result.

The problem with callbacks is it creates something called "Callback Hell." Basically, you start nesting functions within functions within functions, and it starts to get really hard to read the code.

Here it is in callback style:

```
function addString(previous, current, callback){
  setTimeout(() => {
    callback((previous + ' ' + current))
  }, Math.floor(Math.random() * 100) + 1
)}
```

And in order to call it:

```
function addAll(){
  addString('', 'A', result => {
    addString(result, 'B', result => {
      addString(result, 'C', result => {
        console.log(result) // Prints out " A B C"
      })
    })
  })
}
addAll()
```

Promises: Promises try to fix this nesting problem. Let's change our function to use Promises

```
function printString(string){
  return new Promise((resolve, reject) => {
    setTimeout( () => {
      console.log(string)
      resolve()
    }, Math.floor(Math.random() * 100) + 1
  ))
}
```

You can see that it still looks pretty similar. You wrap the whole function in a Promise, and instead of calling the callback, you call resolve (or reject if there is an error). The function returns this Promise object.

Again, let's try to print the letters A, B, C in that order:

```
function printAll(){
  printString("A")
  .then(() => {
    return printString("B")
  })
  .then(() => {
```

```

    return printString("C")
  })
}

```

```
printAll()
```

This is called a Promise Chain. You can see that the code returns the result of the function (which will be a Promise), and this gets sent to the next function in the chain.

The code is no longer nested but it still looks messy!

By using features of arrow functions, we can remove the “wrapper” function. The code becomes cleaner, but still has a lot of unnecessary parenthesis:

```

function printAll(){
  printString("A")
  .then(() => printString("B"))
  .then(() => printString("C"))
}
printAll()

```

Here it is in Promise style:

```

function addString(previous, current){
  return new Promise((resolve, reject) => {
    setTimeout(
      () => {
        resolve(previous + ' ' + current)
      },
      Math.floor(Math.random() * 100) + 1
    )
  })
}

```

And in order to call it:

```

function addAll(){
  addString('', 'A')
  .then(result => {
    return addString(result, 'B')
  })
  .then(result => {
    return addString(result, 'C')
  })
}

```

```

.then(result => {
  console.log(result) // Prints out " A B C"
})
}
addAll()

```

Using arrow functions means we can make the code a little nicer:

```

function addAll(){
  addString('', 'A')
  .then(result => addString(result, 'B'))
  .then(result => addString(result, 'C'))
  .then(result => {
    console.log(result) // Prints out " A B C"
  })
}
addAll()

```

This is definitely more readable, especially if you add more to the chain, but still a mess of parenthesis.

Await: Await is basically syntactic sugar for Promises. It makes your asynchronous code look more like synchronous/procedural code, which is easier for humans to understand.

The printString function doesn't change at all from the promise version.

Again, let's try to print the letters A, B, C in that order:

```

async function printAll(){
  await printString("A")
  await printString("B")
  await printString("C")
}
printAll()

```

You might notice that we use the "async" keyword for the wrapper function printAll. This lets JavaScript know that we are using async/await syntax, and is necessary if you want to use Await. This means you can't use Await at the global level; it always needs a wrapper function. Most JavaScript code runs inside a function, so this isn't a big deal.

The function stays the same as the Promise version.

And in order to call it:

```

async function addAll(){

```

```

let toPrint = await addString('', 'A')
toPrint = await addString(toPrint, 'B')
toPrint = await addString(toPrint, 'C')
console.log(toPrint) // Prints out " A B C"
}
addAll()

```

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.

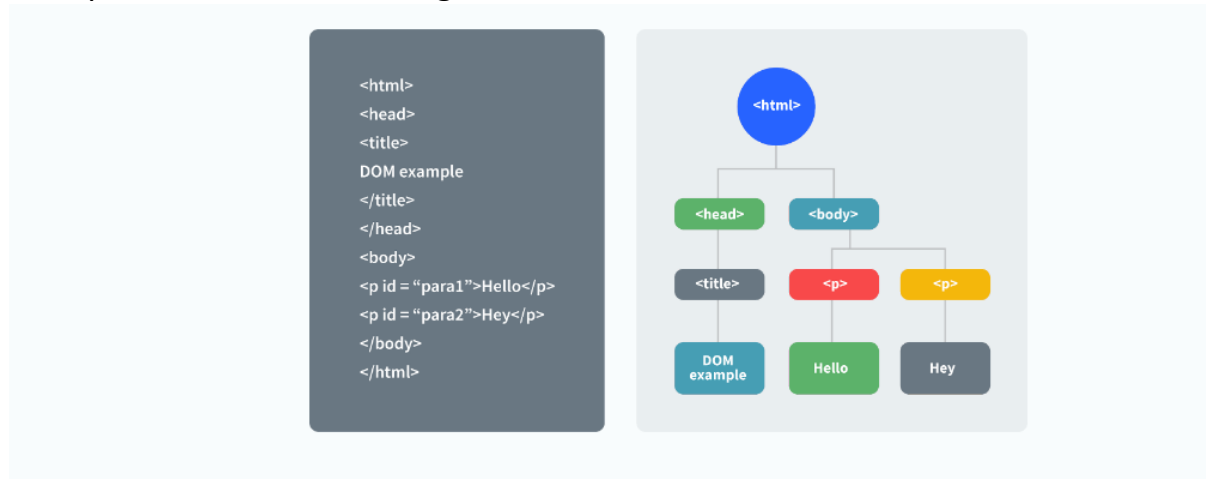
20. What is DOM?

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:



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. What is the difference between let, var, and const?

I'm explaining the importance of these three keywords in JavaScript and TypeScript. I also provided various examples to deeply understand and use them and what happened when?

Let see the detail about the - var vs. let vs. const

var: -

1. var is function-scoped
2. var returns undefined when accessing a variable before it's declared

let: -

1. let is block-scoped
2. let throw ReferenceError when accessing a variable before it's declared

const:-

1. Const is block-scoped
2. Const throw ReferenceError when accessing a variable before it's declared
3. Const cannot be reassigned

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.section}';
}
```

```
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:${this.section}';
  }
}
```

```
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} = classDetails;
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 .

31. What Is the Type of JavaScript?

There are different of Type as given below.

1. String,

2. Number,

3. Boolean,
4. Function
5. Object,
6. Null,
7. Undefined etc.

32. What Types of Boolean Operators in JavaScript?

There are three types of Boolean operators as given below.

1. AND (&&) operator,
2. OR (||) operator and
3. NOT (!) Operator

33. What Is the Difference Between “==” and “===”?

The double equal “==” is an auto-type conversion and it checks only value not type.

The three equal “===” is not auto-type conversion and it checks value and types both.

The example looks like–

```
if(1 == "1")
```

//Its returns true because it's an auto-type conversion and it checks only value not type.

```
if(1 === "1")
```

//Its returns false because it's not auto-type conversion and it checks value and types both.

```
if(1=== parseInt("1"))
```

// its returns true.

```
// alert(0 == false); // return true, because both are same type.
```

```
// alert(0 === false); // return false, because both are of a different type.
```

```
// alert(1 == "1"); // return true
```

34. What Is an Object?

The object is a collection of properties and each property associated with the name-value pairs. The object can contain any data types (numbers, arrays, object, etc.)

The example as given below –

```
var myObject= {empId : "001", empCode : "X0091"};
```

In the above example, here are two properties one is empId and other is empCode and its values are "001" and "X0091".

The properties name can be string or number. If a property name is number i.e.

```
var numObject= {1 : "001", 2 : "X0091"};
```

```
Console.log(numObject.1); //This line throw an error.
```

```
Console.log(numObject["1"]); // will access to this line not get any error.
```

As per my thought, the number of property name should be avoided.

Types of creating an object

1. Object Literals
2. Object constructor

Object Literals: This is the most common way to create the object with an object literal and the example as given below.

The empty object initialized using object literal i.e.

```
var emptyObj = {};
```

This is an object with 4 items using object literal i.e.

```
var emptyObj = {  
  empId: "Red",  
  empCode: "X0091",  
  empDetail : function(){  
    alert("Hi");  
  };  
};
```

Object Constructor: The second way to create an object using object constructor and the constructor is a function used to initialize a new object.

The example as given below -

```
var obj = new Object();  
Obj.empId="001";  
Obj.empCode="X0091";  
Obj.empAddressDetail = function(){  
  console.log("Hi, I Anil");  
};
```

35. How To Achieve Inheritance in JavaScript?

In the JavaScript, we can implement the inheritance using some alternative ways and we cannot define a class keyword but we create a constructor function and using new keyword achieves it.

Some alternative ways as given below –

1. Pseudo-classical inheritance
2. Prototype inheritance

Pseudo-classical inheritance is the most popular way. In this way, create a constructor function using the new operator and add the members function with the help for constructor function prototype.

Prototype-based programming is a technique of object-oriented programming. In this mechanism, we can reuse the existing objects as prototypes. The prototype inheritance also is known as prototypal, Classless or instance based inheritances.

The Inheritance example for prototype based as given below –

//create a helper function.

```
if (typeof Object.create !== 'function') {  
  Object.create = function (obj) {  
    function fun() {};  
    fun.prototype = obj;  
    return new fun();  
  };  
}
```

//This is a parent class.

```
var parent = {  
  sayHi: function () {  
    alert('Hi, I am parent!');  
  },  
  sayHiToWalk: function () {  
    alert('Hi, I am parent! and going to walk!');  
  }  
};
```

//This is child class and the parent class is inherited in the child class.

```
var child = Object.create(parent);  
child.sayHi = function () {  
  alert('Hi, I am a child!');  
};
```

HTML –

```
<button type="submit" onclick="child.sayHi()"> click to oops</button>
```

//The output is: Hi, I am a child!

36. What Is the typeof Operator?

The type of operator is used to find the type of variables.

The example as given below -

```
typeof "Anil Singh" // Returns string
typeof 3.33         // Returns number
typeof true         // Returns Boolean
typeof { name: 'Anil', age: 30 } // Returns object
typeof [10, 20, 30, 40] // Returns object
```

37. What Is a Public, Private and Static Variable in JavaScript?

I am going to explain like strongly typed object-oriented language (OOPs) like (C#, C++, and Java, etc.).

Fist I am creating a conductor class and trying to achieve to declare the public, private and static variables and detail as given below –

function myEmpConsepts() { // This myEmpConsepts is a constructor function.

var empId = "00201"; //This is a private variable.

this.empName = "Anil Singh"; //This is a public variable.

this.getEmpSalary = function () { //This is a public method

console.log("The getEmpSalary method is a public method")

}

}

//This is an instance method and its call at the only one time when the call is instantiate.

myEmpConsepts.prototype.empPublicDetail = function () {

console.log("I am calling public variable in the instance method : " +

this.empName);

}

//This is a static variable and it's shared by all instances.

myEmpConsepts.empStaticVaiable = "Department";

var instanciateToClass = new myEmpConsepts();

38. How to Add/Remove Properties to Object in run-time in JavaScript?

I am going to explain by example for add and remove properties from JavaScript objects as give below.

This example for delete property -

//This is the JSON object.


```

var objectJSON = {
  id: 1,
  name: "Anil Singh",
  dept: "IT"
};
//This is the process to delete
delete objectJSON.dept;
//Delete property by the array collection
MyArrayColection.prototype.remove = function (index) {
  this.splice(index, 3);
}
This example for add property -
//This is used to add the property.
objectJSON.age = 30;
console.log(objectJSON.age); //The result is 30;
//This is the JSON object.
var objectJSON = {
  id: 1,
  name: "Anil Singh",
  dept: "IT",
  age: 30
};

```

39. Why Never Use New Array in JavaScript?

We have some fundamental issues with new Array () the example in detail for array constructor function as given below.

When Array Have more the one Integer?

```

var newArray = new Array(10, 20, 30, 40, 50);
console.log(newArray[0]); //returns 10.
console.log(newArray.length); //returns 5.

```

When Array Have Only One Integer?

```

var newArray = new Array(10);
console.log(newArray[0]); //returns undefined
console.log(newArray.length); //returns 10 because it has an error "array
index out of bound";
//This is the fundamental deference to need to avoid the new array ();

```

40. What is eval() and floor() functions in JavaScript?

The eval() function used to execute an argument as an expression or we can say that evaluate a string as expression and it used to parse the JSON.

The example over eval() function as given below -

```
var x = 14;  
eval('x + 10'); //The output is 24.  
Another over eval() function example -  
eval('var myEval = 10');  
console.log(myEval); // The output is 10.
```

The floor () function is a static method of Math and we can write as Math.floor() and used to round the number of downwards i.e.

```
Math.floor(1.6); //The output is 1.
```

41. What is join() and isNaN() functions in JavaScript?

The is join() function used to join the separator in the array.

Syntax -

```
myArray.join(mySeparator);  
The example as given below -  
var alphabets = ["A", "B", "C", "D"];  
//Join without separator  
var result1 = alphabets.join(); //The output is A B C D.  
//Join with separator.  
var result2 = alphabets.join(','); //The output is A, B, C, D.
```

The isNaN() function is used to check the value is not-a-number.

The example as given below -

```
var var1 = isNaN(-1.23); //The output is false.  
var var2 = isNaN(3); //The output is false.  
var var3 = isNaN(0); //The output is false.  
var var3 = isNaN("10/03/1984"); //The output is true.
```

42. What Is Function Overloading in JavaScript?

There is no real function overloading in JavaScript and it allows passing any number of parameters of any type.

You have to check inside the function how many arguments have been passed and what is the type arguments using typeof.

The example for function overloading not supporting in JavaScript as gives below -

```
function sum(a, b) {  
    alert(a + b);  
}  
function sum(c) {  
    alert(c);  
}
```

sum(3);//The output is 3.

sum(2, 4);//The output is 2.

In the JavaScript, when we write more than one functions with the same name that time JavaScript considers the last define function and override the previous functions. You can see the above example output for the same.

We can achieve using the several different techniques as given below -

- 1.You can check the declared argument name value is undefined.
- 2.We can check the total arguments with arguments.length.
- 3.Checking the type of passing arguments.
- 4.Using a number of arguments
- 5.Using optional arguments like x=x || 'default'
- 6.Using a different name in the first place
- 7.We can use the arguments array to access any given argument by using arguments[i]

43. How to find length of Keys values in object

```
var obj = {  
    id: 1,  
    name: "Anil Singh",  
    dept: "IT",  
    age: 30  
};
```

Object.keys(obj).length

Object.values(obj).length

44. Difference between Anonymous and Named functions in JavaScript

Anonymous Functions: As the name suggests, “anonymous function” refers to a function that does not have a name or a title. In JavaScript, an

anonymous function is something that is declared without an identification. It's the distinction between a regular and an anonymous function. An anonymous function cannot be accessed after it is created; it can only be retrieved by a variable in which it is stored as a function value. There can be several arguments to an anonymous function, but only one expression.

```
var test = function () {  
    console.log("This is an anonymous function!");  
};  
test();
```

Named Functions: In JavaScript, named functions are simply a way of referring to a function that employs the function keyword followed by a name that can be used as a callback to that function. Normal functions with a name or identifier are known as named functions. They can be employed in an expression or declared in a statement. The function's name is stored within its body, which can be handy. And we may use the name to get a function to call itself, or to retrieve its attributes like every object.

```
function test() {  
    console.log(`This is a named function!`);  
};
```

45. Differences between ViewState and SessionState

SessionState: It can be used to store information that you wish to access on different web pages.

ViewState: It can be used to store information that you wish to access from same web page.

ViewState	SessionState
Maintained at page level only.	Maintained at session level.
View state can only be visible from a single page and not multiple pages.	Session state value availability is across all pages available in a user session.
It will retain values in the event of a postback operation occurring.	In session state, user data remains in the server. Data is available to user until the browser is closed or there is session expiration.
Information is stored on the client's end only.	Information is stored on the server.
used to allow the persistence of page-instance-specific data.	used for the persistence of user-specific data on the server's end.
ViewState values are lost/cleared when new page is loaded.	SessionState can be cleared by programmer or user or in case of timeouts.

46. setTimeout() & setInterval() Method

The setTimeout() method executes a function, after waiting a specified number of milliseconds.

There are two parameter that accepted by this method

function : first parameter is a function to be executed

milliseconds : indicates the number of milliseconds before execution takes place.

```
setTimeout(() => {
  console.log('shiva');
}, 5000);
var inter = setTimeout (() =>{
  console.log('shiva');
}, 100);
```

To stop setTimeout
clearTimeout(inter);

The setInterval() method repeats a given function at every given time-interval.

There are two parameter that accepted by this method

function : first parameter is the function to be executed

milliseconds :indicates the length of the time-interval between each execution.

```
setInterval(() =>{  
  console.log('shiva');  
}, 100);  
var inter = setInterval(() =>{  
  console.log('shiva');  
}, 100);
```

To stop setInterval
clearInterval(inter)

47. What are the Event Handlers in JavaScript?

when an event, such as clicking an element or pressing a keyboard key, occurs on an HTML or DOM element, we can invoke certain functions based on these events. So, how do the HTML element knows when to execute the mentioned JavaScript function or JavaScript code? The event handlers handle this. The event handlers are the properties of the HTML or DOM elements, which manages how the element should react to a specific event. when the user performs a particular mouse or keyword action on the browser, it triggers the corresponding event handler associated with that HTML element. The event handler, in turn, executes a piece of JavaScript code, which performs a particular action on the webpage, and the browser displays the results of those actions to the end-users.

Event Handler	
onclick	This event handler invokes a JavaScript code when a click action happens on an image or image map is selected, it can trigger the onClick event handler.
onload	This event handler invokes a JavaScript code when a window or image finish loading.
onmouseover	This event handler invokes a JavaScript code when we place the mouse over an element.
onmouseout	This event handler invokes a JavaScript code when the mouse leaves a part of the page.
onkeypress	This event handler invokes a JavaScript code when the user presses a key.
onkeydown	This event handler invokes a JavaScript code when during the keyboard action.
onkeyup	This event handler invokes a JavaScript code when during the keyboard action.

48. What is an event loop in JavaScript?

The **event loop** is the secret behind JavaScript's asynchronous programming. JS executes all operations on a single thread, but using a few smart data structures, it gives us the illusion of multi-threading. Let's take a look at what happens on the back-end.

The **call stack** is responsible for keeping track of all the operations in line to be executed. Whenever a function is finished, it is popped from the stack.

The **event queue** is responsible for sending new functions to the stack for processing. It follows the queue data structure to maintain the correct sequence in which all operations should be sent for execution.

Whenever an async function is called, it is sent to a browser API. These are APIs built into the browser. Based on the command received from the call stack, the API starts its own single-threaded operation.

An example of this is the **setTimeout** method. When a setTimeout operation is processed in the stack, it is sent to the corresponding API which waits till the specified time to send this operation back in for processing.

Where does it send the operation? The event queue. Hence, we have a cyclic system for running async operations in JavaScript. The language itself is single-threaded, but the browser APIs act as separate threads.

The event loop facilitates this process; it constantly checks whether or not the call stack is empty. If it is empty, new functions are added from the event queue. If it is not, then the current function call is processed.

49. Event Bubbling and Capturing in JavaScript

Understanding the Event Flow

Before jumping to bubbling and capturing, let's find out how an event is propagated inside the DOM.

If we have several nested elements handling the same event, we get confused about which event handler will trigger first. This is where understanding the event order becomes necessary.

Usually, an event is propagated towards the target element starting from its parents, and then it will propagate back towards its parent element.

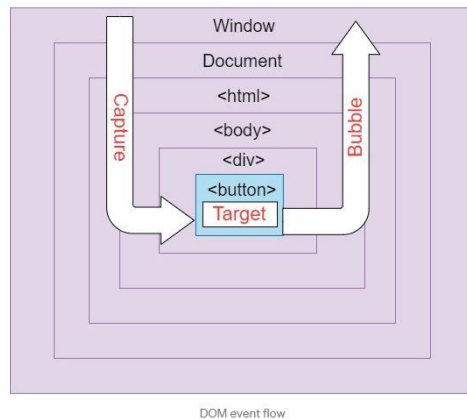
There are three phases in a JavaScript event,

Capture Phase: Event propagates starting from ancestors towards the parent of the target. Propagation starts from Window object.

Target Phase: The event reaches the target element or the element which started the event.

Bubble Phase: This is the reverse of capture. Event is propagated towards ancestors until Window object.

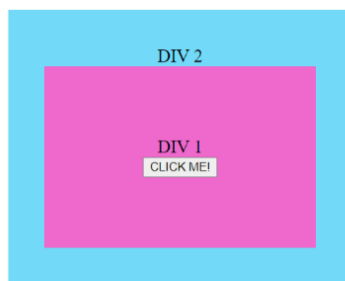
The following diagram will give you a further understanding of the event propagation life cycle.



What is Event Capturing?

Event capturing is the scenario in which the events will propagate, starting from the wrapper elements down to the target element that initiated the event cycle.

If you had an event bound to browser's Window, it would be the first to execute. So, in the following example, the order of event handling will be Window, Document, DIV 2, DIV 1, and finally, the button.



Here we can see that event capturing occurs only until the clicked element or the target. The event will not propagate to child elements.

We can use the `useCapture` argument of the `addEventListener()` method to register events for capturing phase.

`target.addEventListener(type, listener, useCapture)`

You can use the following snippet to test out the above example and get hands-on experience in event capturing.

```
window.addEventListener("click", () => {  
  console.log('Window');
```



```
    },true);

document.addEventListener("click", () => {
    console.log('Document');
},true);

document.querySelector(".div2").addEventListener("click", () => {
    console.log('DIV 2');
},true);

document.querySelector(".div1").addEventListener("click", () => {
    console.log('DIV 1');
},true);

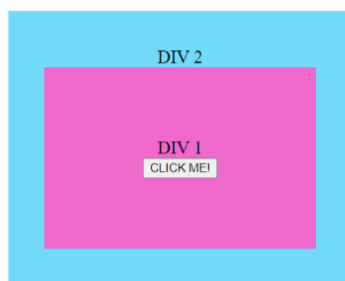
document.querySelector("button").addEventListener("click", () => {
    console.log('CLICK ME!');
},true);
```

What is Event Bubbling?

Event bubbling is pretty simple to understand if you know event capturing. It is the exact opposite of event capturing.

Event bubbling will start from a child element and propagate up the DOM tree until the topmost ancestor's event is handled.

Omitting or setting the `useCapture` argument to 'false' inside `addEventListener()` will register events for bubbling. So, the default behavior of Event Listeners is to bubble events.



In our examples, we used either event capturing or event bubbling for all events. But what if we want to handle events inside both phases? Let's take an example of handling the click events of Document and DIV 2 during the bubbling phase while others are handled during capturing.

The click events attached to Window, DIV 1, and button will fire respectively during the capturing, while DIV 2 and Document listeners are fired in order during the bubbling phase.

```
window.addEventListener("click", () => {  
    console.log('Window');  
},true);
```

```
document.addEventListener("click", () => {  
    console.log('Document');  
}); //registered for bubbling
```

```
document.querySelector(".div2").addEventListener("click", () => {  
    console.log('DIV 2');  
}); //registered for bubbling
```

```
document.querySelector(".div1").addEventListener("click", () => {  
    console.log('DIV 1');  
},true);
```

```
document.querySelector("button").addEventListener("click", () => {  
    console.log('CLICK ME!');  
},true);
```

We can use the **stopPropagation()** method to avoid this behavior. It will stop events from propagating further up or down along the DOM tree.

50. JavaScript String Methods

Methods	Description
charAt()	It provides the char value present at the specified index.
charCodeAt()	It provides the Unicode value of a character present at the specified index.
concat()	It provides a combination of two or more strings.
indexOf()	It provides the position of a char value present in the given string.
lastIndexOf()	It provides the position of a char value present in the given string.

	searching a character from the last position.
search()	It searches a specified regular expression in a given string and returns the position if a match occurs.
match()	It searches a specified regular expression in a given string and returns that regular expression if a match occurs.
replace()	It replaces a given string with the specified replacement.
substr()	It is used to fetch the part of the given string on the basis of specified starting position and length.
substring()	It is used to fetch the part of the given string on the basis of specified index.
slice()	It is used to fetch the part of the given string. It allows us to use a positive as well as a negative index.
toLowerCase()	It converts the given string into lowercase letters.
toLocaleLowerCase()	It converts the given string into lowercase letters on the basis of the current locale.
toUpperCase()	It converts the given string into uppercase letters.
toLocaleUpperCase()	It converts the given string into uppercase letters on the basis of the current locale.
toString()	It provides a string representing the particular object.
valueOf()	It provides the primitive value of the string object.
split()	It splits a string into a substring array, then returns that newly created array.
trim()	It trims the white space from the left and right side of the string.

51. JavaScript Array Methods

Methods	Description
concat()	It returns a new array object that contains two or more merged arrays.
copyWithin()	It copies the part of the given array with its own elements and returns a modified array.
entries()	It creates an iterator object and a loop that iterates over each key/value.
every()	It determines whether all the elements of an array are satisfying the provided function conditions.
flat()	It creates a new array carrying sub-array elements concatenated recursively up to the specified depth.
flatMap()	It maps all array elements via mapping function, then flattens the result into a new array.
fill()	It fills elements into an array with static values.
from()	It creates a new array carrying the exact copy of another array element.
filter()	It returns the new array containing the elements that pass the provided function conditions.
find()	It returns the value of the first element in the given array that satisfies the specified condition.
findIndex()	It returns the index value of the first element in the given array that satisfies the specified condition.
forEach()	It invokes the provided function once for each element of an array.
includes()	It checks whether the given array contains the specified element.
indexOf()	It searches the specified element in the given array and returns the index of the first match.

isArray()	It tests if the passed value is an array.
join()	It joins the elements of an array as a string.
keys()	It creates an iterator object that contains only the keys of the array, then you can iterate through these keys.
lastIndexOf()	It searches the specified element in the given array and returns the index of the last match.
map()	It calls the specified function for every array element and returns the new array.
of()	It creates a new array from a variable number of arguments, holding all arguments of the array of argument.
pop()	It removes and returns the last element of an array.
push()	It adds one or more elements to the end of an array.
reverse()	It reverses the elements of given array.
reduce(function, initial)	It executes a provided function for each value from left to right and reduces the array to a single value.
reduceRight()	It executes a provided function for each value from right to left and reduces the array to a single value.
some()	It determines if any element of the array passes the test of the implemented function.
shift()	It removes and returns the first element of an array.
slice()	It returns a new array containing the copy of the part of the given array.
sort()	It returns the element of the given array in a sorted order.
splice()	It add/remove elements to/from the given array.
toLocaleString()	It returns a string containing all the elements of a specified array.

toString()	It converts the elements of a specified array into string form, without a the original array.
unshift()	It adds one or more elements in the beginning of the given array.
values()	It creates a new iterator object carrying values for each index in the array.

52. Storing data in the browser

- 1.Cookies, document.cookie
- 2.LocalStorage, sessionStorage
- 3.IndexedDB

	Cookies	Local Storage	Session Storage
Capacity	4 kb	10 mb	5 mb
Browsers	HTML 4 / HTML 5	HTML 5	HTML 5
Accessible from	Any window	Any window	Same tab
Expires	Manually set	Never	On tab close
Storage Location	Browser and server	Browser only	Browser only
Sent with requests	Yes	No	No

Cookies: Cookies are the oldest way of storing data in the browser. It is available in both HTML4 and HTML5. Cookies are basically some text-based data with a name-value pair. Cookies can only store 4 KB of data which is much smaller than local storage and session storage.

Cookies are a convenient way to carry data from one session to another on a website. It can be used for authentication purposes and also for identifying the state of a user. For example —

When a user logs in to a website, the server sets a cookie. So the server can recognize that the user is logged in for other sessions.

When shopping from an e-commerce website, if a user adds an item to the cart, usually a cookie is set. So if the user refreshes the page the item will still be in the cart list, and the user can add more items from this state.

```
document.cookie = "user=Shiva"; // update only cookie named 'user'  
alert(document.cookie); // show all cookies
```

Local Storage

Depending on the browser, local storage has a capacity of 5-10 MB. It was introduced in HTML5. Local storage stores data only in the browser and never expires unless manually removed. Local storage can be accessed from any window. That means if a user closes the browser local storage will be saved next time the user opens it again.

Access local storage with JavaScript

You can access local storage by using the localStorage property. Local storages are string values with key-value pairs. See the example below.

```
localStorage.setItem('key1', 'val1');  
console.log(localStorage.getItem('key1'));  
localStorage.clear() // will clear the entire local storage.  
localStorage.removeItem('key1'); // To remove one item you can use  
The array arr will be stored in the local storage. But the whole array will be  
stored as a string. To solve this problem you can use JSON.stringify() and  
JSON.parse().  
let arr = ['val1', 'val2', 'val3'];  
localStorage.setItem('keys', JSON.stringify(arr));  
x = JSON.parse(localStorage.getItem('keys'));  
console.log(x);
```

Session Storage

Session storage is mostly similar to local storage. The only difference is session storage will be deleted after a session. That means if you save something in session storage and close your browser you will not be able to

see the data after you open the browser again. Session storage has a capacity of 5 MB.

Access session storage with JavaScript

All the operations described in local storage are also valid for session storage. To access session storage you need to use the sessionStorage property. Others are pretty much the same as local storage.

```
sessionStorage.setItem('key1', 'val1');
```

As you can see val1 is added to the session storage.

sessionStorage.getItem(), sessionStorage.removeItem(), and sessionStorage.clear() will work exactly the same as local storage.

If you close the browser tab the session storage will be gone.

Local storage and session storage together can be referred to as web storage. Web storage is used to save data for quick access. Like users custom data. For example —

Suppose a website has light/dark mode. If a user sets the dark mode in the browser, the data can be stored in local storage. The data is saved in local storage for this user only. This way when the user opens the website from the browser again, dark mode will be enabled automatically.

53. Static vs Dynamic typing

JavaScript is a dynamically typed language, but TypeScript is a statically typed language.

In dynamically typed languages all type checks are performed in a runtime, only when your program is executing. So this means you can just assign anything you want to the variable and it will work. When you declare a variable, you do not need to specify what type this variable is. Javascript engine infers what type this variable is based on the value assigned to at run time.

```
let a
```

```
a = 0
```

```
console.log(a) // 0
```

```
a = 'Hello world'
```

```
console.log(a) // Hello world
```

```
a = { 'key': 'value' }
```

```
console.log(a) // {key:'value'}
```


If we take a look at **Typescript**, it is a **statically typed language**, so all checks will be performed during compile/build run before we actually execute our program.

So the previous code with added variable a type won't work. Even from the JavaScript standpoint it is valid (except types) and will run without any errors.

54. Strict mode

Strict mode makes several changes to normal JavaScript semantics:

1. Eliminates some JavaScript silent errors by changing them to throw errors.
2. Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
3. Prohibits some syntax likely to be defined in future versions of ECMAScript.

To invoke strict mode for an entire script, put the exact statement **"use strict"**; (or 'use strict';) before any other statements.

```
'use strict';
```

```
var v = "Hi! I'm a strict mode script!"
```

55. JavaScript ES6 Features

let: new let keyword for declaring variables

There are two critical differences between the var and let. Variables declared with the var keyword are function-scoped and hoisted at the top within its scope, whereas variables declared with let keyword are block-scoped ({}) and they are not hoisted.

```
// ES6 syntax
```

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

```
console.log(i); // undefined
```

```
// ES5 syntax
```

```
for(var i = 0; i < 5; i++) {  
  console.log(i); // 0,1,2,3,4  
}
```

```
console.log(i); // 5
```

const: The new const keyword makes it possible to define constants. Constants are read-only, you cannot reassign new values to them. They are also block-scoped like let.

```
const PI = 3.14;  
console.log(PI); // 3.14  
PI = 10; // error
```

for...of Loop: The new for...of loop allows us to iterate over arrays or other iterable objects very easily. Also, the code inside the loop is executed for each element of the iterable object.

```
// Iterating over array  
let letters = ["a", "b", "c", "d", "e", "f"];  
for(let letter of letters) {  
    console.log(letter); // a,b,c,d,e,f  
}
```

Template Literals: Template literals provide an easy and clean way create multi-line strings and perform string interpolation. Now we can embed variables or expressions into a string at any spot without any hassle. Template literals are created using back-tick (` `) (grave accent) character instead of the usual double or single quotes. Variables or expressions can be placed inside the string using the `${...}` syntax.

```
// Simple multi-line string  
let str = `The quick brown fox  
    jumps over the lazy dog.`;  
// String with embedded variables and expression  
let a = 10;  
let b = 20;  
let result = `The sum of ${a} and ${b} is ${a+b}.`;   
console.log(result); // The sum of 10 and 20 is 30.
```

Default Values for Function Parameters: Now, in ES6 you can specify default values to the function parameters. This means that if no arguments are provided to function when it is called these default parameters values will be used. This is one of the most awaited features in JavaScript. Here's an example:

```
function sayHello(name='World') {  
  return `Hello ${name}!`;  
}  
console.log(sayHello()); // Hello World!  
console.log(sayHello('John')); // Hello John!
```

Arrow Functions: Arrow Functions are another interesting feature in ES6. It provides a more concise syntax for writing function expressions by opting out the function and return keywords.

Arrow functions are defined using a new syntax, the fat arrow (=>) notation. Let's see how it looks:

```
// Function Expression  
var sum = function(a, b) {  
  return a + b;  
}  
console.log(sum(2, 3)); // 5  
// Arrow function  
var sum = (a, b) => a + b;  
console.log(sum(2, 3)); // 5
```

Classes: ES6 classes make it easier to create objects, implement inheritance by using the extends keyword, and reuse the code.

In ES6 you can declare a class using the new class keyword followed by a class-name. By convention class names are written in TitleCase (i.e. capitalizing the first letter of each word).

```
class Rectangle {  
  // Class constructor  
  constructor(length, width) {  
    this.length = length;  
    this.width = width;  
  }  
  
  // Class method  
  getArea() {  
    return this.length * this.width;  
  }  
}
```

```
// Square class inherits from the Rectangle class
class Square extends Rectangle {
  // Child class constructor
  constructor(length) {
    // Call parent's constructor
    super(length, length);
  }

  // Child class method
  getPerimeter() {
    return 2 * (this.length + this.width);
  }
}
```

```
let rectangle = new Rectangle(5, 10);
alert(rectangle.getArea()); // 50
```

The Rest Parameters: ES6 introduces rest parameters that allow us to pass an arbitrary number of parameters to a function in the form of an array. This is particularly helpful in situations when you want to pass parameters to a function but you have no idea how many you will need.

A rest parameter is specified by prefixing a named parameter with rest operator (...) i.e. three dots. Rest parameter can only be the last one in the list of parameters, and there can only be one rest parameter. Take a look at the following example, to see how it works:

```
function sortNames(...names) {
  return names.sort();
}

alert(sortNames("Sarah", "Harry", "Peter")); // Harry,Peter,Sarah
alert(sortNames("Tony", "Ben", "Rick", "Jos")); // John,Jos,Rick,Tony
```

The Spread Operator: The spread operator, which is also denoted by (...), performs the exact opposite function of the rest operator. The spread operator spreads out (i.e. splits up) an array and passes the values into the specified function, as shown in the following example:

```
function addNumbers(a, b, c) {
  return a + b + c;
}
```

```
}  
let numbers = [5, 12, 8];  
// ES5 way of passing array as an argument of a function  
alert(addNumbers.apply(null, numbers)); // 25  
// ES6 spread operator  
alert(addNumbers(...numbers)); // 25  
The spread operator can also be used to insert the elements of an array  
into another array without using the array methods like push(), unshift()  
concat(), etc.
```

Destructuring Assignment: The destructuring assignment is an expression that makes it easy to extract values from arrays, or properties from objects, into distinct variables by providing a shorter syntax.

There are two kinds of destructuring assignment expressions—the array and object destructuring assignment.

The array destructuring assignment

```
// ES6 syntax  
let fruits = ["Apple", "Banana"];  
let [a, b] = fruits; // Array destructuring assignment  
alert(a); // Apple  
alert(b); // Banana  
// ES6 syntax  
let fruits = ["Apple", "Banana", "Mango"];  
let [a, ...r] = fruits;  
alert(a); // Apple  
alert(r); // Banana,Mango  
alert(Array.isArray(r)); // true
```

The object destructuring assignment

```
// ES6 syntax  
let person = {name: "Peter", age: 28};  
let {name, age} = person; // Object destructuring assignment  
alert(name); // Peter  
alert(age); // 28
```

New string methods:

```
'hello'.startsWith('hell') //true  
'hello'.endsWith('ello') // true
```

```
'hello'.includes('ell') // true  
'doo '.repeat(3) // 'doo doo doo '
```

56. Deep copying and Shallow copying in JavaScript

A **deep copy** means that all of the values of the new variable are copied and disconnected from the original variable.

deep copy makes a copy of all attributes of the old object and allocates separate memory addresses for the new object. This helps us to create a cloned object without any worries about changing the values of the old object.

```
const a = 5  
let b = a // this is the copy  
b = 6  
console.log(b) // 6  
console.log(a) // 5
```

By executing `b = a`, you make the copy. Now, when you reassign a new value to `b`, the value of `b` changes, but not of `a`.

A **shallow copy** means that certain (sub-)values are still connected to the original variable. Basically, if any of the fields of the objects are referenced to other objects they share the same memory address.

```
const a = {  
  en: 'Hello',  
  de: 'Hallo',  
  es: 'Hola',  
  pt: 'Olà'  
}  
let b = a  
b.pt = 'Oi'  
console.log(b.pt) // Oi  
console.log(a.pt) // Oi
```

To prevent **shallow copy**, use `JSON.parse(JSON.stringify(someArray))`

57. Promise states

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.

Benefits of Promises

- Improves Code Readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better Error Handling

A Promise has four states:

- fulfilled**: Action related to the promise succeeded
- rejected**: Action related to the promise failed
- pending**: Promise is still pending i.e. not fulfilled or rejected yet
- settled**: Promise has fulfilled or rejected

```
var promise = new Promise(function(resolve, reject) {  
  const x = "shiva";  
  const y = " shiva "  
  if(x === y) {  
    resolve();  
  } else {  
    reject();  
  }  
});  
promise.then(function () {  
  console.log('Success, You are a GEEK');  
}).  
catch(function () {  
  console.log('Some error has occurred');  
});
```

58. Singletons Object

Singletons are used to create an instance of a class if it does not exist or else return the reference of the existing one. This means that singletons are created exactly once during the runtime of the application in the global scope. Based on this definition, singletons seem very similar to global variables

59. Namespace

Namespace refers to the programming paradigm of providing scope to the identifiers (names of types, functions, variables, etc) to prevent collisions

between them. For instance, the same variable name might be required in a program in different contexts. Using namespaces in such a scenario will isolate these contexts such that the same identifier can be used in different namespaces. In this article, we will discuss how namespaces can be initialized and used in JavaScript. JavaScript does not provide namespace by default. However, we can replicate this functionality by making a global object which can contain all functions and variables.

```
var <namespace> = {};
```