

No.	Questions
450	How to use await outside of async function prior to ES2022?

1. What are the possible ways to create objects in JavaScript [↗](#)

There are many ways to create objects in javascript as below

i. Object constructor:

The simplest way to create an empty object is using the Object constructor. Currently this approach is not recommended.

```
var object = new Object();
```



The `Object()` is a built-in constructor function so "new" keyword is not required. the above can be written as:

```
var object = Object();
```



ii. Object's create method:

The create method of Object creates a new object by passing the prototype object as a parameter

```
var object = Object.create(null);
```



iii. Object literal syntax:

The object literal syntax (or object initializer), is a comma-separated set of name-value pairs wrapped in curly braces.

```
var object = {  
  name: "Sudheer",  
  age: 34  
};
```



Object literal property values can be of any data type, including array,



Note: This is an easiest way to create an object

iv. Function constructor:

Create any function and apply the new operator to create object instances,

```
function Person(name) {  
  this.name = name;  
  this.age = 21;  
}  
var object = new Person("Sudheer");
```



v. Function constructor with prototype:

This is similar to function constructor but it uses prototype for their properties and methods,

```
function Person() {}  
Person.prototype.name = "Sudheer";  
var object = new Person();
```



This is equivalent to an instance created with an object create method with a function prototype and then call that function with an instance and parameters as arguments.

```
function func() {}  
  
new func(x, y, z);
```



(OR)

```
// Create a new instance using function prototype.  
var newInstance = Object.create(func.prototype)  
  
// Call the function  
var result = func.call(newInstance, x, y, z),  
  
// If the result is a non-null object then use it otherwise just use the  
console.log(result && typeof result === 'object' ? result : newInstance)
```



vi. ES6 Class syntax:

ES6 introduces class feature to create the objects

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

var object = new Person("Sudheer");
```



vii. Singleton pattern:

A Singleton is an object which can only be instantiated one time. Repeated calls to its constructor return the same instance and this way one can ensure that they don't accidentally create multiple instances.

```
var object = new (function () {
  this.name = "Sudheer";
})();
```

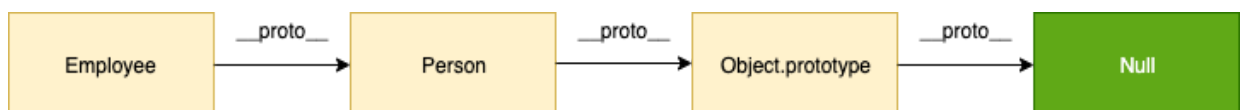


[↑ Back to Top](#)

2. What is a prototype chain [↗](#)

Prototype chaining is used to build new types of objects based on existing ones. It is similar to inheritance in a class based language.

The prototype on object instance is available through **Object.getPrototypeOf(object)** or **__proto__** property whereas prototype on constructors function is available through **Object.prototype**.



[↑ Back to Top](#)

3. What is the difference between Call, Apply and Bind [↗](#)

The difference between Call, Apply and Bind can be explained with below examples,

Call: The call() method invokes a function with a given `this` value and arguments provided one by one

```
var employee1 = { firstName: "John", lastName: "Rodson" };
var employee2 = { firstName: "Jimmy", lastName: "Baily" };

function invite(greeting1, greeting2) {
  console.log(
    greeting1 + " " + this.firstName + " " + this.lastName + ", " + greeting2
  );
}

invite.call(employee1, "Hello", "How are you?"); // Hello John Rodson, How are you?
invite.call(employee2, "Hello", "How are you?"); // Hello Jimmy Baily, How are you?
```



Apply: Invokes the function with a given `this` value and allows you to pass in arguments as an array

```
var employee1 = { firstName: "John", lastName: "Rodson" };
var employee2 = { firstName: "Jimmy", lastName: "Baily" };

function invite(greeting1, greeting2) {
  console.log(
    greeting1 + " " + this.firstName + " " + this.lastName + ", " + greeting2
  );
}

invite.apply(employee1, ["Hello", "How are you?"]); // Hello John Rodson, How are you?
invite.apply(employee2, ["Hello", "How are you?"]); // Hello Jimmy Baily, How are you?
```



bind: returns a new function, allowing you to pass any number of arguments

```
var employee1 = { firstName: "John", lastName: "Rodson" };
var employee2 = { firstName: "Jimmy", lastName: "Baily" };

function invite(greeting1, greeting2) {
  console.log(
    greeting1 + " " + this.firstName + " " + this.lastName + ", " + greeting2
  );
}

var inviteEmployee1 = invite.bind(employee1);
var inviteEmployee2 = invite.bind(employee2);
inviteEmployee1("Hello", "How are you?"); // Hello John Rodson, How are you?
inviteEmployee2("Hello", "How are you?"); // Hello Jimmy Baily, How are you?
```



Call and apply are pretty interchangeable. Both execute the current function immediately. You need to decide whether it's easier to send in an array or a comma separated list of arguments. You can remember by treating Call is for **comma** (separated list) and Apply is for **Array**.

Whereas Bind creates a new function that will have `this` set to the first parameter passed to bind().

[↑ Back to Top](#)

4. What is JSON and its common operations [↗](#)

JSON is a text-based data format following JavaScript object syntax, which was popularized by Douglas Crockford . It is useful when you want to transmit data across a network and it is basically just a text file with an extension of .json, and a MIME type of application/json

Parsing: Converting a string to a native object

```
JSON.parse(text);
```



Stringification: converting a native object to a string so it can be transmitted across the network

```
JSON.stringify(object);
```



[↑ Back to Top](#)

5. What is the purpose of the array slice method [↗](#)

The **slice()** method returns the selected elements in an array as a new array object. It selects the elements starting at the given start argument, and ends at the given optional end argument without including the last element. If you omit the second argument then it selects till the end.

Some of the examples of this method are,

```
let arrayIntegers = [1, 2, 3, 4, 5];
let arrayIntegers1 = arrayIntegers.slice(0, 2); // returns [1,2]
let arrayIntegers2 = arrayIntegers.slice(2, 3); // returns [3]
let arrayIntegers3 = arrayIntegers.slice(4); //returns [5]
```



Note: Slice method won't mutate the original array but it returns the subset as a new array.

[↑ Back to Top](#)

6. What is the purpose of the array splice method [↗](#)

The **splice()** method is used either adds/removes items to/from an array, and then returns the removed item. The first argument specifies the array position for insertion or deletion whereas the optional second argument indicates the number of elements to be deleted. Each additional argument is added to the array.

Some of the examples of this method are,

```
let arrayIntegersOriginal1 = [1, 2, 3, 4, 5];
let arrayIntegersOriginal2 = [1, 2, 3, 4, 5];
let arrayIntegersOriginal3 = [1, 2, 3, 4, 5];

let arrayIntegers1 = arrayIntegersOriginal1.splice(0, 2); // returns [1, 2];
let arrayIntegers2 = arrayIntegersOriginal2.splice(3); // returns [4, 5]; or
let arrayIntegers3 = arrayIntegersOriginal3.splice(3, 1, "a", "b", "c"); //r
```



Note: Splice method modifies the original array and returns the deleted array.

[↑ Back to Top](#)

7. What is the difference between slice and splice [↗](#)

Some of the major difference in a tabular form

Slice	Splice
Doesn't modify the original array(immutable)	Modifies the original array(mutable)
Returns the subset of original array	Returns the deleted elements as array
Used to pick the elements from array	Used to insert or delete elements to/from array

[↑ Back to Top](#)

8. How do you compare Object and Map [↗](#)

Objects are similar to **Maps** in that both let you set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. Due to this reason, Objects have been used as Maps historically. But there are important differences that make using a Map preferable in certain cases.

- i. The keys of an Object are Strings and Symbols, whereas they can be any value for a Map, including functions, objects, and any primitive.
- ii. The keys in Map are ordered while keys added to Object are not. Thus, when iterating over it, a Map object returns keys in order of insertion.
- iii. You can get the size of a Map easily with the size property, while the number of properties in an Object must be determined manually.
- iv. A Map is an iterable and can thus be directly iterated, whereas iterating over an Object requires obtaining its keys in some fashion and iterating over them.
- v. An Object has a prototype, so there are default keys in the map that could collide with your keys if you're not careful. As of ES5 this can be bypassed by using `map = Object.create(null)`, but this is seldom done.
- vi. A Map may perform better in scenarios involving frequent addition and removal of key pairs.

[!\[\]\(d84e7ea36f695d92cb39ec32c307ac93_img.jpg\) Back to Top](#)

9. What is the difference between == and === operators [↗](#)

JavaScript provides both strict(===, !==) and type-converting(==, !=) equality comparison. The strict operators take type of variable in consideration, while non-strict operators make type correction/conversion based upon values of variables. The strict operators follow the below conditions for different types,

- i. Two strings are strictly equal when they have the same sequence of characters, same length, and same characters in corresponding positions.
- ii. Two numbers are strictly equal when they are numerically equal. i.e, Having the same number value. There are two special cases in this,
 - a. NaN is not equal to anything, including NaN.
 - b. Positive and negative zeros are equal to one another.
- iii. Two Boolean operands are strictly equal if both are true or both are false.
- iv. Two objects are strictly equal if they refer to the same Object.
- v. Null and Undefined types are not equal with ===, but equal with ==. i.e, `null===undefined --> false` but `null==undefined --> true`

Some of the example which covers the above cases,

```
0 == false    // true
0 === false   // false
1 == "1"      // true
1 === "1"     // false
null == undefined // true
null === undefined // false
'0' == false  // true
'0' === false // false
[]==[] or []===[] //false, refer different objects in memory
{}=={} or {}==={} //false, refer different objects in memory
```



[↑ Back to Top](#)

10. What are lambda or arrow functions [↗](#)

An arrow function is a shorter syntax for a function expression and does not have its own **this**, **arguments**, **super**, or **new.target**. These functions are best suited for non-method functions, and they cannot be used as constructors.

[↑ Back to Top](#)

11. What is a first class function [↗](#)

In Javascript, functions are first class objects. First-class functions means when functions in that language are treated like any other variable.

For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable. For example, in the below example, handler functions assigned to a listener

```
const handler = () => console.log("This is a click handler function");
document.addEventListener("click", handler);
```



[↑ Back to Top](#)

12. What is a first order function [↗](#)

First-order function is a function that doesn't accept another function as an argument and doesn't return a function as its return value.

```
const firstOrder = () => console.log("I am a first order function!");
```



[↑ Back to Top](#)

13. What is a higher order function [↗](#)

Higher-order function is a function that accepts another function as an argument or returns a function as a return value or both.

```
const firstOrderFunc = () =>
  console.log("Hello, I am a First order function");
const higherOrder = (ReturnFirstOrderFunc) => ReturnFirstOrderFunc();
higherOrder(firstOrderFunc);
```



[↑ Back to Top](#)

14. What is a unary function [↗](#)

Unary function (i.e. monadic) is a function that accepts exactly one argument. It stands for a single argument accepted by a function.

Let us take an example of unary function,

```
const unaryFunction = (a) => console.log(a + 10); // Add 10 to the given argument
```



[↑ Back to Top](#)

15. What is the currying function [↗](#)

Currying is the process of taking a function with multiple arguments and turning it into a sequence of functions each with only a single argument. Currying is named after a mathematician **Haskell Curry**. By applying currying, a n-ary function turns it into a unary function.

Let's take an example of n-ary function and how it turns into a currying function,

```
const multiArgFunction = (a, b, c) => a + b + c;
console.log(multiArgFunction(1, 2, 3)); // 6

const curryUnaryFunction = (a) => (b) => (c) => a + b + c;
curryUnaryFunction(1); // returns a function: b => c => 1 + b + c
curryUnaryFunction(1)(2); // returns a function: c => 3 + c
curryUnaryFunction(1)(2)(3); // returns the number 6
```



Curried functions are great to improve **code reusability** and **functional composition**.

[↑ Back to Top](#)

16. What is a pure function [↗](#)

A **Pure function** is a function where the return value is only determined by its arguments without any side effects. i.e, If you call a function with the same arguments 'n' number of times and 'n' number of places in the application then it will always return the same value.

Let's take an example to see the difference between pure and impure functions,

```
//Impure
let numberArray = [];
const impureAddNumber = (number) => numberArray.push(number);
//Pure
const pureAddNumber = (number) => (argNumberArray) =>
  argNumberArray.concat([number]);

//Display the results
console.log(impureAddNumber(6)); // returns 1
console.log(numberArray); // returns [6]
console.log(pureAddNumber(7)(numberArray)); // returns [6, 7]
console.log(numberArray); // returns [6]
```



As per the above code snippets, the **Push** function is impure itself by altering the array and returning a push number index independent of the parameter value. . Whereas **Concat** on the other hand takes the array and concatenates it with the other array producing a whole new array without side effects. Also, the return value is a concatenation of the previous array.

Remember that Pure functions are important as they simplify unit testing without any side effects and no need for dependency injection. They also avoid tight coupling and make it harder to break your application by not having any side effects. These principles are coming together with **Immutability** concept of ES6 by giving preference to **const** over **let** usage.

[↑ Back to Top](#)

17. What is the purpose of the let keyword [↗](#)

The `let` statement declares a **block scope local variable**. Hence the variables defined with `let` keyword are limited in scope to the block, statement, or expression on which it is used. Whereas variables declared with the `var` keyword used to define a variable globally, or locally to an entire function regardless of block scope.

Let's take an example to demonstrate the usage,

```
let counter = 30;
if (counter === 30) {
  let counter = 31;
  console.log(counter); // 31
}
console.log(counter); // 30 (because the variable in if block won't exist here)
```



[↑ Back to Top](#)

18. What is the difference between `let` and `var` [↗](#)

You can list out the differences in a tabular format

<code>var</code>	<code>let</code>
It is been available from the beginning of JavaScript	Introduced as part of ES6
It has function scope	It has block scope
Variables will be hoisted	Hoisted but not initialized

Let's take an example to see the difference,

```
function userDetails(username) {
  if (username) {
    console.log(salary); // undefined due to hoisting
    console.log(age); // ReferenceError: Cannot access 'age' before initialization
    let age = 30;
    var salary = 10000;
  }
  console.log(salary); //10000 (accessible due to function scope)
  console.log(age); //error: age is not defined(due to block scope)
```



```
}  
userDetails("John");
```

[↑ Back to Top](#)

19. What is the reason to choose the name `let` as a keyword

`let` is a mathematical statement that was adopted by early programming languages like **Scheme** and **Basic**. It has been borrowed from dozens of other languages that use `let` already as a traditional keyword as close to `var` as possible.

[↑ Back to Top](#)

20. How do you redeclare variables in `switch` block without an error

If you try to redeclare variables in a `switch` block then it will cause errors because there is only one block. For example, the below code block throws a syntax error as below,

```
let counter = 1;  
switch (x) {  
  case 0:  
    let name;  
    break;  
  
  case 1:  
    let name; // SyntaxError for redeclaration.  
    break;  
}
```



To avoid this error, you can create a nested block inside a case clause and create a new block scoped lexical environment.

```
let counter = 1;  
switch (x) {  
  case 0: {  
    let name;  
    break;  
  }  
  case 1: {  
    let name; // No SyntaxError for redeclaration.  
  }  
}
```



```
        break;
    }
}
```

[↑ Back to Top](#)

21. What is the Temporal Dead Zone [↗](#)

The Temporal Dead Zone is a behavior in JavaScript that occurs when declaring a variable with the `let` and `const` keywords, but not with `var`. In ECMAScript 6, accessing a `let` or `const` variable before its declaration (within its scope) causes a `ReferenceError`. The time span when that happens, between the creation of a variable's binding and its declaration, is called the temporal dead zone.

Let's see this behavior with an example,

```
function somemethod() {
  console.log(counter1); // undefined
  console.log(counter2); // ReferenceError
  var counter1 = 1;
  let counter2 = 2;
}
```



[↑ Back to Top](#)

22. What is IIFE(Immediately Invoked Function Expression) [↗](#)

IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined. The signature of it would be as below,

```
(function () {
  // logic here
})();
```



The primary reason to use an IIFE is to obtain data privacy because any variables declared within the IIFE cannot be accessed by the outside world. i.e, If you try to access variables with IIFE then it throws an error as below,

```
(function () {
  var message = "IIFE";
  console.log(message);
})
```



```
})();  
console.log(message); //Error: message is not defined
```

[↑ Back to Top](#)

23. How do you decode or encode a URL in JavaScript? [↗](#)

`encodeURIComponent()` function is used to encode an URL. This function requires a URL string as a parameter and return that encoded string. `decodeURIComponent()` function is used to decode an URL. This function requires an encoded URL string as parameter and return that decoded string.

Note: If you want to encode characters such as `/ ? : @ & = + $ #` then you need to use `encodeURIComponent()`.

```
let uri = "employeeDetails?name=john&occupation=manager";  
let encoded_uri = encodeURIComponent(uri);  
let decoded_uri = decodeURIComponent(encoded_uri);
```



[↑ Back to Top](#)

24. What is memoization [↗](#)

Memoization is a programming technique which attempts to increase a function's performance by caching its previously computed results. Each time a memoized function is called, its parameters are used to index the cache. If the data is present, then it can be returned, without executing the entire function. Otherwise the function is executed and then the result is added to the cache. Let's take an example of adding function with memoization,

```
const memoizAddition = () => {  
  let cache = {};  
  return (value) => {  
    if (value in cache) {  
      console.log("Fetching from cache");  
      return cache[value]; // Here, cache.value cannot be used as property name  
    } else {  
      console.log("Calculating result");  
      let result = value + 20;  
      cache[value] = result;  
      return result;  
    }  
  };  
};
```



```
// returned function from memoizAddition
const addition = memoizAddition();
console.log(addition(20)); //output: 40 calculated
console.log(addition(20)); //output: 40 cached
```

[↑ Back to Top](#)

25. What is Hoisting [↗](#)

Hoisting is a JavaScript mechanism where variables, function declarations and classes are moved to the top of their scope before code execution. Remember that JavaScript only hoists declarations, not initialisation. Let's take a simple example of variable hoisting,

```
console.log(message); //output : undefined
var message = "The variable Has been hoisted";
```



The above code looks like as below to the interpreter,

```
var message;
console.log(message);
message = "The variable Has been hoisted";
```



In the same fashion, function declarations are hoisted too

```
message("Good morning"); //Good morning

function message(name) {
  console.log(name);
}
```



This hoisting makes functions to be safely used in code before they are declared.

[↑ Back to Top](#)

26. What are classes in ES6 [↗](#)

In ES6, Javascript classes are primarily syntactic sugar over JavaScript's existing prototype-based inheritance. For example, the prototype based inheritance written in function expression as below,

```
function Bike(model, color) {
  this.model = model;
  this.color = color;
}

Bike.prototype.getDetails = function () {
  return this.model + " bike has" + this.color + " color";
};
```



Whereas ES6 classes can be defined as an alternative

```
class Bike {
  constructor(color, model) {
    this.color = color;
    this.model = model;
  }

  getDetails() {
    return this.model + " bike has" + this.color + " color";
  }
}
```



[↑ Back to Top](#)

27. What are closures [↗](#)

A closure is the combination of a function and the lexical environment within which that function was declared. i.e, It is an inner function that has access to the outer or enclosing function's variables. The closure has three scope chains

- i. Own scope where variables defined between its curly brackets
- ii. Outer function's variables
- iii. Global variables

Let's take an example of closure concept,

```
function Welcome(name) {
  var greetingInfo = function (message) {
    console.log(message + " " + name);
  };
  return greetingInfo;
}

var myFunction = Welcome("John");
```




```
myFunction("Welcome "); //Output: Welcome John  
myFunction("Hello Mr."); //output: Hello Mr.John
```

As per the above code, the inner function(i.e, greetingInfo) has access to the variables in the outer function scope(i.e, Welcome) even after the outer function has returned.

[↑ Back to Top](#)

28. What are modules [↗](#)

Modules refer to small units of independent, reusable code and also act as the foundation of many JavaScript design patterns. Most of the JavaScript modules export an object literal, a function, or a constructor

[↑ Back to Top](#)

29. Why do you need modules [↗](#)

Below are the list of benefits using modules in javascript ecosystem

- i. Maintainability
- ii. Reusability
- iii. Namespacing

[↑ Back to Top](#)

30. What is scope in javascript [↗](#)

Scope is the accessibility of variables, functions, and objects in some particular part of your code during runtime. In other words, scope determines the visibility of variables and other resources in areas of your code.

[↑ Back to Top](#)

31. What is a service worker [↗](#)

A Service worker is basically a script (JavaScript file) that runs in the background, separate from a web page and provides features that don't need a web page or user interaction. Some of the major features of service workers are Rich offline experiences(offline first web application development), periodic background syncs, push notifications, intercept and handle network requests and programmatically managing a cache of responses.

[↑ Back to Top](#)

32. How do you manipulate DOM using a service worker [↗](#)

Service worker can't access the DOM directly. But it can communicate with the pages it controls by responding to messages sent via the `postMessage` interface, and those pages can manipulate the DOM.

[↑ Back to Top](#)

33. How do you reuse information across service worker restarts [↗](#)

The problem with service worker is that it gets terminated when not in use, and restarted when it's next needed, so you cannot rely on global state within a service worker's `onfetch` and `onmessage` handlers. In this case, service workers will have access to IndexedDB API in order to persist and reuse across restarts.

[↑ Back to Top](#)

34. What is IndexedDB [↗](#)

IndexedDB is a low-level API for client-side storage of larger amounts of structured data, including files/blobs. This API uses indexes to enable high-performance searches of this data.

[↑ Back to Top](#)

35. What is web storage [↗](#)

Web storage is an API that provides a mechanism by which browsers can store key/value pairs locally within the user's browser, in a much more intuitive fashion than using cookies. The web storage provides two mechanisms for storing data on the client.

- i. **Local storage:** It stores data for current origin with no expiration date.
- ii. **Session storage:** It stores data for one session and the data is lost when the browser tab is closed.

[↑ Back to Top](#)

36. What is a post message [↗](#)

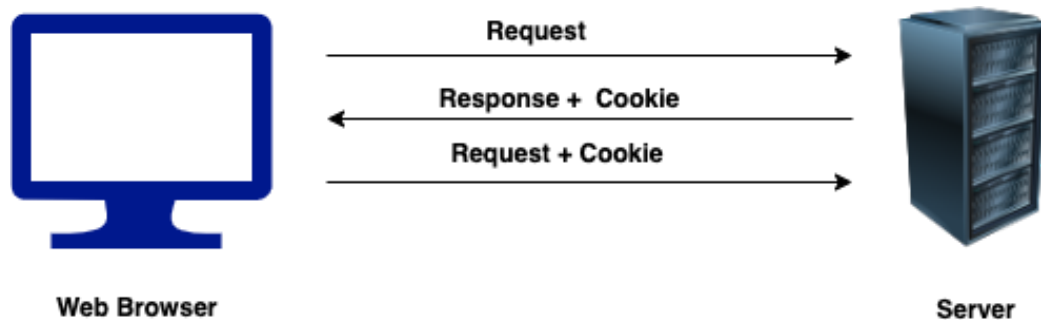
Post message is a method that enables cross-origin communication between Window objects.(i.e, between a page and a pop-up that it spawned, or between a page and an iframe embedded within it). Generally, scripts on different pages are allowed to access each other if and only if the pages follow same-origin policy(i.e, pages share the same protocol, port number, and host).

[↑ Back to Top](#)

37. What is a Cookie [↗](#)

A cookie is a piece of data that is stored on your computer to be accessed by your browser. Cookies are saved as key/value pairs. For example, you can create a cookie named username as below,

```
document.cookie = "username=John";
```



[↑ Back to Top](#)

38. Why do you need a Cookie [↗](#)

Cookies are used to remember information about the user profile(such as username). It basically involves two steps,

- i. When a user visits a web page, the user profile can be stored in a cookie.
- ii. Next time the user visits the page, the cookie remembers the user profile.

[↑ Back to Top](#)

39. What are the options in a cookie [↗](#)

There are few below options available for a cookie,

- i. By default, the cookie is deleted when the browser is closed but you can change this behavior by setting expiry date (in UTC time).

```
document.cookie = "username=John; expires=Sat, 8 Jun 2019 12:00:00 UTC"
```



- i. By default, the cookie belongs to a current page. But you can tell the browser what path the cookie belongs to using a path parameter.

```
document.cookie = "username=John; path=/services";
```



[↑ Back to Top](#)

40. How do you delete a cookie [↗](#)

You can delete a cookie by setting the expiry date as a passed date. You don't need to specify a cookie value in this case. For example, you can delete a username cookie in the current page as below.

```
document.cookie =  
"username=; expires=Fri, 07 Jun 2019 00:00:00 UTC; path=/";
```



Note: You should define the cookie path option to ensure that you delete the right cookie. Some browsers doesn't allow to delete a cookie unless you specify a path parameter.

[↑ Back to Top](#)

41. What are the differences between cookie, local storage and session storage [↗](#)

Below are some of the differences between cookie, local storage and session storage,

Feature	Cookie	Local storage	Session storage
Accessed on client or server side	Both server-side & client-side	client-side only	client-side only
Lifetime	As configured using Expires option	until deleted	until tab is closed
SSL support	Supported	Not supported	Not supported
Maximum data size	4KB	5 MB	5MB

[↑ Back to Top](#)

42. What is the main difference between localStorage and sessionStorage [↗](#)

LocalStorage is the same as SessionStorage but it persists the data even when the browser is closed and reopened(i.e it has no expiration time) whereas in sessionStorage data gets cleared when the page session ends.

[↑ Back to Top](#)

43. How do you access web storage [↗](#)

The Window object implements the `WindowLocalStorage` and `WindowSessionStorage` objects which has `localStorage` (`window.localStorage`) and `sessionStorage` (`window.sessionStorage`) properties respectively. These properties create an instance of the Storage object, through which data items can be set, retrieved and removed for a specific domain and storage type (session or local). For example, you can read and write on local storage objects as below

```
localStorage.setItem("logo", document.getElementById("logo").value);  
localStorage.getItem("logo");
```



[↑ Back to Top](#)

44. What are the methods available on session storage [↗](#)

The session storage provided methods for reading, writing and clearing the session data

```
// Save data to sessionStorage  
sessionStorage.setItem("key", "value");  
  
// Get saved data from sessionStorage  
let data = sessionStorage.getItem("key");  
  
// Remove saved data from sessionStorage  
sessionStorage.removeItem("key");  
  
// Remove all saved data from sessionStorage  
sessionStorage.clear();
```



[↑ Back to Top](#)

45. What is a storage event and its event handler [↗](#)

The StorageEvent is an event that fires when a storage area has been changed in the context of another document. Whereas onstorage property is an EventHandler for processing storage events. The syntax would be as below

```
window.onstorage = functionRef;
```



Let's take the example usage of onstorage event handler which logs the storage key and it's values

```
window.onstorage = function (e) {  
  console.log(  
    "The " +  
    e.key +  
    " key has been changed from " +  
    e.oldValue +  
    " to " +  
    e.newValue +  
    "."  
  );  
};
```



[↑ Back to Top](#)

46. Why do you need web storage [↗](#)

Web storage is more secure, and large amounts of data can be stored locally, without affecting website performance. Also, the information is never transferred to the server. Hence this is a more recommended approach than Cookies.

[↑ Back to Top](#)

47. How do you check web storage browser support [↗](#)

You need to check browser support for localStorage and sessionStorage before using web storage,

```
if (typeof Storage !== "undefined") {  
  // Code for localStorage/sessionStorage.  
} else {
```



```
// Sorry! No Web Storage support..  
}
```

[↑ Back to Top](#)

48. How do you check web workers browser support [↗](#)

You need to check browser support for web workers before using it

```
if (typeof Worker !== "undefined") {  
  // code for Web worker support.  
} else {  
  // Sorry! No Web Worker support..  
}
```



[↑ Back to Top](#)

49. Give an example of a web worker [↗](#)

You need to follow below steps to start using web workers for counting example

- i. Create a Web Worker File: You need to write a script to increment the count value. Let's name it as counter.js

```
let i = 0;  
  
function timedCount() {  
  i = i + 1;  
  postMessage(i);  
  setTimeout("timedCount()", 500);  
}  
  
timedCount();
```



Here postMessage() method is used to post a message back to the HTML page

- i. Create a Web Worker Object: You can create a web worker object by checking for browser support. Let's name this file as web_worker_example.js

```
if (typeof w == "undefined") {  
  w = new Worker("counter.js");  
}
```



and we can receive messages from web worker

```
w.onmessage = function (event) {  
  document.getElementById("message").innerHTML = event.data;  
};
```



- i. Terminate a Web Worker: Web workers will continue to listen for messages (even after the external script is finished) until it is terminated. You can use the terminate() method to terminate listening to the messages.

```
w.terminate();
```



- i. Reuse the Web Worker: If you set the worker variable to undefined you can reuse the code

```
w = undefined;
```



[↑ Back to Top](#)

50. What are the restrictions of web workers on DOM [↗](#)

WebWorkers don't have access to below javascript objects since they are defined in an external files

- i. Window object
- ii. Document object
- iii. Parent object

[↑ Back to Top](#)

51. What is a promise [↗](#)

A promise is an object that may produce a single value some time in the future with either a resolved value or a reason that it's not resolved(for example, network error). It will be in one of the 3 possible states: fulfilled, rejected, or pending.

The syntax of Promise creation looks like below,

```
const promise = new Promise(function (resolve, reject) {  
  // promise description  
});
```



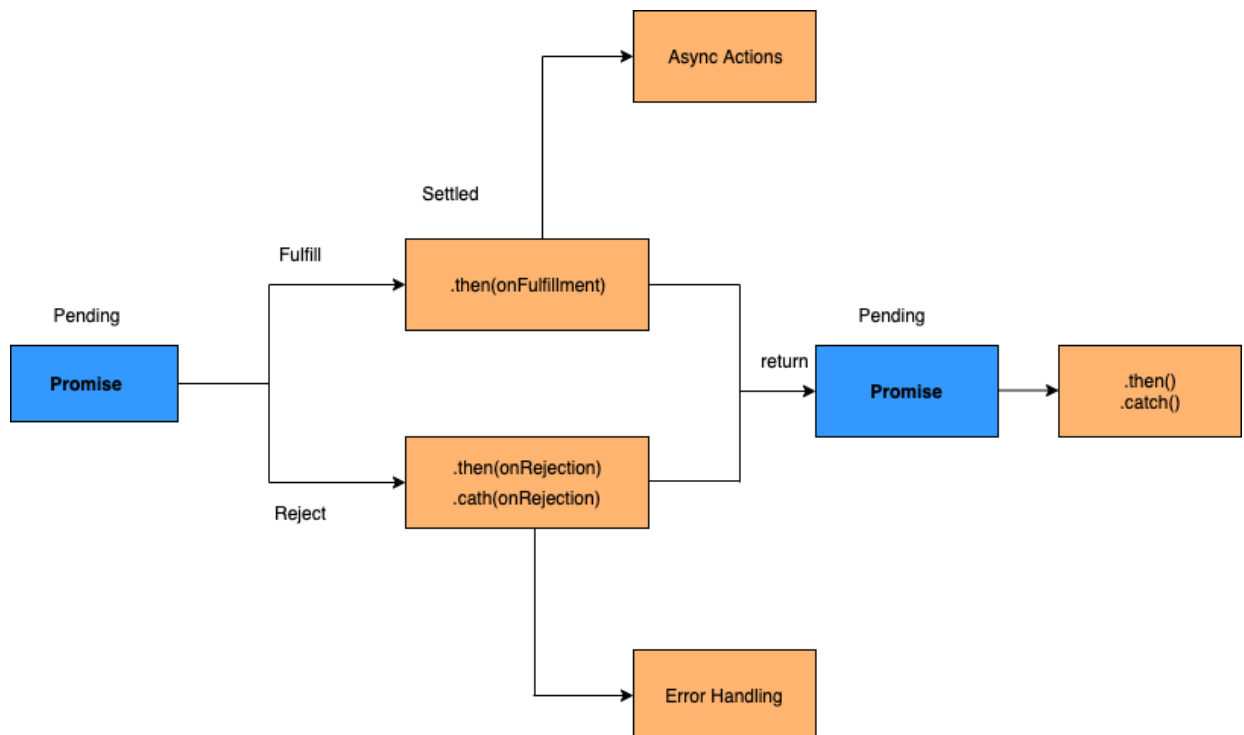
The usage of a promise would be as below,


```
const promise = new Promise(
  (resolve) => {
    setTimeout(() => {
      resolve("I'm a Promise!");
    }, 5000);
  },
  (reject) => {}
);

promise.then((value) => console.log(value));
```



The action flow of a promise will be as below,



[↑ Back to Top](#)

52. Why do you need a promise [↗](#)

Promises are used to handle asynchronous operations. They provide an alternative approach for callbacks by reducing the callback hell and writing the cleaner code.

[↑ Back to Top](#)

53. What are the three states of promise [↗](#)

Promises have three states:

- i. **Pending:** This is an initial state of the Promise before an operation begins
- ii. **Fulfilled:** This state indicates that the specified operation was completed.

- iii. **Rejected:** This state indicates that the operation did not complete. In this case an error value will be thrown.

[↑ Back to Top](#)

54. What is a callback function [↗](#)

A callback function is a function passed into another function as an argument. This function is invoked inside the outer function to complete an action. Let's take a simple example of how to use callback function

```
function callbackFunction(name) {  
    console.log("Hello " + name);  
}  
  
function outerFunction(callback) {  
    let name = prompt("Please enter your name.");  
    callback(name);  
}  
  
outerFunction(callbackFunction);
```



[↑ Back to Top](#)

55. Why do we need callbacks [↗](#)

The callbacks are needed because javascript is an event driven language. That means instead of waiting for a response javascript will keep executing while listening for other events. Let's take an example with the first function invoking an API call(simulated by setTimeout) and the next function which logs the message.

```
function firstFunction() {  
    // Simulate a code delay  
    setTimeout(function () {  
        console.log("First function called");  
    }, 1000);  
}  
function secondFunction() {  
    console.log("Second function called");  
}  
firstFunction();  
secondFunction();
```



Output;

```
// Second function called  
// First function called
```

As observed from the output, javascript didn't wait for the response of the first function and the remaining code block got executed. So callbacks are used in a way to make sure that certain code doesn't execute until the other code finishes execution.

[↑ Back to Top](#)

56. What is a callback hell [↗](#)

Callback Hell is an anti-pattern with multiple nested callbacks which makes code hard to read and debug when dealing with asynchronous logic. The callback hell looks like below,

```
async1(function(){  
  async2(function(){  
    async3(function(){  
      async4(function(){  
        ....  
      });  
    });  
  });  
});
```



[↑ Back to Top](#)

57. What are server-sent events [↗](#)

Server-sent events (SSE) is a server push technology enabling a browser to receive automatic updates from a server via HTTP connection without resorting to polling. These are a one way communications channel - events flow from server to client only. This has been used in Facebook/Twitter updates, stock price updates, news feeds etc.

[↑ Back to Top](#)

58. How do you receive server-sent event notifications [↗](#)

The EventSource object is used to receive server-sent event notifications. For example, you can receive messages from server as below,

```
if (typeof EventSource !== "undefined") {  
    var source = new EventSource("sse_generator.js");  
    source.onmessage = function (event) {  
        document.getElementById("output").innerHTML += event.data + "<br>";  
    };  
}
```



[↑ Back to Top](#)

59. How do you check browser support for server-sent events [↗](#)

You can perform browser support for server-sent events before using it as below,

```
if (typeof EventSource !== "undefined") {  
    // Server-sent events supported. Let's have some code here!  
} else {  
    // No server-sent events supported  
}
```



[↑ Back to Top](#)

60. What are the events available for server sent events [↗](#)

Below are the list of events available for server sent events

Event	Description
onopen	It is used when a connection to the server is opened
onmessage	This event is used when a message is received
onerror	It happens when an error occurs

[↑ Back to Top](#)

61. What are the main rules of promise [↗](#)

A promise must follow a specific set of rules:

- i. A promise is an object that supplies a standard-compliant `.then()` method
- ii. A pending promise may transition into either fulfilled or rejected state
- iii. A fulfilled or rejected promise is settled and it must not transition into any other state.
- iv. Once a promise is settled, the value must not change.

[↑ Back to Top](#)

62. What is callback in callback [↗](#)

You can nest one callback inside in another callback to execute the actions sequentially one by one. This is known as callbacks in callbacks.

```
loadScript("/script1.js", function (script) {
  console.log("first script is loaded");

  loadScript("/script2.js", function (script) {
    console.log("second script is loaded");

    loadScript("/script3.js", function (script) {
      console.log("third script is loaded");
      // after all scripts are loaded
    });
  });
});
```



[↑ Back to Top](#)

63. What is promise chaining [↗](#)

The process of executing a sequence of asynchronous tasks one after another using promises is known as Promise chaining. Let's take an example of promise chaining for calculating the final result,

```
new Promise(function (resolve, reject) {
  setTimeout(() => resolve(1), 1000);
})
  .then(function (result) {
    console.log(result); // 1
    return result * 2;
  })
  .then(function (result) {
    console.log(result); // 2
    return result * 3;
  })
  .then(function (result) {
    console.log(result); // 6
    return result * 4;
  });
```



In the above handlers, the result is passed to the chain of `.then()` handlers with the below work flow,

- i. The initial promise resolves in 1 second,
- ii. After that `.then` handler is called by logging the result(1) and then return a promise with the value of result * 2.
- iii. After that the value passed to the next `.then` handler by logging the result(2) and return a promise with result * 3.
- iv. Finally the value passed to the last `.then` handler by logging the result(6) and return a promise with result * 4.

[↑ Back to Top](#)

64. What is `promise.all` [↗](#)

`Promise.all` is a promise that takes an array of promises as an input (an iterable), and it gets resolved when all the promises get resolved or any one of them gets rejected. For example, the syntax of `promise.all` method is below,

```
Promise.all([Promise1, Promise2, Promise3]).then(result) => { console.log(result); }
```

Note: Remember that the order of the promises(output the result) is maintained as per input order.

[↑ Back to Top](#)

65. What is the purpose of the `race` method in promise [↗](#)

`Promise.race()` method will return the promise instance which is firstly resolved or rejected. Let's take an example of `race()` method where `promise2` is resolved first

```
var promise1 = new Promise(function (resolve, reject) {
  setTimeout(resolve, 500, "one");
});
var promise2 = new Promise(function (resolve, reject) {
  setTimeout(resolve, 100, "two");
});

Promise.race([promise1, promise2]).then(function (value) {
  console.log(value); // "two" // Both promises will resolve, but promise2 is
});
```

[↑ Back to Top](#)

66. What is a strict mode in javascript [↗](#)

Strict Mode is a new feature in ECMAScript 5 that allows you to place a program, or a function, in a "strict" operating context. This way it prevents certain actions from being taken and throws more exceptions. The literal expression `"use strict";` instructs the browser to use the javascript code in the Strict mode.

[↑ Back to Top](#)

67. Why do you need strict mode [↗](#)

Strict mode is useful to write "secure" JavaScript by notifying "bad syntax" into real errors. For example, it eliminates accidentally creating a global variable by throwing an error and also throws an error for assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object.

[↑ Back to Top](#)

68. How do you declare strict mode [↗](#)

The strict mode is declared by adding `"use strict";` to the beginning of a script or a function. If declared at the beginning of a script, it has global scope.

```
"use strict";  
x = 3.14; // This will cause an error because x is not declared
```



and if you declare inside a function, it has local scope

```
x = 3.14; // This will not cause an error.  
myFunction();  
  
function myFunction() {  
  "use strict";  
  y = 3.14; // This will cause an error  
}
```



[↑ Back to Top](#)

69. What is the purpose of double exclamation [↗](#)

The double exclamation or negation(!!) ensures the resulting type is a boolean. If it was falsey (e.g. 0, null, undefined, etc.), it will be false, otherwise, it will be true. For example, you can test IE version using this expression as below,

```
let isIE8 = false;
isIE8 = !!navigator.userAgent.match(/MSIE 8.0/);
console.log(isIE8); // returns true or false
```



If you don't use this expression then it returns the original value.

```
console.log(navigator.userAgent.match(/MSIE 8.0/)); // returns either a
```



Note: The expression !! is not an operator, but it is just twice of ! operator.

[↑ Back to Top](#)

70. What is the purpose of the delete operator [↗](#)

The delete keyword is used to delete the property as well as its value.

```
var user = { name: "John", age: 20 };
delete user.age;

console.log(user); // {name: "John"}
```



[↑ Back to Top](#)

71. What is typeof operator [↗](#)

You can use the JavaScript typeof operator to find the type of a JavaScript variable. It returns the type of a variable or an expression.

```
typeof "John Abraham"; // Returns "string"
typeof (1 + 2); // Returns "number"
typeof [1, 2, 3]; // Returns "object" because all arrays are also objects
```



[↑ Back to Top](#)

72. What is undefined property [↗](#)

The undefined property indicates that a variable has not been assigned a value, or declared but not initialized at all. The type of undefined value is undefined too.

```
var user; // Value is undefined, type is undefined
console.log(typeof user); //undefined
```



Any variable can be emptied by setting the value to undefined.

```
user = undefined;
```



[↑ Back to Top](#)

73. What is null value [↗](#)

The value null represents the intentional absence of any object value. It is one of JavaScript's primitive values. The type of null value is object. You can empty the variable by setting the value to null.

```
var user = null;
console.log(typeof user); //object
```



[↑ Back to Top](#)

74. What is the difference between null and undefined [↗](#)

Below are the main differences between null and undefined,

Null	Undefined
It is an assignment value which indicates that variable points to no object.	It is not an assignment value where a variable has been declared but has not yet been assigned a value.
Type of null is object	Type of undefined is undefined
The null value is a primitive value that represents the null, empty, or non-existent reference.	The undefined value is a primitive value used when a variable has not been assigned a value.
Indicates the absence of a value for a variable	Indicates absence of variable itself

Null	Undefined
Converted to zero (0) while performing primitive operations	Converted to NaN while performing primitive operations

[↑ Back to Top](#)

75. What is eval [↗](#)

The eval() function evaluates JavaScript code represented as a string. The string can be a JavaScript expression, variable, statement, or sequence of statements.

```
console.log(eval("1 + 2")); // 3
```



[↑ Back to Top](#)

76. What is the difference between window and document [↗](#)

Below are the main differences between window and document,

Window	Document
It is the root level element in any web page	It is the direct child of the window object. This is also known as Document Object Model(DOM)
By default window object is available implicitly in the page	You can access it via window.document or document.
It has methods like alert(), confirm() and properties like document, location	It provides methods like getElementById, getElementsByTagName, createElement etc

[↑ Back to Top](#)

77. How do you access history in javascript [↗](#)

The window.history object contains the browser's history. You can load previous and next URLs in the history using back() and next() methods.

```
function goBack() {
  window.history.back();
}
```



```
}  
function goForward() {  
    window.history.forward();  
}
```

Note: You can also access history without window prefix.

[↑ Back to Top](#)

78. How do you detect caps lock key turned on or not [↗](#)

The `MouseEvent` `getModifierState()` is used to return a boolean value that indicates whether the specified modifier key is activated or not. The modifiers such as CapsLock, ScrollLock and NumLock are activated when they are clicked, and deactivated when they are clicked again.

Let's take an input element to detect the CapsLock on/off behavior with an example,

```
<input type="password" onmousedown="enterInput(event)" />  
  
<p id="feedback"></p>  
  
<script>  
    function enterInput(e) {  
        var flag = e.getModifierState("CapsLock");  
        if (flag) {  
            document.getElementById("feedback").innerHTML = "CapsLock activated";  
        } else {  
            document.getElementById("feedback").innerHTML =  
                "CapsLock not activated";  
        }  
    }  
</script>
```



[↑ Back to Top](#)

79. What is NaN [↗](#)

The `isNaN()` function is used to determine whether a value is an illegal number (Not-a-Number) or not. i.e, This function returns true if the value equates to NaN. Otherwise it returns false.

```
isNaN("Hello"); //true  
isNaN("100"); //false
```



[↑ Back to Top](#)

80. What are the differences between undeclared and undefined variables [↗](#)

Below are the major differences between undeclared(not defined) and undefined variables,

undeclared	undefined
These variables do not exist in a program and are not declared	These variables declared in the program but have not assigned any value
If you try to read the value of an undeclared variable, then a runtime error is encountered	If you try to read the value of an undefined variable, an undefined value is returned.

[↑ Back to Top](#)

81. What are global variables [↗](#)

Global variables are those that are available throughout the length of the code without any scope. The var keyword is used to declare a local variable but if you omit it then it will become global variable

```
msg = "Hello"; // var is missing, it becomes global variable
```



[↑ Back to Top](#)

82. What are the problems with global variables [↗](#)

The problem with global variables is the conflict of variable names of local and global scope. It is also difficult to debug and test the code that relies on global variables.

[↑ Back to Top](#)

83. What is NaN property [↗](#)

The NaN property is a global property that represents "Not-a-Number" value. i.e, It indicates that a value is not a legal number. It is very rare to use NaN in a program but it can be used as return value for few cases

```
Math.sqrt(-1);  
parseInt("Hello");
```



[↑ Back to Top](#)

84. What is the purpose of isFinite function [↗](#)

The isFinite() function is used to determine whether a number is a finite, legal number. It returns false if the value is +infinity, -infinity, or NaN (Not-a-Number), otherwise it returns true.

```
isFinite(Infinity); // false  
isFinite(NaN); // false  
isFinite(-Infinity); // false  
  
isFinite(100); // true
```



[↑ Back to Top](#)

85. What is an event flow [↗](#)

Event flow is the order in which event is received on the web page. When you click an element that is nested in various other elements, before your click actually reaches its destination, or target element, it must trigger the click event for each of its parent elements first, starting at the top with the global window object. There are two ways of event flow

- i. Top to Bottom(Event Capturing)
- ii. Bottom to Top (Event Bubbling)

[↑ Back to Top](#)

86. What is event bubbling [↗](#)

Event bubbling is a type of event propagation where the event first triggers on the innermost target element, and then successively triggers on the ancestors (parents) of the target element in the same nesting hierarchy till it reaches the outermost DOM element.

[↑ Back to Top](#)

87. What is event capturing [↗](#)

Event capturing is a type of event propagation where the event is first captured by the outermost element, and then successively triggers on the descendants (children) of the target element in the same nesting hierarchy till it reaches the innermost DOM element.

[↑ Back to Top](#)

88. How do you submit a form using JavaScript [↗](#)

You can submit a form using `document.forms[0].submit()`. All the form input's information is submitted using `onsubmit` event handler

```
function submit() {  
    document.forms[0].submit();  
}
```



[↑ Back to Top](#)

89. How do you find operating system details [↗](#)

The `window.navigator` object contains information about the visitor's browser OS details. Some of the OS properties are available under `platform` property,

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



[↑ Back to Top](#)

90. What is the difference between document load and DOMContentLoaded events [↗](#)

The `DOMContentLoaded` event is fired when the initial HTML document has been completely loaded and parsed, without waiting for assets(stylesheets, images, and subframes) to finish loading. Whereas The `load` event is fired when the whole page has loaded, including all dependent resources(stylesheets, images).

[↑ Back to Top](#)

91. What is the difference between native, host and user objects [↗](#)

Native objects are objects that are part of the JavaScript language defined by the ECMAScript specification. For example, String, Math, RegExp, Object, Function etc core objects defined in the ECMAScript spec. Host objects are objects provided by the browser or runtime environment (Node). For example, window, XMLHttpRequest, DOM nodes etc are considered as host objects. User objects are objects defined in the javascript code. For example, User objects created for profile information.

[↑ Back to Top](#)

92. What are the tools or techniques used for debugging JavaScript code [↗](#)

You can use below tools or techniques for debugging javascript

- i. Chrome Devtools
- ii. debugger statement
- iii. Good old console.log statement

[↑ Back to Top](#)

93. What are the pros and cons of promises over callbacks [↗](#)

Below are the list of pros and cons of promises over callbacks,

Pros:

- i. It avoids callback hell which is unreadable
- ii. Easy to write sequential asynchronous code with .then()
- iii. Easy to write parallel asynchronous code with Promise.all()
- iv. Solves some of the common problems of callbacks(call the callback too late, too early, many times and swallow errors/exceptions)

Cons:

- i. It makes little complex code
- ii. You need to load a polyfill if ES6 is not supported

[↑ Back to Top](#)

94. What is the difference between an attribute and a property [↗](#)

Attributes are defined on the HTML markup whereas properties are defined on the DOM. For example, the below HTML element has 2 attributes type and value,

```
<input type="text" value="Name:">
```



You can retrieve the attribute value as below,

```
const input = document.querySelector("input");  
console.log(input.getAttribute("value")); // Good morning  
console.log(input.value); // Good morning
```



And after you change the value of the text field to "Good evening", it becomes like

```
console.log(input.getAttribute("value")); // Good evening  
console.log(input.value); // Good evening
```



[↑ Back to Top](#)

95. What is same-origin policy [↗](#)

The same-origin policy is a policy that prevents JavaScript from making requests across domain boundaries. An origin is defined as a combination of URI scheme, hostname, and port number. If you enable this policy then it prevents a malicious script on one page from obtaining access to sensitive data on another web page using Document Object Model(DOM).

[↑ Back to Top](#)

96. What is the purpose of void 0 [↗](#)

Void(0) is used to prevent the page from refreshing. This will be helpful to eliminate the unwanted side-effect, because it will return the undefined primitive value. It is commonly used for HTML documents that use href="JavaScript:Void(0);" within an <a> element. i.e, when you click a link, the browser loads a new page or refreshes the same page. But this behavior will be prevented using this expression. For example, the below link notify the message without reloading the page

```
<a href="JavaScript:void(0);" onclick="alert('Well done!')">  
  Click Me!  
</a>
```



[↑ Back to Top](#)

97. Is JavaScript a compiled or interpreted language [↗](#)

JavaScript is an interpreted language, not a compiled language. An interpreter in the browser reads over the JavaScript code, interprets each line, and runs it. Nowadays modern browsers use a technology known as Just-In-Time (JIT) compilation, which compiles JavaScript to executable bytecode just as it is about to run.

[↑ Back to Top](#)

98. Is JavaScript a case-sensitive language [↗](#)

Yes, JavaScript is a case sensitive language. The language keywords, variables, function & object names, and any other identifiers must always be typed with a consistent capitalization of letters.

[↑ Back to Top](#)

99. Is there any relation between Java and JavaScript [↗](#)

No, they are entirely two different programming languages and have nothing to do with each other. But both of them are Object Oriented Programming languages and like many other languages, they follow similar syntax for basic features(if, else, for, switch, break, continue etc).

[↑ Back to Top](#)

100. What are events [↗](#)

Events are "things" that happen to HTML elements. When JavaScript is used in HTML pages, JavaScript can `react` on these events. Some of the examples of HTML events are,

- i. Web page has finished loading
- ii. Input field was changed
- iii. Button was clicked

Let's describe the behavior of click event for button element,

```
<!doctype html>
<html>
  <head>
    <script>
      function greeting() {
        alert('Hello! Good morning');
      }
    </script>
  </head>
  <body>
    <button>Click Me</button>
  </body>
</html>
```



```
    </script>
</head>
<body>
    <button type="button" onclick="greeting()">Click me</button>
</body>
</html>
```

[↑ Back to Top](#)

101. Who created javascript [↗](#)

JavaScript was created by Brendan Eich in 1995 during his time at Netscape Communications. Initially it was developed under the name `Mocha`, but later the language was officially called `LiveScript` when it first shipped in beta releases of Netscape.

[↑ Back to Top](#)

102. What is the use of preventDefault method [↗](#)

The `preventDefault()` method cancels the event if it is cancelable, meaning that the default action or behaviour that belongs to the event will not occur. For example, prevent form submission when clicking on submit button and prevent opening the page URL when clicking on hyperlink are some common use cases.

```
document
    .getElementById("link")
    .addEventListener("click", function (event) {
        event.preventDefault();
    });
```



Note: Remember that not all events are cancelable.

[↑ Back to Top](#)

103. What is the use of stopPropagation method [↗](#)

The `stopPropagation` method is used to stop the event from bubbling up the event chain. For example, the below nested divs with `stopPropagation` method prevents default event propagation when clicking on nested div(Div1)

```
<p>Click DIV1 Element</p>
<div onclick="secondFunc()">DIV 2
    <div onclick="firstFunc(event)">DIV 1</div>
```



</div>

<script>

```
function firstFunc(event) {  
    alert("DIV 1");  
    event.stopPropagation();  
}
```

```
function secondFunc() {  
    alert("DIV 2");  
}
```

</script>

[↑ Back to Top](#)

104. What are the steps involved in return false usage [↗](#)

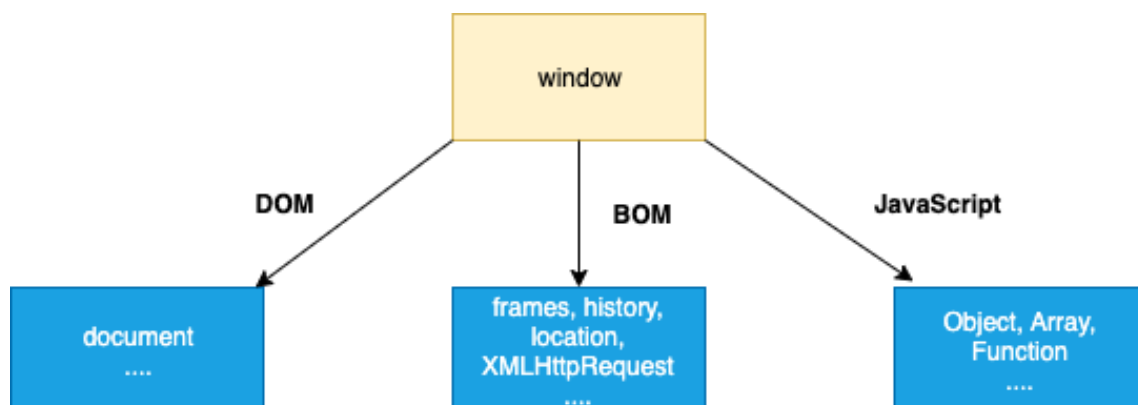
The return false statement in event handlers performs the below steps,

- i. First it stops the browser's default action or behaviour.
- ii. It prevents the event from propagating the DOM
- iii. Stops callback execution and returns immediately when called.

[↑ Back to Top](#)

105. What is BOM [↗](#)

The Browser Object Model (BOM) allows JavaScript to "talk to" the browser. It consists of the objects navigator, history, screen, location and document which are children of the window. The Browser Object Model is not standardized and can change based on different browsers.



[↑ Back to Top](#)

106. What is the use of setTimeout [↗](#)

The `setTimeout()` method is used to call a function or evaluate an expression after a specified number of milliseconds. For example, let's log a message after 2 seconds using `setTimeout` method,

```
setTimeout(function () {  
    console.log("Good morning");  
}, 2000);
```



[↑ Back to Top](#)

107. What is the use of `setInterval` [↗](#)

The `setInterval()` method is used to call a function or evaluate an expression at specified intervals (in milliseconds). For example, let's log a message after 2 seconds using `setInterval` method,

```
setInterval(function () {  
    console.log("Good morning");  
}, 2000);
```



[↑ Back to Top](#)

108. Why is JavaScript treated as Single threaded [↗](#)

JavaScript is a single-threaded language. Because the language specification does not allow the programmer to write code so that the interpreter can run parts of it in parallel in multiple threads or processes. Whereas languages like java, go, C++ can make multi-threaded and multi-process programs.

[↑ Back to Top](#)

109. What is an event delegation [↗](#)

Event delegation is a technique for listening to events where you delegate a parent element as the listener for all of the events that happen inside it.

For example, if you wanted to detect field changes in inside a specific form, you can use event delegation technique,

```
var form = document.querySelector("#registration-form");  
  
// Listen for changes to fields inside the form  
form.addEventListener(  
    "change", function (e) {  
        // Handle the event  
    })
```



```
"input",
function (event) {
    // Log the field that was changed
    console.log(event.target);
},
false
);
```

[↑ Back to Top](#)

110. What is ECMAScript [↗](#)

ECMAScript is the scripting language that forms the basis of JavaScript. ECMAScript standardized by the ECMA International standards organization in the ECMA-262 and ECMA-402 specifications. The first edition of ECMAScript was released in 1997.

[↑ Back to Top](#)

111. What is JSON [↗](#)

JSON (JavaScript Object Notation) is a lightweight format that is used for data interchanging. It is based on a subset of JavaScript language in the way objects are built in JavaScript.

[↑ Back to Top](#)

112. What are the syntax rules of JSON [↗](#)

Below are the list of syntax rules of JSON

- i. The data is in name/value pairs
- ii. The data is separated by commas
- iii. Curly braces hold objects
- iv. Square brackets hold arrays

[↑ Back to Top](#)

113. What is the purpose JSON stringify [↗](#)

When sending data to a web server, the data has to be in a string format. You can achieve this by converting JSON object into a string using stringify() method.

```
var userJSON = { name: "John", age: 31 };  
var userString = JSON.stringify(userJSON);  
console.log(userString); //{"name":"John","age":31}"
```



[↑ Back to Top](#)

114. How do you parse JSON string [↗](#)

When receiving the data from a web server, the data is always in a string format. But you can convert this string value to a javascript object using parse() method.

```
var userString = '{"name":"John","age":31}';  
var userJSON = JSON.parse(userString);  
console.log(userJSON); // {name: "John", age: 31}
```



[↑ Back to Top](#)

115. Why do you need JSON [↗](#)

When exchanging data between a browser and a server, the data can only be text. Since JSON is text only, it can easily be sent to and from a server, and used as a data format by any programming language.

[↑ Back to Top](#)

116. What are PWAs [↗](#)

Progressive web applications (PWAs) are a type of mobile app delivered through the web, built using common web technologies including HTML, CSS and JavaScript. These PWAs are deployed to servers, accessible through URLs, and indexed by search engines.

[↑ Back to Top](#)

117. What is the purpose of clearTimeout method [↗](#)

The clearTimeout() function is used in javascript to clear the timeout which has been set by setTimeout() function before that. i.e, The return value of setTimeout() function is stored in a variable and it's passed into the clearTimeout() function to clear the timer.

For example, the below setTimeout method is used to display the message after 3 seconds. This timeout can be cleared by the clearTimeout() method.

```
<script>
var msg;
function greeting() {
    alert('Good morning');
}
function start() {
    msg =setTimeout(greeting, 3000);

}

function stop() {
    clearTimeout(msg);
}
</script>
```



[↑ Back to Top](#)

118. What is the purpose of clearInterval method [↗](#)

The clearInterval() function is used in javascript to clear the interval which has been set by setInterval() function. i.e, The return value returned by setInterval() function is stored in a variable and it's passed into the clearInterval() function to clear the interval.

For example, the below setInterval method is used to display the message for every 3 seconds. This interval can be cleared by the clearInterval() method.

```
<script>
var msg;
function greeting() {
    alert('Good morning');
}
function start() {
    msg = setInterval(greeting, 3000);

}

function stop() {
    clearInterval(msg);
}
</script>
```



[↑ Back to Top](#)

119. How do you redirect new page in javascript [↗](#)

In vanilla javascript, you can redirect to a new page using the `location` property of window object. The syntax would be as follows,

```
function redirect() {  
    window.location.href = "newPage.html";  
}
```



[↑ Back to Top](#)

120. How do you check whether a string contains a substring [↗](#)

There are 3 possible ways to check whether a string contains a substring or not,

- i. **Using includes:** ES6 provided `String.prototype.includes` method to test a string contains a substring

```
var mainString = "hello",  
    subString = "hell";  
mainString.includes(subString);
```



- i. **Using indexOf:** In an ES5 or older environment, you can use `String.prototype.indexOf` which returns the index of a substring. If the index value is not equal to `-1` then it means the substring exists in the main string.

```
var mainString = "hello",  
    subString = "hell";  
mainString.indexOf(subString) !== -1;
```



- i. **Using RegEx:** The advanced solution is using Regular expression's test method(`RegExp.test`), which allows for testing for against regular expressions

```
var mainString = "hello",  
    regex = /hell/;  
regex.test(mainString);
```



[↑ Back to Top](#)

121. How do you validate an email in javascript [↗](#)

You can validate an email in javascript using regular expressions. It is recommended to do validations on the server side instead of the client side. Because the javascript can be disabled on the client side.


```
function validateEmail(email) {  
    var re =  
        /^[^<>()\[\]\\\.,;:\s@"]+(\.[^<>()\[\]\\\.,;:\s@"]+)*|(".+")@((\[[0-9]
```

[↑ Back to Top](#)

The above regular expression accepts unicode characters.

122. How do you get the current url with javascript [↗](#)

You can use `window.location.href` expression to get the current url path and you can use the same expression for updating the URL too. You can also use `document.URL` for read-only purposes but this solution has issues in FF.

```
console.log("location.href", window.location.href); // Returns full URL
```

[↑ Back to Top](#)

123. What are the various url properties of location object [↗](#)

The below `Location` object properties can be used to access URL components of the page,

- i. href - The entire URL
- ii. protocol - The protocol of the URL
- iii. host - The hostname and port of the URL
- iv. hostname - The hostname of the URL
- v. port - The port number in the URL
- vi. pathname - The path name of the URL
- vii. search - The query portion of the URL
- viii. hash - The anchor portion of the URL

[↑ Back to Top](#)

124. How do get query string values in javascript [↗](#)

You can use `URLSearchParams` to get query string values in javascript. Let's see an example to get the client code value from URL query string,

```
const urlParams = new URLSearchParams(window.location.search);  
const clientId = urlParams.get("clientId");
```



[↑ Back to Top](#)

125. How do you check if a key exists in an object [↗](#)

You can check whether a key exists in an object or not using three approaches,

- i. **Using in operator:** You can use the in operator whether a key exists in an object or not

```
"key" in obj;
```



and If you want to check if a key doesn't exist, remember to use parenthesis,

```
!("key" in obj);
```



- i. **Using hasOwnProperty method:** You can use `hasOwnProperty` to particularly test for properties of the object instance (and not inherited properties)

```
obj.hasOwnProperty("key"); // true
```



- i. **Using undefined comparison:** If you access a non-existing property from an object, the result is undefined. Let's compare the properties against undefined to determine the existence of the property.

```
const user = {  
  name: "John",  
};  
  
console.log(user.name !== undefined); // true  
console.log(user.nickname !== undefined); // false
```



[↑ Back to Top](#)

126. How do you loop through or enumerate javascript object [↗](#)

You can use the `for-in` loop to loop through javascript object. You can also make sure that the key you get is an actual property of an object, and doesn't come from the prototype using `hasOwnProperty` method.

```
var object = {
  k1: "value1",
  k2: "value2",
  k3: "value3",
};

for (var key in object) {
  if (object.hasOwnProperty(key)) {
    console.log(key + " -> " + object[key]); // k1 -> value1 ...
  }
}
```



[↑ Back to Top](#)

127. How do you test for an empty object [↗](#)

There are different solutions based on ECMAScript versions

- i. **Using Object entries(ECMA 7+):** You can use object entries length along with constructor type.

```
Object.entries(obj).length === 0 && obj.constructor === Object; // Since c
```



- i. **Using Object keys(ECMA 5+):** You can use object keys length along with constructor type.

```
Object.keys(obj).length === 0 && obj.constructor === Object; // Since c
```



- i. **Using for-in with hasOwnProperty(Pre-ECMA 5):** You can use a for-in loop along with hasOwnProperty.

```
function isEmpty(obj) {
  for (var prop in obj) {
    if (obj.hasOwnProperty(prop)) {
      return false;
    }
  }

  return JSON.stringify(obj) === JSON.stringify({});
}
```



[↑ Back to Top](#)

128. What is an arguments object [↗](#)

The arguments object is an Array-like object accessible inside functions that contains the values of the arguments passed to that function. For example, let's see how to use arguments object inside sum function,

```
function sum() {  
  var total = 0;  
  for (var i = 0, len = arguments.length; i < len; ++i) {  
    total += arguments[i];  
  }  
  return total;  
}  
  
sum(1, 2, 3); // returns 6
```



Note: You can't apply array methods on arguments object. But you can convert into a regular array as below.

```
var argsArray = Array.prototype.slice.call(arguments);
```



[↑ Back to Top](#)

129. How do you make first letter of the string in an uppercase [↗](#)

You can create a function which uses a chain of string methods such as charAt, toUpperCase and slice methods to generate a string with the first letter in uppercase.

```
function capitalizeFirstLetter(string) {  
  return string.charAt(0).toUpperCase() + string.slice(1);  
}
```



[↑ Back to Top](#)

130. What are the pros and cons of for loop [↗](#)

The for-loop is a commonly used iteration syntax in javascript. It has both pros and cons

Pros [↗](#)

- i. Works on every environment
- ii. You can use break and continue flow control statements

Cons

- i. Too verbose
- ii. Imperative
- iii. You might face one-by-off errors


 [Back to Top](#)

131. How do you display the current date in javascript

You can use `new Date()` to generate a new Date object containing the current date and time. For example, let's display the current date in mm/dd/yyyy

```
var today = new Date();
var dd = String(today.getDate()).padStart(2, "0");
var mm = String(today.getMonth() + 1).padStart(2, "0"); //January is 0!
var yyyy = today.getFullYear();

today = mm + "/" + dd + "/" + yyyy;
document.write(today);
```




 [Back to Top](#)

132. How do you compare two date objects

You need to use `date.getTime()` method to compare date values instead of comparison operators (`==`, `!=`, `===`, and `!==` operators)

```
var d1 = new Date();
var d2 = new Date(d1);
console.log(d1.getTime() === d2.getTime()); //True
console.log(d1 === d2); // False
```



 [Back to Top](#)

133. How do you check if a string starts with another string

You can use ECMAScript 6's `String.prototype.startsWith()` method to check if a string starts with another string or not. But it is not yet supported in all browsers. Let's see an example to see this usage,

```
"Good morning".startsWith("Good"); // true
"Good morning".startsWith("morning"); // false
```



[↑ Back to Top](#)

134. How do you trim a string in javascript [↗](#)

JavaScript provided a trim method on string types to trim any whitespaces present at the beginning or ending of the string.

```
" Hello World ".trim(); //Hello World
```



If your browser(<IE9) doesn't support this method then you can use below polyfill.

```
if (!String.prototype.trim) {
  (function () {
    // Make sure we trim BOM and NBSP
    var rtrim = /^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g;
    String.prototype.trim = function () {
      return this.replace(rtrim, "");
    };
  })();
}
```



[↑ Back to Top](#)

135. How do you add a key value pair in javascript [↗](#)

There are two possible solutions to add new properties to an object. Let's take a simple object to explain these solutions.

```
var object = {
  key1: value1,
  key2: value2,
};
```



- i. **Using dot notation:** This solution is useful when you know the name of the property

```
object.key3 = "value3";
```



- i. **Using square bracket notation:** This solution is useful when the name of the property is dynamically determined.

```
obj["key3"] = "value3";
```



[↑ Back to Top](#)

136. Is the `!--` notation represents a special operator [↗](#)

No, that's not a special operator. But it is a combination of 2 standard operators one after the other,

- i. A logical not (!)
- ii. A prefix decrement (--)

At first, the value decremented by one and then tested to see if it is equal to zero or not for determining the truthy/falsy value.

[↑ Back to Top](#)

137. How do you assign default values to variables [↗](#)

You can use the logical or operator `||` in an assignment expression to provide a default value. The syntax looks like as below,

```
var a = b || c;
```



As per the above expression, variable 'a' will get the value of 'c' only if 'b' is falsy (if is null, false, undefined, 0, empty string, or NaN), otherwise 'a' will get the value of 'b'.

[↑ Back to Top](#)

138. How do you define multiline strings [↗](#)

You can define multiline string literals using the `\` character followed by line terminator.

```
var str =  
    "This is a \  
very lengthy \  
sentence!";
```



But if you have a space after the '\' character, the code will look exactly the same, but it will raise a `SyntaxError`.

[↑ Back to Top](#)

139. What is an app shell model [↗](#)

An application shell (or app shell) architecture is one way to build a Progressive Web App that reliably and instantly loads on your users' screens, similar to what you see in native applications. It is useful for getting some initial HTML to the screen fast without a network.

[↑ Back to Top](#)

140. Can we define properties for functions [↗](#)

Yes, We can define properties for functions because functions are also objects.

```
fn = function (x) {  
  //Function code goes here  
};  
  
fn.name = "John";  
  
fn.profile = function (y) {  
  //Profile code goes here  
};
```



[↑ Back to Top](#)

141. What is the way to find the number of parameters expected by a function [↗](#)

You can use `function.length` syntax to find the number of parameters expected by a function. Let's take an example of `sum` function to calculate the sum of numbers,

```
function sum(num1, num2, num3, num4) {  
  return num1 + num2 + num3 + num4;  
}  
sum.length; // 4 is the number of parameters expected.
```



[↑ Back to Top](#)

142. What is a polyfill [↗](#)

A polyfill is a piece of JS code used to provide modern functionality on older browsers that do not natively support it. For example, Silverlight plugin polyfill can be used to mimic the functionality of an HTML Canvas element on Microsoft Internet Explorer 7.

[↑ Back to Top](#)

143. What are break and continue statements [↗](#)

The break statement is used to "jump out" of a loop. i.e, It breaks the loop and continues executing the code after the loop.

```
for (i = 0; i < 10; i++) {  
  if (i === 5) {  
    break;  
  }  
  text += "Number: " + i + "<br>";  
}
```



The continue statement is used to "jump over" one iteration in the loop. i.e, It breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
for (i = 0; i < 10; i++) {  
  if (i === 5) {  
    continue;  
  }  
  text += "Number: " + i + "<br>";  
}
```



[↑ Back to Top](#)

144. What are js labels [↗](#)

The label statement allows us to name loops and blocks in JavaScript. We can then use these labels to refer back to the code later. For example, the below code with labels avoids printing the numbers when they are same,

```
var i, j;
```



```

loop1: for (i = 0; i < 3; i++) {
  loop2: for (j = 0; j < 3; j++) {
    if (i === j) {
      continue loop1;
    }
    console.log("i = " + i + ", j = " + j);
  }
}

```

```

// Output is:
//   "i = 1, j = 0"
//   "i = 2, j = 0"
//   "i = 2, j = 1"

```

[↑ Back to Top](#)

145. What are the benefits of keeping declarations at the top [↗](#)

It is recommended to keep all declarations at the top of each script or function. The benefits of doing this are,

- i. Gives cleaner code
- ii. It provides a single place to look for local variables
- iii. Easy to avoid unwanted global variables
- iv. It reduces the possibility of unwanted re-declarations

[↑ Back to Top](#)

146. What are the benefits of initializing variables [↗](#)

It is recommended to initialize variables because of the below benefits,

- i. It gives cleaner code
- ii. It provides a single place to initialize variables
- iii. Avoid undefined values in the code

[↑ Back to Top](#)

147. What are the recommendations to create new object [↗](#)

It is recommended to avoid creating new objects using `new Object()`. Instead you can initialize values based on it's type to create the objects.

- i. Assign `{}` instead of `new Object()`
- ii. Assign `""` instead of `new String()`

- iii. Assign 0 instead of new Number()
- iv. Assign false instead of new Boolean()
- v. Assign [] instead of new Array()
- vi. Assign /(())/ instead of new RegExp()
- vii. Assign function (){} instead of new Function()

You can define them as an example,

```
var v1 = {};  
var v2 = "";  
var v3 = 0;  
var v4 = false;  
var v5 = [];  
var v6 = /(())/;  
var v7 = function () {};
```



[↑ Back to Top](#)

148. How do you define JSON arrays [↗](#)

JSON arrays are written inside square brackets and arrays contain javascript objects. For example, the JSON array of users would be as below,

```
"users": [  
  {"firstName": "John", "lastName": "Abrahm"},  
  {"firstName": "Anna", "lastName": "Smith"},  
  {"firstName": "Shane", "lastName": "Warn"}  
]
```



[↑ Back to Top](#)

149. How do you generate random integers [↗](#)

You can use Math.random() with Math.floor() to return random integers. For example, if you want generate random integers between 1 to 10, the multiplication factor should be 10,

```
Math.floor(Math.random() * 10) + 1; // returns a random integer from 1 to 10  
Math.floor(Math.random() * 100) + 1; // returns a random integer from 1 to 100
```



Note: `Math.random()` returns a random number between 0 (inclusive), and 1 (exclusive)

[↑ Back to Top](#)

150. Can you write a random integers function to print integers with in a range [↗](#)

Yes, you can create a proper random function to return a random number between min and max (both included)

```
function randomInteger(min, max) {  
  return Math.floor(Math.random() * (max - min + 1)) + min;  
}  
randomInteger(1, 100); // returns a random integer from 1 to 100  
randomInteger(1, 1000); // returns a random integer from 1 to 1000
```



[↑ Back to Top](#)

151. What is tree shaking [↗](#)

Tree shaking is a form of dead code elimination. It means that unused modules will not be included in the bundle during the build process and for that it relies on the static structure of ES2015 module syntax, (i.e. import and export). Initially this has been popularized by the ES2015 module bundler `rollup`.

[↑ Back to Top](#)

152. What is the need of tree shaking [↗](#)

Tree Shaking can significantly reduce the code size in any application. i.e, The less code we send over the wire the more performant the application will be. For example, if we just want to create a "Hello World" Application using SPA frameworks then it will take around a few MBs, but by tree shaking it can bring down the size to just a few hundred KBs. Tree shaking is implemented in Rollup and Webpack bundlers.

[↑ Back to Top](#)

153. Is it recommended to use eval [↗](#)

No, it allows arbitrary code to be run which causes a security problem. As we know that the `eval()` function is used to run text as code. In most of the cases, it should not be necessary to use it.

[↑ Back to Top](#)

154. What is a Regular Expression [↗](#)

A regular expression is a sequence of characters that forms a search pattern. You can use this search pattern for searching data in a text. These can be used to perform all types of text search and text replace operations. Let's see the syntax format now,

```
/pattern/modifiers;
```



For example, the regular expression or search pattern with case-insensitive username would be,

```
/John/i;
```



[↑ Back to Top](#)

155. What are the string methods available in Regular expression [↗](#)

Regular Expressions has two string methods: `search()` and `replace()`. The `search()` method uses an expression to search for a match, and returns the position of the match.

```
var msg = "Hello John";  
var n = msg.search(/John/i); // 6
```



The `replace()` method is used to return a modified string where the pattern is replaced.

```
var msg = "Hello John";  
var n = msg.replace(/John/i, "Buttler"); // Hello Buttler
```



[↑ Back to Top](#)

156. What are modifiers in regular expression [↗](#)

Modifiers can be used to perform case-insensitive and global searches. Let's list down some of the modifiers,

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match rather than stops at first match
m	Perform multiline matching

Let's take an example of global modifier,

```
var text = "Learn JS one by one";  
var pattern = /one/g;  
var result = text.match(pattern); // one,one
```



[↑ Back to Top](#)

157. What are regular expression patterns [↗](#)

Regular Expressions provide a group of patterns in order to match characters. Basically they are categorized into 3 types,

- i. **Brackets:** These are used to find a range of characters. For example, below are some use cases,
 - a. [abc]: Used to find any of the characters between the brackets(a,b,c)
 - b. [0-9]: Used to find any of the digits between the brackets
 - c. (a|b): Used to find any of the alternatives separated with |
- ii. **Metacharacters:** These are characters with a special meaning For example, below are some use cases,
 - a. \d: Used to find a digit
 - b. \s: Used to find a whitespace character
 - c. \b: Used to find a match at the beginning or ending of a word
- iii. **Quantifiers:** These are useful to define quantities For example, below are some use cases,
 - a. n+: Used to find matches for any string that contains at least one n
 - b. n*: Used to find matches for any string that contains zero or more occurrences of n
 - c. n?: Used to find matches for any string that contains zero or one occurrences of n

[↑ Back to Top](#)

158. What is a RegExp object [↗](#)

RegExp object is a regular expression object with predefined properties and methods. Let's see the simple usage of RegExp object,

```
var regexp = new RegExp("\\w+");  
console.log(regexp);  
// expected output: /\w+/  

```



[↑ Back to Top](#)

159. How do you search a string for a pattern [↗](#)

You can use the test() method of regular expression in order to search a string for a pattern, and return true or false depending on the result.

```
var pattern = /you/;  
console.log(pattern.test("How are you?")); //true
```



[↑ Back to Top](#)

160. What is the purpose of exec method [↗](#)

The purpose of exec method is similar to test method but it executes a search for a match in a specified string and returns a result array, or null instead of returning true/false.

```
var pattern = /you/;  
console.log(pattern.exec("How are you?")); //["you", index: 8, input: "How are you?"]
```



[↑ Back to Top](#)

161. How do you change the style of a HTML element [↗](#)

You can change inline style or classname of a HTML element using javascript

i. **Using style property:** You can modify inline style using style property

```
document.getElementById("title").style.fontSize = "30px";
```



- i. **Using ClassName property:** It is easy to modify element class using className property

```
document.getElementById("title").className = "custom-title";
```



[↑ Back to Top](#)

162. What would be the result of 1+2+'3' [↗](#)

The output is going to be 33 . Since 1 and 2 are numeric values, the result of the first two digits is going to be a numeric value 3 . The next digit is a string type value because of that the addition of numeric value 3 and string type value 3 is just going to be a concatenation value 33 .

[↑ Back to Top](#)

163. What is a debugger statement [↗](#)

The debugger statement invokes any available debugging functionality, such as setting a breakpoint. If no debugging functionality is available, this statement has no effect. For example, in the below function a debugger statement has been inserted. So execution is paused at the debugger statement just like a breakpoint in the script source.

```
function getProfile() {  
    // code goes here  
    debugger;  
    // code goes here  
}
```



[↑ Back to Top](#)

164. What is the purpose of breakpoints in debugging [↗](#)

You can set breakpoints in the javascript code once the debugger statement is executed and the debugger window pops up. At each breakpoint, javascript will stop executing, and let you examine the JavaScript values. After examining values, you can resume the execution of code using the play button.

[↑ Back to Top](#)

165. Can I use reserved words as identifiers [↗](#)

No, you cannot use the reserved words as variables, labels, object or function names. Let's see one simple example,

```
var else = "hello"; // Uncaught SyntaxError: Unexpected token else
```



[↑ Back to Top](#)

166. How do you detect a mobile browser [↗](#)

You can use regex which returns a true or false value depending on whether or not the user is browsing with a mobile.

```
window.mobilecheck = function () {  
  var mobileCheck = false;  
  (function (a) {  
    if (  
      /(android|bb\d+|meego).+mobile|avantgo|bada\/|blackberry|blazer|compal  
      a  
    ) ||  
      /1207|6310|6590|3gso|4thp|50[1-6]i|770s|802s|a wa|abac|ac(er|oo|s\-)|a  
      a.substr(0, 4)  
    )  
  )  
    mobileCheck = true;  
  })(navigator.userAgent || navigator.vendor || window.opera);  
  return mobileCheck;  
};
```



[↑ Back to Top](#)

167. How do you detect a mobile browser without regexp [↗](#)

You can detect mobile browsers by simply running through a list of devices and checking if the useragent matches anything. This is an alternative solution for RegExp usage,

```
function detectmob() {  
  if (  
    navigator.userAgent.match(/Android/i) ||  
    navigator.userAgent.match(/webOS/i) ||  
    navigator.userAgent.match(/iPhone/i) ||  
    navigator.userAgent.match(/iPad/i) ||  
    navigator.userAgent.match(/iPod/i) ||
```



```

        navigator.userAgent.match(/BlackBerry/i) ||
        navigator.userAgent.match(/Windows Phone/i)
    ) {
        return true;
    } else {
        return false;
    }
}

```

[↑ Back to Top](#)

168. How do you get the image width and height using JS [↗](#)

You can programmatically get the image and check the dimensions(width and height) using Javascript.

```

var img = new Image();
img.onload = function () {
    console.log(this.width + "x" + this.height);
};
img.src = "http://www.google.com/intl/en_ALL/images/logo.gif";

```



[↑ Back to Top](#)

169. How do you make synchronous HTTP request [↗](#)

Browsers provide an XMLHttpRequest object which can be used to make synchronous HTTP requests from JavaScript

```

function httpGet(theUrl) {
    var xmlHttpReq = new XMLHttpRequest();
    xmlHttpReq.open("GET", theUrl, false); // false for synchronous request
    xmlHttpReq.send(null);
    return xmlHttpReq.responseText;
}

```



[↑ Back to Top](#)

170. How do you make asynchronous HTTP request [↗](#)

Browsers provide an XMLHttpRequest object which can be used to make asynchronous HTTP requests from JavaScript by passing the 3rd parameter as true.

```
function httpGetAsync(theUrl, callback) {
    var xmlHttpReq = new XMLHttpRequest();
    xmlHttpReq.onreadystatechange = function () {
        if (xmlHttpReq.readyState == 4 && xmlHttpReq.status == 200)
            callback(xmlHttpReq.responseText);
    };
    xmlHttp.open("GET", theUrl, true); // true for asynchronous
    xmlHttp.send(null);
}
```



[↑ Back to Top](#)

171. How do you convert date to another timezone in javascript [↗](#)

You can use the `toLocaleString()` method to convert dates in one timezone to another. For example, let's convert current date to British English timezone as below,

```
console.log(event.toLocaleString("en-GB", { timeZone: "UTC" })); //29/06/2017 11:11:11
```



[↑ Back to Top](#)

172. What are the properties used to get size of window [↗](#)

You can use `innerWidth`, `innerHeight`, `clientWidth`, `clientHeight` properties of windows, document element and document body objects to find the size of a window. Let's use them combination of these properties to calculate the size of a window or document,

```
var width =
    window.innerWidth ||
    document.documentElement.clientWidth ||
    document.body.clientWidth;

var height =
    window.innerHeight ||
    document.documentElement.clientHeight ||
    document.body.clientHeight;
```



[↑ Back to Top](#)

173. What is a conditional operator in javascript [↗](#)

The conditional (ternary) operator is the only JavaScript operator that takes three operands which acts as a shortcut for if statements.

```
var isAuthenticated = false;
console.log(
  isAuthenticated ? "Hello, welcome" : "Sorry, you are not authenticated"
); //Sorry, you are not authenticated
```



[↑ Back to Top](#)

174. Can you apply chaining on conditional operator [↗](#)

Yes, you can apply chaining on conditional operators similar to if ... else if ... else if ... else chain. The syntax is going to be as below,

```
function traceValue(someParam) {
  return condition1
    ? value1
    : condition2
    ? value2
    : condition3
    ? value3
    : value4;
}
```



// The above conditional operator is equivalent to:

```
function traceValue(someParam) {
  if (condition1) {
    return value1;
  } else if (condition2) {
    return value2;
  } else if (condition3) {
    return value3;
  } else {
    return value4;
  }
}
```

[↑ Back to Top](#)

175. What are the ways to execute javascript after page load [↗](#)

You can execute javascript after page load in many different ways,

i. window.onload:

```
window.onload = function ...
```



i. document.onload:

```
document.onload = function ...
```



i. body onload:

```
<body onload="script();">
```



[↑ Back to Top](#)

176. What is the difference between proto and prototype [↗](#)

The `__proto__` object is the actual object that is used in the lookup chain to resolve methods, etc. Whereas `prototype` is the object that is used to build `__proto__` when you create an object with `new`.

```
new Employee().__proto__ === Employee.prototype;  
new Employee().prototype === undefined;
```



There are few more differences,

feature	Prototype	proto
Access	All the function constructors have prototype properties.	All the objects have <code>__proto__</code> property
Purpose	Used to reduce memory wastage with a single copy of function	Used in lookup chain to resolve methods, constructors etc.
ECMAScript	Introduced in ES6	Introduced in ES5
Usage	Frequently used	Rarely used

[↑ Back to Top](#)

177. Give an example where do you really need semicolon [↗](#)

It is recommended to use semicolons after every statement in JavaScript. For example, in the below case it throws an error "... is not a function" at runtime due to missing semicolon.

```
// define a function
var fn = (function () {
  //...
})();
// semicolon missing at this line

// then execute some code inside a closure
function () {
  //...
}
})();
```



and it will be interpreted as

```
var fn = (function () {
  //...
} )(function () {
  //...
})();
```



In this case, we are passing the second function as an argument to the first function and then trying to call the result of the first function call as a function. Hence, the second function will fail with a "... is not a function" error at runtime.

[↑ Back to Top](#)

178. What is a freeze method [↗](#)

The **freeze()** method is used to freeze an object. Freezing an object does not allow adding new properties to an object, prevents from removing and prevents changing the enumerability, configurability, or writability of existing properties. i.e, It returns the passed object and does not create a frozen copy.

```
const obj = {
  prop: 100,
};

Object.freeze(obj);
obj.prop = 200; // Throws an error in strict mode
```



```
console.log(obj.prop); //100
```

Remember freezing is only applied to the top-level properties in objects but not for nested objects. For example, let's try to freeze user object which has employment details as nested object and observe that details have been changed.

```
const user = {  
  name: "John",  
  employment: {  
    department: "IT",  
  },  
};  
  
Object.freeze(user);  
user.employment.department = "HR";
```



Note: It causes a TypeError if the argument passed is not an object.

[↑ Back to Top](#)

179. What is the purpose of freeze method [↗](#)

Below are the main benefits of using freeze method,

- i. It is used for freezing objects and arrays.
- ii. It is used to make an object immutable.

[↑ Back to Top](#)

180. Why do I need to use freeze method [↗](#)

In the Object-oriented paradigm, an existing API contains certain elements that are not intended to be extended, modified, or re-used outside of their current context. Hence it works as the `final` keyword which is used in various languages.

[↑ Back to Top](#)

181. How do you detect a browser language preference [↗](#)

You can use navigator object to detect a browser language preference as below,

```
var language =  
(navigator.languages && navigator.languages[0]) || // Chrome / Firefox
```



```
navigator.language || // All browsers
navigator.userAgent; // IE <= 10

console.log(language);
```

[↑ Back to Top](#)

182. How to convert string to title case with javascript [↗](#)

Title case means that the first letter of each word is capitalized. You can convert a string to title case using the below function,

```
function toTitleCase(str) {
  return str.replace(/\w\S*/g, function (txt) {
    return txt.charAt(0).toUpperCase() + txt.substring(1).toLowerCase();
  });
}
toTitleCase("good morning john"); // Good Morning John
```



[↑ Back to Top](#)

183. How do you detect javascript disabled in the page [↗](#)

You can use the `<noscript>` tag to detect javascript disabled or not. The code block inside `<noscript>` gets executed when JavaScript is disabled, and is typically used to display alternative content when the page generated in JavaScript.

```
<script type="javascript">
  // JS related code goes here
</script>
<noscript>
  <a href="next_page.html?noJS=true">JavaScript is disabled in the page. P.
</noscript>
```



[↑ Back to Top](#)

184. What are various operators supported by javascript [↗](#)

An operator is capable of manipulating (mathematical and logical computations) a certain value or operand. There are various operators supported by JavaScript as below,

- i. **Arithmetic Operators:** Includes + (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus), ++ (Increment) and -- (Decrement)
- ii. **Comparison Operators:** Includes == (Equal), != (Not Equal), === (Equal with type), > (Greater than), >= (Greater than or Equal to), < (Less than), <= (Less than or Equal to)
- iii. **Logical Operators:** Includes && (Logical AND), || (Logical OR), !(Logical NOT)
- iv. **Assignment Operators:** Includes = (Assignment Operator), += (Add and Assignment Operator), -= (Subtract and Assignment Operator), *= (Multiply and Assignment), /= (Divide and Assignment), %= (Modules and Assignment)
- v. **Ternary Operators:** It includes conditional(: ?) Operator
- vi. **typeof Operator:** It uses to find type of variable. The syntax looks like `typeof variable`

[↑ Back to Top](#)

185. What is a rest parameter [↗](#)

Rest parameter is an improved way to handle function parameters which allows us to represent an indefinite number of arguments as an array. The syntax would be as below,

```
function f(a, b, ...theArgs) {  
  // ...  
}
```



For example, let's take a sum example to calculate on dynamic number of parameters,

```
function sum(...args) {  
  let total = 0;  
  for (const i of args) {  
    total += i;  
  }  
  return total;  
}
```



```
console.log(sum(1, 2)); //3  
console.log(sum(1, 2, 3)); //6  
console.log(sum(1, 2, 3, 4)); //13  
console.log(sum(1, 2, 3, 4, 5)); //15
```

Note: Rest parameter is added in ES2015 or ES6

[↑ Back to Top](#)

186. What happens if you do not use rest parameter as a last argument [↗](#)

The rest parameter should be the last argument, as its job is to collect all the remaining arguments into an array. For example, if you define a function like below it doesn't make any sense and will throw an error.

```
function someFunc(a,...b,c){  
  //You code goes here  
  return;  
}
```



[↑ Back to Top](#)

187. What are the bitwise operators available in javascript [↗](#)

Below are the list of bitwise logical operators used in JavaScript

- i. Bitwise AND (&)
- ii. Bitwise OR (|)
- iii. Bitwise XOR (^)
- iv. Bitwise NOT (~)
- v. Left Shift (<<)
- vi. Sign Propagating Right Shift (>>)
- vii. Zero fill Right Shift (>>>)

[↑ Back to Top](#)

188. What is a spread operator [↗](#)

Spread operator allows iterables(arrays / objects / strings) to be expanded into single arguments/elements. Let's take an example to see this behavior,

```
function calculateSum(x, y, z) {  
  return x + y + z;  
}
```

```
const numbers = [1, 2, 3];
```

```
console.log(calculateSum(...numbers)); // 6
```



[↑ Back to Top](#)

189. How do you determine whether object is frozen or not [↗](#)

Object.isFrozen() method is used to determine if an object is frozen or not. An object is frozen if all of the below conditions hold true,

- i. If it is not extensible.
- ii. If all of its properties are non-configurable.
- iii. If all its data properties are non-writable. The usage is going to be as follows,

```
const object = {  
  property: "Welcome JS world",  
};  
Object.freeze(object);  
console.log(Object.isFrozen(object));
```



[↑ Back to Top](#)

190. How do you determine two values same or not using object [↗](#)

The Object.is() method determines whether two values are the same value. For example, the usage with different types of values would be,

```
Object.is("hello", "hello"); // true  
Object.is(window, window); // true  
Object.is([], []); // false
```



Two values are the same if one of the following holds:

- i. both undefined
- ii. both null
- iii. both true or both false
- iv. both strings of the same length with the same characters in the same order
- v. both the same object (means both object have same reference)
- vi. both numbers and both +0 both -0 both NaN both non-zero and both not NaN and both have the same value.

[↑ Back to Top](#)

191. What is the purpose of using object is method [↗](#)

Some of the applications of Object's is method are follows,

- i. It is used for comparison of two strings.
- ii. It is used for comparison of two numbers.
- iii. It is used for comparing the polarity of two numbers.
- iv. It is used for comparison of two objects.

[↑ Back to Top](#)

192. How do you copy properties from one object to other [↗](#)

You can use the `Object.assign()` method which is used to copy the values and properties from one or more source objects to a target object. It returns the target object which has properties and values copied from the source objects. The syntax would be as below,

```
Object.assign(target, ...sources);
```



Let's take example with one source and one target object,

```
const target = { a: 1, b: 2 };
const source = { b: 3, c: 4 };

const returnedTarget = Object.assign(target, source);

console.log(target); // { a: 1, b: 3, c: 4 }

console.log(returnedTarget); // { a: 1, b: 3, c: 4 }
```



As observed in the above code, there is a common property(`b`) from source to target so it's value has been overwritten.

[↑ Back to Top](#)

193. What are the applications of assign method [↗](#)

Below are the some of main applications of `Object.assign()` method,

- i. It is used for cloning an object.
- ii. It is used to merge objects with the same properties.

[↑ Back to Top](#)

194. What is a proxy object [↗](#)

The Proxy object is used to define custom behavior for fundamental operations such as property lookup, assignment, enumeration, function invocation, etc. The syntax would be as follows,

```
var p = new Proxy(target, handler);
```



Let's take an example of proxy object,

```
var handler = {
  get: function (obj, prop) {
    return prop in obj ? obj[prop] : 100;
  },
};

var p = new Proxy({}, handler);
p.a = 10;
p.b = null;

console.log(p.a, p.b); // 10, null
console.log("c" in p, p.c); // false, 100
```



In the above code, it uses `get` handler which define the behavior of the proxy when an operation is performed on it

[↑ Back to Top](#)

195. What is the purpose of seal method

The **Object.seal()** method is used to seal an object, by preventing new properties from being added to it and marking all existing properties as non-configurable. But values of present properties can still be changed as long as they are writable. Let's see the below example to understand more about seal() method

```
const object = {
  property: "Welcome JS world",
};
Object.seal(object);
object.property = "Welcome to object world";
console.log(Object.isSealed(object)); // true
delete object.property; // You cannot delete when sealed
console.log(object.property); //Welcome to object world
```



[↑ Back to Top](#)

196. What are the applications of seal method

Below are the main applications of Object.seal() method,

- i. It is used for sealing objects and arrays.
- ii. It is used to make an object immutable.

 [Back to Top](#)

197. What are the differences between freeze and seal methods

If an object is frozen using the Object.freeze() method then its properties become immutable and no changes can be made in them whereas if an object is sealed using the Object.seal() method then the changes can be made in the existing properties of the object.

 [Back to Top](#)

198. How do you determine if an object is sealed or not

The Object.isSealed() method is used to determine if an object is sealed or not. An object is sealed if all of the below conditions hold true

- i. If it is not extensible.
- ii. If all of its properties are non-configurable.
- iii. If it is not removable (but not necessarily non-writable). Let's see it in the action

```
const object = {  
  property: "Hello, Good morning",  
};  
  
Object.seal(object); // Using seal() method to seal the object  
  
console.log(Object.isSealed(object)); // checking whether the object is sealed
```



 [Back to Top](#)

199. How do you get enumerable key and value pairs

The `Object.entries()` method is used to return an array of a given object's own enumerable string-keyed property `[key, value]` pairs, in the same order as that provided by a `for...in` loop. Let's see the functionality of `object.entries()` method in an example,

```
const object = {
  a: "Good morning",
  b: 100,
};

for (let [key, value] of Object.entries(object)) {
  console.log(`${key}: ${value}`); // a: 'Good morning'
  // b: 100
}
```



Note: The order is not guaranteed as object defined.

[↑ Back to Top](#)

200. What is the main difference between `Object.values` and `Object.entries` method [↗](#)

The `Object.values()` method's behavior is similar to `Object.entries()` method but it returns an array of values instead `[key,value]` pairs.

```
const object = {
  a: "Good morning",
  b: 100,
};

for (let value of Object.values(object)) {
  console.log(`${value}`); // 'Good morning'
  100;
}
```



[↑ Back to Top](#)

201. How can you get the list of keys of any object [↗](#)

You can use the `Object.keys()` method which is used to return an array of a given object's own property names, in the same order as we get with a normal loop. For example, you can get the keys of a user object,

```
const user = {
  name: "John",
  gender: "male",
  age: 40,
};

console.log(Object.keys(user)); //['name', 'gender', 'age']
```



[↑ Back to Top](#)

202. How do you create an object with prototype [↗](#)

The `Object.create()` method is used to create a new object with the specified prototype object and properties. i.e, It uses an existing object as the prototype of the newly created object. It returns a new object with the specified prototype object and properties.

```
const user = {
  name: "John",
  printInfo: function () {
    console.log(`My name is ${this.name}.`);
  },
};

const admin = Object.create(user);

admin.name = "Nick"; // Remember that "name" is a property set on "admin" but
                      // it inherits the printInfo method from "user"

admin.printInfo(); // My name is Nick
```



[↑ Back to Top](#)

203. What is a WeakSet [↗](#)

WeakSet is used to store a collection of weakly(weak references) held objects. The syntax would be as follows,

```
new WeakSet([iterable]);
```



Let's see the below example to explain its behavior,


```
var ws = new WeakSet();
var user = {};
ws.add(user);
ws.has(user); // true
ws.delete(user); // removes user from the set
ws.has(user); // false, user has been removed
```



[↑ Back to Top](#)

204. What are the differences between WeakSet and Set [↗](#)

The main difference is that references to objects in Set are strong while references to objects in WeakSet are weak. i.e, An object in WeakSet can be garbage collected if there is no other reference to it. Other differences are,

- i. Sets can store any value Whereas WeakSets can store only collections of objects
- ii. WeakSet does not have size property unlike Set
- iii. WeakSet does not have methods such as clear, keys, values, entries, forEach.
- iv. WeakSet is not iterable.

[↑ Back to Top](#)

205. List down the collection of methods available on WeakSet [↗](#)

Below are the list of methods available on WeakSet,

- i. add(value): A new object is appended with the given value to the weakset
- ii. delete(value): Deletes the value from the WeakSet collection.
- iii. has(value): It returns true if the value is present in the WeakSet Collection, otherwise it returns false.

Let's see the functionality of all the above methods in an example,

```
var weakSetObject = new WeakSet();
var firstObject = {};
var secondObject = {};
// add(value)
weakSetObject.add(firstObject);
weakSetObject.add(secondObject);
console.log(weakSetObject.has(firstObject)); //true
weakSetObject.delete(secondObject);
```



[↑ Back to Top](#)

206. What is a WeakMap [↗](#)

The WeakMap object is a collection of key/value pairs in which the keys are weakly referenced. In this case, keys must be objects and the values can be arbitrary values. The syntax is looking like as below,

```
new WeakMap([iterable]);
```



Let's see the below example to explain its behavior,

```
var ws = new WeakMap();  
var user = {};  
ws.set(user);  
ws.has(user); // true  
ws.delete(user); // removes user from the map  
ws.has(user); // false, user has been removed
```



[↑ Back to Top](#)

207. What are the differences between WeakMap and Map [↗](#)

The main difference is that references to key objects in Map are strong while references to key objects in WeakMap are weak. i.e, A key object in WeakMap can be garbage collected if there is no other reference to it. Other differences are,

- i. Maps can store any key type Whereas WeakMaps can store only collections of key objects
- ii. WeakMap does not have size property unlike Map
- iii. WeakMap does not have methods such as clear, keys, values, entries, forEach.
- iv. WeakMap is not iterable.

[↑ Back to Top](#)

208. List down the collection of methods available on WeakMap [↗](#)

Below are the list of methods available on WeakMap,

- i. set(key, value): Sets the value for the key in the WeakMap object. Returns the WeakMap object.
- ii. delete(key): Removes any value associated to the key.

- iii. `has(key)`: Returns a Boolean asserting whether a value has been associated to the key in the WeakMap object or not.
 - iv. `get(key)`: Returns the value associated to the key, or undefined if there is none.
- Let's see the functionality of all the above methods in an example,

```
var weakMapObject = new WeakMap();
var firstObject = {};
var secondObject = {};
// set(key, value)
weakMapObject.set(firstObject, "John");
weakMapObject.set(secondObject, 100);
console.log(weakMapObject.has(firstObject)); //true
console.log(weakMapObject.get(firstObject)); // John
weakMapObject.delete(secondObject);
```



[↑ Back to Top](#)

209. What is the purpose of `uneval` [↗](#)

The `uneval()` is an inbuilt function which is used to create a string representation of the source code of an Object. It is a top-level function and is not associated with any object. Let's see the below example to know more about its functionality,

```
var a = 1;
uneval(a); // returns a String containing 1
uneval(function user() {}); // returns "(function user(){})"
```



The `uneval()` function has been deprecated. It is recommended to use `toString()` for functions and `JSON.stringify()` for other cases.

```
function user() {}
console.log(user.toString()); // returns "(function user(){})"
```



[↑ Back to Top](#)

210. How do you encode an URL [↗](#)

The `encodeURIComponent()` function is used to encode complete URI which has special characters except `(/ ? : @ & = + $ #)` characters.

```
var uri = "https://mozilla.org/?x=шеллы";
var encoded = encodeURIComponent(uri);
```



```
console.log(encoded); // https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%
```

[↑ Back to Top](#)

211. How do you decode an URL [↗](#)

The `decodeURI()` function is used to decode a Uniform Resource Identifier (URI) previously created by `encodeURI()`.

```
var uri = "https://mozilla.org/?x=шеллы";
var encoded = encodeURI(uri);
console.log(encoded); // https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%
try {
  console.log(decodeURI(encoded)); // "https://mozilla.org/?x=шеллы"
} catch (e) {
  // catches a malformed URI
  console.error(e);
}
```

[↑ Back to Top](#)

212. How do you print the contents of web page [↗](#)

The `window` object provided a `print()` method which is used to print the contents of the current window. It opens a Print dialog box which lets you choose between various printing options. Let's see the usage of `print` method in an example,

```
<input type="button" value="Print" onclick="window.print()" />
```

Note: In most browsers, it will block while the print dialog is open.

[↑ Back to Top](#)

213. What is the difference between `uneval` and `eval` [↗](#)

The `uneval` function returns the source of a given object; whereas the `eval` function does the opposite, by evaluating that source code in a different memory area. Let's see an example to clarify the difference,

```
var msg = uneval(function greeting() {
  return "Hello, Good morning";
});
```

```
});  
var greeting = eval(msg);  
greeting(); // returns "Hello, Good morning"
```

[↑ Back to Top](#)

214. What is an anonymous function [↗](#)

An anonymous function is a function without a name! Anonymous functions are commonly assigned to a variable name or used as a callback function. The syntax would be as below,

```
function (optionalParameters) {  
    //do something  
}  
  
const myFunction = function(){ //Anonymous function assigned to a variable  
    //do something  
};  
  
[1, 2, 3].map(function(element){ //Anonymous function used as a callback fun  
    //do something  
});
```

Let's see the above anonymous function in an example,

```
var x = function (a, b) {  
    return a * b;  
};  
var z = x(5, 10);  
console.log(z); // 50
```

[↑ Back to Top](#)

215. What is the precedence order between local and global variables [↗](#)

A local variable takes precedence over a global variable with the same name. Let's see this behavior in an example.

```
var msg = "Good morning";  
function greeting() {  
    msg = "Good Evening";
```

```
    console.log(msg); // Good Evening
  }
  greeting();
```

[↑ Back to Top](#)

216. What are javascript accessors [↗](#)

ECMAScript 5 introduced javascript object accessors or computed properties through getters and setters. Getters uses the `get` keyword whereas Setters uses the `set` keyword.

```
var user = {
  firstName: "John",
  lastName: "Abraham",
  language: "en",
  get lang() {
    return this.language;
  },
  set lang(lang) {
    this.language = lang;
  },
};
console.log(user.lang); // getter access lang as en
user.lang = "fr";
console.log(user.lang); // setter used to set lang as fr
```



[↑ Back to Top](#)

217. How do you define property on Object constructor [↗](#)

The `Object.defineProperty()` static method is used to define a new property directly on an object, or modify an existing property on an object, and returns the object. Let's see an example to know how to define property,

```
const newObject = {};

Object.defineProperty(newObject, "newProperty", {
  value: 100,
  writable: false,
});

console.log(newObject.newProperty); // 100
```



```
newObject.newProperty = 200; // It throws an error in strict mode due to wri
```

[↑ Back to Top](#)

218. What is the difference between get and defineProperty [↗](#)

Both have similar results until unless you use classes. If you use `get` the property will be defined on the prototype of the object whereas using `Object.defineProperty()` the property will be defined on the instance it is applied to.

[↑ Back to Top](#)

219. What are the advantages of Getters and Setters [↗](#)

Below are the list of benefits of Getters and Setters,

- i. They provide simpler syntax
- ii. They are used for defining computed properties, or accessors in JS.
- iii. Useful to provide equivalence relation between properties and methods
- iv. They can provide better data quality
- v. Useful for doing things behind the scenes with the encapsulated logic.

[↑ Back to Top](#)

220. Can I add getters and setters using defineProperty method [↗](#)

Yes, You can use the `Object.defineProperty()` method to add Getters and Setters. For example, the below counter object uses increment, decrement, add and subtract properties,

```
var obj = { counter: 0 };

// Define getters
Object.defineProperty(obj, "increment", {
  get: function () {
    this.counter++;
  },
});
Object.defineProperty(obj, "decrement", {
  get: function () {
    this.counter--;
```



```

    },
  });

  // Define setters
  Object.defineProperty(obj, "add", {
    set: function (value) {
      this.counter += value;
    },
  });
  Object.defineProperty(obj, "subtract", {
    set: function (value) {
      this.counter -= value;
    },
  });

  obj.add = 10;
  obj.subtract = 5;
  console.log(obj.increment); //6
  console.log(obj.decrement); //5

```

[↑ Back to Top](#)

221. What is the purpose of switch-case [↗](#)

The switch case statement in JavaScript is used for decision making purposes. In a few cases, using the switch case statement is going to be more convenient than if-else statements. The syntax would be as below,

```

switch (expression)
{
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    .
    .
    case valueN:
        statementN;
        break;
    default:
        statementDefault;
}

```



The above multi-way branch statement provides an easy way to dispatch execution to different parts of code based on the value of the expression.

[↑ Back to Top](#)

222. What are the conventions to be followed for the usage of switch case [↗](#)

Below are the list of conventions should be taken care,

- i. The expression can be of type either number or string.
- ii. Duplicate values are not allowed for the expression.
- iii. The default statement is optional. If the expression passed to switch does not match with any case value then the statement within default case will be executed.
- iv. The break statement is used inside the switch to terminate a statement sequence.
- v. The break statement is optional. But if it is omitted, the execution will continue on into the next case.

[↑ Back to Top](#)

223. What are primitive data types [↗](#)

A primitive data type is data that has a primitive value (which has no properties or methods). There are 7 types of primitive data types.

- i. string
- ii. number
- iii. boolean
- iv. null
- v. undefined
- vi. bigint
- vii. symbol

[↑ Back to Top](#)

224. What are the different ways to access object properties [↗](#)

There are 3 possible ways for accessing the property of an object.

- i. **Dot notation:** It uses dot for accessing the properties

`objectName.property;`



i. **Square brackets notation:** It uses square brackets for property access

```
objectName["property"];
```



i. **Expression notation:** It uses expression in the square brackets

```
objectName[expression];
```



[↑ Back to Top](#)

225. What are the function parameter rules [↗](#)

JavaScript functions follow below rules for parameters,

- i. The function definitions do not specify data types for parameters.
- ii. Do not perform type checking on the passed arguments.
- iii. Do not check the number of arguments received. i.e, The below function follows the above rules,

```
function functionName(parameter1, parameter2, parameter3) {  
    console.log(parameter1); // 1  
}  
functionName(1);
```



[↑ Back to Top](#)

226. What is an error object [↗](#)

An error object is a built in error object that provides error information when an error occurs. It has two properties: name and message. For example, the below function logs error details,

```
try {  
    greeting("Welcome");  
} catch (err) {  
    console.log(err.name + "<br>" + err.message);  
}
```



[↑ Back to Top](#)

227. When you get a syntax error [↗](#)

A `SyntaxError` is thrown if you try to evaluate code with a syntax error. For example, the below missing quote for the function parameter throws a syntax error

```
try {  
    eval("greeting('welcome)"); // Missing ' will produce an error  
} catch (err) {  
    console.log(err.name);  
}
```



[↑ Back to Top](#)

228. What are the different error names from error object [↗](#)

There are 6 different types of error names returned from error object,

Error Name	Description
EvalError	An error has occurred in the eval() function
RangeError	An error has occurred with a number "out of range"
ReferenceError	An error due to an illegal reference
SyntaxError	An error due to a syntax error
TypeError	An error due to a type error
URIError	An error due to encodeURI()

[↑ Back to Top](#)

229. What are the various statements in error handling [↗](#)

Below are the list of statements used in an error handling,

- i. **try**: This statement is used to test a block of code for errors
- ii. **catch**: This statement is used to handle the error
- iii. **throw**: This statement is used to create custom errors.
- iv. **finally**: This statement is used to execute code after try and catch regardless of the result.

[↑ Back to Top](#)

230. What are the two types of loops in javascript [↗](#)

- i. **Entry Controlled loops:** In this kind of loop type, the test condition is tested before entering the loop body. For example, For Loop and While Loop comes under this category.
- ii. **Exit Controlled Loops:** In this kind of loop type, the test condition is tested or evaluated at the end of the loop body. i.e, the loop body will execute at least once irrespective of test condition true or false. For example, do-while loop comes under this category.

[↑ Back to Top](#)

231. What is nodejs [↗](#)

Node.js is a server-side platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. It is an event-based, non-blocking, asynchronous I/O runtime that uses Google's V8 JavaScript engine and libuv library.

[↑ Back to Top](#)

232. What is an Intl object [↗](#)

The Intl object is the namespace for the ECMAScript Internationalization API, which provides language sensitive string comparison, number formatting, and date and time formatting. It provides access to several constructors and language sensitive functions.

[↑ Back to Top](#)

233. How do you perform language specific date and time formatting [↗](#)

You can use the `Intl.DateTimeFormat` object which is a constructor for objects that enable language-sensitive date and time formatting. Let's see this behavior with an example,

```
var date = new Date(Date.UTC(2019, 07, 07, 3, 0, 0));  
console.log(new Intl.DateTimeFormat("en-GB").format(date)); // 07/08/2019  
console.log(new Intl.DateTimeFormat("en-AU").format(date)); // 07/08/2019
```



[↑ Back to Top](#)

234. What is an Iterator [↗](#)

An iterator is an object which defines a sequence and a return value upon its termination. It implements the Iterator protocol with a `next()` method which returns an object with two properties: `value` (the next value in the sequence) and `done` (which is true if the last value in the sequence has been consumed).

[↑ Back to Top](#)

235. How does synchronous iteration works [↗](#)

Synchronous iteration was introduced in ES6 and it works with below set of components,

Iterable: It is an object which can be iterated over via a method whose key is `Symbol.iterator`. **Iterator:** It is an object returned by invoking `[Symbol.iterator]()` on an iterable. This iterator object wraps each iterated element in an object and returns it via `next()` method one by one. **IteratorResult:** It is an object returned by `next()` method. The object contains two properties; the `value` property contains an iterated element and the `done` property determines whether the element is the last element or not.

Let's demonstrate synchronous iteration with an array as below,

```
const iterable = ["one", "two", "three"];
const iterator = iterable[Symbol.iterator]();
console.log(iterator.next()); // { value: 'one', done: false }
console.log(iterator.next()); // { value: 'two', done: false }
console.log(iterator.next()); // { value: 'three', done: false }
console.log(iterator.next()); // { value: 'undefined', done: true }
```



[↑ Back to Top](#)

236. What is an event loop [↗](#)

The event loop is a process that continuously monitors both the call stack and the event queue and checks whether or not the call stack is empty. If the call stack is empty and there are pending events in the event queue, the event loop dequeues the event from the event queue and pushes it to the call stack. The call stack executes the event, and any additional events generated during the execution are added to the end of the event queue.

Note: The event loop allows Node.js to perform non-blocking I/O operations, even though JavaScript is single-threaded, by offloading operations to the system kernel whenever possible. Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background.

[↑ Back to Top](#)

237. What is call stack [↗](#)

Call Stack is a data structure for javascript interpreters to keep track of function calls(creates execution context) in the program. It has two major actions,

- i. Whenever you call a function for its execution, you are pushing it to the stack.
- ii. Whenever the execution is completed, the function is popped out of the stack.

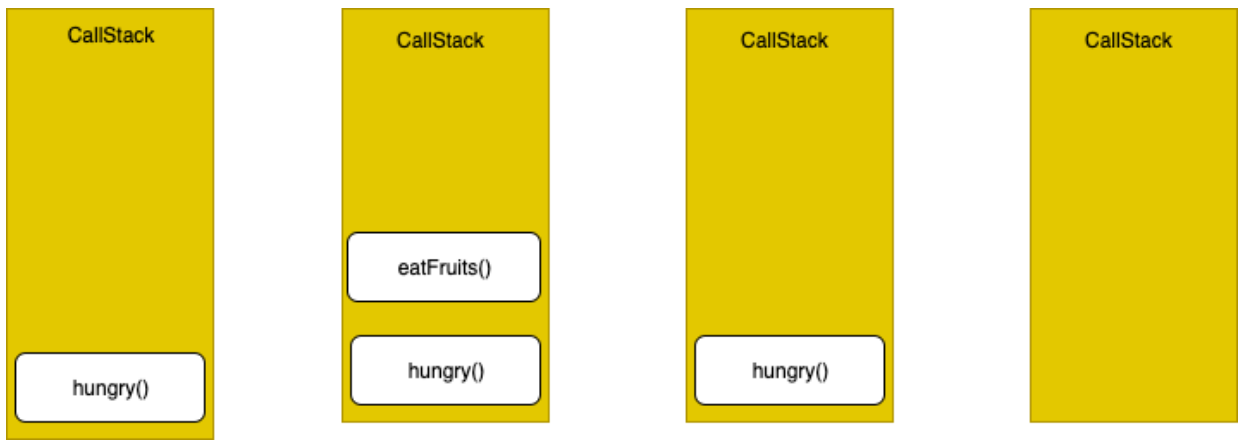
Let's take an example and it's state representation in a diagram format

```
function hungry() {  
  eatFruits();  
}  
function eatFruits() {  
  return "I'm eating fruits";  
}  
  
// Invoke the `hungry` function  
hungry();
```



The above code processed in a call stack as below,

- i. Add the `hungry()` function to the call stack list and execute the code.
- ii. Add the `eatFruits()` function to the call stack list and execute the code.
- iii. Delete the `eatFruits()` function from our call stack list.
- iv. Delete the `hungry()` function from the call stack list since there are no items anymore.



[↑ Back to Top](#)

238. What is an event queue [↗](#)

The event queue follows the queue data structure. It stores async callbacks to be added to the call stack. It is also known as the Callback Queue or Macrotask Queue.

Whenever the call stack receives an async function, it is moved into the Web API. Based on the function, Web API executes it and awaits the result. Once it is finished, it moves the callback into the event queue (the callback of the promise is moved into the microtask queue).

The event loop constantly checks whether or not the call stack is empty. Once the call stack is empty and there is a callback in the event queue, the event loop moves the callback into the call stack. But if there is a callback in the microtask queue as well, it is moved first. The microtask queue has a higher priority than the event queue.

[↑ Back to Top](#)

239. What is a decorator [↗](#)

A decorator is an expression that evaluates to a function and that takes the target, name, and decorator descriptor as arguments. Also, it optionally returns a decorator descriptor to install on the target object. Let's define admin decorator for user class at design time,

```
function admin(isAdmin) {  
  return function(target) {  
    target.isAdmin = isAdmin;  
  }  
}
```



```
@admin(true)
class User() {
}
console.log(User.isAdmin); //true

@admin(false)
class User() {
}
console.log(User.isAdmin); //false
```

[↑ Back to Top](#)

240. What are the properties of Intl object [↗](#)

Below are the list of properties available on Intl object,

- i. **Collator**: These are the objects that enable language-sensitive string comparison.
- ii. **DateTimeFormat**: These are the objects that enable language-sensitive date and time formatting.
- iii. **ListFormat**: These are the objects that enable language-sensitive list formatting.
- iv. **NumberFormat**: Objects that enable language-sensitive number formatting.
- v. **PluralRules**: Objects that enable plural-sensitive formatting and language-specific rules for plurals.
- vi. **RelativeTimeFormat**: Objects that enable language-sensitive relative time formatting.

[↑ Back to Top](#)

241. What is an Unary operator [↗](#)

The unary(+) operator is used to convert a variable to a number. If the variable cannot be converted, it will still become a number but with the value NaN. Let's see this behavior in an action.

```
var x = "100";
var y = +x;
console.log(typeof x, typeof y); // string, number

var a = "Hello";
var b = +a;
console.log(typeof a, typeof b, b); // string, number, NaN
```



[↑ Back to Top](#)

242. How do you sort elements in an array [↗](#)

The `sort()` method is used to sort the elements of an array in place and returns the sorted array. The example usage would be as below,

```
var months = ["Aug", "Sep", "Jan", "June"];  
months.sort();  
console.log(months); // ["Aug", "Jan", "June", "Sep"]
```



[↑ Back to Top](#)

243. What is the purpose of compareFunction while sorting arrays [↗](#)

The `compareFunction` is used to define the sort order. If omitted, the array elements are converted to strings, then sorted according to each character's Unicode code point value. Let's take an example to see the usage of `compareFunction`,

```
let numbers = [1, 2, 5, 3, 4];  
numbers.sort((a, b) => b - a);  
console.log(numbers); // [5, 4, 3, 2, 1]
```



[↑ Back to Top](#)

244. How do you reversing an array [↗](#)

You can use the `reverse()` method to reverse the elements in an array. This method is useful to sort an array in descending order. Let's see the usage of `reverse()` method in an example,

```
let numbers = [1, 2, 5, 3, 4];  
numbers.sort((a, b) => b - a);  
numbers.reverse();  
console.log(numbers); // [1, 2, 3, 4 ,5]
```



[↑ Back to Top](#)

245. How do you find min and max value in an array [↗](#)

You can use `Math.min` and `Math.max` methods on array variables to find the minimum and maximum elements within an array. Let's create two functions to find the min and max value with in an array,

```
var marks = [50, 20, 70, 60, 45, 30];  
function findMin(arr) {  
    return Math.min.apply(null, arr);  
}  
function findMax(arr) {  
    return Math.max.apply(null, arr);  
}  
  
console.log(findMin(marks));  
console.log(findMax(marks));
```



[↑ Back to Top](#)

246. How do you find min and max values without Math functions [↗](#)

You can write functions which loop through an array comparing each value with the lowest value or highest value to find the min and max values. Let's create those functions to find min and max values,

```
var marks = [50, 20, 70, 60, 45, 30];  
function findMin(arr) {  
    var length = arr.length;  
    var min = Infinity;  
    while (length--) {  
        if (arr[length] < min) {  
            min = arr[length];  
        }  
    }  
    return min;  
}  
  
function findMax(arr) {  
    var length = arr.length;  
    var max = -Infinity;  
    while (length--) {  
        if (arr[length] > max) {  
            max = arr[length];  
        }  
    }  
    return max;  
}
```



```
console.log(findMin(marks));  
console.log(findMax(marks));
```

[↑ Back to Top](#)

247. What is an empty statement and purpose of it [↗](#)

The empty statement is a semicolon (;) indicating that no statement will be executed, even if JavaScript syntax requires one. Since there is no action with an empty statement you might think that its usage is quite less, but the empty statement is occasionally useful when you want to create a loop that has an empty body. For example, you can initialize an array with zero values as below,

```
// Initialize an array a  
for (let i = 0; i < a.length; a[i++] = 0);
```



[↑ Back to Top](#)

248. How do you get metadata of a module [↗](#)

You can use the `import.meta` object which is a meta-property exposing context-specific meta data to a JavaScript module. It contains information about the current module, such as the module's URL. In browsers, you might get different meta data than NodeJS.

```
<script type="module" src="welcome-module.js"></script>;  
console.log(import.meta); // { url: "file:///home/user/welcome-module.js" }
```



[↑ Back to Top](#)

249. What is a comma operator [↗](#)

The comma operator is used to evaluate each of its operands from left to right and returns the value of the last operand. This is totally different from comma usage within arrays, objects, and function arguments and parameters. For example, the usage for numeric expressions would be as below,

```
var x = 1;  
x = (x++, x);
```



```
console.log(x); // 2
```

[↑ Back to Top](#)

250. What is the advantage of a comma operator [↗](#)

It is normally used to include multiple expressions in a location that requires a single expression. One of the common usages of this comma operator is to supply multiple parameters in a `for` loop. For example, the below for loop uses multiple expressions in a single location using comma operator,

```
for (var a = 0, b = 10; a <= 10; a++, b--)
```



You can also use the comma operator in a return statement where it processes before returning.

```
function myFunction() {  
  var a = 1;  
  return (a += 10), a; // 11  
}
```



[↑ Back to Top](#)

251. What is typescript [↗](#)

TypeScript is a typed superset of JavaScript created by Microsoft that adds optional types, classes, `async/await`, and many other features, and compiles to plain JavaScript. Angular built entirely in TypeScript and used as a primary language. You can install it globally as

```
npm install -g typescript
```



Let's see a simple example of TypeScript usage,

```
function greeting(name: string): string {  
  return "Hello, " + name;  
}
```



```
let user = "Sudheer";
```

```
console.log(greeting(user));
```

The greeting method allows only string type as argument.

[↑ Back to Top](#)

252. What are the differences between javascript and typescript [↗](#)

Below are the list of differences between javascript and typescript,

feature	typescript	javascript
Language paradigm	Object oriented programming language	Scripting language
Typing support	Supports static typing	It has dynamic typing
Modules	Supported	Not supported
Interface	It has interfaces concept	Doesn't support interfaces
Optional parameters	Functions support optional parameters	No support of optional parameters for functions

[↑ Back to Top](#)

253. What are the advantages of typescript over javascript [↗](#)

Below are some of the advantages of typescript over javascript,

- TypeScript is able to find compile time errors at the development time only and it makes sure less runtime errors. Whereas javascript is an interpreted language.
- TypeScript is strongly-typed or supports static typing which allows for checking type correctness at compile time. This is not available in javascript.
- TypeScript compiler can compile the .ts files into ES3, ES4 and ES5 unlike ES6 features of javascript which may not be supported in some browsers.

[↑ Back to Top](#)

254. What is an object initializer [↗](#)

An object initializer is an expression that describes the initialization of an Object. The syntax for this expression is represented as a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}). This is also known as literal notation. It is one of the ways to create an object.

```
var initObject = { a: "John", b: 50, c: {} };

console.log(initObject.a); // John
```



[↑ Back to Top](#)

255. What is a constructor method [↗](#)

The constructor method is a special method for creating and initializing an object created within a class. If you do not specify a constructor method, a default constructor is used. The example usage of constructor would be as below,

```
class Employee {
  constructor() {
    this.name = "John";
  }
}

var employeeObject = new Employee();

console.log(employeeObject.name); // John
```



[↑ Back to Top](#)

256. What happens if you write constructor more than once in a class [↗](#)

The "constructor" in a class is a special method and it should be defined only once in a class. i.e, If you write a constructor method more than once in a class it will throw a `SyntaxError` error.

```
class Employee {
  constructor() {
    this.name = "John";
  }
  constructor() { // Uncaught SyntaxError: A class may only have one con:
    this.age = 30;
  }
}
```



```
    }  
  }  
  
  var employeeObject = new Employee();  
  
  console.log(employeeObject.name);
```

[↑ Back to Top](#)

257. How do you call the constructor of a parent class [↗](#)

You can use the `super` keyword to call the constructor of a parent class. Remember that `super()` must be called before using 'this' reference. Otherwise it will cause a reference error. Let's see the usage of it,

```
class Square extends Rectangle {  
  constructor(length) {  
    super(length, length);  
    this.name = "Square";  
  }  
  
  get area() {  
    return this.width * this.height;  
  }  
  
  set area(value) {  
    this.area = value;  
  }  
}
```



[↑ Back to Top](#)

258. How do you get the prototype of an object [↗](#)

You can use the `Object.getPrototypeOf(obj)` method to return the prototype of the specified object. i.e. The value of the internal `prototype` property. If there are no inherited properties then `null` value is returned.

```
const newPrototype = {};  
const newObject = Object.create(newPrototype);  
  
console.log(Object.getPrototypeOf(newObject) === newPrototype); // true
```



[↑ Back to Top](#)

259. What happens If I pass string type for getPrototype method [↗](#)

In ES5, it will throw a `TypeError` exception if the `obj` parameter isn't an object. Whereas in ES2015, the parameter will be coerced to an `Object`.

```
// ES5
Object.getPrototypeOf("James"); // TypeError: "James" is not an object
// ES2015
Object.getPrototypeOf("James"); // String.prototype
```



[↑ Back to Top](#)

260. How do you set prototype of one object to another [↗](#)

You can use the `Object.setPrototypeOf()` method that sets the prototype (i.e., the internal `Prototype` property) of a specified object to another object or null. For example, if you want to set prototype of a square object to rectangle object would be as follows,

```
Object.setPrototypeOf(Square.prototype, Rectangle.prototype);
Object.setPrototypeOf({}, null);
```



[↑ Back to Top](#)

261. How do you check whether an object can be extendable or not [↗](#)

The `Object.isExtensible()` method is used to determine if an object is extendable or not. i.e, Whether it can have new properties added to it or not.

```
const newObject = {};
console.log(Object.isExtensible(newObject)); //true
```



Note: By default, all the objects are extendable. i.e, The new properties can be added or modified.

[↑ Back to Top](#)

262. How do you prevent an object to extend [↗](#)

The `Object.preventExtensions()` method is used to prevent new properties from ever being added to an object. In other words, it prevents future extensions to the object. Let's see the usage of this property,

```
const newObject = {};  
Object.preventExtensions(newObject); // NOT extendable  
  
try {  
  Object.defineProperty(newObject, "newProperty", {  
    // Adding new property  
    value: 100,  
  });  
} catch (e) {  
  console.log(e); // TypeError: Cannot define property newProperty, object is not extensible  
}
```

[↑ Back to Top](#)

263. What are the different ways to make an object non-extensible [↗](#)

You can mark an object non-extensible in 3 ways,

- i. `Object.preventExtensions`
- ii. `Object.seal`
- iii. `Object.freeze`

```
var newObject = {};  
  
Object.preventExtensions(newObject); // Prevent objects are non-extensible  
Object.isExtensible(newObject); // false  
  
var sealedObject = Object.seal({}); // Sealed objects are non-extensible  
Object.isExtensible(sealedObject); // false  
  
var frozenObject = Object.freeze({}); // Frozen objects are non-extensible  
Object.isExtensible(frozenObject); // false
```

[↑ Back to Top](#)

264. How do you define multiple properties on an object [↗](#)

The `Object.defineProperties()` method is used to define new or modify existing properties directly on an object and returning the object. Let's define multiple properties on an empty object,

```
const newObject = {};
```



```
Object.defineProperty(newObject, {  
  newProperty1: {  
    value: "John",  
    writable: true,  
  },  
  newProperty2: {},  
});
```

[↑ Back to Top](#)

265. What is MEAN in javascript [↗](#)

The MEAN (MongoDB, Express, AngularJS, and Node.js) stack is the most popular open-source JavaScript software tech stack available for building dynamic web apps where you can write both the server-side and client-side halves of the web project entirely in JavaScript.

[↑ Back to Top](#)

266. What Is Obfuscation in javascript [↗](#)

Obfuscation is the deliberate act of creating obfuscated javascript code(i.e, source or machine code) that is difficult for humans to understand. It is something similar to encryption, but a machine can understand the code and execute it. Let's see the below function before Obfuscation,

```
function greeting() {  
  console.log("Hello, welcome to JS world");  
}
```



And after the code Obfuscation, it would be appeared as below,

```
eval(  
  (function (p, a, c, k, e, d) {  
    e = function (c) {  
      return c;  
    };  
    if (!"".replace(/~/, String)) {  
      while (c--) {  
        d[c] = k[c] || c;  
      }  
      k = [  

```



```

        function (e) {
            return d[e];
        },
    ];
    e = function () {
        return "\\w+";
    };
    c = 1;
}
while (c--) {
    if (k[c]) {
        p = p.replace(new RegExp("\\b" + e(c) + "\\b", "g"), k[c]);
    }
}
return p;
}))(
    "2 1(){0.3('4, 7 6 5 8')}",
    9,
    9,
    "console|greeting|function|log|Hello|JS|to|welcome|world".split("|"),
    0,
    {}
)
);

```

[↑ Back to Top](#)

267. Why do you need Obfuscation [↗](#)

Below are the few reasons for Obfuscation,

- i. The Code size will be reduced. So data transfers between server and client will be fast.
- ii. It hides the business logic from outside world and protects the code from others
- iii. Reverse engineering is highly difficult
- iv. The download time will be reduced

[↑ Back to Top](#)

268. What is Minification [↗](#)

Minification is the process of removing all unnecessary characters(empty spaces are removed) and variables will be renamed without changing it's functionality. It is also a type of obfuscation .

[↑ Back to Top](#)

269. What are the advantages of minification [↗](#)

Normally it is recommended to use minification for heavy traffic and intensive requirements of resources. It reduces file sizes with below benefits,

- i. Decreases loading times of a web page
- ii. Saves bandwidth usages

[↑ Back to Top](#)

270. What are the differences between Obfuscation and Encryption [↗](#)

Below are the main differences between Obfuscation and Encryption,

Feature	Obfuscation	Encryption
Definition	Changing the form of any data in any other form	Changing the form of information to an unreadable format by using a key
A key to decode	It can be decoded without any key	It is required
Target data format	It will be converted to a complex form	Converted into an unreadable format

[↑ Back to Top](#)

271. What are the common tools used for minification [↗](#)

There are many online/offline tools to minify the javascript files,

- i. Google's Closure Compiler
- ii. UglifyJS2
- iii. jsmin
- iv. javascript-minifier.com/
- v. prettydiff.com

[↑ Back to Top](#)

272. How do you perform form validation using javascript [↗](#)

JavaScript can be used to perform HTML form validation. For example, if the form field is empty, the function needs to notify, and return false, to prevent the form being submitted. Lets' perform user login in an html form,

```
<form name="myForm" onsubmit="return validateForm()" method="post">  
  User name: <input type="text" name="uname" />  
  <input type="submit" value="Submit" />  
</form>
```



And the validation on user login is below,

```
function validateForm() {  
  var x = document.forms["myForm"]["uname"].value;  
  if (x == "") {  
    alert("The username shouldn't be empty");  
    return false;  
  }  
}
```



[↑ Back to Top](#)

273. How do you perform form validation without javascript [↗](#)

You can perform HTML form validation automatically without using javascript. The validation enabled by applying the `required` attribute to prevent form submission when the input is empty.

```
<form method="post">  
  <input type="text" name="uname" required />  
  <input type="submit" value="Submit" />  
</form>
```



Note: Automatic form validation does not work in Internet Explorer 9 or earlier.

[↑ Back to Top](#)

274. What are the DOM methods available for constraint validation [↗](#)



The below DOM methods are available for constraint validation on an invalid input,

- i. `checkValidity()`: It returns true if an input element contains valid data.

- ii. `setCustomValidity()`: It is used to set the `validationMessage` property of an input element. Let's take an user login form with DOM validations

```
function myFunction() {  
  var userName = document.getElementById("uname");  
  if (!userName.checkValidity()) {  
    document.getElementById("message").innerHTML =  
      userName.validationMessage;  
  } else {  
    document.getElementById("message").innerHTML =  
      "Entered a valid username";  
  }  
}
```



[↑ Back to Top](#)

275. What are the available constraint validation DOM properties [↗](#)

Below are the list of some of the constraint validation DOM properties available,

- i. `validity`: It provides a list of boolean properties related to the validity of an input element.
- ii. `validationMessage`: It displays the message when the validity is false.
- iii. `willValidate`: It indicates if an input element will be validated or not.

[↑ Back to Top](#)

276. What are the list of validity properties [↗](#)

The `validity` property of an input element provides a set of properties related to the validity of data.

- i. `customError`: It returns true, if a custom validity message is set.
- ii. `patternMismatch`: It returns true, if an element's value does not match its `pattern` attribute.
- iii. `rangeOverflow`: It returns true, if an element's value is greater than its `max` attribute.
- iv. `rangeUnderflow`: It returns true, if an element's value is less than its `min` attribute.
- v. `stepMismatch`: It returns true, if an element's value is invalid according to `step` attribute.
- vi. `tooLong`: It returns true, if an element's value exceeds its `maxLength` attribute.

- vii. typeMismatch: It returns true, if an element's value is invalid according to type attribute.
- viii. valueMissing: It returns true, if an element with a required attribute has no value.
- ix. valid: It returns true, if an element's value is valid.

[↑ Back to Top](#)

277. Give an example usage of rangeOverflow property [↗](#)

If an element's value is greater than its max attribute then rangeOverflow property returns true. For example, the below form submission throws an error if the value is more than 100,

```
<input id="age" type="number" max="100" />  
<button onclick="myOverflowFunction()">OK</button>
```



```
function myOverflowFunction() {  
  if (document.getElementById("age").validity.rangeOverflow) {  
    alert("The mentioned age is not allowed");  
  }  
}
```



[↑ Back to Top](#)

278. Is enums feature available in javascript [↗](#)

No, javascript does not natively support enums. But there are different kinds of solutions to simulate them even though they may not provide exact equivalents. For example, you can use freeze or seal on object,

```
var DaysEnum = Object.freeze({"monday":1, "tuesday":2, "wednesday":3, .
```



[↑ Back to Top](#)

279. What is an enum [↗](#)

An enum is a type restricting variables to one value from a predefined set of constants. JavaScript has no enums but typescript provides built-in enum support.

```
enum Color {  
  RED, GREEN, BLUE  
}
```



[↑ Back to Top](#)

280. How do you list all properties of an object [↗](#)

You can use the `Object.getOwnPropertyNames()` method which returns an array of all properties found directly in a given object. Let's see the usage of it in an example,

```
const newObject = {  
  a: 1,  
  b: 2,  
  c: 3,  
};  
  
console.log(Object.getOwnPropertyNames(newObject));  
["a", "b", "c"];
```



[↑ Back to Top](#)

281. How do you get property descriptors of an object [↗](#)

You can use the `Object.getOwnPropertyDescriptors()` method which returns all own property descriptors of a given object. The example usage of this method is below,

```
const newObject = {  
  a: 1,  
  b: 2,  
  c: 3,  
};  
  
const descriptorsObject = Object.getOwnPropertyDescriptors(newObject);  
console.log(descriptorsObject.a.writable); //true  
console.log(descriptorsObject.a.configurable); //true  
console.log(descriptorsObject.a.enumerable); //true  
console.log(descriptorsObject.a.value); // 1
```



[↑ Back to Top](#)

282. What are the attributes provided by a property descriptor [↗](#)

A property descriptor is a record which has the following attributes

- i. value: The value associated with the property
- ii. writable: Determines whether the value associated with the property can be changed or not
- iii. configurable: Returns true if the type of this property descriptor can be changed and if the property can be deleted from the corresponding object.
- iv. enumerable: Determines whether the property appears during enumeration of the properties on the corresponding object or not.
- v. set: A function which serves as a setter for the property
- vi. get: A function which serves as a getter for the property

[↑ Back to Top](#)

283. How do you extend classes [↗](#)

The `extends` keyword is used in class declarations/expressions to create a class which is a child of another class. It can be used to subclass custom classes as well as built-in objects. The syntax would be as below,

```
class ChildClass extends ParentClass { ... }
```



Let's take an example of Square subclass from Polygon parent class,

```
class Square extends Rectangle {  
  constructor(length) {  
    super(length, length);  
    this.name = "Square";  
  }  
  
  get area() {  
    return this.width * this.height;  
  }  
  
  set area(value) {  
    this.area = value;  
  }  
}
```



[↑ Back to Top](#)

284. How do I modify the url without reloading the page [↗](#)

The `window.location.href` property will be helpful to modify the url but it reloads the page. HTML5 introduced the `history.pushState()` and `history.replaceState()` methods, which allow you to add and modify history entries, respectively. For example, you can use `pushState` as below,

```
window.history.pushState("page2", "Title", "/page2.html");
```



[↑ Back to Top](#)

285. How do you check whether an array includes a particular value or not [↗](#)

The `Array#includes()` method is used to determine whether an array includes a particular value among its entries by returning either `true` or `false`. Let's see an example to find an element(numeric and string) within an array.

```
var numericArray = [1, 2, 3, 4];
console.log(numericArray.includes(3)); // true

var stringArray = ["green", "yellow", "blue"];
console.log(stringArray.includes("blue")); //true
```



[↑ Back to Top](#)

286. How do you compare scalar arrays [↗](#)

You can use `length` and `every` method of arrays to compare two scalar(compared directly using `===`) arrays. The combination of these expressions can give the expected result,

```
const arrayFirst = [1, 2, 3, 4, 5];
const arraySecond = [1, 2, 3, 4, 5];
console.log(
  arrayFirst.length === arraySecond.length &&
  arrayFirst.every((value, index) => value === arraySecond[index])
); // true
```



If you would like to compare arrays irrespective of order then you should sort them before,

```
const arrayFirst = [2, 3, 1, 4, 5];
const arraySecond = [1, 2, 3, 4, 5];
```




```
console.log(
  arrayFirst.length === arraySecond.length &&
  arrayFirst.sort().every((value, index) => value === arraySecond[index])
); //true
```

[↑ Back to Top](#)

287. How to get the value from get parameters [↗](#)

The `new URL()` object accepts the url string and `searchParams` property of this object can be used to access the get parameters. Remember that you may need to use `polyfill` or `window.location` to access the URL in older browsers(including IE).

```
let urlString = "http://www.some-domain.com/about.html?x=1&y=2&z=3"; //  or
let url = new URL(urlString);
let parameterZ = url.searchParams.get("z");
console.log(parameterZ); // 3
```

[↑ Back to Top](#)

288. How do you print numbers with commas as thousand separators [↗](#)

You can use the `Number.prototype.toLocaleString()` method which returns a string with a language-sensitive representation such as thousand separator, currency etc of this number.

```
function convertToThousandFormat(x) {
  return x.toLocaleString(); // 12,345.679
}

console.log(convertToThousandFormat(12345.6789));
```

[↑ Back to Top](#)

289. What is the difference between java and javascript [↗](#)

Both are totally unrelated programming languages and no relation between them. Java is statically typed, compiled, runs on its own VM. Whereas Javascript is dynamically typed, interpreted, and runs in a browser and nodejs environments. Let's see the major differences in a tabular format,

Feature	Java	JavaScript
Typed	It's a strongly typed language	It's a dynamic typed language
Paradigm	Object oriented programming	Prototype based programming
Scoping	Block scoped	Function-scoped
Concurrency	Thread based	event based
Memory	Uses more memory	Uses less memory. Hence it will be used for web pages

[↑ Back to Top](#)

290. Does JavaScript supports namespace [↗](#)

JavaScript doesn't support namespace by default. So if you create any element(function, method, object, variable) then it becomes global and pollutes the global namespace. Let's take an example of defining two functions without any namespace,

```
function func1() {  
    console.log("This is a first definition");  
}  
function func1() {  
    console.log("This is a second definition");  
}  
func1(); // This is a second definition
```




It always calls the second function definition. In this case, namespace will solve the name collision problem.

[↑ Back to Top](#)

291. How do you declare namespace [↗](#)


Even though JavaScript lacks namespaces, we can use Objects , IIFE to create namespaces.

- i. **Using Object Literal Notation:** Let's wrap variables and functions inside an Object literal which acts as a namespace. After that you can access them using object notation



```
var namespaceOne = {
  function func1() {
    console.log("This is a first definition");
  }
}
var namespaceTwo = {
  function func1() {
    console.log("This is a second definition");
  }
}
namespaceOne.func1(); // This is a first definition
namespaceTwo.func1(); // This is a second definition
```


- i. **Using IIFE (Immediately invoked function expression):** The outer pair of parentheses of IIFE creates a local scope for all the code inside of it and makes the anonymous function a function expression. Due to that, you can create the same function in two different function expressions to act as a namespace.



```
(function () {
  function fun1() {
    console.log("This is a first definition");
  }
  fun1();
})();

(function () {
  function fun1() {
    console.log("This is a second definition");
  }
  fun1();
})();
```

- i. **Using a block and a let/const declaration:** In ECMAScript 6, you can simply use a block and a let declaration to restrict the scope of a variable to a block.



```
{
  let myFunction = function fun1() {
    console.log("This is a first definition");
  };
  myFunction();
}
//myFunction(): ReferenceError: myFunction is not defined.

{
  let myFunction = function fun1() {
    console.log("This is a second definition");
  };
}
```

```
    myFunction();  
}  
//myFunction(): ReferenceError: myFunction is not defined.
```

[↑ Back to Top](#)

292. How do you invoke javascript code in an iframe from parent page [↗](#)

Initially iFrame needs to be accessed using either `document.getElementById` or `window.frames`. After that `contentWindow` property of iFrame gives the access for `targetFunction`

```
document.getElementById("targetFrame").contentWindow.targetFunction();  
window.frames[0].frameElement.contentWindow.targetFunction(); // Accessing i
```

[↑ Back to Top](#)

293. How do get the timezone offset from date [↗](#)

You can use the `getTimezoneOffset` method of the date object. This method returns the time zone difference, in minutes, from current locale (host system settings) to UTC

```
var offset = new Date().getTimezoneOffset();  
console.log(offset); // -480
```

[↑ Back to Top](#)

294. How do you load CSS and JS files dynamically [↗](#)

You can create both link and script elements in the DOM and append them as child to head tag. Let's create a function to add script and style resources as below,

```
function loadAssets(filename, filetype) {  
    if (filetype == "css") {  
        // External CSS file  
        var fileReference = document.createElement("link");  
        fileReference.setAttribute("rel", "stylesheet");  
        fileReference.setAttribute("type", "text/css");  
        fileReference.setAttribute("href", filename);  
    } else if (filetype == "js") {
```

```
// External JavaScript file
var fileReference = document.createElement("script");
fileReference.setAttribute("type", "text/javascript");
fileReference.setAttribute("src", filename);
}
if (typeof fileReference != "undefined")
    document.getElementsByTagName("head")[0].appendChild(fileReference);
}
```

[↑ Back to Top](#)

295. What are the different methods to find HTML elements in DOM



If you want to access any element in an HTML page, you need to start with accessing the document object. Later you can use any of the below methods to find the HTML element,

- i. document.getElementById(id): It finds an element by Id
- ii. document.getElementsByTagName(name): It finds an element by tag name
- iii. document.getElementsByClassName(name): It finds an element by class name

[↑ Back to Top](#)

296. What is jQuery

jQuery is a popular cross-browser JavaScript library that provides Document Object Model (DOM) traversal, event handling, animations and AJAX interactions by minimizing the discrepancies across browsers. It is widely famous with its philosophy of "Write less, do more". For example, you can display welcome message on the page load using jQuery as below,

```
$(document).ready(function () {
    // It selects the document and apply the function on page load
    alert("Welcome to jQuery world");
});
```



Note: You can download it from jquery's official site or install it from CDNs, like google.

[↑ Back to Top](#)

297. What is V8 JavaScript engine

V8 is an open source high-performance JavaScript engine used by the Google Chrome browser, written in C++. It is also being used in the node.js project. It implements ECMAScript and WebAssembly, and runs on Windows 7 or later, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors. **Note:** It can run standalone, or can be embedded into any C++ application.

[↑ Back to Top](#)

298. Why do we call javascript as dynamic language [↗](#)

JavaScript is a loosely typed or a dynamic language because variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned/reassigned with values of all types.

```
let age = 50; // age is a number now
age = "old"; // age is a string now
age = true; // age is a boolean
```



[↑ Back to Top](#)

299. What is a void operator [↗](#)

The `void` operator evaluates the given expression and then returns undefined(i.e, without returning value). The syntax would be as below,

```
void expression;
void expression;
```



Let's display a message without any redirection or reload

```
<a href="javascript:void(alert('Welcome to JS world'))">
  Click here to see a message
</a>
```



Note: This operator is often used to obtain the undefined primitive value, using "void(0)".

[↑ Back to Top](#)

300. How to set the cursor to wait [↗](#)

The cursor can be set to wait in JavaScript by using the property "cursor". Let's perform this behavior on page load using the below function.

```
function myFunction() {  
    window.document.body.style.cursor = "wait";  
}
```



and this function invoked on page load

```
<body onload="myFunction()"></body>
```



[↑ Back to Top](#)

301. How do you create an infinite loop [↗](#)

You can create infinite loops using for and while loops without using any expressions. The for loop construct or syntax is better approach in terms of ESLint and code optimizer tools,

```
for (;;) {}  
while (true) {}
```



[↑ Back to Top](#)

302. Why do you need to avoid with statement [↗](#)

JavaScript's with statement was intended to provide a shorthand for writing recurring accesses to objects. So it can help reduce file size by reducing the need to repeat a lengthy object reference without performance penalty. Let's take an example where it is used to avoid redundancy when accessing an object several times.

```
a.b.c.greeting = "welcome";  
a.b.c.age = 32;
```



Using with it turns this into:

```
with (a.b.c) {  
    greeting = "welcome";  
}
```



```
    age = 32;
}
```

But this `with` statement creates performance problems since one cannot predict whether an argument will refer to a real variable or to a property inside the `with` argument.

[↑ Back to Top](#)

303. What is the output of below for loops [↗](#)

```
for (var i = 0; i < 4; i++) {
  // global scope
  setTimeout(() => console.log(i));
}

for (let i = 0; i < 4; i++) {
  // block scope
  setTimeout(() => console.log(i));
}
```



The output of the above for loops is 4 4 4 4 and 0 1 2 3

Explanation: Due to the event queue/loop of javascript, the `setTimeout` callback function is called after the loop has been executed. Since the variable `i` is declared with the `var` keyword it became a global variable and the value was equal to 4 using iteration when the time `setTimeout` function is invoked. Hence, the output of the first loop is 4 4 4 4 .

Whereas in the second loop, the variable `i` is declared as the `let` keyword it becomes a block scoped variable and it holds a new value(0, 1 ,2 3) for each iteration. Hence, the output of the first loop is 0 1 2 3 .

[↑ Back to Top](#)

304. List down some of the features of ES6 [↗](#)

Below are the list of some new features of ES6,

- i. Support for constants or immutable variables
- ii. Block-scope support for variables, constants and functions
- iii. Arrow functions
- iv. Default parameters

- v. Rest and Spread Parameters
- vi. Template Literals
- vii. Multi-line Strings
- viii. Destructuring Assignment
- ix. Enhanced Object Literals
- x. Promises
- xi. Classes
- xii. Modules

[↑ Back to Top](#)

305. What is ES6 [↗](#)

ES6 is the sixth edition of the javascript language and it was released in June 2015. It was initially known as ECMAScript 6 (ES6) and later renamed to ECMAScript 2015. Almost all the modern browsers support ES6 but for the old browsers there are many transpilers, like Babel.js etc.

[↑ Back to Top](#)


306. Can I redeclare let and const variables [↗](#)

No, you cannot redeclare let and const variables. If you do, it throws below error

Uncaught SyntaxError: Identifier 'someVariable' has already been declared 

Explanation: The variable declaration with `var` keyword refers to a function scope and the variable is treated as if it were declared at the top of the enclosing scope due to hoisting feature. So all the multiple declarations contributing to the same hoisted variable without any error. Let's take an example of re-declaring variables in the same scope for both `var` and `let/const` variables.

```
var name = "John";  
function myFunc() {  
    var name = "Nick";  
    var name = "Abraham"; // Re-assigned in the same function block  
    alert(name); // Abraham  
}  
myFunc();  
alert(name); // John
```



The block-scoped multi-declaration throws syntax error,

```
let name = "John";
function myFunc() {
  let name = "Nick";
  let name = "Abraham"; // Uncaught SyntaxError: Identifier 'name' has already been declared
  alert(name);
}

myFunc();
alert(name);
```

[↑ Back to Top](#)

307. Is const variable makes the value immutable [↗](#)

No, the const variable doesn't make the value immutable. But it disallows subsequent assignments(i.e, You can declare with assignment but can't assign another value later)

```
const userList = [];
userList.push("John"); // Can mutate even though it can't re-assign
console.log(userList); // ['John']
```

[↑ Back to Top](#)

308. What are default parameters [↗](#)

In ES5, we need to depend on logical OR operators to handle default values of function parameters. Whereas in ES6, Default function parameters feature allows parameters to be initialized with default values if no value or undefined is passed. Let's compare the behavior with an examples,

```
//ES5
var calculateArea = function (height, width) {
  height = height || 50;
  width = width || 60;

  return width * height;
};
console.log(calculateArea()); //300
```

The default parameters makes the initialization more simpler,

```
//ES6
var calculateArea = function (height = 50, width = 60) {
  return width * height;
};

console.log(calculateArea()); //300
```



[↑ Back to Top](#)

309. What are template literals [↗](#)

Template literals or template strings are string literals allowing embedded expressions. These are enclosed by the back-tick (`) character instead of double or single quotes. In ES6, this feature enables using dynamic expressions as below,

```
var greeting = `Welcome to JS World, Mr. ${firstName} ${lastName}.`;
```



In ES5, you need break string like below,

```
var greeting = 'Welcome to JS World, Mr. ' + firstName + ' ' + lastName
```



Note: You can use multi-line strings and string interpolation features with template literals.

[↑ Back to Top](#)

310. How do you write multi-line strings in template literals [↗](#)

In ES5, you would have to use newline escape characters(`\n`) and concatenation symbols(+) in order to get multi-line strings.

```
console.log("This is string sentence 1\n" + "This is string sentence 2")
```



Whereas in ES6, You don't need to mention any newline sequence character,

```
console.log(`This is string sentence
'This is string sentence 2`);
```



[↑ Back to Top](#)

311. What are nesting templates [↗](#)

The nesting template is a feature supported within template literals syntax to allow inner backticks inside a placeholder `${ }` within the template. For example, the below nesting template is used to display the icons based on user permissions whereas outer template checks for platform type,

```
const iconStyles = `icon ${
  isMobilePlatform()
    ? ""
    : `icon-${user.isAuthorized ? "submit" : "disabled"}`
}`;
```



You can write the above use case without nesting template features as well. However, the nesting template feature is more compact and readable.

```
//Without nesting templates
const iconStyles = `icon ${
  isMobilePlatform()
    ? ""
    : user.isAuthorized
      ? "icon-submit"
      : "icon-disabled"
}`;
```



[↑ Back to Top](#)

312. What are tagged templates [↗](#)

Tagged templates are the advanced form of templates in which tags allow you to parse template literals with a function. The tag function accepts the first parameter as an array of strings and remaining parameters as expressions. This function can also return manipulated strings based on parameters. Let's see the usage of this tagged template behavior of an IT professional skill set in an organization,

```
var user1 = "John";
var skill1 = "JavaScript";
var experience1 = 15;

var user2 = "Kane";
var skill2 = "JavaScript";
var experience2 = 5;
```



```
function myInfoTag(strings, userExp, experienceExp, skillExp) {
  var str0 = strings[0]; // "Mr/Ms. "
  var str1 = strings[1]; // " is a/an "
  var str2 = strings[2]; // "in"

  var expertiseStr;
  if (experienceExp > 10) {
    expertiseStr = "expert developer";
  } else if (skillExp > 5 && skillExp <= 10) {
    expertiseStr = "senior developer";
  } else {
    expertiseStr = "junior developer";
  }

  return `${str0}${userExp}${str1}${expertiseStr}${str2}${skillExp}`;
}

var output1 = myInfoTag`Mr/Ms. ${user1} is a/an ${experience1} in ${skill1}`
var output2 = myInfoTag`Mr/Ms. ${user2} is a/an ${experience2} in ${skill2}`

console.log(output1); // Mr/Ms. John is a/an expert developer in JavaScript
console.log(output2); // Mr/Ms. Kane is a/an junior developer in JavaScript
```

[↑ Back to Top](#)

313. What are raw strings [↗](#)

ES6 provides a raw strings feature using the `String.raw()` method which is used to get the raw string form of template strings. This feature allows you to access the raw strings as they were entered, without processing escape sequences. For example, the usage would be as below,

```
var calculationString = String.raw`The sum of numbers is \n${
  1 + 2 + 3 + 4
}!`;
console.log(calculationString); // The sum of numbers is \n10!
```



If you don't use raw strings, the newline character sequence will be processed by displaying the output in multiple lines

```
var calculationString = `The sum of numbers is \n${1 + 2 + 3 + 4}!`;
console.log(calculationString);
// The sum of numbers is
// 10!
```



Also, the raw property is available on the first argument to the tag function

```
function tag(strings) {  
  console.log(strings.raw[0]);  
}
```



[↑ Back to Top](#)

314. What is destructuring assignment [↗](#)

The destructuring assignment is a JavaScript expression that makes it possible to unpack values from arrays or properties from objects into distinct variables. Let's get the month values from an array using destructuring assignment

```
var [one, two, three] = ["JAN", "FEB", "MARCH"];  
  
console.log(one); // "JAN"  
console.log(two); // "FEB"  
console.log(three); // "MARCH"
```



and you can get user properties of an object using destructuring assignment,

```
var { name, age } = { name: "John", age: 32 };  
  
console.log(name); // John  
console.log(age); // 32
```



[↑ Back to Top](#)

315. What are default values in destructuring assignment [↗](#)

A variable can be assigned a default value when the value unpacked from the array or object is undefined during destructuring assignment. It helps to avoid setting default values separately for each assignment. Let's take an example for both arrays and object use cases,

Arrays destructuring:

```
var x, y, z;  
  
[x = 2, y = 4, z = 6] = [10];  
console.log(x); // 10
```




```
console.log(y); // 4
console.log(z); // 6
```

Objects destructuring:

```
var { x = 2, y = 4, z = 6 } = { x: 10 };
```



```
console.log(x); // 10
console.log(y); // 4
console.log(z); // 6
```

[↑ Back to Top](#)

316. How do you swap variables in destructuring assignment [↗](#)

If you don't use destructuring assignment, swapping two values requires a temporary variable. Whereas using a destructuring feature, two variable values can be swapped in one destructuring expression. Let's swap two number variables in array destructuring assignment,

```
var x = 10,
    y = 20;

[x, y] = [y, x];
console.log(x); // 20
console.log(y); // 10
```



[↑ Back to Top](#)

317. What are enhanced object literals [↗](#)

Object literals make it easy to quickly create objects with properties inside the curly braces. For example, it provides shorter syntax for common object property definition as below.

```
//ES6
var x = 10,
    y = 20;
obj = { x, y };
console.log(obj); // {x: 10, y:20}
//ES5
var x = 10,
    y = 20;
```



```
obj = { x: x, y: y };  
console.log(obj); // {x: 10, y:20}
```

[↑ Back to Top](#)

318. What are dynamic imports [↗](#)

The dynamic imports using `import()` function syntax allows us to load modules on demand by using promises or the `async/await` syntax. Currently this feature is in [stage4 proposal](#). The main advantage of dynamic imports is reduction of our bundle's sizes, the size/payload response of our requests and overall improvements in the user experience. The syntax of dynamic imports would be as below,

```
import("./Module").then((Module) => Module.method());
```



[↑ Back to Top](#)

319. What are the use cases for dynamic imports [↗](#)

Below are some of the use cases of using dynamic imports over static imports,

- i. Import a module on-demand or conditionally. For example, if you want to load a polyfill on legacy browser

```
if (isLegacyBrowser()) {  
  import(...)  
  .then(...);  
}
```



- i. Compute the module specifier at runtime. For example, you can use it for internationalization.

```
import(`messages_${getLocale()}.js`).then(...);
```



- i. Import a module from within a regular script instead a module.

[↑ Back to Top](#)

320. What are typed arrays [↗](#)

Typed arrays are array-like objects from ECMAScript 6 API for handling binary data. JavaScript provides 8 Typed array types,

- i. Int8Array: An array of 8-bit signed integers
- ii. Int16Array: An array of 16-bit signed integers
- iii. Int32Array: An array of 32-bit signed integers
- iv. Uint8Array: An array of 8-bit unsigned integers
- v. Uint16Array: An array of 16-bit unsigned integers
- vi. Uint32Array: An array of 32-bit unsigned integers
- vii. Float32Array: An array of 32-bit floating point numbers
- viii. Float64Array: An array of 64-bit floating point numbers

For example, you can create an array of 8-bit signed integers as below

```
const a = new Int8Array();  
// You can pre-allocate n bytes  
const bytes = 1024;  
const a = new Int8Array(bytes);
```



[↑ Back to Top](#)

321. What are the advantages of module loaders [↗](#)

The module loaders provides the below features,

- i. Dynamic loading
- ii. State isolation
- iii. Global namespace isolation
- iv. Compilation hooks
- v. Nested virtualization

[↑ Back to Top](#)

322. What is collation [↗](#)

Collation is used for sorting a set of strings and searching within a set of strings. It is parameterized by locale and aware of Unicode. Let's take comparison and sorting features,

i. Comparison:

```
var list = ["ä", "a", "z"]; // In German, "ä" sorts with "a" Whereas in English it doesn't  
var l10nDE = new Intl.Collator("de");  
var l10nSV = new Intl.Collator("sv");
```



```
console.log(l10nDE.compare("ä", "z") === -1); // true
console.log(l10nSV.compare("ä", "z") === +1); // true
```

i. Sorting:

```
var list = ["ä", "a", "z"]; // In German, "ä" sorts with "a" Whereas in Swedish it doesn't
var l10nDE = new Intl.Collator("de");
var l10nSV = new Intl.Collator("sv");
console.log(list.sort(l10nDE.compare)); // [ "a", "ä", "z" ]
console.log(list.sort(l10nSV.compare)); // [ "a", "z", "ä" ]
```

[↑ Back to Top](#)

323. What is for...of statement [↗](#)

The for...of statement creates a loop iterating over iterable objects or elements such as built-in String, Array, Array-like objects (like arguments or NodeList), TypedArray, Map, Set, and user-defined iterables. The basic usage of for...of statement on arrays would be as below,

```
let arrayIterable = [10, 20, 30, 40, 50];

for (let value of arrayIterable) {
  value++;
  console.log(value); // 11 21 31 41 51
}
```

[↑ Back to Top](#)

324. What is the output of below spread operator array [↗](#)

```
[..."John Resig"];
```

The output of the array is ['J', 'o', 'h', 'n', ' ', 'R', 'e', 's', 'i', 'g'] **Explanation:** The string is an iterable type and the spread operator within an array maps every character of an iterable to one element. Hence, each character of a string becomes an element within an Array.

[↑ Back to Top](#)

325. Is PostMessage secure [↗](#)

Yes, `postMessages` can be considered very secure as long as the programmer/developer is careful about checking the origin and source of an arriving message. But if you try to send/receive a message without verifying its source will create cross-site scripting attacks.

[↑ Back to Top](#)

326. What are the problems with `postmessage` target origin as wildcard [↗](#)

The second argument of `postMessage` method specifies which origin is allowed to receive the message. If you use the wildcard `"*"` as an argument then any origin is allowed to receive the message. In this case, there is no way for the sender window to know if the target window is at the target origin when sending the message. If the target window has been navigated to another origin, the other origin would receive the data. Hence, this may lead to XSS vulnerabilities.

```
targetWindow.postMessage(message, "*");
```



[↑ Back to Top](#)

327. How do you avoid receiving `postMessages` from attackers [↗](#)

Since the listener listens for any message, an attacker can trick the application by sending a message from the attacker's origin, which gives an impression that the receiver received the message from the actual sender's window. You can avoid this issue by validating the origin of the message on the receiver's end using the `"message.origin"` attribute. For examples, let's check the sender's origin <http://www.some-sender.com> on receiver side www.some-receiver.com,

```
//Listener on http://www.some-receiver.com/  
window.addEventListener("message", function(message){  
    if(/^http://www\.some-sender\.com$/i.test(message.origin)){  
        console.log('You received the data from valid sender', message.data)  
    }  
});
```



[↑ Back to Top](#)

328. Can I avoid using `postMessages` completely [↗](#)

You cannot avoid using `postMessages` completely(or 100%). Even though your application doesn't use `postMessage` considering the risks, a lot of third party scripts use `postMessage` to communicate with the third party service. So your application might be using `postMessage` without your knowledge.

[↑ Back to Top](#)

329. Is `postMessages` synchronous [↗](#)

The `postMessages` are synchronous in IE8 browser but they are asynchronous in IE9 and all other modern browsers (i.e, IE9+, Firefox, Chrome, Safari).Due to this asynchronous behaviour, we use a callback mechanism when the `postMessage` is returned.

[↑ Back to Top](#)

330. What paradigm is Javascript [↗](#)

JavaScript is a multi-paradigm language, supporting imperative/procedural programming, Object-Oriented Programming and functional programming. JavaScript supports Object-Oriented Programming with prototypical inheritance.

[↑ Back to Top](#)

331. What is the difference between internal and external javascript [↗](#)

Internal JavaScript: It is the source code within the script tag. **External JavaScript:** The source code is stored in an external file(stored with .js extension) and referred with in the tag.

[↑ Back to Top](#)

332. Is JavaScript faster than server side script [↗](#)

Yes, JavaScript is faster than server side scripts. Because JavaScript is a client-side script it does not require any web server's help for its computation or calculation. So JavaScript is always faster than any server-side script like ASP, PHP, etc.

[↑ Back to Top](#)

333. How do you get the status of a checkbox [↗](#)

You can apply the `checked` property on the selected checkbox in the DOM. If the value is `true` it means the checkbox is checked, otherwise it is unchecked. For example, the below HTML checkbox element can be access using javascript as below:

```
<input type="checkbox" id="checkboxname" value="Agree" /> Agree the conditions<br />
```



```
console.log(document.getElementById('checkboxname').checked); // true c 1:
```



[↑ Back to Top](#)

334. What is the purpose of double tilde operator [↗](#)

The double tilde operator(`~~`) is known as double NOT bitwise operator. This operator is a slightly quicker substitute for `Math.floor()`.

[↑ Back to Top](#)

335. How do you convert character to ASCII code [↗](#)

You can use the `String.prototype.charCodeAt()` method to convert string characters to ASCII numbers. For example, let's find ASCII code for the first letter of 'ABC' string,

```
"ABC".charCodeAt(0); // returns 65
```



Whereas `String.fromCharCode()` method converts numbers to equal ASCII characters.

```
String.fromCharCode(65, 66, 67); // returns 'ABC'
```



[↑ Back to Top](#)

336. What is ArrayBuffer [↗](#)

An `ArrayBuffer` object is used to represent a generic, fixed-length raw binary data buffer. You can create it as below,

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16
alert(buffer.byteLength); // 16
```



To manipulate an ArrayBuffer, we need to use a "view" object.

```
//Create a DataView referring to the buffer
let view = new DataView(buffer);
```



[↑ Back to Top](#)

337. What is the output of below string expression [↗](#)

```
console.log("Welcome to JS world"[0]);
```



The output of the above expression is "W". **Explanation:** The bracket notation with specific index on a string returns the character at a specific location. Hence, it returns the character "W" of the string. Since this is not supported in IE7 and below versions, you may need to use the `.charAt()` method to get the desired result.

[↑ Back to Top](#)

338. What is the purpose of Error object [↗](#)

The Error constructor creates an error object and the instances of error objects are thrown when runtime errors occur. The Error object can also be used as a base object for user-defined exceptions. The syntax of error object would be as below,

```
new Error([message[, fileName[, lineNumber]]])
```



You can throw user defined exceptions or errors using Error object in try...catch block as below,

```
try {
  if (withdraw > balance)
    throw new Error("Oops! You don't have enough balance");
} catch (e) {
  console.log(e.name + ": " + e.message);
}
```



[↑ Back to Top](#)

339. What is the purpose of EvalError object

The EvalError object indicates an error regarding the global `eval()` function. Even though this exception is not thrown by JavaScript anymore, the EvalError object remains for compatibility. The syntax of this expression would be as below,

```
new EvalError([message[, fileName[, lineNumber]]])
```



You can throw EvalError with in try...catch block as below,

```
try {  
    throw new EvalError('Eval function error', 'someFile.js', 100);  
} catch (e) {  
    console.log(e.message, e.name, e.fileName);           // "Eval function
```



 [Back to Top](#)

340. What are the list of cases error thrown from non-strict mode to strict mode

When you apply 'use strict'; syntax, some of the below cases will throw a SyntaxError before executing the script

i. When you use Octal syntax

```
var n = 022;
```



i. Using with statement

ii. When you use delete operator on a variable name

iii. Using eval or arguments as variable or function argument name

iv. When you use newly reserved keywords

v. When you declare a function in a block

```
if (someCondition) {  
    function f() {}  
}
```



Hence, the errors from above cases are helpful to avoid errors in development/production environments.

[↑ Back to Top](#)

341. Do all objects have prototypes [↗](#)

No. All objects have prototypes except for the base object which is created by the user, or an object that is created using the new keyword.

[↑ Back to Top](#)

342. What is the difference between a parameter and an argument [↗](#)

Parameter is the variable name of a function definition whereas an argument represents the value given to a function when it is invoked. Let's explain this with a simple function

```
function myFunction(parameter1, parameter2, parameter3) {  
  console.log(arguments[0]); // "argument1"  
  console.log(arguments[1]); // "argument2"  
  console.log(arguments[2]); // "argument3"  
}  
myFunction("argument1", "argument2", "argument3");
```



[↑ Back to Top](#)

343. What is the purpose of some method in arrays [↗](#)

The some() method is used to test whether at least one element in the array passes the test implemented by the provided function. The method returns a boolean value. Let's take an example to test for any odd elements,

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
  
var odd = (element) => element % 2 !== 0;  
  
console.log(array.some(odd)); // true (the odd element exists)
```



[↑ Back to Top](#)

344. How do you combine two or more arrays [↗](#)

The concat() method is used to join two or more arrays by returning a new array containing all the elements. The syntax would be as below,

```
array1.concat(array2, array3, ..., arrayX)
```



Let's take an example of array's concatenation with veggies and fruits arrays,

```
var veggies = ["Tomato", "Carrot", "Cabbage"];  
var fruits = ["Apple", "Orange", "Pears"];  
var veggiesAndFruits = veggies.concat(fruits);  
console.log(veggiesAndFruits); // Tomato, Carrot, Cabbage, Apple, Orange, Pe
```



[↑ Back to Top](#)

345. What is the difference between Shallow and Deep copy [↗](#)

There are two ways to copy an object,

Shallow Copy: Shallow copy is a bitwise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

Example

```
var empDetails = {  
  name: "John",  
  age: 25,  
  expertise: "Software Developer",  
};
```



to create a duplicate

```
var empDetailsShallowCopy = empDetails; //Shallow copying!
```



if we change some property value in the duplicate one like this:

```
empDetailsShallowCopy.name = "Johnson";
```



The above statement will also change the name of `empDetails`, since we have a shallow copy. That means we're losing the original data as well.

Deep copy: A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

Example

```
var empDetails = {  
  name: "John",  
  age: 25,  
  expertise: "Software Developer",  
};
```



Create a deep copy by using the properties from the original object into new variable

```
var empDetailsDeepCopy = {  
  name: empDetails.name,  
  age: empDetails.age,  
  expertise: empDetails.expertise,  
};
```



Now if you change `empDetailsDeepCopy.name`, it will only affect `empDetailsDeepCopy` & not `empDetails`

[↑ Back to Top](#)

346. How do you create specific number of copies of a string [↗](#)

The `repeat()` method is used to construct and return a new string which contains the specified number of copies of the string on which it was called, concatenated together. Remember that this method has been added to the ECMAScript 2015 specification. Let's take an example of Hello string to repeat it 4 times,

```
"Hello".repeat(4); // 'HelloHelloHelloHello'
```



347. How do you return all matching strings against a regular expression [↗](#)

The `matchAll()` method can be used to return an iterator of all results matching a string against a regular expression. For example, the below example returns an array of matching string results against a regular expression,

```
let regexp = /Hello(\d?))/g;
let greeting = "Hello1Hello2Hello3";

let greetingList = [...greeting.matchAll(regexp)];

console.log(greetingList[0]); //Hello1
console.log(greetingList[1]); //Hello2
console.log(greetingList[2]); //Hello3
```



[↑ Back to Top](#)

348. How do you trim a string at the beginning or ending [↗](#)

The `trim` method of string prototype is used to trim on both sides of a string. But if you want to trim especially at the beginning or ending of the string then you can use `trimStart/trimLeft` and `trimEnd/trimRight` methods. Let's see an example of these methods on a greeting message,

```
var greeting = "  Hello, Goodmorning!  ";

console.log(greeting); // "  Hello, Goodmorning!  "
console.log(greeting.trimStart()); // "Hello, Goodmorning!  "
console.log(greeting.trimLeft()); // "Hello, Goodmorning!  "

console.log(greeting.trimEnd()); // "  Hello, Goodmorning!"
console.log(greeting.trimRight()); // "  Hello, Goodmorning!"
```



[↑ Back to Top](#)

349. What is the output of below console statement with unary operator [↗](#)

Let's take console statement with unary operator as given below,

```
console.log(+ "Hello");
```



The output of the above console log statement returns NaN. Because the element is prefixed by the unary operator and the JavaScript interpreter will try to convert that element into a number type. Since the conversion fails, the value of the statement results in NaN value.

[↑ Back to Top](#)

350. Does javascript uses mixins [↗](#)

Mixin is a generic object-oriented programming term - is a class containing methods that can be used by other classes without a need to inherit from it. In JavaScript we can only inherit from a single object. ie. There can be only one `[[prototype]]` for an object.

But sometimes we require to extend more than one, to overcome this we can use Mixin which helps to copy methods to the prototype of another class.

Say for instance, we've two classes `User` and `CleanRoom`. Suppose we need to add `CleanRoom` functionality to `User`, so that user can clean the room at demand. Here's where concept called mixins comes into picture.

```
// mixin
let cleanRoomMixin = {
  cleanRoom() {
    alert(`Hello ${this.name}, your room is clean now`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  },
};

// usage:
class User {
  constructor(name) {
    this.name = name;
  }
}

// copy the methods
Object.assign(User.prototype, cleanRoomMixin);

// now User can clean the room
new User("Dude").cleanRoom(); // Hello Dude, your room is clean now!
```



[↑ Back to Top](#)

351. What is a thunk function [↗](#)

A thunk is just a function which delays the evaluation of the value. It doesn't take any arguments but gives the value whenever you invoke the thunk. i.e, It is used not to execute now but it will be sometime in the future. Let's take a synchronous example,

```
const add = (x, y) => x + y;

const thunk = () => add(2, 3);

thunk(); // 5
```



[↑ Back to Top](#)

352. What are asynchronous thunks [↗](#)

The asynchronous thunks are useful to make network requests. Let's see an example of network requests,

```
function fetchData(fn) {
  fetch("https://jsonplaceholder.typicode.com/todos/1")
    .then((response) => response.json())
    .then((json) => fn(json));
}

const asyncThunk = function () {
  return fetchData(function getData(data) {
    console.log(data);
  });
};

asyncThunk();
```



The `getData` function won't be called immediately but it will be invoked only when the data is available from API endpoint. The `setTimeout` function is also used to make our code asynchronous. The best real time example is `redux` state management library which uses the asynchronous thunks to delay the actions to dispatch.

[↑ Back to Top](#)

353. What is the output of below function calls [↗](#)

Code snippet:

```
const circle = {
  radius: 20,
  diameter() {
    return this.radius * 2;
  },
};
```



```
    perimeter: () => 2 * Math.PI * this.radius,  
  };
```

```
console.log(circle.diameter());  
console.log(circle.perimeter());
```



Output:

The output is 40 and NaN. Remember that diameter is a regular function, whereas the value of perimeter is an arrow function. The `this` keyword of a regular function(i.e, diameter) refers to the surrounding scope which is a class(i.e, Shape object). Whereas this keyword of perimeter function refers to the surrounding scope which is a window object. Since there is no radius property on window objects it returns an undefined value and the multiple of number value returns NaN value.

[↑ Back to Top](#)

354. How to remove all line breaks from a string [↗](#)

The easiest approach is using regular expressions to detect and replace newlines in the string. In this case, we use replace function along with string to replace with, which in our case is an empty string.

```
function remove_linebreaks( var message ) {  
    return message.replace( /\r\n/gm, "" );  
}
```



In the above expression, g and m are for global and multiline flags.

[↑ Back to Top](#)

355. What is the difference between reflow and repaint [↗](#)

A *repaint* occurs when changes are made which affect the visibility of an element, but not its layout. Examples of this include outline, visibility, or background color. A *reflow* involves changes that affect the layout of a portion of the page (or the whole page). Resizing the browser window, changing the font, content changing (such as user typing text), using JavaScript methods involving computed styles, adding or removing elements from the DOM, and changing an element's classes are a few of the things that can trigger reflow. Reflow of an element causes the subsequent reflow of all child and ancestor elements as well as any elements following it in the DOM.

[↑ Back to Top](#)

356. What happens with negating an array [↗](#)

Negating an array with `!` character will coerce the array into a boolean. Since Arrays are considered to be truthy So negating it will return `false`.

```
console.log(![]); // false
```



[↑ Back to Top](#)

357. What happens if we add two arrays [↗](#)

If you add two arrays together, it will convert them both to strings and concatenate them. For example, the result of adding arrays would be as below,

```
console.log(["a"] + ["b"]); // "ab"
console.log([] + []); // ""
console.log(![] + []); // "false", because ![] returns false.
```



[↑ Back to Top](#)

358. What is the output of prepend additive operator on falsy values [↗](#)

If you prepend the additive(+) operator on falsy values(null, undefined, NaN, false, ""), the falsy value converts to a number value zero. Let's display them on browser console as below,

```
console.log(+null); // 0
console.log(+undefined); // NaN
console.log(+false); // 0
```



```
console.log(+NaN); // NaN
console.log(+""); // 0
```

[↑ Back to Top](#)

359. How do you create self string using special characters [↗](#)

The self string can be formed with the combination of `[]()!+` characters. You need to remember the below conventions to achieve this pattern.

- i. Since Arrays are truthful values, negating the arrays will produce false: `![] === false`
- ii. As per JavaScript coercion rules, the addition of arrays together will toString them: `[] + [] === ""`
- iii. Prepend an array with + operator will convert an array to false, the negation will make it true and finally converting the result will produce value '1': `+(!(+[])) === 1`

By applying the above rules, we can derive below conditions

```
(![] + [] === "false" + !+[]) === 1;
```



Now the character pattern would be created as below,

[illegible]

[Back to Top](#)

360. How do you remove falsy values from an array [↗](#)

You can apply the filter method on the array by passing Boolean as a parameter. This way it removes all falsy values(0, undefined, null, false and "") from the array.

```
const myArray = [false, null, 1, 5, undefined];  
myArray.filter(Boolean); // [1, 5] // is same as myArray.filter(x => x);
```



[↑ Back to Top](#)

361. How do you get unique values of an array [↗](#)

You can get unique values of an array with the combination of `Set` and rest expression/spread(...) syntax.

```
console.log([...new Set([1, 2, 4, 4, 3])]); // [1, 2, 4, 3]
```



[↑ Back to Top](#)

362. What is destructuring aliases [↗](#)

Sometimes you would like to have a destructured variable with a different name than the property name. In that case, you'll use a `: newName` to specify a name for the variable. This process is called destructuring aliases.

```
const obj = { x: 1 };  
// Grabs obj.x as as { otherName }  
const { x: otherName } = obj;
```



[↑ Back to Top](#)

363. How do you map the array values without using map method [↗](#)

You can map the array values without using the `map` method by just using the `from` method of Array. Let's map city names from Countries array,

```
const countries = [  
  { name: "India", capital: "Delhi" },  
  { name: "US", capital: "Washington" },  
  { name: "Russia", capital: "Moscow" },  
  { name: "Singapore", capital: "Singapore" },  
  { name: "China", capital: "Beijing" },  
  { name: "France", capital: "Paris" },  
];
```



```
const cityNames = Array.from(countries, ({ capital }) => capital);
console.log(cityNames); // ['Delhi', 'Washington', 'Moscow', 'Singapore', 'Be:
```

[↑ Back to Top](#)

364. How do you empty an array [↗](#)

You can empty an array quickly by setting the array length to zero.

```
let cities = ["Singapore", "Delhi", "London"];
cities.length = 0; // cities becomes []
```



[↑ Back to Top](#)

365. How do you rounding numbers to certain decimals [↗](#)

You can round numbers to a certain number of decimals using `toFixed` method from native javascript.

```
let pie = 3.141592653;
pie = pie.toFixed(3); // 3.142
```



[↑ Back to Top](#)

366. What is the easiest way to convert an array to an object [↗](#)

You can convert an array to an object with the same data using `spread(...)` operator.

```
var fruits = ["banana", "apple", "orange", "watermelon"];
var fruitsObject = { ...fruits };
console.log(fruitsObject); // {0: "banana", 1: "apple", 2: "orange", 3: "wat
```



[↑ Back to Top](#)

367. How do you create an array with some data [↗](#)

You can create an array with some data or an array with the same values using `fill` method.

```
var newArray = new Array(5).fill("0");
console.log(newArray); // ["0", "0", "0", "0", "0"]
```



[↑ Back to Top](#)

368. What are the placeholders from console object [↗](#)

Below are the list of placeholders available from console object,

- i. %o — It takes an object,
 - ii. %s — It takes a string,
 - iii. %d — It is used for a decimal or integer
- These placeholders can be represented in the console.log as below

```
const user = { name: "John", id: 1, city: "Delhi" };
console.log(
  "Hello %s, your details %o are available in the object form",
  "John",
  user
); // Hello John, your details {name: "John", id: 1, city: "Delhi"} are available
```



[↑ Back to Top](#)

369. Is it possible to add CSS to console messages [↗](#)

Yes, you can apply CSS styles to console messages similar to html text on the web page.

```
console.log(
  "%c The text has blue color, with large font and red background",
  "color: blue; font-size: x-large; background: red"
);
```



The text will be displayed as below,

```
> console.log('%c Color of the text', 'color: blue; font-size: x-large; background: red');
Color of the text
```

vendors~main.51281d83.chunk.js:1

Note: All CSS styles can be applied to console messages.

[↑ Back to Top](#)

370. What is the purpose of dir method of console object

The `console.dir()` is used to display an interactive list of the properties of the specified JavaScript object as JSON.

```
const user = { name: "John", id: 1, city: "Delhi" };  
console.dir(user);
```



The user object displayed in JSON representation

```
> const user = { "name": "John", "id": 1, "city": "Delhi"};  
   console.dir(user);
```

```
▼ Object ⓘ  
  name: "John"  
  id: 1  
  city: "Delhi"  
  ► __proto__: Object
```

[↑ Back to Top](#)

371. Is it possible to debug HTML elements in console

Yes, it is possible to get and debug HTML elements in the console just like inspecting elements.

```
const element = document.getElementsByTagName("body")[0];  
console.log(element);
```



It prints the HTML element in the console,

```
> const element = document.getElementsByTagName("body")[0];  
< undefined  
> console.log(element);  
  ► <body class="question-page unified-theme">...</body>  
< undefined  
> |
```

[↑ Back to Top](#)

372. How do you display data in a tabular format using console object

The `console.table()` is used to display data in the console in a tabular format to visualize complex arrays or objects.

```
const users = [
  { name: "John", id: 1, city: "Delhi" },
  { name: "Max", id: 2, city: "London" },
  { name: "Rod", id: 3, city: "Paris" },
];
console.table(users);
```



The data visualized in a table format,

```
> const users = [{ "name": "John", "id": 1, "city": "Delhi"},
                  { "name": "Max", "id": 2, "city": "London"},
                  { "name": "Rod", "id": 3, "city": "Paris"}];
< undefined
> console.table(users);
```

VM92:1

(index)	name	id	city
0	John	1	Delhi
1	Max	2	London
2	Rod	3	Paris

► Array(3)

Not: Remember that `console.table()` is not supported in IE.

[↑ Back to Top](#)

373. How do you verify that an argument is a Number or not [↗](#)

The combination of `isNaN` and `isFinite` methods are used to confirm whether an argument is a number or not.

```
function isNumber(n) {
  return !isNaN(parseFloat(n)) && isFinite(n);
}
```



[↑ Back to Top](#)

374. How do you create copy to clipboard button [↗](#)

You need to select the content(using `.select()` method) of the input element and execute the copy command with `execCommand` (i.e, `execCommand('copy')`). You can also execute other system commands like cut and paste.

```
document.querySelector("#copy-button").onclick = function () {
  // Select the content
  document.querySelector("#copy-input").select();
}
```



```
// Copy to the clipboard
document.execCommand("copy");
};
```

[↑ Back to Top](#)

375. What is the shortcut to get timestamp [↗](#)

You can use `new Date().getTime()` to get the current timestamp. There is an alternative shortcut to get the value.

```
console.log(+new Date());
console.log(Date.now());
```



[↑ Back to Top](#)

376. How do you flattening multi dimensional arrays [↗](#)

Flattening bi-dimensional arrays is trivial with Spread operator.

```
const biDimensionalArr = [11, [22, 33], [44, 55], [66, 77], 88, 99];
const flattenArr = [].concat(...biDimensionalArr); // [11, 22, 33, 44, 55, 66, 77, 88, 99]
```



But you can make it work with multi-dimensional arrays by recursive calls,

```
function flattenMultiArray(arr) {
  const flattened = [].concat(...arr);
  return flattened.some((item) => Array.isArray(item))
    ? flattenMultiArray(flattened)
    : flattened;
}
const multiDimensionalArr = [11, [22, 33], [44, [55, 66, [77, [88]], 99]]];
const flatArr = flattenMultiArray(multiDimensionalArr); // [11, 22, 33, 44, 55, 66, 77, 88, 99]
```



Also you can use the `flat` method of Array.

```
const arr = [1, [2, 3], 4, 5, [6, 7]];
const flatArr = arr.flat(); // [1, 2, 3, 4, 5, 6, 7]
```



```
// And for multiDemensional arrays
const multiDimensionalArr = [11, [22, 33], [44, [55, 66, [77, [88]], 99]]];
```



```
const oneStepFlat = multiDimensionalArr.flat(1); // [11, 22, 33, 44, [55, 66
const towStep = multiDimensionalArr.flat(2); // [11, 22, 33, 44, 55, 66, [77
const fullyFlatArray = multiDimensionalArr.flat(Infinity); // [11, 22, 33, 4
```

[↑ Back to Top](#)

377. What is the easiest multi condition checking [↗](#)

You can use `indexOf` to compare input with multiple values instead of checking each value as one condition.

```
// Verbose approach
if (
  input === "first" ||
  input === 1 ||
  input === "second" ||
  input === 2
) {
  someFunction();
}
// Shortcut
if (["first", 1, "second", 2].indexOf(input) !== -1) {
  someFunction();
}
```



[↑ Back to Top](#)

378. How do you capture browser back button [↗](#)

The `beforeunload` event is triggered when the window, the document and its resources are about to be unloaded. This event is helpful to warn users about losing the current data and detect back button event.

```
window.addEventListener('beforeunload', () => {
  console.log('Clicked browser back button');
});
```



You can also use `popstate` event to detect the browser back button. **Note:** The history entry has been activated using `history.pushState` method.

```
window.addEventListener('popstate', () => {
  console.log('Clicked browser back button');
  box.style.backgroundColor = 'white';
});
```



```
});

const box = document.getElementById('div');

box.addEventListener('click', () => {
  box.style.backgroundColor = 'blue';
  window.history.pushState({}, null, null);
});
```

In the preceeding code, When the box element clicked, its background color appears in blue color and changed to while color upon clicking the browser back button using `popstate` event handler. The `state` property of `popstate` contains the copy of history entry's state object.

[↑ Back to Top](#)(#table-of-contents)

379. How do you disable right click in the web page [↗](#)

The right click on the page can be disabled by returning false from the `oncontextmenu` attribute on the body element.

```
<body oncontextmenu="return false;"></body>
```

[↑ Back to Top](#)

380. What are wrapper objects [↗](#)

Primitive Values like string,number and boolean don't have properties and methods but they are temporarily converted or coerced to an object(Wrapper object) when you try to perform actions on them. For example, if you apply `toUpperCase()` method on a primitive string value, it does not throw an error but returns uppercase of the string.

```
let name = "john";

console.log(name.toUpperCase()); // Behind the scenes treated as console.log
```

i.e, Every primitive except null and undefined have Wrapper Objects and the list of wrapper objects are String,Number,Boolean,Symbol and BigInt.

[↑ Back to Top](#)

381. What is AJAX [↗](#)

AJAX stands for Asynchronous JavaScript and XML and it is a group of related technologies(HTML, CSS, JavaScript, XMLHttpRequest API etc) used to display data asynchronously. i.e. We can send data to the server and get data from the server without reloading the web page.

[↑ Back to Top](#)

382. What are the different ways to deal with Asynchronous Code [↗](#)

Below are the list of different ways to deal with Asynchronous code.

- i. Callbacks
- ii. Promises
- iii. Async/await
- iv. Third-party libraries such as async.js,bluebird etc

[↑ Back to Top](#)

383. How to cancel a fetch request [↗](#)

Until a few days back, One shortcoming of native promises is no direct way to cancel a fetch request. But the new `AbortController` from js specification allows you to use a signal to abort one or multiple fetch calls. The basic flow of cancelling a fetch request would be as below,

- i. Create an `AbortController` instance
- ii. Get the `signal` property of an instance and pass the signal as a fetch option for signal
- iii. Call the `AbortController`'s `abort` property to cancel all fetches that use that signal For example, let's pass the same signal to multiple fetch calls will cancel all requests with that signal,

```
const controller = new AbortController();
const { signal } = controller;

fetch("http://localhost:8000", { signal })
  .then((response) => {
    console.log(`Request 1 is complete!`);
  })
  .catch((e) => {
```



```

    if (e.name === "AbortError") {
      // We know it's been canceled!
    }
  });

fetch("http://localhost:8000", { signal })
  .then((response) => {
    console.log(`Request 2 is complete!`);
  })
  .catch((e) => {
    if (e.name === "AbortError") {
      // We know it's been canceled!
    }
  });

// Wait 2 seconds to abort both requests
setTimeout(() => controller.abort(), 2000);

```

[↑ Back to Top](#)

384. What is web speech API [↗](#)

Web speech API is used to enable modern browsers recognize and synthesize speech (i.e, voice data into web apps). This API has been introduced by W3C Community in the year 2012. It has two main parts,

i. **SpeechRecognition (Asynchronous Speech Recognition or Speech-to-Text):**

It provides the ability to recognize voice context from an audio input and respond accordingly. This is accessed by the `SpeechRecognition` interface. The below example shows on how to use this API to get text from speech,

```

window.SpeechRecognition =
  window.webkitSpeechRecognition || window.SpeechRecognition; // webkitSpeech
const recognition = new window.SpeechRecognition();
recognition.onresult = (event) => {
  // SpeechRecognitionEvent type
  const speechToText = event.results[0][0].transcript;
  console.log(speechToText);
};
recognition.start();

```

In this API, browser is going to ask you for permission to use your microphone

i. **SpeechSynthesis (Text-to-Speech):** It provides the ability to recognize voice context from an audio input and respond. This is accessed by the

SpeechSynthesis interface. For example, the below code is used to get voice/speech from text,

```
if ("speechSynthesis" in window) {  
    var speech = new SpeechSynthesisUtterance("Hello World!");  
    speech.lang = "en-US";  
    window.speechSynthesis.speak(speech);  
}
```



The above examples can be tested on chrome(33+) browser's developer console.

Note: This API is still a working draft and only available in Chrome and Firefox browsers(ofcourse Chrome only implemented the specification)

[↑ Back to Top](#)

385. What is minimum timeout throttling [↗](#)

Both browser and NodeJS javascript environments throttles with a minimum delay that is greater than 0ms. That means even though setting a delay of 0ms will not happen instantaneously. **Browsers:** They have a minimum delay of 4ms. This throttle occurs when successive calls are triggered due to callback nesting(certain depth) or after a certain number of successive intervals. Note: The older browsers have a minimum delay of 10ms. **Nodejs:** They have a minimum delay of 1ms. This throttle happens when the delay is larger than 2147483647 or less than 1. The best example to explain this timeout throttling behavior is the order of below code snippet.

```
function runMeFirst() {  
    console.log("My script is initialized");  
}  
setTimeout(runMeFirst, 0);  
console.log("Script loaded");
```



and the output would be in

```
Script loaded  
My script is initialized
```



If you don't use `setTimeout` , the order of logs will be sequential.

```
function runMeFirst() {  
    console.log("My script is initialized");  
}
```



```
runMeFirst();  
console.log("Script loaded");
```

and the output is,

```
My script is initialized  
Script loaded
```



[↑ Back to Top](#)

386. How do you implement zero timeout in modern browsers [↗](#)

You can't use `setTimeout(fn, 0)` to execute the code immediately due to minimum delay of greater than 0ms. But you can use `window.postMessage()` to achieve this behavior.

[↑ Back to Top](#)

387. What are tasks in event loop [↗](#)

A task is any javascript code/program which is scheduled to be run by the standard mechanisms such as initially starting to run a program, run an event callback, or an interval or timeout being fired. All these tasks are scheduled on a task queue. Below are the list of use cases to add tasks to the task queue,

- i. When a new javascript program is executed directly from console or running by the `<script>` element, the task will be added to the task queue.
- ii. When an event fires, the event callback added to task queue
- iii. When a `setTimeout` or `setInterval` is reached, the corresponding callback added to task queue

[↑ Back to Top](#)

388. What is microtask [↗](#)

Microtask is the javascript code which needs to be executed immediately after the currently executing task/microtask is completed. They are kind of blocking in nature. i.e, The main thread will be blocked until the microtask queue is empty. The main sources of microtasks are `Promise.resolve`, `Promise.reject`, `MutationObservers`, `IntersectionObservers` etc

Note: All of these microtasks are processed in the same turn of the event loop. [↑](#)

[Back to Top](#)

389. What are different event loops [↗](#)

In JavaScript, there are multiple event loops that can be used depending on the context of your application. The most common event loops are:

390. The Browser Event Loop

391. The Node.js Event Loop

- Browser Event Loop: The Browser Event Loop is used in client-side JavaScript applications and is responsible for handling events that occur within the browser environment, such as user interactions (clicks, keypresses, etc.), HTTP requests, and other asynchronous actions.

- The Node.js Event Loop is used in server-side JavaScript applications and is responsible for handling events that occur within the Node.js runtime environment, such as file I/O, network I/O, and other asynchronous actions.

****[↑ Back to Top](#table-of-contents)****

390. What is the purpose of queueMicrotask [↗](#)

The `queueMicrotask` function is used to schedule a microtask, which is a function that will be executed asynchronously in the microtask queue. The purpose of `queueMicrotask` is to ensure that a function is executed after the current task has finished, but before the browser performs any rendering or handles user events.

Example:

```
console.log("Start"); //1

queueMicrotask(() => {
  console.log("Inside microtask"); // 3
});

console.log("End"); //2
```

By using `queueMicrotask`, you can ensure that certain tasks or callbacks are executed at the earliest opportunity during the JavaScript event loop, making it useful for performing work that needs to be done asynchronously but with higher priority than regular `setTimeout` or `setInterval` callbacks.

391. How do you use javascript libraries in typescript file [↗](#)

It is known that not all JavaScript libraries or frameworks have TypeScript declaration files. But if you still want to use libraries or frameworks in our TypeScript files without getting compilation errors, the only solution is `declare` keyword along with a variable declaration. For example, let's imagine you have a library called `customLibrary` that doesn't have a TypeScript declaration and have a namespace called `customLibrary` in the global namespace. You can use this library in typescript code as below,

```
declare var customLibrary;
```



In the runtime, typescript will provide the type to the `customLibrary` variable as any type. The another alternative without using `declare` keyword is below

```
var customLibrary: any;
```



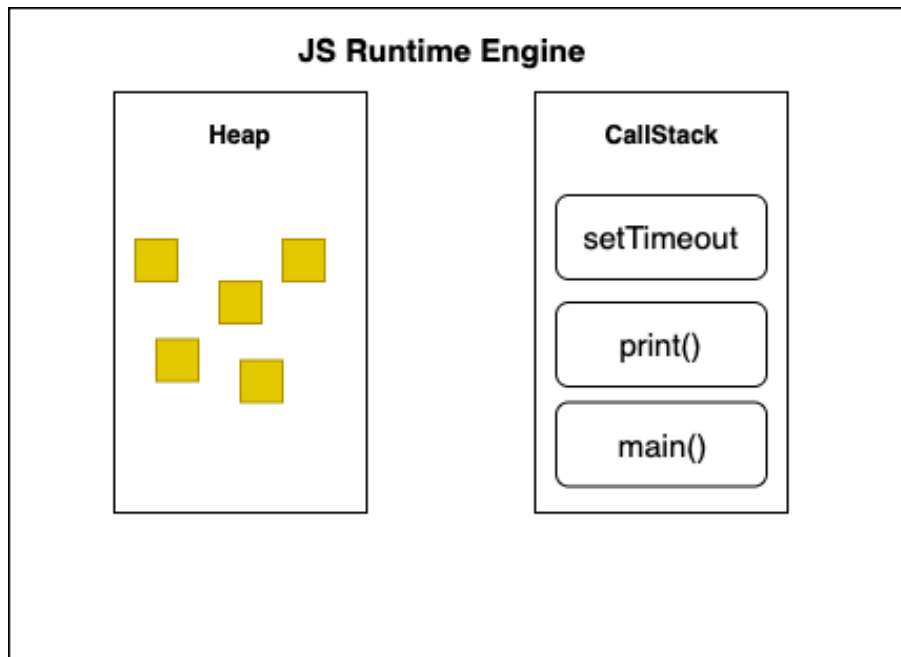
392. What are the differences between promises and observables [↗](#)

Some of the major difference in a tabular form

Promises	Observables
Emits only a single value at a time	Emits multiple values over a period of time(stream of values ranging from 0 to multiple)
Eager in nature; they are going to be called immediately	Lazy in nature; they require subscription to be invoked
Promise is always asynchronous even though it resolved immediately	Observable can be either synchronous or asynchronous
Doesn't provide any operators	Provides operators such as map, forEach, filter, reduce, retry, and retryWhen etc
Cannot be canceled	Canceled by using unsubscribe() method

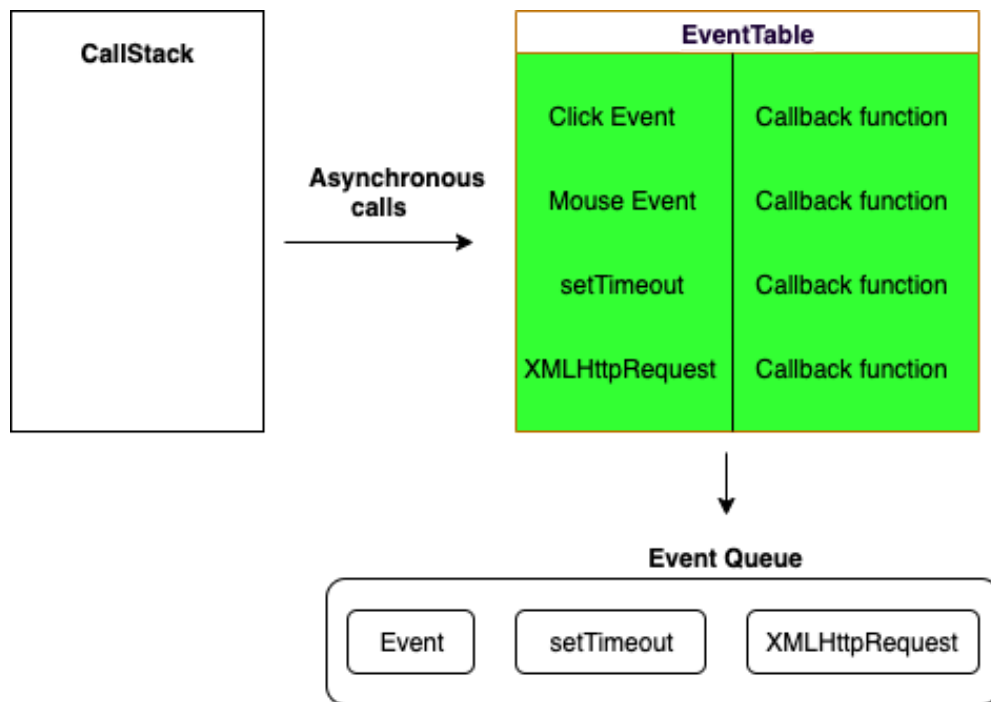
393. What is heap [↗](#)

Heap(Or memory heap) is the memory location where objects are stored when we define variables. i.e, This is the place where all the memory allocations and de-allocation take place. Both heap and call-stack are two containers of JS runtime. Whenever runtime comes across variables and function declarations in the code it stores them in the Heap.



394. What is an event table [↗](#)

Event Table is a data structure that stores and keeps track of all the events which will be executed asynchronously like after some time interval or after the resolution of some API requests. i.e Whenever you call a `setTimeout` function or invoke async operation, it is added to the Event Table. It doesn't not execute functions on it's own. The main purpose of the event table is to keep track of events and send them to the Event Queue as shown in the below diagram.



[↑ Back to Top](#)

395. What is a microTask queue [↗](#)

Microtask Queue is the new queue where all the tasks initiated by promise objects get processed before the callback queue. The microtasks queue are processed before the next rendering and painting jobs. But if these microtasks are running for a long time then it leads to visual degradation.

[↑ Back to Top](#)

396. What is the difference between shim and polyfill [↗](#)

A shim is a library that brings a new API to an older environment, using only the means of that environment. It isn't necessarily restricted to a web application. For example, es5-shim.js is used to emulate ES5 features on older browsers (mainly pre IE9). Whereas polyfill is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively. In a simple sentence, A polyfill is a shim for a browser API.

[↑ Back to Top](#)

397. How do you detect primitive or non primitive value type [↗](#)

In JavaScript, primitive types include boolean, string, number, BigInt, null, Symbol and undefined. Whereas non-primitive types include the Objects. But you can easily identify them with the below function,

```
var myPrimitive = 30;
var myNonPrimitive = {};
function isPrimitive(val) {
    return Object(val) !== val;
}

isPrimitive(myPrimitive);
isPrimitive(myNonPrimitive);
```



If the value is a primitive data type, the Object constructor creates a new wrapper object for the value. But If the value is a non-primitive data type (an object), the Object constructor will give the same object.

[↑ Back to Top](#)

398. What is babel [↗](#)

Babel is a JavaScript transpiler to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Some of the main features are listed below,

- i. Transform syntax
- ii. Polyfill features that are missing in your target environment (using @babel/polyfill)
- iii. Source code transformations (or codemods)

[↑ Back to Top](#)

399. Is Node.js completely single threaded [↗](#)

Node is a single thread, but some of the functions included in the Node.js standard library(e.g, fs module functions) are not single threaded. i.e, Their logic runs outside of the Node.js single thread to improve the speed and performance of a program.

[↑ Back to Top](#)

400. What are the common use cases of observables [↗](#)

Some of the most common use cases of observables are web sockets with push notifications, user input changes, repeating intervals, etc

[↑ Back to Top](#)

401. What is RxJS [↗](#)

RxJS (Reactive Extensions for JavaScript) is a library for implementing reactive programming using observables that makes it easier to compose asynchronous or callback-based code. It also provides utility functions for creating and working with observables.

[↑ Back to Top](#)

402. What is the difference between Function constructor and function declaration [↗](#)

The functions which are created with `Function` constructor do not create closures to their creation contexts but they are always created in the global scope. i.e, the function can access its own local variables and global scope variables only. Whereas function declarations can access outer function variables(closures) too.

Let's see this difference with an example,

Function Constructor:

```
var a = 100;
function createFunction() {
  var a = 200;
  return new Function("return a;");
}
console.log(createFunction()); // 100
```



Function declaration:

```
var a = 100;
function createFunction() {
  var a = 200;
  return function func() {
    return a;
  };
}
console.log(createFunction()); // 200
```



[↑ Back to Top](#)

403. What is a Short circuit condition [↗](#)

Short circuit conditions are meant for condensed way of writing simple if statements. Let's demonstrate the scenario using an example. If you would like to login to a portal with an authentication condition, the expression would be as below,

```
if (authenticate) {  
    loginToPorta();  
}
```



Since the javascript logical operators evaluated from left to right, the above expression can be simplified using && logical operator

```
authenticate && loginToPorta();
```



[↑ Back to Top](#)

404. What is the easiest way to resize an array [↗](#)

The length property of an array is useful to resize or empty an array quickly. Let's apply length property on number array to resize the number of elements from 5 to 2,

```
var array = [1, 2, 3, 4, 5];  
console.log(array.length); // 5  
  
array.length = 2;  
console.log(array.length); // 2  
console.log(array); // [1,2]
```



and the array can be emptied too

```
var array = [1, 2, 3, 4, 5];  
array.length = 0;  
console.log(array.length); // 0  
console.log(array); // []
```



[↑ Back to Top](#)

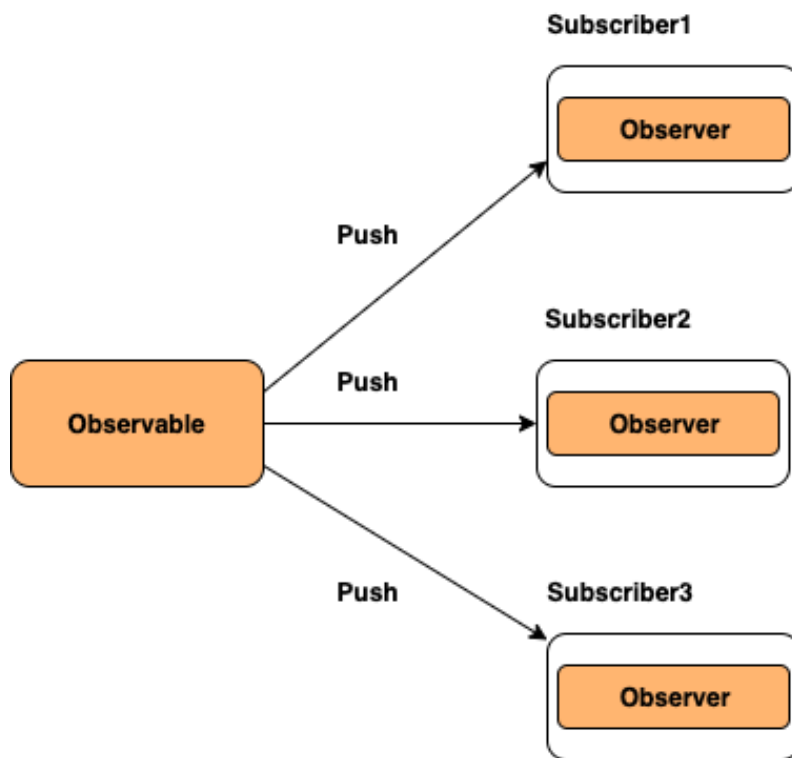
405. What is an observable [↗](#)

An Observable is basically a function that can return a stream of values either synchronously or asynchronously to an observer over time. The consumer can get the value by calling `subscribe()` method. Let's look at a simple example of an Observable

```
import { Observable } from "rxjs";

const observable = new Observable((observer) => {
  setTimeout(() => {
    observer.next("Message from a Observable!");
  }, 3000);
});

observable.subscribe((value) => console.log(value));
```



Note: Observables are not part of the JavaScript language yet but they are being proposed to be added to the language

[↑ Back to Top](#)

406. What is the difference between function and class declarations



The main difference between function declarations and class declarations is hoisting . The function declarations are hoisted but not class declarations.

Classes:

```
const user = new User(); // ReferenceError
```



```
class User {}
```

Constructor Function:

```
const user = new User(); // No error
```



```
function User() {}
```

[↑ Back to Top](#)

407. What is an async function [↗](#)

An async function is a function declared with the `async` keyword which enables asynchronous, promise-based behavior to be written in a cleaner style by avoiding promise chains. These functions can contain zero or more `await` expressions.

Let's take a below async function example,

```
async function logger() {  
  let data = await fetch("http://someapi.com/users"); // pause until fetch r  
  console.log(data);  
}  
logger();
```



It is basically syntax sugar over ES2015 promises and generators.

[↑ Back to Top](#)

408. How do you prevent promises swallowing errors [↗](#)

While using asynchronous code, JavaScript's ES6 promises can make your life a lot easier without having callback pyramids and error handling on every second line. But Promises have some pitfalls and the biggest one is swallowing errors by default.

Let's say you expect to print an error to the console for all the below cases,

```
Promise.resolve("promised value").then(function () {  
  throw new Error("error");  
});
```



```
Promise.reject("error value").catch(function () {  
    throw new Error("error");  
});  
  
new Promise(function (resolve, reject) {  
    throw new Error("error");  
});
```

But there are many modern JavaScript environments that won't print any errors. You can fix this problem in different ways,

- i. **Add catch block at the end of each chain:** You can add catch block to the end of each of your promise chains

```
Promise.resolve("promised value")  
    .then(function () {  
        throw new Error("error");  
    })  
    .catch(function (error) {  
        console.error(error.stack);  
    });
```



But it is quite difficult to type for each promise chain and verbose too.

- ii. **Add done method:** You can replace first solution's then and catch blocks with done method

```
Promise.resolve("promised value").done(function () {  
    throw new Error("error");  
});
```



Let's say you want to fetch data using HTTP and later perform processing on the resulting data asynchronously. You can write `done` block as below,

```
getDataFromHttp()  
    .then(function (result) {  
        return processDataAsync(result);  
    })  
    .done(function (processed) {  
        displayData(processed);  
    });
```



In future, if the processing library API changed to synchronous then you can remove `done` block as below,

```
getDataFromHttp().then(function (result) {  
    return displayData(processDataAsync(result));  
});
```



and then you forgot to add `done` block to `then` block leads to silent errors.

- iii. **Extend ES6 Promises by Bluebird:** Bluebird extends the ES6 Promises API to avoid the issue in the second solution. This library has a "default" `onRejection` handler which will print all errors from rejected Promises to `stderr`. After installation, you can process unhandled rejections

```
Promise.onPossiblyUnhandledRejection(function (error) {  
    throw error;  
});
```



and discard a rejection, just handle it with an empty catch

```
Promise.reject("error value").catch(function () {});
```



[↑ Back to Top](#)

409. What is deno [↗](#)

Deno is a simple, modern and secure runtime for JavaScript and TypeScript that uses V8 JavaScript engine and the Rust programming language.

[↑ Back to Top](#)

410. How do you make an object iterable in javascript [↗](#)

By default, plain objects are not iterable. But you can make the object iterable by defining a `Symbol.iterator` property on it.

Let's demonstrate this with an example,

```
const collection = {  
    one: 1,  
    two: 2,  
    three: 3,
```



```

[Symbol.iterator]() {
  const values = Object.keys(this);
  let i = 0;
  return {
    next: () => {
      return {
        value: this[values[i++]],
        done: i > values.length,
      };
    },
  };
},
};

const iterator = collection[Symbol.iterator]();

console.log(iterator.next()); // → {value: 1, done: false}
console.log(iterator.next()); // → {value: 2, done: false}
console.log(iterator.next()); // → {value: 3, done: false}
console.log(iterator.next()); // → {value: undefined, done: true}

```

The above process can be simplified using a generator function,

```

const collection = {
  one: 1,
  two: 2,
  three: 3,
  [Symbol.iterator]: function* () {
    for (let key in this) {
      yield this[key];
    }
  },
};

const iterator = collection[Symbol.iterator]();
console.log(iterator.next()); // {value: 1, done: false}
console.log(iterator.next()); // {value: 2, done: false}
console.log(iterator.next()); // {value: 3, done: false}
console.log(iterator.next()); // {value: undefined, done: true}

```



[↑ Back to Top](#)

411. What is a Proper Tail Call [↗](#)

First, we should know about tail call before talking about "Proper Tail Call". A tail call is a subroutine or function call performed as the final action of a calling function. Whereas **Proper tail call(PTC)** is a technique where the program or code will not create additional stack frames for a recursion when the function call is a tail call.

For example, the below classic or head recursion of factorial function relies on stack for each step. Each step need to be processed upto $n * \text{factorial}(n - 1)$

```
function factorial(n) {  
  if (n === 0) {  
    return 1;  
  }  
  return n * factorial(n - 1);  
}  
console.log(factorial(5)); //120
```



But if you use Tail recursion functions, they keep passing all the necessary data it needs down the recursion without relying on the stack.

```
function factorial(n, acc = 1) {  
  if (n === 0) {  
    return acc;  
  }  
  return factorial(n - 1, n * acc);  
}  
console.log(factorial(5)); //120
```



The above pattern returns the same output as the first one. But the accumulator keeps track of total as an argument without using stack memory on recursive calls.

[↑ Back to Top](#)

412. How do you check an object is a promise or not [↗](#)

If you don't know if a value is a promise or not, wrapping the value as `Promise.resolve(value)` which returns a promise

```
function isPromise(object) {  
  if (Promise && Promise.resolve) {  
    return Promise.resolve(object) == object;  
  } else {  
    throw "Promise not supported in your environment";  
  }  
}
```



```

}

var i = 1;
var promise = new Promise(function (resolve, reject) {
  resolve();
});

console.log(isPromise(i)); // false
console.log(isPromise(promise)); // true

```

Another way is to check for `.then()` handler type

```

function isPromise(value) {
  return Boolean(value && typeof value.then === "function");
}

var i = 1;
var promise = new Promise(function (resolve, reject) {
  resolve();
});

console.log(isPromise(i)); // false
console.log(isPromise(promise)); // true

```



[↑ Back to Top](#)

413. How to detect if a function is called as constructor [↗](#)

You can use `new.target` pseudo-property to detect whether a function was called as a constructor(using the new operator) or as a regular function call.

- i. If a constructor or function invoked using the new operator, `new.target` returns a reference to the constructor or function.
- ii. For function calls, `new.target` is undefined.

```

function Myfunc() {
  if (new.target) {
    console.log("called with new");
  } else {
    console.log("not called with new");
  }
}

new Myfunc(); // called with new
Myfunc(); // not called with new
Myfunc.call({}); // not called with new

```



[↑ Back to Top](#)

414. What are the differences between arguments object and rest parameter [↗](#)

There are three main differences between arguments object and rest parameters

- i. The arguments object is an array-like but not an array. Whereas the rest parameters are array instances.
- ii. The arguments object does not support methods such as sort, map, forEach, or pop. Whereas these methods can be used in rest parameters.
- iii. The rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function

[↑ Back to Top](#)

415. What are the differences between spread operator and rest parameter [↗](#)

Rest parameter collects all remaining elements into an array. Whereas Spread operator allows iterables(arrays / objects / strings) to be expanded into single arguments/elements. i.e, Rest parameter is opposite to the spread operator.

[↑ Back to Top](#)

416. What are the different kinds of generators [↗](#)

There are five kinds of generators,

i. Generator function declaration:

```
function* myGenFunc() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
const genObj = myGenFunc();
```



ii. Generator function expressions:

```
const myGenFunc = function* () {  
  yield 1;  
}
```



```
    yield 2;
    yield 3;
};
const genObj = myGenFunc();
```

iii. Generator method definitions in object literals:

```
const myObj = {
  *myGeneratorMethod() {
    yield 1;
    yield 2;
    yield 3;
  },
};
const genObj = myObj.myGeneratorMethod();
```



iv. Generator method definitions in class:

```
class MyClass {
  *myGeneratorMethod() {
    yield 1;
    yield 2;
    yield 3;
  }
}
const myObject = new MyClass();
const genObj = myObject.myGeneratorMethod();
```



v. Generator as a computed property:

```
const SomeObj = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
  },
};

console.log(Array.from(SomeObj)); // [ 1, 2, 3 ]
```



[↑ Back to Top](#)

Below are the list of built-in iterables in javascript,

- i. Arrays and TypedArrays
- ii. Strings: Iterate over each character or Unicode code-points
- iii. Maps: iterate over its key-value pairs
- iv. Sets: iterates over their elements
- v. arguments: An array-like special variable in functions
- vi. DOM collection such as NodeList

[↑ Back to Top](#)

418. What are the differences between for...of and for...in statements



Both for...in and for...of statements iterate over js data structures. The only difference is over what they iterate:

- i. for..in iterates over all enumerable property keys of an object
- ii. for..of iterates over the values of an iterable object.

Let's explain this difference with an example,

```
let arr = ["a", "b", "c"];

arr.newProp = "newVlue";

// key are the property keys
for (let key in arr) {
  console.log(key); // 0, 1, 2 & newValue
}

// value are the property values
for (let value of arr) {
  console.log(value); // a, b, c
}
```



Since for..in loop iterates over the keys of the object, the first loop logs 0, 1, 2 and newProp while iterating over the array object. The for..of loop iterates over the values of a arr data structure and logs a, b, c in the console.

[↑ Back to Top](#)

419. How do you define instance and non-instance properties [↗](#)

The Instance properties must be defined inside of class methods. For example, name and age properties defined inside constructor as below,

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  }  
}
```



But Static(class) and prototype data properties must be defined outside of the ClassBody declaration. Let's assign the age value for Person class as below,

```
Person.staticAge = 30;  
Person.prototype.prototypeAge = 40;
```



[↑ Back to Top](#)

420. What is the difference between isNaN and Number.isNaN? [↗](#)

- i. **isNaN**: The global function `isNaN` converts the argument to a Number and returns true if the resulting value is NaN.
- ii. **Number.isNaN**: This method does not convert the argument. But it returns true when the type is a Number and value is NaN.

Let's see the difference with an example,

```
isNaN('hello'); // true  
Number.isNaN('hello'); // false
```



[↑ Back to Top](#)

421. How to invoke an IIFE without any extra brackets? [↗](#)

Immediately Invoked Function Expressions(IIFE) requires a pair of parenthesis to wrap the function which contains set of statements.

```
(function (dt) {  
  console.log(dt.toLocaleTimeString());  
})(new Date());
```



Since both IIFE and void operator discard the result of an expression, you can avoid the extra brackets using void operator for IIFE as below,

```
void (function (dt) {  
  console.log(dt.toLocaleTimeString());  
})(new Date());
```



[↑ Back to Top](#)

422. Is that possible to use expressions in switch cases? [↗](#)

You might have seen expressions used in switch condition but it is also possible to use for switch cases by assigning true value for the switch condition. Let's see the weather condition based on temperature as an example,

```
const weather = (function getWeather(temp) {  
  switch (true) {  
    case temp < 0:  
      return "freezing";  
    case temp < 10:  
      return "cold";  
    case temp < 24:  
      return "cool";  
    default:  
      return "unknown";  
  }  
})(10);
```



[↑ Back to Top](#)

423. What is the easiest way to ignore promise errors? [↗](#)

The easiest and safest way to ignore promise errors is void that error. This approach is ESLint friendly too.

```
await promise.catch((e) => void e);
```



[↑ Back to Top](#)

424. How do style the console output using CSS? [↗](#)

You can add CSS styling to the console output using the CSS format content specifier %c. The console string message can be appended after the specifier and CSS style in another argument. Let's print the red the color text using console.log and CSS specifier as below,

```
console.log("%cThis is a red text", "color:red");
```



It is also possible to add more styles for the content. For example, the font-size can be modified for the above text

```
console.log(
  "%cThis is a red text with bigger font",
  "color:red; font-size:20px"
);
```



[↑ Back to Top](#)

425. What is nullish coalescing operator (??)? [↗](#)

It is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand. This can be contrasted with the logical OR (||) operator, which returns the right-hand side operand if the left operand is any falsy value, not only null or undefined.

```
console.log(null ?? true); // true
console.log(false ?? true); // false
console.log(undefined ?? true); // true
```



[↑ Back to Top](#)

426. How do you group and nest console output? [↗](#)

The console.group() can be used to group related log messages to be able to easily read the logs and use console.groupEnd() to close the group. Along with this, you can also nest groups which allows to output message in hierarchical manner.

For example, if you're logging a user's details:

```
console.group("User Details");
console.log("name: Sudheer Jonna");
console.log("job: Software Developer");
```



```
// Nested Group
console.group("Address");
console.log("Street: Commonwealth");
console.log("City: Los Angeles");
console.log("State: California");

// Close nested group
console.groupEnd();

// Close outer group
console.groupEnd();
```

You can also use `console.groupCollapsed()` instead of `console.group()` if you want the groups to be collapsed by default.

[↑ Back to Top](#)

427. What is the difference between dense and sparse arrays? [↗](#)

An array contains items at each index starting from `first(0)` to `last(array.length - 1)` is called as Dense array. Whereas if at least one item is missing at any index, the array is called as sparse.

Let's see the below two kind of arrays,

```
const avengers = ["Ironman", "Hulk", "CaptainAmerica"];
console.log(avengers[0]); // 'Ironman'
console.log(avengers[1]); // 'Hulk'
console.log(avengers[2]); // 'CaptainAmerica'
console.log(avengers.length); // 3

const justiceLeague = ["Superman", "Aquaman", , "Batman"];
console.log(justiceLeague[0]); // 'Superman'
console.log(justiceLeague[1]); // 'Aquaman'
console.log(justiceLeague[2]); // undefined
console.log(justiceLeague[3]); // 'Batman'
console.log(justiceLeague.length); // 4
```



[↑ Back to Top](#)

428. What are the different ways to create sparse arrays? [↗](#)

There are 4 different ways to create sparse arrays in JavaScript

- i. **Array literal:** Omit a value when using the array literal

```
const justiceLeague = ["Superman", "Aquaman", , "Batman"];
console.log(justiceLeague); // ['Superman', 'Aquaman', empty , 'Batman']
```



ii. **Array() constructor:** Invoking Array(length) or new Array(length)

```
const array = Array(3);
console.log(array); // [empty, empty ,empty]
```



iii. **Delete operator:** Using delete array[index] operator on the array

```
const justiceLeague = ["Superman", "Aquaman", "Batman"];
delete justiceLeague[1];
console.log(justiceLeague); // ['Superman', empty, , 'Batman']
```



iv. **Increase length property:** Increasing length property of an array

```
const justiceLeague = ["Superman", "Aquaman", "Batman"];
justiceLeague.length = 5;
console.log(justiceLeague); // ['Superman', 'Aquaman', 'Batman', empty,
```



[↑ Back to Top](#)

429. What is the difference between setTimeout, setImmediate and process.nextTick? [↗](#)

- i. **Set Timeout:** setTimeout() is to schedule execution of a one-time callback after delay milliseconds.
- ii. **Set Immediate:** The setImmediate function is used to execute a function right after the current event loop finishes.
- iii. **Process NextTick:** If process.nextTick() is called in a given phase, all the callbacks passed to process.nextTick() will be resolved before the event loop continues. This will block the event loop and create I/O Starvation if process.nextTick() is called recursively.

[↑ Back to Top](#)

430. How do you reverse an array without modifying original array? [↗](#)



The `reverse()` method reverses the order of the elements in an array but it mutates the original array. Let's take a simple example to demonstrate this case,

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reverse();

console.log(newArray); // [ 5, 4, 3, 2, 1]
console.log(originalArray); // [ 5, 4, 3, 2, 1]
```



There are few solutions that won't mutate the original array. Let's take a look.

- i. **Using slice and reverse methods:** In this case, just invoke the `slice()` method on the array to create a shallow copy followed by `reverse()` method call on the copy.

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.slice().reverse(); //Slice an array gives

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```



- ii. **Using spread and reverse methods:** In this case, let's use the spread syntax (...) to create a copy of the array followed by `reverse()` method call on the copy.

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = [...originalArray].reverse();

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```



- iii. **Using reduce and spread methods:** Here execute a reducer function on an array elements and append the accumulated array on right side using spread syntax

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reduce((accumulator, value) => {
  return [value, ...accumulator];
}, []);

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```



- iv. **Using reduceRight and spread methods:** Here execute a right reducer function(i.e. opposite direction of reduce method) on an array elements and append the accumulated array on left side using spread syntax

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reduceRight((accumulator, value) => {
  return [...accumulator, value];
}, []);

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```

- v. **Using reduceRight and push methods:** Here execute a right reducer function(i.e. opposite direction of reduce method) on an array elements and push the iterated value to the accumulator

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reduceRight((accumulator, value) => {
  accumulator.push(value);
  return accumulator;
}, []);

console.log(originalArray); // [1, 2, 3, 4, 5]
console.log(newArray); // [ 5, 4, 3, 2, 1]
```

[↑ Back to Top](#)

431. How do you create custom HTML element? [↗](#)

The creation of custom HTML elements involves two main steps,

- i. **Define your custom HTML element:** First you need to define some custom class by extending HTMLElement class. After that define your component properties (styles, text etc) using `connectedCallback` method. **Note:** The browser exposes a function called `customElements.define` in order to reuse the element.

```
class CustomElement extends HTMLElement {
  connectedCallback() {
    this.innerHTML = "This is a custom element";
  }
}
customElements.define("custom-element", CustomElement);
```

- ii. **Use custom element just like other HTML element:** Declare your custom element as a HTML tag.

```
<body>  
  <custom-element>  
</body>
```



[↑ Back to Top](#)

432. **What is global execution context?** [↗](#)

The global execution context is the default or first execution context that is created by the JavaScript engine before any code is executed (i.e, when the file first loads in the browser). All the global code that is not inside a function or object will be executed inside this global execution context. Since JS engine is single threaded there will be only one global environment and there will be only one global execution context.

For example, the below code other than code inside any function or object is executed inside the global execution context.

```
var x = 10;  
  
function A() {  
  console.log("Start function A");  
  
  function B() {  
    console.log("In function B");  
  }  
  
  B();  
}  
  
A();  
  
console.log("GlobalContext");
```



[↑ Back to Top](#)

433. **What is function execution context?** [↗](#)

Whenever a function is invoked, the JavaScript engine creates a different type of Execution Context known as a Function Execution Context (FEC) within the Global Execution Context (GEC) to evaluate and execute the code within that function.

434. What is debouncing? [↗](#)

Debouncing is a programming pattern that allows delaying execution of some piece of code until a specified time to avoid unnecessary *CPU cycles, API calls and improve performance*. The debounce function make sure that your code is only triggered once per user input. The common usecases are Search box suggestions, text-field auto-saves, and eliminating double-button clicks.

Let's say you want to show suggestions for a search query, but only after a visitor has finished typing it. So here you write a debounce function where the user keeps writing the characters with in 500ms then previous timer cleared out using `clearTimeout` and reschedule API call/DB query for a new time—300 ms in the future.

```
function debounce(func, timeout = 500) {  
  let timer;  
  return (...args) => {  
    clearTimeout(timer);  
    timer = setTimeout(() => {  
      func.apply(this, args);  
    }, timeout);  
  };  
}  
  
function fetchResults() {  
  console.log("Fetching input suggestions");  
}  
  
const processChange = debounce(() => fetchResults());
```



The `debounce()` function can be used on input, button and window events

Input:

```
<input type="text" onkeyup="processChange()" />
```



Button:

```
<button onclick="processChange()">Click me</button>
```



Windows event:


```
window.addEventListener("scroll", processChange);
```



[↑ Back to Top](#)

435. What is throttling? [↗](#)

Throttling is a technique used to limit the execution of an event handler function, even when this event triggers continuously due to user actions. The common use cases are browser resizing, window scrolling etc.

The below example creates a throttle function to reduce the number of events for each pixel change and trigger scroll event for each 100ms except for the first event.

```
const throttle = (func, limit) => {  
  let inThrottle;  
  return (...args) => {  
    if (!inThrottle) {  
      func.apply(this, args);  
      inThrottle = true;  
      setTimeout(() => (inThrottle = false), limit);  
    }  
  };  
};  
window.addEventListener("scroll", () => {  
  throttle(handleScrollAnimation, 100);  
});
```



[↑ Back to Top](#)

436. What is optional chaining? [↗](#)

According to MDN official docs, the optional chaining operator (?.) permits reading the value of a property located deep within a chain of connected objects without having to expressly validate that each reference in the chain is valid.

The ?. operator is like the . chaining operator, except that instead of causing an error if a reference is nullish (null or undefined), the expression short-circuits with a return value of undefined. When used with function calls, it returns undefined if the given function does not exist.

```
const adventurer = {  
  name: "Alice",
```



```

    cat: {
      name: "Dinah",
    },
  };

const dogName = adventurer.dog?.name;
console.log(dogName);
// expected output: undefined

console.log(adventurer.someNonExistentMethod?.());
// expected output: undefined

```

[↑ Back to Top](#)

437. What is an environment record? [↗](#)

According to ECMAScript specification 262 (9.1):

Environment Record is a specification type used to define the association of Identifiers to specific variables and functions, based upon the lexical nesting structure of ECMAScript code.

Usually an Environment Record is associated with some specific syntactic structure of ECMAScript code such as a FunctionDeclaration, a BlockStatement, or a Catch clause of a TryStatement.

Each time such code is evaluated, a new Environment Record is created to record the identifier bindings that are created by that code.

[↑ Back to Top](#)

438. How to verify if a variable is an array? [↗](#)

It is possible to check if a variable is an array instance using 3 different ways,

i. `Array.isArray()` method:

The `Array.isArray(value)` utility function is used to determine whether value is an array or not. This function returns a true boolean value if the variable is an array and a false value if it is not.

```

const numbers = [1, 2, 3];
const user = { name: "John" };
Array.isArray(numbers); // true
Array.isArray(user); //false

```



ii. instanceof operator:

The instanceof operator is used to check the type of an array at run time. It returns true if the type of a variable is an Array other false for other type.

```
const numbers = [1, 2, 3];
const user = { name: "John" };
console.log(numbers instanceof Array); // true
console.log(user instanceof Array); // false
```



iii. Checking constructor type:

The constructor property of the variable is used to determine whether the variable Array type or not.

```
const numbers = [1, 2, 3];
const user = { name: "John" };
console.log(numbers.constructor === Array); // true
console.log(user.constructor === Array); // false
```



[↑ Back to Top](#)

439. What is pass by value and pass by reference? [↗](#)

Pass-by-value creates a new space in memory and makes a copy of a value. Primitives such as string, number, boolean etc will actually create a new copy. Hence, updating one value doesn't impact the other value. i.e, The values are independent of each other.

```
let a = 5;
let b = a;

b++;
console.log(a, b); //5, 6
```



In the above code snippet, the value of `a` is assigned to `b` and the variable `b` has been incremented. Since there is a new space created for variable `b`, any update on this variable doesn't impact the variable `a`.

Pass by reference doesn't create a new space in memory but the new variable adopts a memory address of an initial variable. Non-primitives such as objects, arrays and functions gets the reference of the initial variable. i.e, updating one value will impact the other variable.

```
let user1 = {  
  name: "John",  
  age: 27,  
};  
let user2 = user1;  
user2.age = 30;  
  
console.log(user1.age, user2.age); // 30, 30
```



In the above code snippet, updating the `age` property of one object will impact the other property due to the same reference.

[↑ Back to Top](#)

440. What are the differences between primitives and non-primitives? [↗](#)

JavaScript language has both primitives and non-primitives but there are few differences between them as below,

Primitives	Non-primitives
These types are predefined	Created by developer
These are immutable	Mutable
Compare by value	Compare by reference
Stored in Stack	Stored in heap
Contain certain value	Can contain NULL too

[↑ Back to Top](#)

441. How do you create your own bind method using either call or apply method? [↗](#)

The custom bind function needs to be created on Function prototype inorder to use it as other builtin functions. This custom function should return a function similar to original bind method and the implementation of inner function needs to use apply method call.

The function which is going to bind using custom `myOwnBind` method act as the attached function(`boundTargetFunction`) and argument as the object for `apply` method call.

```
Function.prototype.myOwnBind = function (whoIsCallingMe) {  
  if (typeof this !== "function") {  
    throw new Error(this + "cannot be bound as it's not callable");  
  }  
  const boundTargetFunction = this;  
  return function () {  
    boundTargetFunction.apply(whoIsCallingMe, arguments);  
  };  
};
```



[↑ Back to Top](#)

442. What are the differences between pure and impure functions? [↗](#)

Some of the major differences between pure and impure function are as below,

Pure function	Impure function
It has no side effects	It causes side effects
It is always return the same result	It returns different result on each call
Easy to read and debug	Difficult to read and debug because they are affected by extenal code

[↑ Back to Top](#)

445. What is referential transparency? [↗](#)

An expression in javascript that can be replaced by its value without affecting the behaviour of the program is called referential transparency. Pure functions are referentially transparent.

```
const add = (x, y) => x + y;
const multiplyBy2 = (x) => x * 2;
```



//Now add (2, 3) can be replaced by 5.

```
multiplyBy2(add(2, 3));
```

[↑ Back to Top](#)

446. What are the possible side-effects in javascript? [↗](#)

A side effect is the modification of the state through the invocation of a function or expression. These side effects make our function impure by default. Below are some side effects which make function impure,

- Making an HTTP request. Asynchronous functions such as fetch and promise are impure.
- DOM manipulations
- Mutating the input data
- Printing to a screen or console: For example, console.log() and alert()
- Fetching the current time
- Math.random() calls: Modifies the internal state of Math object

[↑ Back to Top](#)

447. What are compose and pipe functions? [↗](#)

The "compose" and "pipe" are two techniques commonly used in functional programming to simplify complex operations and make code more readable. They are not native to JavaScript and higher-order functions. the `compose()` applies right to left any number of functions to the output of the previous function.

[↑ Back to Top](#)

448. What is module pattern? [↗](#)

Module pattern is a designed pattern used to wrap a set of variables and functions together in a single scope returned as an object. JavaScript doesn't have access specifiers similar to other languages(Java, Python, etc) to provide private scope. It uses IIFE (Immediately invoked function expression) to allow for private scopes. i.e., a closure that protect variables and methods.

The module pattern looks like below,

```
(function () {  
  // Private variables or functions goes here.  
  
  return {  
    // Return public variables or functions here.  
  };  
})();
```



Let's see an example of a module pattern for an employee with private and public access,

```
const createEmployee = (function () {  
  // Private  
  const name = "John";  
  const department = "Sales";  
  const getEmployeeName = () => name;  
  const getDepartmentName = () => department;  
  
  // Public  
  return {  
    name,  
    department,  
    getName: () => getEmployeeName(),  
    getDepartment: () => getDepartmentName(),  
  };  
})();  
  
console.log(createEmployee.name);  
console.log(createEmployee.department);  
console.log(createEmployee.getName());  
console.log(createEmployee.getDepartment());
```



Note: It mimics the concepts of classes with private variables and methods.

[↑ Back to Top](#)

449. What is Function Composition? [↗](#)

It is an approach where the result of one function is passed on to the next function, which is passed to another until the final function is executed for the final result.

```
//example  
const double = (x) => x * 2;
```



```
const square = (x) => x * x;

var output1 = double(2);
var output2 = square(output1);
console.log(output2);

var output_final = square(double(2));
console.log(output_final);
```

[↑ Back to Top](#)

450. How to use await outside of async function prior to ES2022? [↗](#)

Prior to ES2022, if you attempted to use an await outside of an async function resulted in a SyntaxError.

```
await Promise.resolve(console.log("Hello await")); // SyntaxError: await
```

But you can fix this issue with an alternative IIFE (Immediately Invoked Function Expression) to get access to the feature.

```
(async function () {
  await Promise.resolve(console.log("Hello await")); // Hello await
})();
```

In ES2022, you can write top-level await without writing any hacks.

```
await Promise.resolve(console.log("Hello await")); //Hello await
```

[↑ Back to Top](#)

Coding Exercise [↗](#)

1. What is the output of below code [↗](#)

```
var car = new Vehicle("Honda", "white", "2010", "UK");
console.log(car);

function Vehicle(model, color, year, country) {
  this.model = model;
  this.color = color;
```



```
this.year = year;
this.country = country;
}
```

- 1: Undefined
- 2: ReferenceError
- 3: null
- 4: {model: "Honda", color: "white", year: "2010", country: "UK"}

► Answer

[↑ Back to Top](#)

2. What is the output of below code [↗](#)

```
function foo() {
  let x = (y = 0);
  x++;
  y++;
  return x;
}
```

```
console.log(foo(), typeof x, typeof y);
```

- 1: 1, undefined and undefined
- 2: ReferenceError: X is not defined
- 3: 1, undefined and number
- 4: 1, number and number

► Answer

[↑ Back to Top](#)

3. What is the output of below code [↗](#)

```
function main() {
  console.log("A");
  setTimeout(function print() {
    console.log("B");
  }, 0);
}
```

```
    console.log("C");  
  }  
  main();
```

- 1: A, B and C
- 2: B, A and C
- 3: A and C
- 4: A, C and B

► Answer

[↑ Back to Top](#)

4. What is the output of below equality check [↗](#)

```
console.log(0.1 + 0.2 === 0.3);
```



- 1: false
- 2: true

► Answer

[↑ Back to Top](#)

5. What is the output of below code [↗](#)

```
var y = 1;  
if (function f() {}) {  
  y += typeof f;  
}  
console.log(y);
```



- 1: 1function
- 2: 1object
- 3: ReferenceError
- 4: 1undefined

► Answer

[↑ Back to Top](#)

6. What is the output of below code [↗](#)

```
function foo() {  
  return;  
  {  
    message: "Hello World";  
  }  
}  
console.log(foo());
```



- 1: Hello World
- 2: Object {message: "Hello World"}
- 3: Undefined
- 4: SyntaxError

► Answer

[↑ Back to Top](#)

7. What is the output of below code [↗](#)

```
var myChars = ["a", "b", "c", "d"];  
delete myChars[0];  
console.log(myChars);  
console.log(myChars[0]);  
console.log(myChars.length);
```



- 1: [empty, 'b', 'c', 'd'], empty, 3
- 2: [null, 'b', 'c', 'd'], empty, 3
- 3: [empty, 'b', 'c', 'd'], undefined, 4
- 4: [null, 'b', 'c', 'd'], undefined, 4

► Answer

[↑ Back to Top](#)

8. What is the output of below code in latest Chrome [↗](#)

```
var array1 = new Array(3);  
console.log(array1);
```

```
var array2 = [];  
array2[2] = 100;  
console.log(array2);
```

```
var array3 = [, , ,];  
console.log(array3);
```

- 1: [undefined × 3], [undefined × 2, 100], [undefined × 3]
- 2: [empty × 3], [empty × 2, 100], [empty × 3]
- 3: [null × 3], [null × 2, 100], [null × 3]
- 4: [], [100], []

► Answer

[↑ Back to Top](#)

9. What is the output of below code [↗](#)

```
const obj = {  
  prop1: function () {  
    return 0;  
  },  
  prop2() {  
    return 1;  
  },  
  ["prop" + 3]() {  
    return 2;  
  },  
};
```

```
console.log(obj.prop1());  
console.log(obj.prop2());  
console.log(obj.prop3());
```

- 1: 0, 1, 2
- 2: 0, { return 1 }, 2
- 3: 0, { return 1 }, { return 2 }
- 4: 0, 1, undefined

► Answer

[↑ Back to Top](#)

10. What is the output of below code [↗](#)

```
console.log(1 < 2 < 3);  
console.log(3 > 2 > 1);
```



- 1: true, true
- 2: true, false
- 3: SyntaxError, SyntaxError,
- 4: false, false

► Answer

[↑ Back to Top](#)

11. What is the output of below code in non-strict mode [↗](#)

```
function printNumbers(first, second, first) {  
  console.log(first, second, first);  
}  
printNumbers(1, 2, 3);
```



- 1: 1, 2, 3
- 2: 3, 2, 3
- 3: SyntaxError: Duplicate parameter name not allowed in this context
- 4: 1, 2, 1

► Answer

[↑ Back to Top](#)

12. What is the output of below code [↗](#)

```
const printNumbersArrow = (first, second, first) => {  
  console.log(first, second, first);  
};  
printNumbersArrow(1, 2, 3);
```



- 1: 1, 2, 3
- 2: 3, 2, 3
- 3: SyntaxError: Duplicate parameter name not allowed in this context
- 4: 1, 2, 1

► Answer

[↑ Back to Top](#)

13. What is the output of below code [↗](#)

```
const arrowFunc = () => arguments.length;  
console.log(arrowFunc(1, 2, 3));
```



- 1: ReferenceError: arguments is not defined
- 2: 3
- 3: undefined
- 4: null

► Answer

[↑ Back to Top](#)

14. What is the output of below code [↗](#)

```
console.log(String.prototype.trimLeft.name === "trimLeft");  
console.log(String.prototype.trimLeft.name === "trimStart");
```



- 1: True, False
- 2: False, True

► Answer

[↑ Back to Top](#)

15. What is the output of below code [↗](#)

```
console.log(Math.max());
```



- 1: undefined
- 2: Infinity
- 3: 0
- 4: -Infinity

► Answer

[↑ Back to Top](#)

16. What is the output of below code [↗](#)

```
console.log(10 == [10]);  
console.log(10 == [[[[[[[10]]]]]]]);
```



- 1: True, True
- 2: True, False
- 3: False, False
- 4: False, True

► Answer

[↑ Back to Top](#)

17. What is the output of below code [↗](#)

```
console.log(10 + "10");  
console.log(10 - "10");
```



- 1: 20, 0
- 2: 1010, 0
- 3: 1010, 10-10

- 4: NaN, NaN

► Answer

[↑ Back to Top](#)

18. What is the output of below code [↗](#)

```
console.log([0] == false);  
if ([0]) {  
  console.log("I'm True");  
} else {  
  console.log("I'm False");  
}
```



- 1: True, I'm True
- 2: True, I'm False
- 3: False, I'm True
- 4: False, I'm False

► Answer

19. What is the output of below code [↗](#)

```
console.log([1, 2] + [3, 4]);
```



- 1: [1,2,3,4]
- 2: [1,2][3,4]
- 3: SyntaxError
- 4: 1,23,4

► Answer

[↑ Back to Top](#)

20. What is the output of below code [↗](#)

```
const numbers = new Set([1, 1, 2, 3, 4]);  
console.log(numbers);
```




```
const browser = new Set("Firefox");  
console.log(browser);
```

- 1: {1, 2, 3, 4}, {"F", "i", "r", "e", "f", "o", "x"}
- 2: {1, 2, 3, 4}, {"F", "i", "r", "e", "o", "x"}
- 3: [1, 2, 3, 4], ["F", "i", "r", "e", "o", "x"]
- 4: {1, 1, 2, 3, 4}, {"F", "i", "r", "e", "f", "o", "x"}

► Answer

[↑ Back to Top](#)

21. What is the output of below code [↗](#)

```
console.log(NaN === NaN);
```



- 1: True
- 2: False

► Answer

[↑ Back to Top](#)

22. What is the output of below code [↗](#)

```
let numbers = [1, 2, 3, 4, NaN];  
console.log(numbers.indexOf(NaN));
```



- 1: 4
- 2: NaN
- 3: SyntaxError
- 4: -1

► Answer

[↑ Back to Top](#)

23. What is the output of below code [↗](#)

```
let [a, ...b,] = [1, 2, 3, 4, 5];  
console.log(a, b);
```



- 1: 1, [2, 3, 4, 5]
- 2: 1, {2, 3, 4, 5}
- 3: SyntaxError
- 4: 1, [2, 3, 4]

► Answer

[↑ Back to Top](#)

25. What is the output of below code [↗](#)

```
async function func() {  
  return 10;  
}  
console.log(func());
```



- 1: Promise {<fulfilled>: 10}
- 2: 10
- 3: SyntaxError
- 4: Promise {<rejected>: 10}

► Answer

[↑ Back to Top](#)

26. What is the output of below code [↗](#)

```
async function func() {  
  await 10;  
}  
console.log(func());
```



- 1: Promise {<fulfilled>: 10}

- 2: 10
- 3: SyntaxError
- 4: Promise {<resolved>: undefined}

► Answer

[↑ Back to Top](#)

27. What is the output of below code [↗](#)

```
function delay() {  
  return new Promise(resolve => setTimeout(resolve, 2000));  
}  
  
async function delayedLog(item) {  
  await delay();  
  console.log(item);  
}  
  
async function processArray(array) {  
  array.forEach(item => {  
    await delayedLog(item);  
  })  
}  
  
processArray([1, 2, 3, 4]);
```



- 1: SyntaxError
- 2: 1, 2, 3, 4
- 3: 4, 4, 4, 4
- 4: 4, 3, 2, 1

► Answer

[↑ Back to Top](#)

28. What is the output of below code [↗](#)

```
function delay() {  
  return new Promise((resolve) => setTimeout(resolve, 2000));  
}
```



```

async function delayedLog(item) {
  await delay();
  console.log(item);
}

async function process(array) {
  array.forEach(async (item) => {
    await delayedLog(item);
  });
  console.log("Process completed!");
}
process([1, 2, 3, 5]);

```

- 1: 1 2 3 5 and Process completed!
- 2: 5 5 5 5 and Process completed!
- 3: Process completed! and 5 5 5 5
- 4: Process completed! and 1 2 3 5

► Answer

[↑ Back to Top](#)

29. What is the output of below code [↗](#)

```

var set = new Set();
set.add("+0").add("-0").add(NaN).add(undefined).add(NaN);
console.log(set);

```



- 1: Set(4) {"+0", "-0", NaN, undefined}
- 2: Set(3) {"+0", NaN, undefined}
- 3: Set(5) {"+0", "-0", NaN, undefined, NaN}
- 4: Set(4) {"+0", NaN, undefined, NaN}

► Answer

[↑ Back to Top](#)

30. What is the output of below code [↗](#)

```
const sym1 = Symbol("one");
const sym2 = Symbol("one");

const sym3 = Symbol.for("two");
const sym4 = Symbol.for("two");

console.log(sym1 === sym2, sym3 === sym4);
```



- 1: true, true
- 2: true, false
- 3: false, true
- 4: false, false

► Answer

[↑ Back to Top](#)

31. What is the output of below code [↗](#)

```
const sym1 = new Symbol("one");
console.log(sym1);
```



- 1: SyntaxError
- 2: one
- 3: Symbol('one')
- 4: Symbol

► Answer

[↑ Back to Top](#)

32. What is the output of below code [↗](#)

```
let myNumber = 100;
let myString = "100";

if (!typeof myNumber === "string") {
  console.log("It is not a string!");
} else {
  console.log("It is a string!");
}
```



```

}

if (!typeof myString === "number") {
  console.log("It is not a number!");
} else {
  console.log("It is a number!");
}

```

- 1: SyntaxError
- 2: It is not a string!, It is not a number!
- 3: It is not a string!, It is a number!
- 4: It is a string!, It is a number!

► Answer

[↑ Back to Top](#)

33. What is the output of below code [↗](#)

```

console.log(
  JSON.stringify({ myArray: ["one", undefined, function () {}, Symbol("")] })
);
console.log(
  JSON.stringify({ [Symbol.for("one")]: "one" }, [Symbol.for("one")])
);

```

- 1: {"myArray":["one', undefined, {}, Symbol]], {}
- 2: {"myArray":["one', null,null,null]], {}
- 3: {"myArray":["one', null,null,null]], "{ [Symbol.for('one')]: 'one' }, [Symbol.for('one')]"
- 4: {"myArray":["one', undefined, function() {}, Symbol(")], {}

► Answer

[↑ Back to Top](#)

34. What is the output of below code [↗](#)

```

class A {
  constructor() {
    console.log(new.target.name);
  }
}

```

```

    }
}

class B extends A {
    constructor() {
        super();
    }
}

```

```

new A();
new B();

```

- 1: A, A
- 2: A, B

► Answer

[↑ Back to Top](#)

35. What is the output of below code [↗](#)

```

const [x, ...y, z] = [1, 2, 3, 4];
console.log(x, y, z);

```



- 1: 1, [2, 3], 4
- 2: 1, [2, 3, 4], undefined
- 3: 1, [2], 3
- 4: SyntaxError

► Answer

[↑ Back to Top](#)

36. What is the output of below code [↗](#)

```

const { a: x = 10, b: y = 20 } = { a: 30 };

console.log(x);
console.log(y);

```



- 1: 30, 20
- 2: 10, 20
- 3: 10, undefined
- 4: 30, undefined

► Answer

[↑ Back to Top](#)

37. What is the output of below code [↗](#)

```
function area({ length = 10, width = 20 }) {  
  console.log(length * width);  
}
```



```
area();
```

- 1: 200
- 2: Error
- 3: undefined
- 4: 0

► Answer

[↑ Back to Top](#)

38. What is the output of below code [↗](#)

```
const props = [  
  { id: 1, name: "John" },  
  { id: 2, name: "Jack" },  
  { id: 3, name: "Tom" },  
];  
  
const [, , { name }] = props;  
console.log(name);
```



- 1: Tom
- 2: Error

- 3: undefined
- 4: John

► Answer

[↑ Back to Top](#)

39. What is the output of below code [↗](#)

```
function checkType(num = 1) {  
  console.log(typeof num);  
}
```



```
checkType();  
checkType(undefined);  
checkType("");  
checkType(null);
```

- 1: number, undefined, string, object
- 2: undefined, undefined, string, object
- 3: number, number, string, object
- 4: number, number, number, number

► Answer

[↑ Back to Top](#)

40. What is the output of below code [↗](#)

```
function add(item, items = []) {  
  items.push(item);  
  return items;  
}
```



```
console.log(add("Orange"));  
console.log(add("Apple"));
```

- 1: ['Orange'], ['Orange', 'Apple']
- 2: ['Orange'], ['Apple']

► Answer

[↑ Back to Top](#)

41. What is the output of below code [↗](#)

```
function greet(greeting, name, message = greeting + " " + name) {  
  console.log([greeting, name, message]);  
}  
  
greet("Hello", "John");  
greet("Hello", "John", "Good morning!");
```



- 1: SyntaxError
- 2: ['Hello', 'John', 'Hello John'], ['Hello', 'John', 'Good morning!']

► Answer

[↑ Back to Top](#)

42. What is the output of below code [↗](#)

```
function outer(f = inner()) {  
  function inner() {  
    return "Inner";  
  }  
}  
outer();
```



- 1: ReferenceError
- 2: Inner

► Answer

[↑ Back to Top](#)

43. What is the output of below code [↗](#)

```
function myFun(x, y, ...manyMoreArgs) {  
  console.log(manyMoreArgs);  
}
```



```
myFun(1, 2, 3, 4, 5);  
myFun(1, 2);
```

- 1: [3, 4, 5], undefined
- 2: SyntaxError
- 3: [3, 4, 5], []
- 4: [3, 4, 5], [undefined]

► Answer

[↑ Back to Top](#)

44. What is the output of below code [↗](#)

```
const obj = { key: "value" };  
const array = [...obj];  
console.log(array);
```



- 1: ['key', 'value']
- 2: TypeError
- 3: []
- 4: ['key']

► Answer

[↑ Back to Top](#)

45. What is the output of below code [↗](#)

```
function* myGenFunc() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
var myGenObj = new myGenFunc();  
console.log(myGenObj.next().value);
```



- 1: 1
- 2: undefined

- 3: SyntaxError
- 4: TypeError

► Answer

[↑ Back to Top](#)

46. What is the output of below code [↗](#)

```
function* yieldAndReturn() {  
  yield 1;  
  return 2;  
  yield 3;  
}  
  
var myGenObj = yieldAndReturn();  
console.log(myGenObj.next());  
console.log(myGenObj.next());  
console.log(myGenObj.next());
```



- 1: { value: 1, done: false }, { value: 2, done: true }, { value: undefined, done: true }
- 2: { value: 1, done: false }, { value: 2, done: false }, { value: undefined, done: true }
- 3: { value: 1, done: false }, { value: 2, done: true }, { value: 3, done: true }
- 4: { value: 1, done: false }, { value: 2, done: false }, { value: 3, done: true }

► Answer

[↑ Back to Top](#)

47. What is the output of below code [↗](#)

```
const myGenerator = (function* () {  
  yield 1;  
  yield 2;  
  yield 3;  
})();  
for (const value of myGenerator) {  
  console.log(value);  
  break;  
}  
  
for (const value of myGenerator) {
```



```
    console.log(value);  
  }  
}
```

- 1: 1,2,3 and 1,2,3
- 2: 1,2,3 and 4,5,6
- 3: 1 and 1
- 4: 1

► Answer

[↑ Back to Top](#)

48. What is the output of below code [↗](#)

```
const num = 0o38;  
console.log(num);
```



- 1: SyntaxError
- 2: 38

► Answer

[↑ Back to Top](#)

49. What is the output of below code [↗](#)

```
const squareObj = new Square(10);  
console.log(squareObj.area);  
  
class Square {  
  constructor(length) {  
    this.length = length;  
  }  
  
  get area() {  
    return this.length * this.length;  
  }  
  
  set area(value) {  
    this.area = value;  
  }  
}
```



```
}  
}
```

- 1: 100
- 2: ReferenceError

► Answer

[↑ Back to Top](#)

50. What is the output of below code [↗](#)

```
function Person() {}  
  
Person.prototype.walk = function () {  
  return this;  
};  
  
Person.run = function () {  
  return this;  
};  
  
let user = new Person();  
let walk = user.walk;  
console.log(walk());  
  
let run = Person.run;  
console.log(run());
```



- 1: undefined, undefined
- 2: Person, Person
- 3: SyntaxError
- 4: Window, Window

► Answer

[↑ Back to Top](#)

51. What is the output of below code [↗](#)



```
class Vehicle {
  constructor(name) {
    this.name = name;
  }

  start() {
    console.log(`${this.name} vehicle started`);
  }
}

class Car extends Vehicle {
  start() {
    console.log(`${this.name} car started`);
    super.start();
  }
}

const car = new Car("BMW");
console.log(car.start());
```

- 1: SyntaxError
- 2: BMW vehicle started, BMW car started
- 3: BMW car started, BMW vehicle started
- 4: BMW car started, BMW car started

► Answer

[↑ Back to Top](#)

52. What is the output of below code [↗](#)



```
const USER = { age: 30 };
USER.age = 25;
console.log(USER.age);
```

- 1: 30
- 2: 25
- 3: Uncaught TypeError
- 4: SyntaxError

► Answer

[↑ Back to Top](#)

53. What is the output of below code [↗](#)

```
console.log("😊" === "😊");
```



- 1: false
- 2: true

► Answer

[↑ Back to Top](#)

54. What is the output of below code? [↗](#)

```
console.log(typeof typeof typeof true);
```



- 1: string
- 2: boolean
- 3: NaN
- 4: number

► Answer

[↑ Back to Top](#)

55. What is the output of below code? [↗](#)

```
let zero = new Number(0);
```

```
if (zero) {  
  console.log("If");  
} else {  
  console.log("Else");  
}
```



- 1: If
- 2: Else

- 3: NaN
- 4: SyntaxError

► Answer

[↑ Back to Top](#)

55. What is the output of below code in non strict mode? [↗](#)

```
let msg = "Good morning!!";  
  
msg.name = "John";  
  
console.log(msg.name);
```



- 1: ""
- 2: Error
- 3: John
- 4: Undefined

► Answer

[↑ Back to Top](#)

56. What is the output of below code? [↗](#)

```
let count = 10;  
  
(function innerFunc() {  
  if (count === 10) {  
    let count = 11;  
    console.log(count);  
  }  
  console.log(count);  
})();
```



- 1: 11, 10
- 2: 11, 11
- 3: 10, 11
- 4: 10, 10

► Answer

[↑ Back to Top](#)

57. What is the output of below code ? [↗](#)

- 1: console.log(true && 'hi');
- 2: console.log(true && 'hi' && 1);
- 3: console.log(true && " " && 0);

► Answer

[↑ Back to Top](#)

58. What is the output of below code ? [↗](#)

```
let arr = [1, 2, 3];  
let str = "1,2,3";  
  
console.log(arr == str);
```



- 1: false
- 2: Error
- 3: true

► Answer

[↑ Back to Top](#)

59. What is the output of below code? [↗](#)

```
getMessage();  
  
var getMessage = () => {  
  console.log("Good morning");  
};
```



- 1: Good morning

- 2: getMessage is not a function
- 3: getMessage is not defined
- 4: Undefined

► Answer

[↑ Back to Top](#)

60. What is the output of below code? [↗](#)

```
let quickPromise = Promise.resolve();

quickPromise.then(() => console.log("promise finished"));

console.log("program finished");
```



- 1: program finished
- 2: Cannot predict the order
- 3: program finished, promise finished
- 4: promise finished, program finished

► Answer

[↑ Back to Top](#)

61. What is the output of below code? [↗](#)

```
console
  .log("First line")
  [("a", "b", "c")].forEach((element) => console.log(element));
console.log("Third line");
```



- 1: First line , then print a, b, c in a new line, and finally print Third line as next line
- 2: First line , then print a, b, c in a first line, and print Third line as next line
- 3: Missing semi-colon error
- 4: Cannot read properties of undefined

► Answer

[↑ Back to Top](#)

62. Write a function that returns a random HEX color [↗](#)

► Solution 1 (Iterative generation)

► Solution 2 (One-liner)

[↑ Back to Top](#)

63. What is the output of below code? [↗](#)

```
var of = ["of"];  
for (var of of of) {  
  console.log(of);  
}
```



- 1: of
- 2: SyntaxError: Unexpected token of
- 3: SyntaxError: Identifier 'of' has already been declared
- 4: ReferenceError: of is not defined

► Answer

[↑ Back to Top](#)

64. What is the output of below code? [↗](#)

```
const numbers = [11, 25, 31, 23, 33, 18, 200];  
numbers.sort();  
console.log(numbers);
```



- 1: [11, 18, 23, 25, 31, 33, 200]
- 2: [11, 18, 200, 23, 25, 31, 33]
- 3: [11, 25, 31, 23, 33, 18, 200]
- 4: Cannot sort numbers

► Answer

[↑ Back to Top](#)

65. What is the output order of below code? [↗](#)

```
setTimeout(() => {  
  console.log("1");  
}, 0);  
Promise.resolve("hello").then(() => console.log("2"));  
console.log("3");
```



- 1: 1, 2, 3
- 2: 1, 3, 2
- 3: 3, 1, 2
- 4: 3, 2, 1

► Answer

[↑ Back to Top](#)

66. What is the output of below code? [↗](#)

```
console.log(name);  
console.log(message());  
var name = "John";  
(function message() {  
  console.log("Hello John: Welcome");  
});
```



- 1: John, Hello John: Welcome
- 2: undefined, Hello John, Welcome
- 3: Reference error: name is not defined, Reference error: message is not defined
- 4: undefined, Reference error: message is not defined

► Answer

[↑ Back to Top](#)

67. What is the output of below code? [↗](#)

```
message();
```



```
function message() {  
  console.log("Hello");  
}  
function message() {  
  console.log("Bye");  
}
```

- 1: Reference error: message is not defined
- 2: Hello
- 3: Bye
- 4: Compile time error

► Answer

[↑ Back to Top](#)

68. What is the output of below code? [↗](#)

```
var currentCity = "NewYork";  
  
var changeCurrentCity = function () {  
  console.log("Current City:", currentCity);  
  var currentCity = "Singapore";  
  console.log("Current City:", currentCity);  
};  
  
changeCurrentCity();
```



- 1: NewYork, Singapore
- 2: NewYork, NewYork
- 3: undefined, Singapore
- 4: Singapore, Singapore

► Answer

[↑ Back to Top](#)

69. What is the output of below code in an order? [↗](#)



```
function second() {  
  var message;  
  console.log(message);  
}  
  
function first() {  
  var message = "first";  
  second();  
  console.log(message);  
}  
  
var message = "default";  
first();  
console.log(message);
```

- 1: undefined, first, default
- 2: default, default, default
- 3: first, first, default
- 4: undefined, undefined, undefined

► Answer

[↑ Back to Top](#)

70. What is the output of below code? [↗](#)

```
var expressionOne = function functionOne() {  
  console.log("functionOne");  
};  
functionOne();
```



- 1: functionOne is not defined
- 2: functionOne
- 3: console.log("functionOne")
- 4: undefined

► Answer

[↑ Back to Top](#)

71. What is the output of below code? [↗](#)

```
const user = {  
  name: "John",  
  eat() {  
    console.log(this);  
    var eatFruit = function () {  
      console.log(this);  
    };  
    eatFruit();  
  },  
};  
user.eat();
```



- 1: {name: "John", eat: f}, {name: "John", eat: f}
- 2: Window {...}, Window {...}
- 3: {name: "John", eat: f}, undefined
- 4: {name: "John", eat: f}, Window {...}

► Answer

[↑ Back to Top](#)

72. What is the output of below code? [↗](#)

```
let message = "Hello World!";  
message[0] = "J";  
console.log(message);  
  
let name = "John";  
name = name + " Smith";  
console.log(name);
```



- 1: Jello World!, John Smith
- 2: Jello World!, John
- 3: Hello World!, John Smith
- 4: Hello World!, John

► Answer

[↑ Back to Top](#)

73. What is the output of below code? [↗](#)

```
let user1 = {  
  name: "Jacob",  
  age: 28,  
};  
  
let user2 = {  
  name: "Jacob",  
  age: 28,  
};  
  
console.log(user1 === user2);
```



- 1: True
- 2: False
- 3: Compile time error

► Answer

[↑ Back to Top](#)

74. What is the output of below code? [↗](#)

```
function greeting() {  
  setTimeout(function () {  
    console.log(message);  
  }, 5000);  
  const message = "Hello, Good morning";  
}  
greeting();
```



- 1: Undefined
- 2: Reference error:
- 3: Hello, Good morning
- 4: null

► Answer

[↑ Back to Top](#)

75. What is the output of below code? [↗](#)

```
const a = new Number(10);  
const b = 10;  
console.log(a === b);
```



- 1: False
- 2: True

► Answer

[↑ Back to Top](#)

76. What is the type of below function? [↗](#)

```
function add(a, b) {  
  console.log("The input arguments are: ", a, b);  
  return a + b;  
}
```



- 1: Pure function
- 2: Impure function

► Answer

[↑ Back to Top](#)

77. What is the output of below code? [↗](#)

```
const promiseOne = new Promise((resolve, reject) => setTimeout(resolve, 400  
const promiseTwo = new Promise((resolve, reject) => setTimeout(reject, 4000)));
```



```
Promise.all([promiseOne, promiseTwo]).then((data) => console.log(data));
```



- 1: [{status: "fullfilled", value: undefined}, {status: "rejected", reason: undefined}]
- 2: [{status: "fullfilled", value: undefined}, Uncaught(in promise)]
- 3: Uncaught (in promise)
- 4: [Uncaught(in promise), Uncaught(in promise)]

► Answer

[↑ Back to Top](#)

78. What is the output of below code? [↗](#)

```
try {  
  setTimeout(() => {  
    console.log("try block");  
    throw new Error(`An exception is thrown`);  
  }, 1000);  
} catch (err) {  
  console.log("Error: ", err);  
}
```



- 1: try block, Error: An exception is thrown
- 2: Error: An exception is thrown
- 3: try block, Uncaught Error: Exception is thrown
- 4: Uncaught Error: Exception is thrown

► Answer

[↑ Back to Top](#)

79. What is the output of below code? [↗](#)

```
let a = 10;  
if (true) {  
  let a = 20;  
  console.log(a, "inside");  
}  
console.log(a, "outside");
```



- 1: 20, "inside" and 20, "outside"
- 2: 20, "inside" and 10, "outside"
- 3: 10, "inside" and 10, "outside"
- 4: 10, "inside" and 20, "outside"

► Answer

[↑ Back to Top](#)

80. What is the output of below code? [↗](#)

```
let arr = [1, 2, 3, 4, 5, -6, 7];  
arr.length = 0;  
console.log(arr);
```



- 1: 0
- 2: Undefined
- 3: null
- 4: []

► Answer

[↑ Back to Top](#)

Disclaimer [↗](#)

The questions provided in this repository are the summary of frequently asked questions across numerous companies. We cannot guarantee that these questions will actually be asked during your interview process, nor should you focus on memorizing all of them. The primary purpose is for you to get a sense of what some companies might ask — do not get discouraged if you don't know the answer to all of them — that is ok!

Good luck with your interview 😊

Releases

No releases published

Packages

No packages published

Contributors 86
