JavaScript Interview Questions & Answers - 400

1.

   ### What are the possible ways to create objects in JavaScript

   There are many ways to create objects in javascript as below

   1. **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();
      ```

   2. **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);
      ```

   3. **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, function,
      ```

      Note: This is an easiest way to create an object

   4. **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");
      ```

   5. **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 new instan
console.log(result && typeof result === 'object' ? result : newInstance);
```

6. **ES6 Class syntax:**

   ES6 introduces class feature to create the objects

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

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

7. **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";
   })();
   ```

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**.

Screenshot

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

```javascript
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

```javascript
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().

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);
```

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.

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]; original a
let arrayIntegers2 = arrayIntegersOriginal2.splice(3); // returns [4, 5]; original arra
let arrayIntegers3 = arrayIntegersOriginal3.splice(3, 1, "a", "b", "c"); //returns [4];
```

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

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

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.

   1. The keys of an Object are Strings and Symbols, whereas they can be any value for a Map, including functions, objects, and any primitive.
   2. 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.
   3. 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.
   4. 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.
   5. 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.
   6. A Map may perform better in scenarios involving frequent addition and removal of key pairs.

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,

   1. Two strings are strictly equal when they have the same sequence of characters, same length, and same characters in corresponding

positions.
2. Two numbers are strictly equal when they are numerically equal. i.e, Having the same number value. There are two special cases in this,
   1. NaN is not equal to anything, including NaN.
   2. Positive and negative zeros are equal to one another.
3. Two Boolean operands are strictly equal if both are true or both are false.
4. Two objects are strictly equal if they refer to the same Object.
5. 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
```

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.

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);
```

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!");
```

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);
```

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 and d
```

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**.

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.

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)
```

18.

What is the difference between let and var

You can list out the differences in a tabular format

var	let
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 to due function scope)
  console.log(age); //error: age is not defined(due to block scope)
}
userDetails("John");
```

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.

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;
  }
}
```

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;
}
```

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
```

23.

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.

24.

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,

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

In the same fashion, function declarations are hoisted too

```javascript
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.

25.

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,

```javascript
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

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

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

26.

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

1. Own scope where variables defined between its curly brackets
2. Outer function's variables
3. 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.

27.

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

28.

Why do you need modules

Below are the list of benefits using modules in javascript ecosystem

1. Maintainability
2. Reusability
3. Namespacing

29.

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.

30.

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.

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

31.

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).

32.

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";
```

Screenshot

33.

Why do you need a Cookie

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

1. When a user visits a web page, the user profile can be stored in a cookie.
2. Next time the user visits the page, the cookie remembers the user profile.

34.

What are the options in a cookie

There are few below options available for a cookie,

1. 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";
```

1. 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";
```

35.

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.

36.

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

37.

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.

38.

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");
```

39.

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();
```

40.

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.

41.

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,

Screenshot

42.

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.

43.

What are the three states of promise

Promises have three states:

1. **Pending:** This is an initial state of the Promise before an operation begins

2. **Fulfilled:** This state indicates that the specified operation was completed.
3. **Rejected:** This state indicates that the operation did not complete. In this case an error value will be thrown.

44.

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);
```

45.

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.

46.

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(){
                ....
            });
        });
    });
});
```

47.

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.

48.

What are the main rules of promise

A promise must follow a specific set of rules,

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

49.

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
    });
  });
});
```

50.

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,

1. The initial promise resolves in 1 second,

2. After that `.then` handler is called by logging the result(1) and then return a promise with the value of result * 2.
3. After that the value passed to the next `.then` handler by logging the result(2) and return a promise with result * 3.
4. Finally the value passed to the last `.then` handler by logging the result(6) and return a promise with result * 4.

51.

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.

52.

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 faster
});
```

53.

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.

54.

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.

55.

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
}
```

56.

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, 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 an Array or null
```

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

57.

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"}
```

58.

What is the 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"
```

59.

What is undefined property

The undefined property indicates that a variable has not been assigned a value, or not declared 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;
```

60.

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
```

61.

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

62.

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
```

63.

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

64.

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.

65.

What is isNaN

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
```

66.

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.

67.

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
```

68.

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.

69.

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");
```

70.

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
```

71.

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

1. Top to Bottom(Event Capturing)
2. Bottom to Top (Event Bubbling)

72.

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.

73.

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.

74.

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();
}
```

75.

What are the tools or techniques used for debugging JavaScript code

You can use below tools or techniques for debugging javascript

1. Chrome Devtools
2. debugger statement
3. Good old console.log statement

76.

What are the pros and cons of promises over callbacks

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

Pros:

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

Cons:

1. It makes little complex code
2. You need to load a polyfill if ES6 is not supported

77.

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
```

78.

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).

79.

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>
```

80.

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.

81.

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.

82.

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).

83.

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,

1. Web page has finished loading
2. Input field was changed
3. 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 type="button" onclick="greeting()">Click me</button>
 </body>
</html>
```

84.

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.

85.

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.

86.

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>
```

87.

What are the steps involved in return false usage

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

1. First it stops the browser's default action or behaviour.
2. It prevents the event from propagating the DOM
3. Stops callback execution and returns immediately when called.

88.

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.

Screenshot

89.

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);
```

90.

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);
```

91.

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.

92.

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,

```javascript
var form = document.querySelector("#registration-form");

// Listen for changes to fields inside the form
form.addEventListener(
  "input",
  function (event) {
    // Log the field that was changed
    console.log(event.target);
  },
  false
);
```

93.

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.

94.

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.

95.

What are the syntax rules of JSON

Below are the list of syntax rules of JSON

1. The data is in name/value pairs
2. The data is separated by commas
3. Curly braces hold objects
4. Square brackets hold arrays

96.

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}"
```

97.

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}
```

98.

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.

99.

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>
```

100.

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>
```

101.

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";
}
```

102.

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,

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

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

1. **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;
```

1. **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);
```

103.

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,

1. **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);
```

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

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

1. **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
```

104.

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 ...
  }
}
```

105.

How do you test for an empty object

There are different solutions based on ECMAScript versions

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

```
Object.entries(obj).length === 0 && obj.constructor === Object; // Since date object le
```

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

```
Object.keys(obj).length === 0 && obj.constructor === Object; // Since date object lengt
```

1. **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({});
}
```

106.

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);
```

107.

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);
}
```

108.

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

1. Works on every environment
2. You can use break and continue flow control statements

Cons

1. Too verbose
2. Imperative
3. You might face one-by-off errors

109.

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);
```

110.

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
```

111.

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
```

112.

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, "");
    };
  })();
}
```

113.

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,
};
```

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

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

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

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

114.

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,

1. A logical not (!)
2. 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.

115.

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'.

116.

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.

117.

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
};
```

118.

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.
```

119.

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.

120.

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>";
}
```

121.

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"
```

122.

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,

1. Gives cleaner code
2. It provides a single place to look for local variables
3. Easy to avoid unwanted global variables
4. It reduces the possibility of unwanted re-declarations

123.

What are the benefits of initializing variables

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

1. It gives cleaner code

2. It provides a single place to initialize variables
3. Avoid undefined values in the code

124.

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.

1. Assign {} instead of new Object()
2. Assign ”” instead of new String()
3. Assign 0 instead of new Number()
4. Assign false instead of new Boolean()
5. Assign [] instead of new Array()
6. Assign /()/ instead of new RegExp()
7. 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 () {};
```

125.

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"}
]
```

126.

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)

127.

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
```

128.

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`.

129.

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.

130.

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.

131.

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;
```

132.

How do you change the style of a HTML element

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

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

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

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

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

133.

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`.

134.

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
}
```

135.

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.

136.

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
```

137.

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";
```

138.

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
```

139.

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;
  }
}
```

140.

What are the ways to execute javascript after page load

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

1. **window.onload:**

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

1. **document.onload:**

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

1. **body onload:**

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

141.

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;
```

142.

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.

143.

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
```

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

144.

What is the purpose of freeze method

Below are the main benefits of using freeze method,

1. It is used for freezing objects and arrays.
2. It is used to make an object immutable.

145.

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.

146.

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.substr(1).toLowerCase();
  });
}
toTitleCase("good morning john"); // Good Morning John
```

147.

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. Please click
</noscript>
```

148.

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,

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

149.

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 total(…args){
let sum = 0;
for(let i of args){
sum+=i;
}
return sum;
}
console.log(fun(1,2)); //3
console.log(fun(1,2,3)); //6
console.log(fun(1,2,3,4)); //13
console.log(fun(1,2,3,4,5)); //15
```

Note: Rest parameter is added in ES2015 or ES6

150.

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;
}
```

151.

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
```

152.

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,

1. If it is not extensible.
2. If all of its properties are non-configurable.
3. 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));
```

153.

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:

1. both undefined
2. both null
3. both true or both false
4. both strings of the same length with the same characters in the same order
5. both the same object (means both object have same reference)
6. both numbers and both +0 both -0 both NaN both non-zero and both not NaN and both have the same value.

154.

What is the purpose of using object is method

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

1. It is used for comparison of two strings.
2. It is used for comparison of two numbers.
3. It is used for comparing the polarity of two numbers.
4. It is used for comparison of two objects.

155.

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 target object. 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.

156.

What are the applications of assign method

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

1. It is used for cloning an object.
2. It is used to merge objects with the same properties.

157.

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
```

158.

What are the applications of seal method

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

1. It is used for sealing objects and arrays.
2. It is used to make an object immutable.

159.

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.

160.

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

1. If it is not extensible.
2. If all of its properties are non-configurable.
3. 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 or not
```

161.

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.

162.

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;
}
```

163.

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']
```

164.

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 not on "u

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

165.

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 function
  //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
```

166.

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);
}
greeting();
```

167.

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
```

168.

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 writable setti
```

169.

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.

170.

What are the advantages of Getters and Setters

Below are the list of benefits of Getters and Setters,

1. They provide simpler syntax
2. They are used for defining computed properties, or accessors in JS.
3. Useful to provide equivalence relation between properties and methods
4. They can provide better data quality
5. Useful for doing things behind the scenes with the encapsulated logic.

171.

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
```

172.

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.

173.

What are the conventions to be followed for the usage of switch case

Below are the list of conventions should be taken care,

1. The expression can be of type either number or string.
2. Duplicate values are not allowed for the expression.
3. 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.
4. The break statement is used inside the switch to terminate a statement sequence.
5. The break statement is optional. But if it is omitted, the execution will continue on into the next case.

174.

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.

1. string
2. number
3. boolean
4. null
5. undefined
6. bigint
7. symbol

175.

What are the different ways to access object properties

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

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

```
objectName.property;
```

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

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

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

```
objectName[expression];
```

176.

What are the function parameter rules

JavaScript functions follow below rules for parameters,

1. The function definitions do not specify data types for parameters.
2. Do not perform type checking on the passed arguments.
3. 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);
```

177.

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);
}
```

178.

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);
}
```

179.

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() |

180.

What are the various statements in error handling

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

1. **try:** This statement is used to test a block of code for errors
2. **catch:** This statement is used to handle the error
3. **throw:** This statement is used to create custom errors.
4. **finally:** This statement is used to execute code after try and catch regardless of the result.

181.

What are the two types of loops in javascript

1. **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.
2. **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.

182.

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.

183.

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.

184.

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).

185.

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 }
```

186.

What is an event loop

The Event Loop is a queue of callback functions. When an async function executes, the callback function is pushed into the queue. The JavaScript engine doesn't start processing the event loop until the async function has finished executing the code. **Note:** It allows Node.js to perform non-blocking I/O operations even though JavaScript is single-threaded.

187.

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,

1. Whenever you call a function for its execution, you are pushing it to the stack.
2. 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,

1. Add the `hungry()` function to the call stack list and execute the code.
2. Add the `eatFruits()` function to the call stack list and execute the code.
3. Delete the `eatFruits()` function from our call stack list.
4. Delete the `hungry()` function from the call stack list since there are no items anymore.

Screenshot

188.

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
```

189.

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"]
```

190.

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]
```

191.

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]
```

192.

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));
```

193.

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[len];
    }
  }
  return min;
}

function findMax(arr) {
  var length = arr.length;
  var max = -Infinity;
  while (len--) {
    if (arr[length] > max) {
      max = arr[length];
    }
  }
  return max;
}

console.log(findMin(marks));
console.log(findMax(marks));
```

194.

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 it's 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(int i=0; i < a.length; a[i++] = 0) ;
```

195.

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" }
```

196.

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
```

197.

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
}
```

198.

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.

199.

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

200.

What are the advantages of typescript over javascript

Below are some of the advantages of typescript over javascript,

1. TypeScript is able to find compile time errors at the development time only and it makes sures less runtime errors. Whereas javascript is an interpreted language.
2. TypeScript is strongly-typed or supports static typing which allows for checking type correctness at compile time. This is not available in javascript.
3. 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.

201.

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
```

202.

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
```

203.

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 constructor
    this.age = 30;
  }
}
```

```
 var employeeObject = new Employee();

 console.log(employeeObject.name);
```

204.

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 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;
  }
}
```

205.

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
```

206.

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
```

207.

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);
```

208.

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.

209.

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 exter
}
```

210.

What are the different ways to make an object non-extensible

You can mark an object non-extensible in 3 ways,

1. Object.preventExtensions
2. Object.seal
3. Object.freeze

```javascript
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
```

211.

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,

```javascript
const newObject = {};

Object.defineProperties(newObject, {
  newProperty1: {
    value: "John",
    writable: true,
  },
  newProperty2: {},
});
```

212.

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.

213.

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. Below are the few reasons for Obfuscation,

1. The Code size will be reduced. So data transfers between server and client will be fast.
2. It hides the business logic from outside world and protects the code from others
3. Reverse engineering is highly difficult
4. The download time will be reduced

214.

What is Minification

```
Minification is the process of removing all unnecessary characters(empty spaces are re
```

Normally it is recommended to use minification for heavy traffic and intensive requirements of resources. It reduces file sizes with below benefits,

```
1. Decreases loading times of a web page
2. Saves bandwidth usages
```

215.

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

216.

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;
  }
}
```

217.

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.

218.

Is enums feature available in javascript

n enum is a type restricting variables to one value from a predefined set of constants.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, ...})
```

219.

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 the usage of it in an example,

```
const newObject = {
  a: 1,
  b: 2,
  c: 3,
};

console.log(Object.getOwnPropertyNames(newObject));
["a", "b", "c"];
```

220.

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
```

221.

What are the attributes provided by a property descriptor

A property descriptor is a record which has the following attributes

1. value: The value associated with the property
2. writable: Determines whether the value associated with the property can be changed or not
3. configurable: Returns true if the type of this property descriptor can be changed and if the property can be deleted from the corresponding object.
4. enumerable: Determines whether the property appears during enumeration of the properties on the corresponding object or not.
5. set: A function which serves as a setter for the property

6. get: A function which serves as a getter for the property

222.

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,

```javascript
class ChildClass extends ParentClass { ... }
```

Let's take an example of Square subclass from Polygon parent class,

```javascript
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;
  }
}
```

223.

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.

```javascript
var numericArray = [1, 2, 3, 4];
console.log(numericArray.includes(3)); // true

var stringArray = ["green", "yellow", "blue"];
console.log(stringArray.includes("blue")); //true
```

224.

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
```

225.

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));
```

226.

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 |

227.

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.

228.

How do you declare namespace

Even though JavaScript lacks namespaces, we can use Objects , IIFE to create namespaces.

1. **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
```

1. **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();
})();
```

1. **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.
```

229.

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,

1. document.getElementById(id): It finds an element by Id

2. document.getElementsByTagName(name): It finds an element by tag name
3. document.getElementsByClassName(name): It finds an element by class name

230.

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,

```javascript
$(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.

231.

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.

232.

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.

```javascript
let age = 50; // age is a number now
age = "old"; // age is a string now
age = true; // age is a boolean
```

233.

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)".

234.

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>
```

235.

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) {}
```

236.

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.

```javascript
a.b.c.greeting = "welcome";
a.b.c.age = 32;
```

Using `with` it turns this into:

```javascript
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.

237.

What is the output of below for loops

```javascript
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`.

238.

List down some of the features of ES6

Below are the list of some new features of ES6,

1. Support for constants or immutable variables
2. Block-scope support for variables, constants and functions
3. Arrow functions
4. Default parameters
5. Rest and Spread Parameters

6. Template Literals
7. Multi-line Strings
8. Destructuring Assignment
9. Enhanced Object Literals
10. Promises
11. Classes
12. Modules

239.

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.

240.

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 dec
  alert(name);
}
```

```
myFunc();
alert(name);
```

241.

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']
```

242.

What are default parameters

In E5, 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
```

243.

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 E6, 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.

244.

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`);
```

245.

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'}`;
```

246.

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
```

247.

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
```

248.

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
```

249.

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}
```

250.

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());
```

251.

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
}
```

252.

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.

253.

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.

254.

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.

255.

Is JavaScript faster than server side script

Yes, JavaScript is faster than server side script. 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.

256.

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` 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" name="checkboxname" value="Agree" /> Agree the
conditions<br />
```

```
console.log(document.getElementById('checkboxname').checked); // true or false
```

257.

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.

258.

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);
}
```

259.

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

1. When you use Octal syntax

```
var n = 022;
```

1. Using `with` statement
2. When you use delete operator on a variable name
3. Using eval or arguments as variable or function argument name
4. When you use newly reserved keywords
5. 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.

260.

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.

261.

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");
```

262.

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)
```

263.

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, Pears
```

264.

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`

265.

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'
```

266.

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!"
```

267.

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.

268.

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
```

269.

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.

270.

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.

271.

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
```

272.

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.
```

273.

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
```

274.

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);
```

275.

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]
```

276.

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;
```

277.

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', 'Beijing', 'Pa
```

278.

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 []
```

279.

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
```

280.

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: "watermelon"}
```

281.

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"]
```

282.

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, Screenshot

Note: All CSS styles can be applied to console messages.

283.

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 Screenshot

284.

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,

Screenshot

285.

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,

Screenshot **Not:** Remember that `console.table()` is not supported in IE.

286.

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);
}
```

287.

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());
```

288.

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,
```

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,
```

289.

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();
}
```

290.

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(new String
```

i.e, Every primitive except null and undefined have Wrapper Objects and the list of wrapper objects are String,Number,Boolean,Symbol and BigInt.

291.

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.

292.

What are the different ways to deal with Asynchronous Code

Below are the list of different ways to deal with Asynchronous code.

1. Callbacks
2. Promises
3. Async/await
4. Third-party libraries such as async.js,bluebird etc

293.

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,

1. Create an `AbortController` instance
2. Get the signal property of an instance and pass the signal as a fetch option for signal
3. 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);
```

294.

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
```

295.

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.

296.

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,

1. 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.
2. When an event fires, the event callback added to task queue
3. When a setTimeout or setInterval is reached, the corresponding callback added to task queue

297.

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.

298.

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.

Screenshot

299.

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.

Screenshot

300.

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.

301.

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.

302.

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,

1. Transform syntax
2. Polyfill features that are missing in your target environment (using @babel/polyfill)
3. Source code transformations (or codemods)

303.

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.

304.

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

305.

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
```

306.

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();
```

307.

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); // []
```

308.

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() {}
```

309.

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 returns
  console.log(data);
}
logger();
```

It is basically syntax sugar over ES2015 promises and generators.

310.

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");
});
```

```javascript
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,

1. **Add catch block at the end of each chain:** You can add catch block to the end of each of your promise chains

   ```javascript
   Promise.resolve("promised value")
     .then(function () {
       throw new Error("error");
     })
     .catch(function (error) {
       console.error(error.stack);
     });
   ```

311.

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,

```javascript
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}
```

312.

Write a factorial function

```
function factorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
console.log(factorial(5)); //120
```

313.

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
```

314.

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.

1. If a constructor or function invoked using the new operator, new.target returns a reference to the constructor or function.
2. 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
```

315.

What are the differences between arguments object and rest parameter

There are three main differences between arguments object and rest parameters

1. The arguments object is an array-like but not an array. Whereas the rest parameters are array instances.
2. The arguments object does not support methods such as sort, map, forEach, or pop. Whereas these methods can be used in rest parameters.
3. 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

316.

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.

317.

What are the different kinds of generators

There are five kinds of generators,

1. **Generator function declaration:**

   ```
   function* myGenFunc() {
     yield 1;
     yield 2;
     yield 3;
   }
   const genObj = myGenFunc();
   ```

2. **Generator function expressions:**

   ```
   const myGenFunc = function* () {
     yield 1;
     yield 2;
     yield 3;
   };
   const genObj = myGenFunc();
   ```

3. **Generator method definitions in object literals:**

```
const myObj = {
  *myGeneratorMethod() {
    yield 1;
    yield 2;
    yield 3;
  },
};
const genObj = myObj.myGeneratorMethod();
```

4. **Generator method definitions in class:**

```
class MyClass {
  *myGeneratorMethod() {
    yield 1;
    yield 2;
    yield 3;
  }
}
const myObject = new MyClass();
const genObj = myObject.myGeneratorMethod();
```

318.

What are the built-in iterables

Below are the list of built-in iterables in javascript,

1. Arrays and TypedArrays
2. Strings: Iterate over each character or Unicode code-points
3. Maps: iterate over its key-value pairs
4. Sets: iterates over their elements
5. arguments: An array-like special variable in functions
6. DOM collection such as NodeList

319.

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:

1. for..in iterates over all enumerable property keys of an object
2. 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);
}

// value are the property values
for (let value of arr) {
  console.log(value);
}
```

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.

320.

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;
```

321.

What is the difference between isNaN and Number.isNaN?

1. **isNaN**: The global function `isNaN` converts the argument to a Number and returns true if the resulting value is NaN.
2. **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
```

322.

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 temparature 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);
```

323.

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);
```

324.

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
```

325.

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");

console.groupEnd();
```

326.

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 demonistrate 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.

1. **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 a new copy

   console.log(originalArray); // [1, 2, 3, 4, 5]
   console.log(newArray); // [ 5, 4, 3, 2, 1]
   ```

2. **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]
```

327.

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");
```

328.

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.

329.

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
```

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

Answer: 4 The function declarations are hoisted similar to any variables. So the placement for `Vehicle` function declaration doesn’t make any difference.

-

2. What is the output of below code

```javascript
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

Answer: 3 Of course the return value of `foo()` is 1 due to the increment operator. But the statement `let x = y = 0` declares a local variable x. Whereas y declared as a global variable accidentally. This statement is equivalent to,

```javascript
let x;
window.y = 0;
x = window.y;
```

Since the block scoped variable x is undefined outside of the function, the type will be undefined too. Whereas the global variable `y` is available outside the function, the value is 0 and type is number.

-

3. What is the output of below code

```javascript
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

Answer: 4 The statements order is based on the event loop mechanism. The order of statements follows the below order,

1. At first, the main function is pushed to the stack.
2. Then the browser pushes the fist statement of the main function(i.e, A's console.log) to the stack, executing and popping out immediately.
3. But `setTimeout` statement moved to Browser API to apply the delay for callback.
4. In the meantime, C's console.log added to stack, executed and popped out.
5. The callback of `setTimeout` moved from Browser API to message queue.
6. The `main` function popped out from stack because there are no statements to execute
7. The callback moved from message queue to the stack since the stack is empty.
8. The console.log for B is added to the stack and display on the console.

-

4. What is the output of below equality check

```
console.log(0.1 + 0.2 === 0.3);
```

- 1: false
- 2: true

Answer

Answer: 1 This is due to the float point math problem. Since the floating point numbers are encoded in binary format, the addition operations on them lead to rounding errors. Hence, the comparison of floating points doesn't give expected results. You can find more details about the explanation here 0.30000000000000004.com/

-

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

Answer: 4 The main points in the above code snippets are,

1. You can see function expression instead function declaration inside if statement. So it always returns true.
2. Since it is not declared(or assigned) anywhere, f is undefined and typeof f is undefined too.

In other words, it is same as

```
var y = 1;
if ("foo") {
  y += typeof f;
}
console.log(y);
```

Note: It returns 1object for MS Edge browser

-

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

Answer: 3 This is a semicolon issue. Normally semicolons are optional in JavaScript. So if there are any statements(in this case, return) missing semicolon, it is automatically inserted immediately. Hence, the function returned as undefined.

Whereas if the opening curly brace is along with the return keyword then the function is going to be returned as expected.

```
function foo() {
  return {
    message: "Hello World",
  };
}
console.log(foo()); // {message: "Hello World"}
```

-

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

Answer: 3 The `delete` operator will delete the object property but it will not reindex the array or change its length. So the number or elements or length of the array won’t be changed. If you try to print myChars then you can observe that it doesn’t set an undefined value, rather the property is removed from the array. The newer versions of Chrome use `empty` instead of `undefined` to make the difference a bit clearer.

-

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

Answer: 2 The latest chrome versions display `sparse array`(they are filled with holes) using this empty x n notation. Whereas the older versions have undefined x n notation. **Note:** The latest version of FF displays `n empty slots` notation.

-

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

Answer: 1 ES6 provides method definitions and property shorthands for objects. So both prop2 and prop3 are treated as regular function values.

-

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

Answer: 2 The important point is that if the statement contains the same operators(e.g, < or >) then it can be evaluated from left to right. The first statement follows the below order,

1. console.log(1 < 2 < 3);

2. console.log(true < 3);
3. console.log(1 < 3); // True converted as `1` during comparison
4. True

Whereas the second statement follows the below order,

1. console.log(3 > 2 > 1);
2. console.log(true > 1);
3. console.log(1 > 1); // False converted as `0` during comparison
4. False

-

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

Answer: 2 In non-strict mode, the regular JavaScript functions allow duplicate named parameters. The above code snippet has duplicate parameters on 1st and 3rd parameters. The value of the first parameter is mapped to the third argument which is passed to the function. Hence, the 3rd argument overrides the first parameter.

Note: In strict mode, duplicate parameters will throw a Syntax Error.

-

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

Answer: 3 Unlike regular functions, the arrow functions doesn't not allow duplicate parameters in either strict or non-strict mode. So you can see `SyntaxError` in the console.

-

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

Answer: 1 Arrow functions do not have an `arguments, super, this, or new.target` bindings. So any reference to `arguments` variable tries to resolve to a binding in a lexically enclosing environment. In this case, the arguments variable is not defined outside of the arrow function. Hence, you will receive a reference error.

Where as the normal function provides the number of arguments passed to the function

```
const func = function () {
  return arguments.length;
};
console.log(func(1, 2, 3));
```

But If you still want to use an arrow function then rest operator on arguments provides the expected arguments

```
const arrowFunc = (...args) => args.length;
console.log(arrowFunc(1, 2, 3));
```

-

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

Answer: 2 In order to be consistent with functions like `String.prototype.padStart`, the standard method name for trimming the whitespaces is considered as `trimStart`. Due to web web compatibility reasons, the old method name 'trimLeft' still acts as an alias for 'trimStart'. Hence, the prototype for 'trimLeft' is always 'trimStart'

-

15. What is the output of below code

```
console.log(Math.max());
```

- 1: undefined
- 2: Infinity
- 3: 0
- 4: -Infinity

Answer

Answer: 4 -Infinity is the initial comparant because almost every other value is bigger. So when no arguments are provided, -Infinity is going to be returned. **Note:** Zero number of arguments is a valid case.

-

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

Answer: 1 As per the comparison algorithm in the ECMAScript specification(ECMA-262), the above expression converted into JS as below

```
10 === Number([10].valueOf().toString()); // 10
```

So it doesn't matter about number brackets([]) around the number, it is always converted to a number in the expression.

-

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

Answer: 2 The concatenation operator(+) is applicable for both number and string types. So if any operand is string type then both operands concatenated as strings. Whereas subtract(-) operator tries to convert the operands as number type.

-

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

Answer: 1 In comparison operators, the expression `[0]` converted to Number([0].valueOf().toString()) which is resolved to false. Whereas `[0]` just becomes a truthy value without any conversion because there is no comparison operator.

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

Answer: 4 The + operator is not meant or defined for arrays. So it converts arrays into strings and concatenates them.

-

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

Answer: 1 Since `Set` object is a collection of unique values, it won't allow duplicate values in the collection. At the same time, it is case sensitive data structure.

-

21. What is the output of below code

```
console.log(NaN === NaN);
```

- 1: True
- 2: False

Answer

Answer: 2 JavaScript follows IEEE 754 spec standards. As per this spec, NaNs are never equal for floating-point numbers.

-

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

Answer: 4 The `indexOf` uses strict equality operator(===) internally and `NaN === NaN` evaluates to false. Since indexOf won't be able to find NaN inside an array, it returns -1 always. But you can use `Array.prototype.findIndex` method to find out the index of NaN in an array or You can use `Array.prototype.includes` to check if NaN is present in an array or not.

```
let numbers = [1, 2, 3, 4, NaN];
console.log(numbers.findIndex(Number.isNaN)); // 4

console.log(numbers.includes(NaN)); // true
```

-

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

Answer: 3 When using rest parameters, trailing commas are not allowed and will throw a SyntaxError. If you remove the trailing comma then it displays 1st answer

```
let [a, ...b] = [1, 2, 3, 4, 5];
console.log(a, b); // 1, [2, 3, 4, 5]
```

-

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

Answer: 1 Async functions always return a promise. But even if the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise. The above async function is equivalent to below expression,

```
function func() {
  return Promise.resolve(10);
}
```

-

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

Answer: 4 The await expression returns value 10 with promise resolution and the code after each await expression can be treated as existing in a `.then` callback. In this case, there is no return expression at the end of the function. Hence, the default return value of `undefined` is returned as the resolution of the promise. The above async function is equivalent to below expression,

```
function func() {
  return Promise.resolve(10).then(() => undefined);
}
```

-

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

Answer: 1 Even though "processArray" is an async function, the anonymous function that we use for `forEach` is synchronous. If you use await inside a synchronous function then it throws a syntax error.

-

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

Answer: 4 The forEach method will not wait until all items are finished but it just runs the tasks and goes next. Hence, the last statement is displayed first followed by a sequence of promise resolutions.

But you control the array sequence using for..of loop,

```
async function processArray(array) {
  for (const item of array) {
    await delayedLog(item);
  }
  console.log("Process completed!");
}
```

-

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

Answer: 1 Set has few exceptions from equality check,

1. All NaN values are equal
2. Both +0 and -0 considered as different values

-

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");

cnsooe.log(sym1 === sym2, sym3 === sym4);
```

- 1: true, true
- 2: true, false
- 3: false, true
- 4: false, false

Answer

Answer: 3 Symbol follows below conventions,

1. Every symbol value returned from Symbol() is unique irrespective of the optional string.
2. `Symbol.for()` function creates a symbol in a global symbol registry list. But it doesn't necessarily create a new symbol on every call, it checks first if a symbol with the given key is already present in the registry and returns the symbol if it is found. Otherwise a new symbol created in the registry.

Note: The symbol description is just useful for debugging purposes.

-

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

Answer: 1 `Symbol` is a just a standard function and not an object constructor(unlike other primitives new Boolean, new String and new Number). So if you try to call it with the new operator will result in a TypeError

-

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

Answer: 4 The return value of `typeof myNumber (OR) typeof myString` is always the truthy value (either "number" or "string"). Since ! operator converts the value to a boolean value, the value of both `!typeof myNumber or !typeof myString` is always false. Hence the if condition fails and control goes to else block.

-

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

Answer: 2 The symbols has below constraints,

1. The undefined, Functions, and Symbols are not valid JSON values. So those values are either omitted (in an object) or changed to null (in an array). Hence, it returns null values for the value array.
2. All Symbol-keyed properties will be completely ignored. Hence it returns an empty object({}).

-

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

Answer: 2 Using constructors, `new.target` refers to the constructor (points to the class definition of class which is initialized) that was directly invoked by new. This also applies to the case if the constructor is in a parent class and was delegated from a child constructor.

-

35. What is the output of below code

```
const [x, ...y] = [1, 2, 3, 4];
console.log(x, y);
```

- 1: 1, [2, 3, 4]
- 2: 1, [2, 3]
- 3: 1, [2]
- 4: SyntaxError

Answer

Answer: 4 It throws a syntax error because the rest element should not have a trailing comma. You should always consider using a rest operator as the last element.

-

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

Answer: 1 The object property follows below rules,

1. The object properties can be retrieved and assigned to a variable with a different name
2. The property assigned a default value when the retrieved value is `undefined`

-

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

Answer: 2 If you leave out the right-hand side assignment for the destructuring object, the function will look for at least one argument to be supplied when invoked. Otherwise you will receive an error `Error: Cannot read property 'length' of undefined` as mentioned above.

You can avoid the error with either of the below changes,

1. **Pass at least an empty object:**

```
function area({ length = 10, width = 20 }) {
  console.log(length * width);
}

area({});
```

2. **Assign default empty object:**

```
function area({ length = 10, width = 20 } = {}) {
  console.log(length * width);
}

area();
```

-

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

Answer: 1 It is possible to combine Array and Object destructuring. In this case, the third element in the array props accessed first followed by name property in the object.

-

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

Answer: 3 If the function argument is set implicitly(not passing argument) or explicitly to undefined, the value of the argument is the default parameter. Whereas for other falsy values('' or null), the value of the argument is passed as a parameter.

Hence, the result of function calls categorized as below,

1. The first two function calls logs number type since the type of default value is number
2. The type of ' ' and null values are string and object type respectively.

-

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

Answer: 2 Since the default argument is evaluated at call time, a new object is created each time the function is called. So in this case, the new array is created and an element pushed to the default empty array.

-

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

Answer: 2 Since parameters defined earlier are available to later default parameters, this code snippet doesn't throw any error.

-

42. What is the output of below code

```javascript
function outer(f = inner()) {
  function inner() {
    return "Inner";
  }
}
outer();
```

- 1: ReferenceError
- 2: Inner

Answer

Answer: 1 The functions and variables declared in the function body cannot be referred from default value parameter initializers. If you still try to access, it throws a run-time ReferenceError(i.e, `inner` is not defined).

-

43. What is the output of below code

```javascript
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

Answer: 3 The rest parameter is used to hold the remaining parameters of a function and it becomes an empty array if the argument is not provided.

-

44. What is the output of below code

```javascript
const obj = { key: "value" };
const array = [...obj];
console.log(array);
```

- 1: ['key', 'value']
- 2: TypeError
- 3: []

- 4: [‘key’]

Answer

Answer: 2 Spread syntax can be applied only to iterable objects. By default, Objects are not iterable, but they become iterable when used in an Array, or with iterating functions such as `map(), reduce(), and assign()`. If you still try to do it, it still throws `TypeError: obj is not iterable`.

-

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

Answer: 4 Generators are not constructible type. But if you still proceed to do, there will be an error saying “TypeError: myGenFunc is not a constructor”

-

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

Answer: 1 A return statement in a generator function will make the generator finish. If a value is returned, it will be set as the value property of the object and done property to true. When a generator is finished, subsequent next() calls return an object of this form: `{value: undefined, done: true}`.

-

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

Answer: 4 The generator should not be re-used once the iterator is closed. i.e, Upon exiting a loop(on completion or using break & return), the generator is closed and trying to iterate over it again does not yield any more results. Hence, the second loop doesn't print any value.

-

48. What is the output of below code

```
const num = 0o38;
console.log(num);
```

- 1: SyntaxError
- 2: 38

Answer

Answer: 1 If you use an invalid number(outside of 0-7 range) in the octal literal, JavaScript will throw a SyntaxError. In ES5, it treats the octal literal as a decimal number.

-

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

Answer: 2 Unlike function declarations, class declarations are not hoisted. i.e, First You need to declare your class and then access it, otherwise it will throw a ReferenceError "Uncaught ReferenceError: Square is not defined".

Note: Class expressions also applies to the same hoisting restrictions of class declarations.

-

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

Answer: 4 When a regular or prototype method is called without a value for **this**, the methods return an initial this value if the value is not undefined. Otherwise global window object will be returned. In our case, the initial `this` value is undefined so both methods return window objects.

-

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

Answer: 3 The super keyword is used to call methods of a superclass. Unlike other languages the super invocation doesn't need to be a first statement. i.e, The statements will be executed in the same order of code.

-

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

Answer: 2 Even though we used constant variables, the content of it is an object and the object's contents (e.g properties) can be altered. Hence, the change is going to be valid in this case.

-

53. What is the output of below code

```
console.log(" " === " ");
```

- 1: false
- 2: true

Answer

Answer: 2 Emojis are unicodes and the unicode for smile symbol is "U+1F642". The unicode comparision of same emojies is equivalent to string comparison. Hence, the output is always true.

-

54. What is the output of below code?

```
console.log(typeof typeof typeof true);
```

- 1: string
- 2: boolean
- 3: NaN
- 4: number

Answer

Answer: 1 The typeof operator on any primitive returns a string value. So even if you apply the chain of typeof operators on the return value, it is always string.

-

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

Answer: 1

1. The type of operator on new Number always returns object. i.e, typeof new Number(0) –> object.
2. Objects are always truthy in if block

Hence the above code block always goes to if section.

-

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

Answer: 4 It returns undefined for non-strict mode and returns Error for strict mode. In non-strict mode, the wrapper object is going to be created and get the mentioned property. But the object get disappeared after accessing the property in next line.

-

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

Answer: 1 11 and 10 is logged to the console.

The innerFunc is a closure which captures the count variable from the outerscope. i.e, 10. But the conditional has another local variable `count` which overwrites the ourter `count` variable. So the first console.log displays value 11. Whereas the second console.log logs 10 by capturing the count variable from outerscope.

Answer

- 1: hi
- 2: 1
- 3: ''

Reason : The operator returns the value of the first falsy operand encountered when evaluating from left to right, or the value of the last operand if they are all truthy.

Note: Below these values are consider as falsy value

- 1: 0
- 2: ' '
- 3: null
- 4: undefined
- 5: NAN

---

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

Answer: 3 Arrays have their own implementation of `toString` method that returns a comma-separated list of elements. So the above code snippet returns true. In order to avoid conversion of array type, we should use === for comparison.

---

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

Answer: 2 Hoisting will move variables and functions to be the top of scope. Even though getMessage is an arrow function the above function will considered as a varible due to it's variable declaration or assignment. So the variables will have undefined value in memory phase and throws an error '`getMessage` is not a function' at the code execution phase.

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

Answer: 3 Even though a promise is resolved immediately, it won't be executed immediately because its **.then/catch/finally** handlers or callbacks(aka task) are pushed into the queue. Whenever the JavaScript engine becomes free from the current program, it pulls a task from the queue and executes it. This is the reason why last statement is printed first before the log of promise handler.

Note: We call the above queue as "MicroTask Queue"

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

Answer: 4 When JavaScript encounters a line break without a semicolon, the JavaScript parser will automatically add a semicolon based on a set of rules called `Automatic Semicolon Insertion` which determines whether line break as end of statement or not to insert semicolon. But it does not assume a semicolon before square brackets [...]. So the first two lines considered as a single statement as below.

```
console
  .log("First line")
  [("a", "b", "c")].forEach((element) => console.log(element));
```

Hence, there will be **cannot read properties of undefined** error while applying the array square bracket on log function.

##