```
weakMapObject.set(secondObject, 100);
console.log(weakMapObject.has(firstObject)); //true
console.log(weakMapObject.get(firstObject)); // John
weakMapObject.delete(secondObject);
```

Back to Top

209. What is the purpose of uneval

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

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

Back to Top

210. How do you encode an URL

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

```
var uri = "https://mozilla.org/?x=шеллы";
var encoded = encodeURI(uri);
console.log(encoded); // https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%8B
```

Back to Top

211. How do you decode an URL

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

```
var uri = "https://mozilla.org/?x=шеллы";
var encoded = encodeURI(uri);
console.log(encoded); // https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%8B
try {
  console.log(decodeURI(encoded)); // "https://mozilla.org/?x=шеллы"
} catch (e) {
```

```
  // catches a malformed URI
  console.error(e);
}
```

Back to Top

212. How do you print the contents of web page

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

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

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

Back to Top

213. What is the difference between uneval and eval

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

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

Back to Top

214. What is an anonymous function

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

```
function (optionalParameters) {
  //do something
}
```

```
const myFunction = function(){ //Anonymous function assigned to a variable
  //do something
};

[1, 2, 3].map(function(element){ //Anonymous function used as a callback functic
  //do something
});
```

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

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

Back to Top

215. What is the precedence order between local and global variables

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

```
var msg = "Good morning";
function greeting() {
  msg = "Good Evening";
  console.log(msg);
}
greeting();
```

Back to Top

216. What are javascript accessors

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

```
var user = {
  firstName: "John",
  lastName : "Abraham",
```

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

Back to Top

217. How do you define property on Object constructor

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

```
const newObject = {};

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

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

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

Back to Top

218. What is the difference between get and defineProperty

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

Back to Top

219. What are the advantages of Getters and Setters

Below are the list of benefits of Getters and Setters,

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

↑ Back to Top

220. Can I add getters and setters using defineProperty method

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

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

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

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

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

⬆ Back to Top

221. What is the purpose of switch-case

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

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

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

⬆ Back to Top

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

Below are the list of conventions should be taken care,

i. The expression can be of type either number or string.
ii. Duplicate values are not allowed for the expression.
iii. The default statement is optional. If the expression passed to switch does not match with any case value then the statement within default case will be executed.
iv. The break statement is used inside the switch to terminate a statement sequence.

v. The break statement is optional. But if it is omitted, the execution will continue on into the next case.

⬆ Back to Top

223. What are primitive data types

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

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

⬆ Back to Top

224. What are the different ways to access object properties

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

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

```
objectName.property;
```

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

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

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

```
objectName[expression];
```

⬆ Back to Top

225. What are the function parameter rules

JavaScript functions follow below rules for parameters,

i. The function definitions do not specify data types for parameters.

ii. Do not perform type checking on the passed arguments.

iii. Do not check the number of arguments received. i.e, The below function follows the above rules,

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

⬆ Back to Top

226. What is an error object

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

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

⬆ Back to Top

227. When you get a syntax error

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

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

⬆ Back to Top

228. What are the different error names from error object

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

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

⬆ Back to Top

229. What are the various statements in error handling

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

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

⬆ Back to Top

230. What are the two types of loops in javascript

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

⬆ Back to Top

231. What is nodejs

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

⬆ Back to Top

232. What is an Intl object

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

⬆ Back to Top

233. How do you perform language specific date and time formatting

You can use the `Intl.DateTimeFormat` object which is a constructor for objects that enable language-sensitive date and time formatting. Let's see this behavior with an example,

```
var date = new Date(Date.UTC(2019, 07, 07, 3, 0, 0));
console.log(new Intl.DateTimeFormat("en-GB").format(date)); // 07/08/2019
console.log(new Intl.DateTimeFormat("en-AU").format(date)); // 07/08/2019
```

⬆ Back to Top

234. What is an Iterator

An iterator is an object which defines a sequence and a return value upon its termination. It implements the Iterator protocol with a `next()` method which returns an object with two properties: `value` (the next value in the sequence) and `done` (which is true if the last value in the sequence has been consumed).

⬆ Back to Top

235. How does synchronous iteration works

Synchronous iteration was introduced in ES6 and it works with below set of components,

Iterable: It is an object which can be iterated over via a method whose key is Symbol.iterator. **Iterator:** It is an object returned by invoking `[Symbol.iterator]()` on an iterable. This iterator object wraps each iterated element in an object and returns it via `next()` method one by one. **IteratorResult:** It is an object returned by `next()` method. The object contains two properties; the `value` property contains an iterated element and the `done` property determines whether the element is the last element or not.

Let's demonstrate synchronous iteration with an array as below,

```
const iterable = ["one", "two", "three"];
const iterator = iterable[Symbol.iterator]();
console.log(iterator.next()); // { value: 'one', done: false }
console.log(iterator.next()); // { value: 'two', done: false }
console.log(iterator.next()); // { value: 'three', done: false }
console.log(iterator.next()); // { value: 'undefined, done: true }
```

⬆ Back to Top

236. What is an event loop

The Event Loop is a 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.

⬆ Back to Top

237. What is call stack

Call Stack is a data structure for javascript interpreters to keep track of function calls(creates execution context) in the program. It has two major actions,

i. Whenever you call a function for its execution, you are pushing it to the stack.
ii. Whenever the execution is completed, the function is popped out of the stack.

Let's take an example and it's state representation in a diagram format

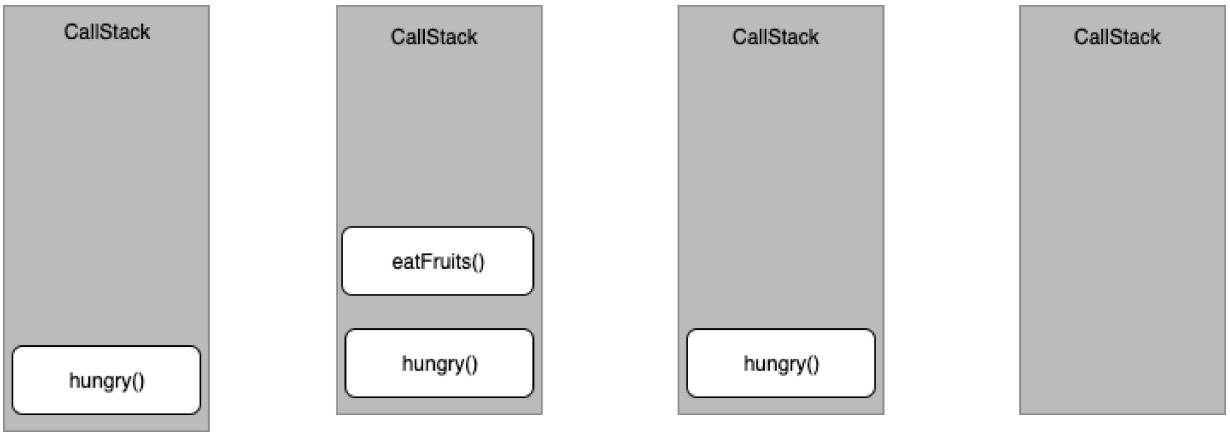
```
function hungry() {
  eatFruits();
}
function eatFruits() {
```

```
  return "I'm eating fruits";
}

// Invoke the `hungry` function
hungry();
```

The above code processed in a call stack as below,

i. Add the `hungry()` function to the call stack list and execute the code.
ii. Add the `eatFruits()` function to the call stack list and execute the code.
iii. Delete the `eatFruits()` function from our call stack list.
iv. Delete the `hungry()` function from the call stack list since there are no items anymore.



↑ Back to Top

238. What is an event queue

↑ Back to Top

239. What is a decorator

A decorator is an expression that evaluates to a function and that takes the target, name, and decorator descriptor as arguments. Also, it optionally returns a decorator descriptor to install on the target object. Let's define admin decorator for user class at design time,

```
function admin(isAdmin) {
   return function(target) {
       target.isAdmin = isAdmin;
   }
}
```

```
@admin(true)
class User() {
}
console.log(User.isAdmin); //true

 @admin(false)
 class User() {
 }
 console.log(User.isAdmin); //false
```

⬆ Back to Top

240. What are the properties of Intl object

Below are the list of properties available on Intl object,

i. **Collator:** These are the objects that enable language-sensitive string comparison.
ii. **DateTimeFormat:** These are the objects that enable language-sensitive date and time formatting.
iii. **ListFormat:** These are the objects that enable language-sensitive list formatting.
iv. **NumberFormat:** Objects that enable language-sensitive number formatting.
v. **PluralRules:** Objects that enable plural-sensitive formatting and language-specific rules for plurals.
vi. **RelativeTimeFormat:** Objects that enable language-sensitive relative time formatting.

⬆ Back to Top

241. What is an Unary operator

The unary(+) operator is used to convert a variable to a number.If the variable cannot be converted, it will still become a number but with the value NaN. Let's see this behavior in an action.

```
var x = "100";
var y = +x;
console.log(typeof x, typeof y); // string, number

var a = "Hello";
var b = +a;
console.log(typeof a, typeof b, b); // string, number, NaN
```

⬆ Back to Top

242. How do you sort elements in an array

The sort() method is used to sort the elements of an array in place and returns the sorted array. The example usage would be as below,

```
var months = ["Aug", "Sep", "Jan", "June"];
months.sort();
console.log(months); //  ["Aug", "Jan", "June", "Sep"]
```

⬆ Back to Top

243. What is the purpose of compareFunction while sorting arrays

The compareFunction is used to define the sort order. If omitted, the array elements are converted to strings, then sorted according to each character's Unicode code point value. Let's take an example to see the usage of compareFunction,

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

⬆ Back to Top

244. How do you reversing an array

You can use the reverse() method to reverse the elements in an array. This method is useful to sort an array in descending order. Let's see the usage of reverse() method in an example,

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

⬆ Back to Top

245. How do you find min and max value in an array

You can use `Math.min` and `Math.max` methods on array variables to find the minimum and maximum elements within an array. Let's create two functions to find the min and max value with in an array,

```
var marks = [50, 20, 70, 60, 45, 30];
function findMin(arr) {
  return Math.min.apply(null, arr);
}
function findMax(arr) {
  return Math.max.apply(null, arr);
}

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

⬆ Back to Top

246. How do you find min and max values without Math functions

You can write functions which loop through an array comparing each value with the lowest value or highest value to find the min and max values. Let's create those functions to find min and max values,

```
var marks = [50, 20, 70, 60, 45, 30];
function findMin(arr) {
  var length = arr.length;
  var min = Infinity;
  while (length--) {
    if (arr[length] < min) {
      min = arr[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));
```

⬆ Back to Top

247. What is an empty statement and purpose of it

The empty statement is a semicolon (;) indicating that no statement will be executed, even if JavaScript syntax requires one. Since there is no action with an empty statement you might think that 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) ;
```

⬆ Back to Top

248. How do you get metadata of a module

You can use the `import.meta` object which is a meta-property exposing context-specific meta data to a JavaScript module. It contains information about the current module, such as the module's URL. In browsers, you might get different meta data than NodeJS.

```
<script type="module" src="welcome-module.js"></script>;
console.log(import.meta); // { url: "file:///home/user/welcome-module.js" }
```

⬆ Back to Top

249. What is a comma operator

The comma operator is used to evaluate each of its operands from left to right and returns the value of the last operand. This is totally different from comma usage within arrays, objects, and function arguments and parameters. For example, the usage for numeric expressions would be as below,

```
var x = 1;
x = (x++, x);
```

```
console.log(x); // 2
```

⬆ Back to Top

250. What is the advantage of a comma operator

It is normally used to include multiple expressions in a location that requires a single expression. One of the common usages of this comma operator is to supply multiple parameters in a `for` loop. For example, the below for loop uses multiple expressions in a single location using comma operator,

```
for (var a = 0, b =10; a <= 10; a++, b--)
```

You can also use the comma operator in a return statement where it processes before returning.

```
function myFunction() {
  var a = 1;
  return (a += 10), a; // 11
}
```

⬆ Back to Top

251. What is typescript

TypeScript is a typed superset of JavaScript created by Microsoft that adds optional types, classes, async/await, and many other features, and compiles to plain JavaScript. Angular built entirely in TypeScript and used as a primary language. You can install it globally as

```
npm install -g typescript
```

Let's see a simple example of TypeScript usage,

```
function greeting(name: string): string {
  return "Hello, " + name;
}

let user = "Sudheer";
```

```
console.log(greeting(user));
```

The greeting method allows only string type as argument.

⬆ Back to Top

252. What are the differences between javascript and typescript

Below are the list of differences between javascript and typescript,

feature	typescript	javascript
Language paradigm	Object oriented programming language	Scripting language
Typing support	Supports static typing	It has dynamic typing
Modules	Supported	Not supported
Interface	It has interfaces concept	Doesn't support interfaces
Optional parameters	Functions support optional parameters	No support of optional parameters for functions

⬆ Back to Top

253. What are the advantages of typescript over javascript

Below are some of the advantages of typescript over javascript,

i. 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.
ii. TypeScript is strongly-typed or supports static typing which allows for checking type correctness at compile time. This is not available in javascript.
iii. TypeScript compiler can compile the .ts files into ES3,ES4 and ES5 unlike ES6 features of javascript which may not be supported in some browsers.

⬆ Back to Top

254. What is an object initializer

An object initializer is an expression that describes the initialization of an Object. The syntax for this expression is represented as a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}). This is also known as literal notation. It is one of the ways to create an object.

```
var initObject = { a: "John", b: 50, c: {} };

console.log(initObject.a); // John
```

⬆ Back to Top

255. What is a constructor method

The constructor method is a special method for creating and initializing an object created within a class. If you do not specify a constructor method, a default constructor is used. The example usage of constructor would be as below,

```
class Employee {
  constructor() {
    this.name = "John";
  }
}

var employeeObject = new Employee();

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

⬆ Back to Top

256. What happens if you write constructor more than once in a class

The "constructor" in a class is a special method and it should be defined only once in a class. i.e, If you write a constructor method more than once in a class it will throw a `SyntaxError` error.

```
class Employee {
  constructor() {
    this.name = "John";
  }
  constructor() {   //  Uncaught SyntaxError: A class may only have one constru
    this.age = 30;
  }
}
```

```
var employeeObject = new Employee();

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

Back to Top

257. How do you call the constructor of a parent class

You can use the `super` keyword to call the constructor of a parent class. Remember that `super()` must be called before using 'this' reference. Otherwise it will cause a reference error. Let's the usage of it,

```
class Square extends Rectangle {
  constructor(length) {
    super(length, length);
    this.name = "Square";
  }

  get area() {
    return this.width * this.height;
  }

  set area(value) {
    this.area = value;
  }
}
```

Back to Top

258. How do you get the prototype of an object

You can use the `Object.getPrototypeOf(obj)` method to return the prototype of the specified object. i.e. The value of the internal `prototype` property. If there are no inherited properties then `null` value is returned.

```
const newPrototype = {};
const newObject = Object.create(newPrototype);

console.log(Object.getPrototypeOf(newObject) === newPrototype); // true
```

Back to Top

259. What happens If I pass string type for getPrototype method

In ES5, it will throw a TypeError exception if the obj parameter isn't an object. Whereas in ES2015, the parameter will be coerced to an `Object`.

```
// ES5
Object.getPrototypeOf("James"); // TypeError: "James" is not an object
// ES2015
Object.getPrototypeOf("James"); // String.prototype
```

⬆ Back to Top

260. How do you set prototype of one object to another

You can use the `Object.setPrototypeOf()` method that sets the prototype (i.e., the internal `Prototype` property) of a specified object to another object or null. For example, if you want to set prototype of a square object to rectangle object would be as follows,

```
Object.setPrototypeOf(Square.prototype, Rectangle.prototype);
Object.setPrototypeOf({}, null);
```

⬆ Back to Top

261. How do you check whether an object can be extendable or not

The `Object.isExtensible()` method is used to determine if an object is extendable or not. i.e, Whether it can have new properties added to it or not.

```
const newObject = {};
console.log(Object.isExtensible(newObject)); //true
```

Note: By default, all the objects are extendable. i.e, The new properties can be added or modified.

⬆ Back to Top

262. How do you prevent an object to extend

The `Object.preventExtensions()` method is used to prevent new properties from ever being added to an object. In other words, it prevents future extensions to the object. Let's see the usage of this property,

```
const newObject = {};
Object.preventExtensions(newObject); // NOT extendable

try {
  Object.defineProperty(newObject, "newProperty", {
    // Adding new property
    value: 100,
  });
} catch (e) {
  console.log(e); // TypeError: Cannot define property newProperty, object is nc
}
```

Back to Top

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

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

i. Object.preventExtensions
ii. Object.seal
iii. Object.freeze

```
var newObject = {};

Object.preventExtensions(newObject); // Prevent objects are non-extensible
Object.isExtensible(newObject); // false

var sealedObject = Object.seal({}); // Sealed objects are non-extensible
Object.isExtensible(sealedObject); // false

var frozenObject = Object.freeze({}); // Frozen objects are non-extensible
Object.isExtensible(frozenObject); // false
```

Back to Top

264. How do you define multiple properties on an object

The `Object.defineProperties()` method is used to define new or modify existing properties directly on an object and returning the object. Let's define multiple properties on an empty object,

```
const newObject = {};

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

⬆ Back to Top

265. What is MEAN in javascript

The MEAN (MongoDB, Express, AngularJS, and Node.js) stack is the most popular open-source JavaScript software tech stack available for building dynamic web apps where you can write both the server-side and client-side halves of the web project entirely in JavaScript.

⬆ Back to Top

266. What Is Obfuscation in javascript

Obfuscation is the deliberate act of creating obfuscated javascript code(i.e, source or machine code) that is difficult for humans to understand. It is something similar to encryption, but a machine can understand the code and execute it. Let's see the below function before Obfuscation,

```
function greeting() {
  console.log("Hello, welcome to JS world");
}
```

And after the code Obfuscation, it would be appeared as below,

```
eval(
  (function (p, a, c, k, e, d) {
    e = function (c) {
      return c;
```

```
    };
    if (!"".replace(/^/, String)) {
      while (c--) {
        d[c] = k[c] || c;
      }
      k = [
        function (e) {
          return d[e];
        },
      ];
      e = function () {
        return "\\w+";
      };
      c = 1;
    }
    while (c--) {
      if (k[c]) {
        p = p.replace(new RegExp("\\b" + e(c) + "\\b", "g"), k[c]);
      }
    }
    return p;
  })(
    "2 1(){0.3('4, 7 6 5 8')}",
    9,
    9,
    "console|greeting|function|log|Hello|JS|to|welcome|world".split("|"),
    0,
    {}
  )
);
```

↑ Back to Top

267. Why do you need Obfuscation

Below are the few reasons for Obfuscation,

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

↑ Back to Top

268. What is Minification

Minification is the process of removing all unnecessary characters(empty spaces are removed) and variables will be renamed without changing it's functionality. It is also a type of obfuscation .

⬆ Back to Top

269. What are the advantages of minification

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

i. Decreases loading times of a web page
ii. Saves bandwidth usages

⬆ Back to Top

270. What are the differences between Obfuscation and Encryption

Below are the main differences between Obfuscation and Encryption,

Feature	Obfuscation	Encryption
Definition	Changing the form of any data in any other form	Changing the form of information to an unreadable format by using a key
A key to decode	It can be decoded without any key	It is required
Target data format	It will be converted to a complex form	Converted into an unreadable format

⬆ Back to Top

271. What are the common tools used for minification

There are many online/offline tools to minify the javascript files,

i. Google's Closure Compiler
ii. UglifyJS2
iii. jsmin
iv. javascript-minifier.com/

v. prettydiff.com

⬆ Back to Top

272. How do you perform form validation using javascript

JavaScript can be used to perform HTML form validation. For example, if the form field is empty, the function needs to notify, and return false, to prevent the form being submitted. Lets' perform user login in an html form,

```
<form name="myForm" onsubmit="return validateForm()" method="post">
  User name: <input type="text" name="uname" />
  <input type="submit" value="Submit" />
</form>
```

And the validation on user login is below,

```
function validateForm() {
  var x = document.forms["myForm"]["uname"].value;
  if (x == "") {
    alert("The username shouldn't be empty");
    return false;
  }
}
```

⬆ Back to Top

273. How do you perform form validation without javascript

You can perform HTML form validation automatically without using javascript. The validation enabled by applying the `required` attribute to prevent form submission when the input is empty.

```
<form method="post">
  <input type="text" name="uname" required />
  <input type="submit" value="Submit" />
</form>
```

Note: Automatic form validation does not work in Internet Explorer 9 or earlier.

⬆ Back to Top

274. What are the DOM methods available for constraint validation

The below DOM methods are available for constraint validation on an invalid input,

i. checkValidity(): It returns true if an input element contains valid data.
ii. setCustomValidity(): It is used to set the validationMessage property of an input element. Let's take an user login form with DOM validations

```
function myFunction() {
  var userName = document.getElementById("uname");
  if (!userName.checkValidity()) {
    document.getElementById("message").innerHTML =
      userName.validationMessage;
  } else {
    document.getElementById("message").innerHTML =
      "Entered a valid username";
  }
}
```

⬆ Back to Top

275. What are the available constraint validation DOM properties

Below are the list of some of the constraint validation DOM properties available,

i. validity: It provides a list of boolean properties related to the validity of an input element.
ii. validationMessage: It displays the message when the validity is false.
iii. willValidate: It indicates if an input element will be validated or not.

⬆ Back to Top

276. What are the list of validity properties

The validity property of an input element provides a set of properties related to the validity of data.

i. customError: It returns true, if a custom validity message is set.
ii. patternMismatch: It returns true, if an element's value does not match its pattern attribute.
iii. rangeOverflow: It returns true, if an element's value is greater than its max attribute.
iv. rangeUnderflow: It returns true, if an element's value is less than its min attribute.

v. stepMismatch: It returns true, if an element's value is invalid according to step attribute.

vi. tooLong: It returns true, if an element's value exceeds its maxLength attribute.

vii. typeMismatch: It returns true, if an element's value is invalid according to type attribute.

viii. valueMissing: It returns true, if an element with a required attribute has no value.

ix. valid: It returns true, if an element's value is valid.

⬆ Back to Top

277. Give an example usage of rangeOverflow property

If an element's value is greater than its max attribute then rangeOverflow property returns true. For example, the below form submission throws an error if the value is more than 100,

```
<input id="age" type="number" max="100" />
<button onclick="myOverflowFunction()">OK</button>


function myOverflowFunction() {
  if (document.getElementById("age").validity.rangeOverflow) {
    alert("The mentioned age is not allowed");
  }
}
```

⬆ Back to Top

278. Is enums feature available in javascript

No, javascript does not natively support enums. But there are different kinds of solutions to simulate them even though they may not provide exact equivalents. For example, you can use freeze or seal on object,

```
var DaysEnum = Object.freeze({"monday":1, "tuesday":2, "wednesday":3, ...})
```

⬆ Back to Top

279. What is an enum

An enum is a type restricting variables to one value from a predefined set of constants. JavaScript has no enums but typescript provides built-in enum support.

```
enum Color {
 RED, GREEN, BLUE
}
```

⬆ Back to Top

280. How do you list all properties of an object

You can use the `Object.getOwnPropertyNames()` method which returns an array of all properties found directly in a given object. Let's the usage of it in an example,

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

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

⬆ Back to Top

281. How do you get property descriptors of an object

You can use the `Object.getOwnPropertyDescriptors()` method which returns all own property descriptors of a given object. The example usage of this method is below,

```
const newObject = {
  a: 1,
  b: 2,
  c: 3,
};
const descriptorsObject = Object.getOwnPropertyDescriptors(newObject);
console.log(descriptorsObject.a.writable); //true
console.log(descriptorsObject.a.configurable); //true
console.log(descriptorsObject.a.enumerable); //true
console.log(descriptorsObject.a.value); // 1
```

⬆ Back to Top

282. What are the attributes provided by a property descriptor

A property descriptor is a record which has the following attributes

i. value: The value associated with the property
ii. writable: Determines whether the value associated with the property can be changed or not
iii. configurable: Returns true if the type of this property descriptor can be changed and if the property can be deleted from the corresponding object.
iv. enumerable: Determines whether the property appears during enumeration of the properties on the corresponding object or not.
v. set: A function which serves as a setter for the property
vi. get: A function which serves as a getter for the property

⬆ Back to Top

283. How do you extend classes

The `extends` keyword is used in class declarations/expressions to create a class which is a child of another class. It can be used to subclass custom classes as well as built-in objects. The syntax would be as below,

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

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

```
class Square extends Rectangle {
  constructor(length) {
    super(length, length);
    this.name = "Square";
  }

  get area() {
    return this.width * this.height;
  }

  set area(value) {
    this.area = value;
  }
}
```

⬆ Back to Top

284. How do I modify the url without reloading the page

The `window.location.url` property will be helpful to modify the url but it reloads the page. HTML5 introduced the `history.pushState()` and `history.replaceState()` methods, which allow you to add and modify history entries, respectively. For example, you can use pushState as below,

```
window.history.pushState("page2", "Title", "/page2.html");
```

⬆ Back to Top

285. How do you check whether an array includes a particular value or not

The `Array#includes()` method is used to determine whether an array includes a particular value among its entries by returning either true or false. Let's see an example to find an element(numeric and string) within an array.

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

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

⬆ Back to Top

286. How do you compare scalar arrays

You can use length and every method of arrays to compare two scalar(compared directly using ===) arrays. The combination of these expressions can give the expected result,

```
const arrayFirst = [1, 2, 3, 4, 5];
const arraySecond = [1, 2, 3, 4, 5];
console.log(
  arrayFirst.length === arraySecond.length &&
    arrayFirst.every((value, index) => value === arraySecond[index])
); // true
```

If you would like to compare arrays irrespective of order then you should sort them before,

```
const arrayFirst = [2, 3, 1, 4, 5];
const arraySecond = [1, 2, 3, 4, 5];
console.log(
  arrayFirst.length === arraySecond.length &&
    arrayFirst.sort().every((value, index) => value === arraySecond[index])
); //true
```

Back to Top

287. How to get the value from get parameters

The `new URL()` object accepts the url string and `searchParams` property of this object can be used to access the get parameters. Remember that you may need to use polyfill or `window.location` to access the URL in older browsers(including IE).

```
let urlString = "http://www.some-domain.com/about.html?x=1&y=2&z=3"; //window.lo
let url = new URL(urlString);
let parameterZ = url.searchParams.get("z");
console.log(parameterZ); // 3
```

Back to Top

288. How do you print numbers with commas as thousand separators

You can use the `Number.prototype.toLocaleString()` method which returns a string with a language-sensitive representation such as thousand separator,currency etc of this number.

```
function convertToThousandFormat(x) {
  return x.toLocaleString(); // 12,345.679
}

console.log(convertToThousandFormat(12345.6789));
```

Back to Top

289. What is the difference between java and javascript

Both are totally unrelated programming languages and no relation between them. Java is statically typed, compiled, runs on its own VM. Whereas Javascript is dynamically typed, interpreted, and runs in a browser and nodejs environments. Let's see the major differences in a tabular format,

Feature	Java	JavaScript
Typed	It's a strongly typed language	It's a dynamic typed language
Paradigm	Object oriented programming	Prototype based programming
Scoping	Block scoped	Function-scoped
Concurrency	Thread based	event based
Memory	Uses more memory	Uses less memory. Hence it will be used for web pages

⬆ Back to Top

290. Does JavaScript supports namespace

JavaScript doesn't support namespace by default. So if you create any element(function, method, object, variable) then it becomes global and pollutes the global namespace. Let's take an example of defining two functions without any namespace,

```
function func1() {
  console.log("This is a first definition");
}
function func1() {
  console.log("This is a second definition");
}
func1(); // This is a second definition
```

It always calls the second function definition. In this case, namespace will solve the name collision problem.

⬆ Back to Top

291. How do you declare namespace

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

i. **Using Object Literal Notation:** Let's wrap variables and functions inside an Object literal which acts as a namespace. After that you can access them using object notation

```
var namespaceOne = {
   function func1() {
       console.log("This is a first definition");
   }
}
var namespaceTwo = {
     function func1() {
         console.log("This is a second definition");
     }
 }
namespaceOne.func1(); // This is a first definition
namespaceTwo.func1(); // This is a second definition
```

i. **Using IIFE (Immediately invoked function expression):** The outer pair of parentheses of IIFE creates a local scope for all the code inside of it and makes the anonymous function a function expression. Due to that, you can create the same function in two different function expressions to act as a namespace.

```
(function () {
  function fun1() {
    console.log("This is a first definition");
  }
  fun1();
})();

(function () {
  function fun1() {
    console.log("This is a second definition");
  }
  fun1();
})();
```

i. **Using a block and a let/const declaration:** In ECMAScript 6, you can simply use a block and a let declaration to restrict the scope of a variable to a block.

```
{
  let myFunction = function fun1() {
    console.log("This is a first definition");
  };
```

```
  myFunction();
}
//myFunction(): ReferenceError: myFunction is not defined.

{
  let myFunction = function fun1() {
    console.log("This is a second definition");
  };
  myFunction();
}
//myFunction(): ReferenceError: myFunction is not defined.
```

Back to Top

292. How do you invoke javascript code in an iframe from parent page

Initially iFrame needs to be accessed using either `document.getElementBy` or `window.frames` . After that `contentWindow` property of iFrame gives the access for targetFunction

```
document.getElementById("targetFrame").contentWindow.targetFunction();
window.frames[0].frameElement.contentWindow.targetFunction(); // Accessing ifram
```

Back to Top

293. How do get the timezone offset from date

You can use the `getTimezoneOffset` method of the date object. This method returns the time zone difference, in minutes, from current locale (host system settings) to UTC

```
var offset = new Date().getTimezoneOffset();
console.log(offset); // -480
```

Back to Top

294. How do you load CSS and JS files dynamically

You can create both link and script elements in the DOM and append them as child to head tag. Let's create a function to add script and style resources as below,

```
function loadAssets(filename, filetype) {
  if (filetype == "css") {
    // External CSS file
    var fileReference = document.createElement("link");
    fileReference.setAttribute("rel", "stylesheet");
    fileReference.setAttribute("type", "text/css");
    fileReference.setAttribute("href", filename);
  } else if (filetype == "js") {
    // External JavaScript file
    var fileReference = document.createElement("script");
    fileReference.setAttribute("type", "text/javascript");
    fileReference.setAttribute("src", filename);
  }
  if (typeof fileReference != "undefined")
    document.getElementsByTagName("head")[0].appendChild(fileReference);
}
```

⬆ Back to Top

295. What are the different methods to find HTML elements in DOM

If you want to access any element in an HTML page, you need to start with accessing the document object. Later you can use any of the below methods to find the HTML element,

i. document.getElementById(id): It finds an element by Id
ii. document.getElementsByTagName(name): It finds an element by tag name
iii. document.getElementsByClassName(name): It finds an element by class name

⬆ Back to Top

296. What is jQuery

jQuery is a popular cross-browser JavaScript library that provides Document Object Model (DOM) traversal, event handling, animations and AJAX interactions by minimizing the discrepancies across browsers. It is widely famous with its philosophy of “Write less, do more”. For example, you can display welcome message on the page load using jQuery as below,

```
$(document).ready(function () {
  // It selects the document and apply the function on page load
  alert("Welcome to jQuery world");
});
```

Note: You can download it from jquery's official site or install it from CDNs, like google.

⬆ Back to Top

297. What is V8 JavaScript engine

V8 is an open source high-performance JavaScript engine used by the Google Chrome browser, written in C++. It is also being used in the node.js project. It implements ECMAScript and WebAssembly, and runs on Windows 7 or later, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors. **Note:** It can run standalone, or can be embedded into any C++ application.

⬆ Back to Top

298. Why do we call javascript as dynamic language

JavaScript is a loosely typed or a dynamic language because variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned/reassigned with values of all types.

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

⬆ Back to Top

299. What is a void operator

The `void` operator evaluates the given expression and then returns undefined(i.e, without returning value). The syntax would be as below,

```
void expression;
void expression;
```

Let's display a message without any redirection or reload

```
<a href="javascript:void(alert('Welcome to JS world'))">
  Click here to see a message
</a>
```

Note: This operator is often used to obtain the undefined primitive value, using "void(0)".

⬆ Back to Top

300. How to set the cursor to wait

The cursor can be set to wait in JavaScript by using the property "cursor". Let's perform this behavior on page load using the below function.

```
function myFunction() {
  window.document.body.style.cursor = "wait";
}
```

and this function invoked on page load

```
<body onload="myFunction()"></body>
```

⬆ Back to Top

301. How do you create an infinite loop

You can create infinite loops using for and while loops without using any expressions. The for loop construct or syntax is better approach in terms of ESLint and code optimizer tools,

```
for (;;) {}
while (true) {}
```

⬆ Back to Top

302. Why do you need to avoid with statement

JavaScript's with statement was intended to provide a shorthand for writing recurring accesses to objects. So it can help reduce file size by reducing the need to repeat a lengthy object reference without performance penalty. Let's take an example where it is used to avoid redundancy when accessing an object several times.

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

Using `with` it turns this into:

```
with (a.b.c) {
  greeting = "welcome";
  age = 32;
}
```

But this `with` statement creates performance problems since one cannot predict whether an argument will refer to a real variable or to a property inside the with argument.

⬆ Back to Top

303. What is the output of below for loops

```
for (var i = 0; i < 4; i++) {
  // global scope
  setTimeout(() => console.log(i));
}

for (let i = 0; i < 4; i++) {
  // block scope
  setTimeout(() => console.log(i));
}
```

The output of the above for loops is 4 4 4 4 and 0 1 2 3

Explanation: Due to the event queue/loop of javascript, the `setTimeout` callback function is called after the loop has been executed. Since the variable i is declared with the `var` keyword it became a global variable and the value was equal to 4 using iteration when the time `setTimeout` function is invoked. Hence, the output of the first loop is `4 4 4 4`.

Whereas in the second loop, the variable i is declared as the `let` keyword it becomes a block scoped variable and it holds a new value(0, 1 ,2 3) for each iteration. Hence, the output of the first loop is `0 1 2 3`.

⬆ Back to Top

304. List down some of the features of ES6

Below are the list of some new features of ES6,

i. Support for constants or immutable variables
ii. Block-scope support for variables, constants and functions
iii. Arrow functions
iv. Default parameters
v. Rest and Spread Parameters
vi. Template Literals
vii. Multi-line Strings
viii. Destructuring Assignment
ix. Enhanced Object Literals
x. Promises
xi. Classes
xii. Modules

⬆ Back to Top

305. What is ES6

ES6 is the sixth edition of the javascript language and it was released in June 2015. It was initially known as ECMAScript 6 (ES6) and later renamed to ECMAScript 2015. Almost all the modern browsers support ES6 but for the old browsers there are many transpilers, like Babel.js etc.

⬆ Back to Top

306. Can I redeclare let and const variables

No, you cannot redeclare let and const variables. If you do, it throws below error

```
Uncaught SyntaxError: Identifier 'someVariable' has already been declared
```

Explanation: The variable declaration with `var` keyword refers to a function scope and the variable is treated as if it were declared at the top of the enclosing scope due to hoisting feature. So all the multiple declarations contributing to the same hoisted variable without any error. Let's take an example of re-declaring variables in the same scope for both var and let/const variables.

```
var name = "John";
function myFunc() {
  var name = "Nick";
```

```
  var name = "Abraham"; // Re-assigned in the same function block
  alert(name); // Abraham
}
myFunc();
alert(name); // John
```

The block-scoped multi-declaration throws syntax error,

```
let name = "John";
function myFunc() {
  let name = "Nick";
  let name = "Abraham"; // Uncaught SyntaxError: Identifier 'name' has already b
  alert(name);
}

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

Back to Top

307. Is const variable makes the value immutable

No, the const variable doesn't make the value immutable. But it disallows subsequent assignments(i.e, You can declare with assignment but can't assign another value later)

```
const userList = [];
userList.push("John"); // Can mutate even though it can't re-assign
console.log(userList); // ['John']
```

Back to Top

308. What are default parameters

In 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
```

⬆ Back to Top

309. What are template literals

Template literals or template strings are string literals allowing embedded expressions. These are enclosed by the back-tick (`) character instead of double or single quotes. In 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.

⬆ Back to Top

310. How do you write multi-line strings in template literals

In ES5, you would have to use newline escape characters('\n') and concatenation symbols(+) in order to get multi-line strings.

```
console.log("This is string sentence 1\n" + "This is string sentence 2");
```

Whereas in ES6, You don't need to mention any newline sequence character,

```
console.log(`This is string sentence
'This is string sentence 2`);
```

⬆ Back to Top

311. What are nesting templates

The nesting template is a feature supported within template literals syntax to allow inner backticks inside a placeholder ${ } within the template. For example, the below nesting template is used to display the icons based on user permissions whereas outer template checks for platform type,

```
const iconStyles = `icon ${
  isMobilePlatform()
    ? ""
    : `icon-${user.isAuthorized ? "submit" : "disabled"}`
}`;
```

You can write the above use case without nesting template features as well. However, the nesting template feature is more compact and readable.

```
//Without nesting templates
 const iconStyles = `icon ${ isMobilePlatform() ? '' :
  (user.isAuthorized ? 'icon-submit' : 'icon-disabled'}`;
```

⬆ Back to Top

312. What are tagged templates

Tagged templates are the advanced form of templates in which tags allow you to parse template literals with a function. The tag function accepts the first parameter as an array of strings and remaining parameters as expressions. This function can also return manipulated strings based on parameters. Let's see the usage of this tagged template behavior of an IT professional skill set in an organization,

```
var user1 = "John";
var skill1 = "JavaScript";
var experience1 = 15;
```

```
var user2 = "Kane";
var skill2 = "JavaScript";
var experience2 = 5;

function myInfoTag(strings, userExp, experienceExp, skillExp) {
  var str0 = strings[0]; // "Mr/Ms. "
  var str1 = strings[1]; // " is a/an "
  var str2 = strings[2]; // "in"

  var expertiseStr;
  if (experienceExp > 10) {
    expertiseStr = "expert developer";
  } else if (skillExp > 5 && skillExp <= 10) {
    expertiseStr = "senior developer";
  } else {
    expertiseStr = "junior developer";
  }

  return `${str0}${userExp}${str1}${expertiseStr}${str2}${skillExp}`;
}

var output1 = myInfoTag`Mr/Ms. ${user1} is a/an ${experience1} in ${skill1}`;
var output2 = myInfoTag`Mr/Ms. ${user2} is a/an ${experience2} in ${skill2}`;

console.log(output1); // Mr/Ms. John is a/an expert developer in JavaScript
console.log(output2); // Mr/Ms. Kane is a/an junior developer in JavaScript
```

⬆ Back to Top

313. What are raw strings

ES6 provides a raw strings feature using the `String.raw()` method which is used to get the raw string form of template strings. This feature allows you to access the raw strings as they were entered, without processing escape sequences. For example, the usage would be as below,

```
var calculationString = String.raw`The sum of numbers is \n${
  1 + 2 + 3 + 4
}!`;
console.log(calculationString); // The sum of numbers is 10
```

If you don't use raw strings, the newline character sequence will be processed by displaying the output in multiple lines

```
var calculationString = `The sum of numbers is \n${1 + 2 + 3 + 4}!`;
console.log(calculationString);
// The sum of numbers is
// 10
```

Also, the raw property is available on the first argument to the tag function

```
function tag(strings) {
  console.log(strings.raw[0]);
}
```

⬆ Back to Top

314. What is destructuring assignment

The destructuring assignment is a JavaScript expression that makes it possible to unpack values from arrays or properties from objects into distinct variables. Let's get the month values from an array using destructuring assignment

```
var [one, two, three] = ["JAN", "FEB", "MARCH"];

console.log(one); // "JAN"
console.log(two); // "FEB"
console.log(three); // "MARCH"
```

and you can get user properties of an object using destructuring assignment,

```
var { name, age } = { name: "John", age: 32 };

console.log(name); // John
console.log(age); // 32
```

⬆ Back to Top

315. What are default values in destructuring assignment

A variable can be assigned a default value when the value unpacked from the array or object is undefined during destructuring assignment. It helps to avoid setting default values separately for each assignment. Let's take an example for both arrays and object use cases,

Arrays destructuring:

```
var x, y, z;

[x = 2, y = 4, z = 6] = [10];
console.log(x); // 10
console.log(y); // 4
console.log(z); // 6
```

Objects destructuring:

```
var { x = 2, y = 4, z = 6 } = { x: 10 };

console.log(x); // 10
console.log(y); // 4
console.log(z); // 6
```

⬆ Back to Top

316. How do you swap variables in destructuring assignment

If you don't use destructuring assignment, swapping two values requires a temporary variable. Whereas using a destructuring feature, two variable values can be swapped in one destructuring expression. Let's swap two number variables in array destructuring assignment,

```
var x = 10,
  y = 20;

[x, y] = [y, x];
console.log(x); // 20
console.log(y); // 10
```

⬆ Back to Top

317. What are enhanced object literals

Object literals make it easy to quickly create objects with properties inside the curly braces. For example, it provides shorter syntax for common object property definition as below.

```
//ES6
var x = 10,
  y = 20;
obj = { x, y };
console.log(obj); // {x: 10, y:20}
//ES5
var x = 10,
  y = 20;
obj = { x: x, y: y };
console.log(obj); // {x: 10, y:20}
```

⬆ Back to Top

318. What are dynamic imports

The dynamic imports using `import()` function syntax allows us to load modules on demand by using promises or the async/await syntax. Currently this feature is in stage4 proposal. The main advantage of dynamic imports is reduction of our bundle's sizes, the size/payload response of our requests and overall improvements in the user experience. The syntax of dynamic imports would be as below,

```
import("./Module").then((Module) => Module.method());
```

⬆ Back to Top

319. What are the use cases for dynamic imports

Below are some of the use cases of using dynamic imports over static imports,

i. Import a module on-demand or conditionally. For example, if you want to load a polyfill on legacy browser

```
if (isLegacyBrowser()) {
    import(···)
    .then(···);
}
```

i. Compute the module specifier at runtime. For example, you can use it for internationalization.

```
import(`messages_${getLocale()}.js`).then(···);
```

i. Import a module from within a regular script instead a module.

⬆ Back to Top

320. What are typed arrays

Typed arrays are array-like objects from ECMAScript 6 API for handling binary data. JavaScript provides 8 Typed array types,

i. Int8Array: An array of 8-bit signed integers
ii. Int16Array: An array of 16-bit signed integers
iii. Int32Array: An array of 32-bit signed integers
iv. Uint8Array: An array of 8-bit unsigned integers
v. Uint16Array: An array of 16-bit unsigned integers
vi. Uint32Array: An array of 32-bit unsigned integers
vii. Float32Array: An array of 32-bit floating point numbers
viii. Float64Array: An array of 64-bit floating point numbers

For example, you can create an array of 8-bit signed integers as below

```
const a = new Int8Array();
// You can pre-allocate n bytes
const bytes = 1024;
const a = new Int8Array(bytes);
```

⬆ Back to Top

321. What are the advantages of module loaders

The module loaders provides the below features,

i. Dynamic loading
ii. State isolation
iii. Global namespace isolation
iv. Compilation hooks
v. Nested virtualization

⬆ Back to Top

322. What is collation

Collation is used for sorting a set of strings and searching within a set of strings. It is parameterized by locale and aware of Unicode. Let's take comparison and sorting features,

i. **Comparison:**

```
var list = ["ä", "a", "z"]; // In German,  "ä" sorts with "a" Whereas in Swedish
var l10nDE = new Intl.Collator("de");
var l10nSV = new Intl.Collator("sv");
console.log(l10nDE.compare("ä", "z") === -1); // true
console.log(l10nSV.compare("ä", "z") === +1); // true
```

i. **Sorting:**

```
var list = ["ä", "a", "z"]; // In German,  "ä" sorts with "a" Whereas in Swedish
var l10nDE = new Intl.Collator("de");
var l10nSV = new Intl.Collator("sv");
console.log(list.sort(l10nDE.compare)); // [ "a", "ä", "z" ]
console.log(list.sort(l10nSV.compare)); // [ "a", "z", "ä" ]
```

⬆ Back to Top

323. What is for...of statement

The for...of statement creates a loop iterating over iterable objects or elements such as built-in String, Array, Array-like objects (like arguments or NodeList), TypedArray, Map, Set, and user-defined iterables. The basic usage of for...of statement on arrays would be as below,

```
let arrayIterable = [10, 20, 30, 40, 50];

for (let value of arrayIterable) {
  value++;
  console.log(value); // 11 21 31 41 51
}
```

⬆ Back to Top

324. What is the output of below spread operator array

```
[..."John Resig"];
```

The output of the array is ['J', 'o', 'h', 'n', '', 'R', 'e', 's', 'i', 'g'] **Explanation:** The string is an iterable type and the spread operator within an array maps every character of an iterable to one element. Hence, each character of a string becomes an element within an Array.

⬆ Back to Top

325. Is PostMessage secure

Yes, postMessages can be considered very secure as long as the programmer/developer is careful about checking the origin and source of an arriving message. But if you try to send/receive a message without verifying its source will create cross-site scripting attacks.

⬆ Back to Top

326. What are the problems with postmessage target origin as wildcard

The second argument of postMessage method specifies which origin is allowed to receive the message. If you use the wildcard "*" as an argument then any origin is allowed to receive the message. In this case, there is no way for the sender window to know if the target window is at the target origin when sending the message. If the target window has been navigated to another origin, the other origin would receive the data. Hence, this may lead to XSS vulnerabilities.

```
targetWindow.postMessage(message, "*");
```

⬆ Back to Top

327. How do you avoid receiving postMessages from attackers

Since the listener listens for any message, an attacker can trick the application by sending a message from the attacker's origin, which gives an impression that the receiver received the message from the actual sender's window. You can avoid this issue by validating the origin of the message on the receiver's end using the "message.origin" attribute. For examples, let's check the sender's origin http://www.some-sender.com on receiver side www.some-receiver.com,

```
//Listener on http://www.some-receiver.com/
window.addEventListener("message", function(message){
    if(/^http://www\.some-sender\.com$/.test(message.origin)){
```

```
            console.log('You received the data from valid sender', message.data);
        }
    });
```

Back to Top

328. Can I avoid using postMessages completely

You cannot avoid using postMessages completely(or 100%). Even though your application doesn't use postMessage considering the risks, a lot of third party scripts use postMessage to communicate with the third party service. So your application might be using postMessage without your knowledge.

Back to Top

329. Is postMessages synchronous

The postMessages are synchronous in IE8 browser but they are asynchronous in IE9 and all other modern browsers (i.e, IE9+, Firefox, Chrome, Safari).Due to this asynchronous behaviour, we use a callback mechanism when the postMessage is returned.

Back to Top

330. What paradigm is Javascript

JavaScript is a multi-paradigm language, supporting imperative/procedural programming, Object-Oriented Programming and functional programming. JavaScript supports Object-Oriented Programming with prototypical inheritance.

Back to Top

331. What is the difference between internal and external javascript

Internal JavaScript: It is the source code within the script tag. **External JavaScript:** The source code is stored in an external file(stored with .js extension) and referred with in the tag.

Back to Top

332. Is JavaScript faster than server side script

Yes, JavaScript is faster than server side 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.

⬆ Back to Top

333. How do you get the status of a checkbox

You can apply the `checked` property on the selected checkbox in the DOM. If the value is `True` 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
```

⬆ Back to Top

334. What is the purpose of double tilde operator

The double tilde operator(~~) is known as double NOT bitwise operator. This operator is going to be a quicker substitute for Math.floor().

⬆ Back to Top

335. How do you convert character to ASCII code

You can use the `String.prototype.charCodeAt()` method to convert string characters to ASCII numbers. For example, let's find ASCII code for the first letter of 'ABC' string,

```
"ABC".charCodeAt(0); // returns 65
```

Whereas `String.fromCharCode()` method converts numbers to equal ASCII characters.

```
String.fromCharCode(65, 66, 67); // returns 'ABC'
```

⬆ Back to Top

336. What is ArrayBuffer

An ArrayBuffer object is used to represent a generic, fixed-length raw binary data buffer. You can create it as below,

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16
alert(buffer.byteLength); // 16
```

To manipulate an ArrayBuffer, we need to use a “view” object.

```
//Create a DataView referring to the buffer
let view = new DataView(buffer);
```

⬆ Back to Top

337. What is the output of below string expression

```
console.log("Welcome to JS world"[0]);
```

The output of the above expression is "W". **Explanation:** The bracket notation with specific index on a string returns the character at a specific location. Hence, it returns the character "W" of the string. Since this is not supported in IE7 and below versions, you may need to use the .charAt() method to get the desired result.

⬆ Back to Top

338. What is the purpose of Error object

The Error constructor creates an error object and the instances of error objects are thrown when runtime errors occur. The Error object can also be used as a base object for user-defined exceptions. The syntax of error object would be as below,

```
new Error([message[, fileName[, lineNumber]]])
```

You can throw user defined exceptions or errors using Error object in try...catch block as below,

```
try {
  if (withdraw > balance)
    throw new Error("Oops! You don't have enough balance");
```

```
} catch (e) {
  console.log(e.name + ": " + e.message);
}
```

Back to Top

339. What is the purpose of EvalError object

The EvalError object indicates an error regarding the global `eval()` function. Even though this exception is not thrown by JavaScript anymore, the EvalError object remains for compatibility. The syntax of this expression would be as below,

```
new EvalError([message[, fileName[, lineNumber]]])
```

You can throw EvalError with in try...catch block as below,

```
try {
  throw new EvalError('Eval function error', 'someFile.js', 100);
} catch (e) {
  console.log(e.message, e.name, e.fileName);              // "Eval function err
```

Back to Top

340. What are the list of cases error thrown from non-strict mode to strict mode

When you apply 'use strict'; syntax, some of the below cases will throw a SyntaxError before executing the script

i. When you use Octal syntax

```
var n = 022;
```

i. Using `with` statement
ii. When you use delete operator on a variable name
iii. Using eval or arguments as variable or function argument name
iv. When you use newly reserved keywords
v. When you declare a function in a block

```
if (someCondition) {
  function f() {}
}
```

Hence, the errors from above cases are helpful to avoid errors in development/production environments.

⬆ Back to Top

341. Do all objects have prototypes

No. All objects have prototypes except for the base object which is created by the user, or an object that is created using the new keyword.

⬆ Back to Top

342. What is the difference between a parameter and an argument

Parameter is the variable name of a function definition whereas an argument represents the value given to a function when it is invoked. Let's explain this with a simple function

```
function myFunction(parameter1, parameter2, parameter3) {
  console.log(arguments[0]); // "argument1"
  console.log(arguments[1]); // "argument2"
  console.log(arguments[2]); // "argument3"
}
myFunction("argument1", "argument2", "argument3");
```

⬆ Back to Top

343. What is the purpose of some method in arrays

The some() method is used to test whether at least one element in the array passes the test implemented by the provided function. The method returns a boolean value. Let's take an example to test for any odd elements,

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var odd = (element) => element % 2 !== 0;

console.log(array.some(odd)); // true (the odd element exists)
```

⬆ Back to Top

344. How do you combine two or more arrays

The concat() method is used to join two or more arrays by returning a new array containing all the elements. The syntax would be as below,

```
array1.concat(array2, array3, ..., arrayX)
```

Let's take an example of array's concatenation with veggies and fruits arrays,

```
var veggies = ["Tomato", "Carrot", "Cabbage"];
var fruits = ["Apple", "Orange", "Pears"];
var veggiesAndFruits = veggies.concat(fruits);
console.log(veggiesAndFruits); // Tomato, Carrot, Cabbage, Apple, Orange, Pears
```

⬆ Back to Top

345. What is the difference between Shallow and Deep copy

There are two ways to copy an object,

Shallow Copy: Shallow copy is a bitwise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

Example

```
var empDetails = {
  name: "John",
  age: 25,
  expertise: "Software Developer",
};
```

to create a duplicate

```
var empDetailsShallowCopy = empDetails; //Shallow copying!
```

if we change some property value in the duplicate one like this:

```
empDetailsShallowCopy.name = "Johnson";
```

The above statement will also change the name of `empDetails`, since we have a shallow copy. That means we're losing the original data as well.

Deep copy: A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

Example

```
var empDetails = {
  name: "John",
  age: 25,
  expertise: "Software Developer",
};
```

Create a deep copy by using the properties from the original object into new variable

```
var empDetailsDeepCopy = {
  name: empDetails.name,
  age: empDetails.age,
  expertise: empDetails.expertise,
};
```

Now if you change `empDetailsDeepCopy.name`, it will only affect `empDetailsDeepCopy` & not `empDetails`

⬆ Back to Top

346. How do you create specific number of copies of a string

The `repeat()` method is used to construct and return a new string which contains the specified number of copies of the string on which it was called, concatenated together. Remember that this method has been added to the ECMAScript 2015 specification. Let's take an example of Hello string to repeat it 4 times,

```
"Hello".repeat(4); // 'HelloHelloHelloHello'
```

347. How do you return all matching strings against a regular expression

The `matchAll()` method can be used to return an iterator of all results matching a string against a regular expression. For example, the below example returns an array of matching string results against a regular expression,

```
let regexp = /Hello(\d?))/g;
let greeting = "Hello1Hello2Hello3";

let greetingList = [...greeting.matchAll(regexp)];

console.log(greetingList[0]); //Hello1
console.log(greetingList[1]); //Hello2
console.log(greetingList[2]); //Hello3
```

⬆ Back to Top

348. How do you trim a string at the beginning or ending

The `trim` method of string prototype is used to trim on both sides of a string. But if you want to trim especially at the beginning or ending of the string then you can use `trimStart/trimLeft` and `trimEnd/trimRight` methods. Let's see an example of these methods on a greeting message,

```
var greeting = "   Hello, Goodmorning!   ";

console.log(greeting); // "   Hello, Goodmorning!   "
console.log(greeting.trimStart()); // "Hello, Goodmorning!   "
console.log(greeting.trimLeft()); // "Hello, Goodmorning!   "

console.log(greeting.trimEnd()); // "   Hello, Goodmorning!"
console.log(greeting.trimRight()); // "   Hello, Goodmorning!"
```

⬆ Back to Top

349. What is the output of below console statement with unary operator

Let's take console statement with unary operator as given below,

```
console.log(+"Hello");
```

The output of the above console log statement returns NaN. Because the element is prefixed by the unary operator and the JavaScript interpreter will try to convert that element into a number type. Since the conversion fails, the value of the statement results in NaN value.

⬆ Back to Top

350. Does javascript uses mixins

Mixin is a generic object-oriented programming term - is a class containing methods that can be used by other classes without a need to inherit from it. In JavaScript we can only inherit from a single object. ie. There can be only one `[[prototype]]` for an object.

But sometimes we require to extend more than one, to overcome this we can use Mixin which helps to copy methods to the prototype of another class.

Say for instance, we've two classes `User` and `CleanRoom`. Suppose we need to add `CleanRoom` functionality to `User`, so that user can clean the room at demand. Here's where concept called mixins comes into picture.

```
// mixin
let cleanRoomMixin = {
  cleanRoom() {
    alert(`Hello ${this.name}, your room is clean now`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  },
};

// usage:
class User {
  constructor(name) {
    this.name = name;
  }
}

// copy the methods
Object.assign(User.prototype, cleanRoomMixin);

// now User can clean the room
new User("Dude").cleanRoom(); // Hello Dude, your room is clean now!
```

⬆ Back to Top

351. What is a thunk function

A thunk is just a function which delays the evaluation of the value. It doesn't take any arguments but gives the value whenever you invoke the thunk. i.e, It is used not to execute now but it will be sometime in the future. Let's take a synchronous example,

```
const add = (x, y) => x + y;

const thunk = () => add(2, 3);

thunk(); // 5
```

⬆ Back to Top

352. What are asynchronous thunks

The asynchronous thunks are useful to make network requests. Let's see an example of network requests,

```
function fetchData(fn) {
  fetch("https://jsonplaceholder.typicode.com/todos/1")
    .then((response) => response.json())
    .then((json) => fn(json));
}

const asyncThunk = function () {
  return fetchData(function getData(data) {
    console.log(data);
  });
};

asyncThunk();
```

The `getData` function won't be called immediately but it will be invoked only when the data is available from API endpoint. The setTimeout function is also used to make our code asynchronous. The best real time example is redux state management library which uses the asynchronous thunks to delay the actions to dispatch.

⬆ Back to Top

353. What is the output of below function calls

Code snippet:

```
const circle = {
  radius: 20,
  diameter() {
    return this.radius * 2;
  },
  perimeter: () => 2 * Math.PI * this.radius,
};


console.log(circle.diameter());
console.log(circle.perimeter());
```

Output:

The output is 40 and NaN. Remember that diameter is a regular function, whereas the value of perimeter is an arrow function. The `this` keyword of a regular function(i.e, diameter) refers to the surrounding scope which is a class(i.e, Shape object). Whereas this keyword of perimeter function refers to the surrounding scope which is a window object. Since there is no radius property on window objects it returns an undefined value and the multiple of number value returns NaN value.

⬆ Back to Top

354. How to remove all line breaks from a string

The easiest approach is using regular expressions to detect and replace newlines in the string. In this case, we use replace function along with string to replace with, which in our case is an empty string.

```
function remove_linebreaks( var message ) {
    return message.replace( /[\r\n]+/gm, "" );
}
```

In the above expression, g and m are for global and multiline flags.

⬆ Back to Top

355. What is the difference between reflow and repaint

A *repaint* occurs when changes are made which affect the visibility of an element, but not its layout. Examples of this include outline, visibility, or background color. A *reflow* involves changes that affect the layout of a portion of the page (or the whole page). Resizing the browser window, changing the font, content changing (such as user typing text), using JavaScript methods involving computed styles, adding or removing elements from the DOM, and changing an element's classes are a few of the things that can trigger reflow. Reflow of an element causes the subsequent reflow of all child and ancestor elements as well as any elements following it in the DOM.

⬆ Back to Top

356. What happens with negating an array

Negating an array with `!` character will coerce the array into a boolean. Since Arrays are considered to be truthy So negating it will return `false`.

```
console.log(![]); // false
```

⬆ Back to Top

357. What happens if we add two arrays

If you add two arrays together, it will convert them both to strings and concatenate them. For example, the result of adding arrays would be as below,

```
console.log(["a"] + ["b"]); // "ab"
console.log([] + []); // ""
console.log(![] + []); // "false", because ![] returns false.
```

⬆ Back to Top

358. What is the output of prepend additive operator on falsy values

If you prepend the additive(+) operator on falsy values(null, undefined, NaN, false, ""), the falsy value converts to a number value zero. Let's display them on browser console as below,

```
console.log(+null); // 0
console.log(+undefined); // NaN
console.log(+false); // 0
```

```
console.log(+NaN); // NaN
console.log(+""); // 0
```

Back to Top

359. How do you create self string using special characters

The self string can be formed with the combination of `[]()!+` characters. You need to remember the below conventions to achieve this pattern.

i. Since Arrays are truthful values, negating the arrays will produce false: ![] === false
ii. As per JavaScript coercion rules, the addition of arrays together will toString them: [] + [] === ""
iii. Prepend an array with + operator will convert an array to false, the negation will make it true and finally converting the result will produce value '1': +(!(+[])) === 1

By applying the above rules, we can derive below conditions

```
(![] + [] === "false" + !+[]) === 1;
```

Now the character pattern would be created as below,

```
     s              e              l              f
 ^^^^^^^^^^^^^  ^^^^^^^^^^^^^  ^^^^^^^^^^^^^  ^^^^^^^^^^^^^

 (![] + [])[3] + (![] + [])[4] + (![] + [])[2] + (![] + [])[0]
 ^^^^^^^^^^^^^  ^^^^^^^^^^^^^  ^^^^^^^^^^^^^  ^^^^^^^^^^^^^
(![] + [])[+!+[]+!+[]+!+[]] +
(![] + [])[+!+[]+!+[]+!+[]+!+[]] +
(![] + [])[+!+[]+!+[]] +
(![] + [])[+[]]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
(![]+[])[+!+[]+!+[]+!+[]]+(![]+[])[+!+[]+!+[]+!+[]+!+[]]+(![]+[])[+!+[]+!+[]]+(!
```

Back to Top

360. How do you remove falsy values from an array

You can apply the filter method on the array by passing Boolean as a parameter. This way it removes all falsy values(0, undefined, null, false and "") from the array.

```
const myArray = [false, null, 1, 5, undefined];
myArray.filter(Boolean); // [1, 5] // is same as myArray.filter(x => x);
```

⬆ Back to Top

361. How do you get unique values of an array

You can get unique values of an array with the combination of `Set` and rest expression/spread(...) syntax.

```
console.log([...new Set([1, 2, 4, 4, 3])]); // [1, 2, 4, 3]
```

⬆ Back to Top

362. What is destructuring aliases

Sometimes you would like to have a destructured variable with a different name than the property name. In that case, you'll use a `: newName` to specify a name for the variable. This process is called destructuring aliases.

```
const obj = { x: 1 };
// Grabs obj.x as as { otherName }
const { x: otherName } = obj;
```

⬆ Back to Top

363. How do you map the array values without using map method

You can map the array values without using the `map` method by just using the `from` method of Array. Let's map city names from Countries array,

```
const countries = [
  { name: "India", capital: "Delhi" },
  { name: "US", capital: "Washington" },
  { name: "Russia", capital: "Moscow" },
  { name: "Singapore", capital: "Singapore" },
  { name: "China", capital: "Beijing" },
  { name: "France", capital: "Paris" },
];
```

```
const cityNames = Array.from(countries, ({ capital }) => capital);
console.log(cityNames); // ['Delhi, 'Washington', 'Moscow', 'Singapore', 'Beijir
```

⬆ Back to Top

364. How do you empty an array

You can empty an array quickly by setting the array length to zero.

```
let cities = ["Singapore", "Delhi", "London"];
cities.length = 0; // cities becomes []
```

⬆ Back to Top

365. How do you rounding numbers to certain decimals

You can round numbers to a certain number of decimals using `toFixed` method from native javascript.

```
let pie = 3.141592653;
pie = pie.toFixed(3); // 3.142
```

⬆ Back to Top

366. What is the easiest way to convert an array to an object

You can convert an array to an object with the same data using spread(...) operator.

```
var fruits = ["banana", "apple", "orange", "watermelon"];
var fruitsObject = { ...fruits };
console.log(fruitsObject); // {0: "banana", 1: "apple", 2: "orange", 3: "waterme
```

⬆ Back to Top

367. How do you create an array with some data

You can create an array with some data or an array with the same values using `fill` method.

```
var newArray = new Array(5).fill("0");
console.log(newArray); // ["0", "0", "0", "0", "0"]
```

Back to Top

368. What are the placeholders from console object

Below are the list of placeholders available from console object,

i. %o — It takes an object,
ii. %s — It takes a string,
iii. %d — It is used for a decimal or integer These placeholders can be represented in the console.log as below

```
const user = { name: "John", id: 1, city: "Delhi" };
console.log(
  "Hello %s, your details %o are available in the object form",
  "John",
  user
); // Hello John, your details {name: "John", id: 1, city: "Delhi"} are availabl
```

Back to Top

369. Is it possible to add CSS to console messages

Yes, you can apply CSS styles to console messages similar to html text on the web page.

```
console.log(
  "%c The text has blue color, with large font and red background",
  "color: blue; font-size: x-large; background: red"
);
```

The text will be displayed as below,

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

Back to Top

370. What is the purpose of dir method of console object

The `console.dir()` is used to display an interactive list of the properties of the specified JavaScript object as JSON.

```
const user = { name: "John", id: 1, city: "Delhi" };
console.dir(user);
```

The user object displayed in JSON representation

Back to Top

371. Is it possible to debug HTML elements in console

Yes, it is possible to get and debug HTML elements in the console just like inspecting elements.

```
const element = document.getElementsByTagName("body")[0];
console.log(element);
```

It prints the HTML element in the console,

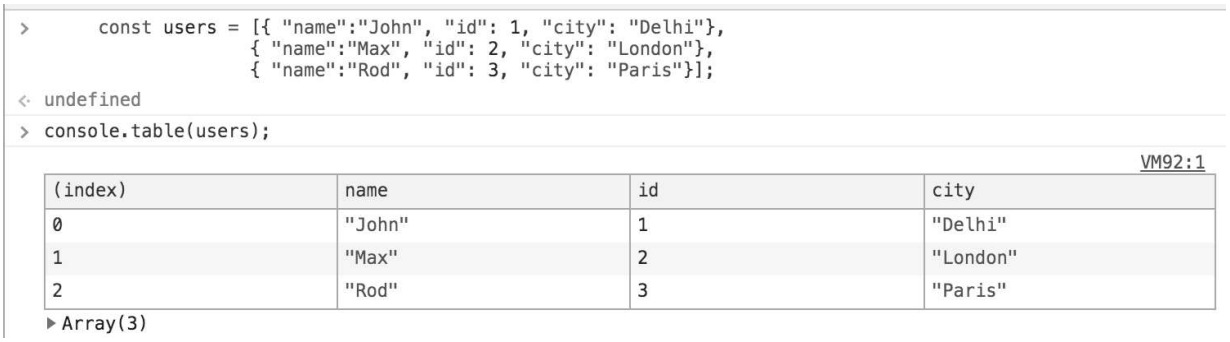
Back to Top

372. How do you display data in a tabular format using console object

The `console.table()` is used to display data in the console in a tabular format to visualize complex arrays or objects.

```
const users = [
  { name: "John", id: 1, city: "Delhi" },
  { name: "Max", id: 2, city: "London" },
  { name: "Rod", id: 3, city: "Paris" },
];
console.table(users);
```

The data visualized in a table format,



```
> const users = [{ "name":"John", "id": 1, "city": "Delhi"},
                 { "name":"Max", "id": 2, "city": "London"},
                 { "name":"Rod", "id": 3, "city": "Paris"}];
< undefined
> console.table(users);
```

VM92:1

(index)	name	id	city
0	"John"	1	"Delhi"
1	"Max"	2	"London"
2	"Rod"	3	"Paris"

▶ Array(3)

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

⬆ Back to Top

373. How do you verify that an argument is a Number or not

The combination of IsNaN and isFinite methods are used to confirm whether an argument is a number or not.

```
function isNumber(n) {
  return !isNaN(parseFloat(n)) && isFinite(n);
}
```

⬆ Back to Top

374. How do you create copy to clipboard button

You need to select the content(using .select() method) of the input element and execute the copy command with execCommand (i.e, execCommand('copy')). You can also execute other system commands like cut and paste.

```
document.querySelector("#copy-button").onclick = function () {
  // Select the content
```

```
  document.querySelector("#copy-input").select();
  // Copy to the clipboard
  document.execCommand("copy");
};
```

Back to Top

375. What is the shortcut to get timestamp

You can use `new Date().getTime()` to get the current timestamp. There is an alternative shortcut to get the value.

```
console.log(+new Date());
console.log(Date.now());
```

Back to Top

376. How do you flattening multi dimensional arrays

Flattening bi-dimensional arrays is trivial with Spread operator.

```
const biDimensionalArr = [11, [22, 33], [44, 55], [66, 77], 88, 99];
const flattenArr = [].concat(...biDimensionalArr); // [11, 22, 33, 44, 55, 66, 7
```

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,
```

Back to Top

377. What is the easiest multi condition checking

You can use `indexOf` to compare input with multiple values instead of checking each value as one condition.

```
// Verbose approach
if (
  input === "first" ||
  input === 1 ||
  input === "second" ||
  input === 2
) {
  someFunction();
}
// Shortcut
if (["first", 1, "second", 2].indexOf(input) !== -1) {
  someFunction();
}
```

⬆ Back to Top

378. How do you capture browser back button

The `window.onbeforeunload` method is used to capture browser back button events. This is helpful to warn users about losing the current data.

```
window.onbeforeunload = function () {
  alert("You work will be lost");
};
```

⬆ Back to Top

379. How do you disable right click in the web page

The right click on the page can be disabled by returning false from the `oncontextmenu` attribute on the body element.

```
<body oncontextmenu="return false;"></body>
```

⬆ Back to Top

380. What are wrapper objects

Primitive Values like string,number and boolean don't have properties and methods but they are temporarily converted or coerced to an object(Wrapper object) when you try to perform actions on them. For example, if you apply toUpperCase() method on a primitive string value, it does not throw an error but returns uppercase of the string.

```
let name = "john";

console.log(name.toUpperCase()); // Behind the scenes treated as console.log(new
```

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

⬆ Back to Top

381. What is AJAX

AJAX stands for Asynchronous JavaScript and XML and it is a group of related technologies(HTML, CSS, JavaScript, XMLHttpRequest API etc) used to display data asynchronously. i.e. We can send data to the server and get data from the server without reloading the web page.

⬆ Back to Top

382. What are the different ways to deal with Asynchronous Code

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

i. Callbacks
ii. Promises
iii. Async/await
iv. Third-party libraries such as async.js,bluebird etc

⬆ Back to Top

383. How to cancel a fetch request

Until a few days back, One shortcoming of native promises is no direct way to cancel a fetch request. But the new `AbortController` from js specification allows you to use a signal to abort one or multiple fetch calls. The basic flow of cancelling a fetch request would be as below,

i. Create an `AbortController` instance

ii. Get the signal property of an instance and pass the signal as a fetch option for signal

iii. Call the AbortController's abort property to cancel all fetches that use that signal For example, let's pass the same signal to multiple fetch calls will cancel all requests with that signal,

```
const controller = new AbortController();
const { signal } = controller;

fetch("http://localhost:8000", { signal })
  .then((response) => {
    console.log(`Request 1 is complete!`);
  })
  .catch((e) => {
    if (e.name === "AbortError") {
      // We know it's been canceled!
    }
  });

fetch("http://localhost:8000", { signal })
  .then((response) => {
    console.log(`Request 2 is complete!`);
  })
  .catch((e) => {
    if (e.name === "AbortError") {
      // We know it's been canceled!
    }
  });

// Wait 2 seconds to abort both requests
setTimeout(() => controller.abort(), 2000);
```

⬆ Back to Top

384. What is web speech API

Web speech API is used to enable modern browsers recognize and synthesize speech(i.e, voice data into web apps). This API has been introduced by W3C Community in the year 2012. It has two main parts,

i. **SpeechRecognition (Asynchronous Speech Recognition or Speech-to-Text):** It provides the ability to recognize voice context from an audio input and respond accordingly. This is accessed by the `SpeechRecognition` interface. The below example shows on how to use this API to get text from speech,

```
window.SpeechRecognition =
  window.webkitSpeechRecognition || window.SpeechRecognition; // webkitSpeechRec
const recognition = new window.SpeechRecognition();
recognition.onresult = (event) => {
  // SpeechRecognitionEvent type
  const speechToText = event.results[0][0].transcript;
  console.log(speechToText);
};
recognition.start();
```

In this API, browser is going to ask you for permission to use your microphone

i. **SpeechSynthesis (Text-to-Speech):** It provides the ability to recognize voice context from an audio input and respond. This is accessed by the `SpeechSynthesis` interface. For example, the below code is used to get voice/speech from text,

```
if ("speechSynthesis" in window) {
  var speech = new SpeechSynthesisUtterance("Hello World!");
  speech.lang = "en-US";
  window.speechSynthesis.speak(speech);
}
```

The above examples can be tested on chrome(33+) browser's developer console.
Note: This API is still a working draft and only available in Chrome and Firefox browsers(ofcourse Chrome only implemented the specification)

⬆ Back to Top

385. What is minimum timeout throttling

Both browser and NodeJS javascript environments throttles with a minimum delay that is greater than 0ms. That means even though setting a delay of 0ms will not happen instantaneously. **Browsers:** They have a minimum delay of 4ms. This throttle occurs when successive calls are triggered due to callback nesting(certain depth) or after a certain number of successive intervals. Note: The older browsers have a minimum delay of 10ms. **Nodejs:** They have a minimum delay of 1ms. This throttle happens when the delay is larger than 2147483647 or less than 1. The best example to explain this timeout throttling behavior is the order of below code snippet.

```
function runMeFirst() {
  console.log("My script is initialized");
```

```
}
setTimeout(runMeFirst, 0);
console.log("Script loaded");
```

and the output would be in

```
Script loaded
My script is initialized
```

If you don't use `setTimeout`, the order of logs will be sequential.

```
function runMeFirst() {
  console.log("My script is initialized");
}
runMeFirst();
console.log("Script loaded");
```

and the output is,

```
My script is initialized
Script loaded
```

⬆ Back to Top

386. How do you implement zero timeout in modern browsers

You can't use setTimeout(fn, 0) to execute the code immediately due to minimum delay of greater than 0ms. But you can use window.postMessage() to achieve this behavior.

⬆ Back to Top

387. What are tasks in event loop

A task is any javascript code/program which is scheduled to be run by the standard mechanisms such as initially starting to run a program, run an event callback, or an interval or timeout being fired. All these tasks are scheduled on a task queue. Below are the list of use cases to add tasks to the task queue,

i. When a new javascript program is executed directly from console or running by the `<script>` element, the task will be added to the task queue.

ii. When an event fires, the event callback added to task queue

iii. When a setTimeout or setInterval is reached, the corresponding callback added to task queue

⬆ Back to Top

388. What is microtask

Microtask is the javascript code which needs to be executed immediately after the currently executing task/microtask is completed. They are kind of blocking in nature. i.e, The main thread will be blocked until the microtask queue is empty. The main sources of microtasks are Promise.resolve, Promise.reject, MutationObservers, IntersectionObservers etc

Note: All of these microtasks are processed in the same turn of the event loop. ⬆ Back to Top

389. What are different event loops

⬆ Back to Top

390. What is the purpose of queueMicrotask

⬆ Back to Top

391. How do you use javascript libraries in typescript file

It is known that not all JavaScript libraries or frameworks have TypeScript declaration files. But if you still want to use libraries or frameworks in our TypeScript files without getting compilation errors, the only solution is `declare` keyword along with a variable declaration. For example, let's imagine you have a library called `customLibrary` that doesn't have a TypeScript declaration and have a namespace called `customLibrary` in the global namespace. You can use this library in typescript code as below,

```
declare var customLibrary;
```

In the runtime, typescript will provide the type to the `customLibrary` variable as `any` type. The another alternative without using declare keyword is below

```
var customLibrary: any;
```

⬆ Back to Top

392. What are the differences between promises and observables

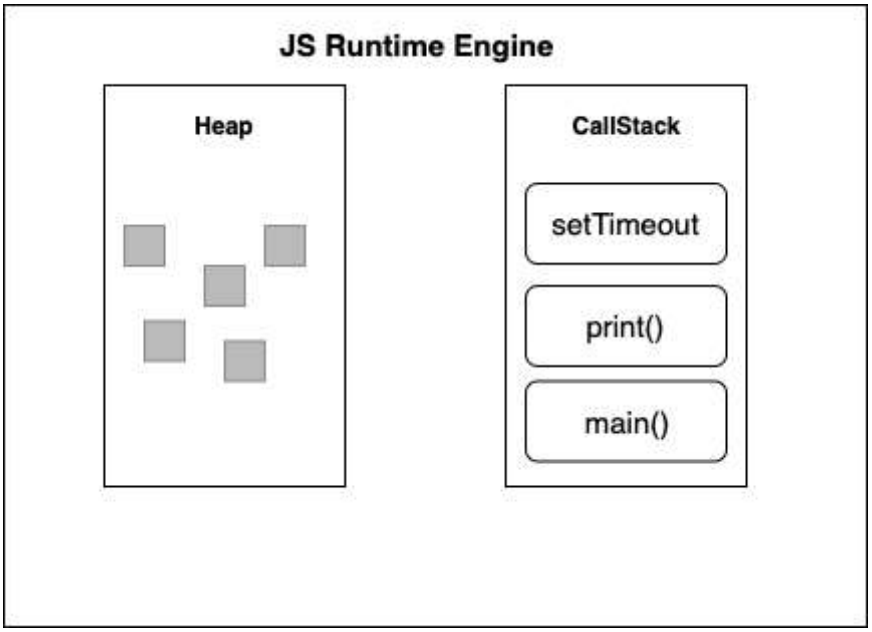
Some of the major difference in a tabular form

Promises	Observables
Emits only a single value at a time	Emits multiple values over a period of time(stream of values ranging from 0 to multiple)
Eager in nature; they are going to be called immediately	Lazy in nature; they require subscription to be invoked
Promise is always asynchronous even though it resolved immediately	Observable can be either synchronous or asynchronous
Doesn't provide any operators	Provides operators such as map, forEach, filter, reduce, retry, and retryWhen etc
Cannot be canceled	Canceled by using unsubscribe() method

⬆ Back to Top

393. What is heap

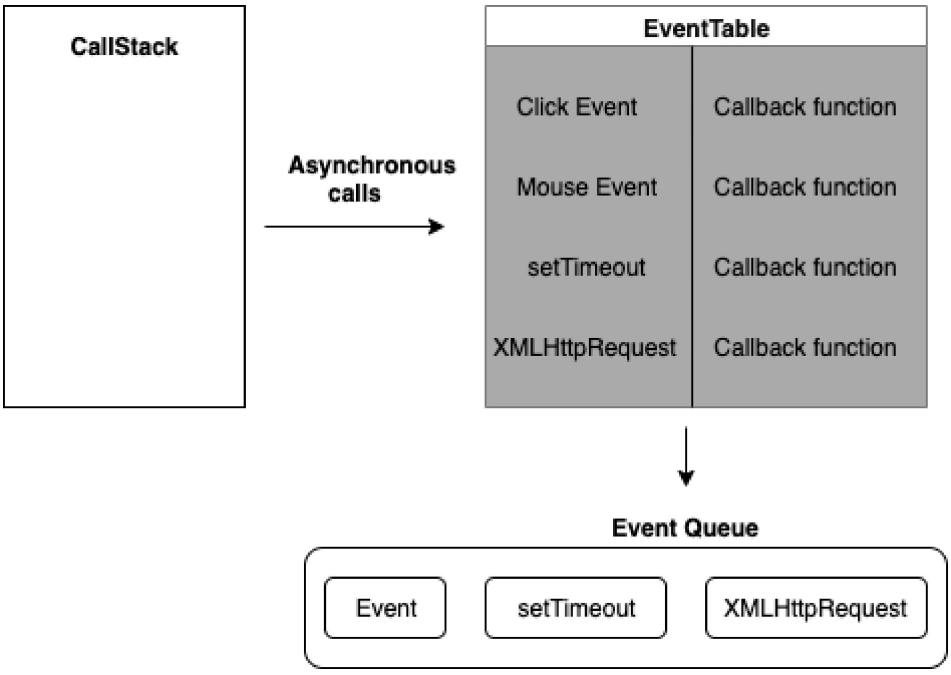
Heap(Or memory heap) is the memory location where objects are stored when we define variables. i.e, This is the place where all the memory allocations and de-allocation take place. Both heap and call-stack are two containers of JS runtime. Whenever runtime comes across variables and function declarations in the code it stores them in the Heap.



Back to Top

394. What is an event table

Event Table is a data structure that stores and keeps track of all the events which will be executed asynchronously like after some time interval or after the resolution of some API requests. i.e Whenever you call a setTimeout function or invoke async operation, it is added to the Event Table. It doesn't not execute functions on it's own. The main purpose of the event table is to keep track of events and send them to the Event Queue as shown in the below diagram.



Back to Top

395. What is a microTask queue

Microtask Queue is the new queue where all the tasks initiated by promise objects get processed before the callback queue. The microtasks queue are processed before the next rendering and painting jobs. But if these microtasks are running for a long time then it leads to visual degradation.

⬆ Back to Top

396. What is the difference between shim and polyfill

A shim is a library that brings a new API to an older environment, using only the means of that environment. It isn't necessarily restricted to a web application. For example, es5-shim.js is used to emulate ES5 features on older browsers (mainly pre IE9). Whereas polyfill is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively. In a simple sentence, A polyfill is a shim for a browser API.

⬆ Back to Top

397. How do you detect primitive or non primitive value type

In JavaScript, primitive types include boolean, string, number, BigInt, null, Symbol and undefined. Whereas non-primitive types include the Objects. But you can easily identify them with the below function,

```
var myPrimitive = 30;
var myNonPrimitive = {};
function isPrimitive(val) {
  return Object(val) !== val;
}

isPrimitive(myPrimitive);
isPrimitive(myNonPrimitive);
```

If the value is a primitive data type, the Object constructor creates a new wrapper object for the value. But If the value is a non-primitive data type (an object), the Object constructor will give the same object.

⬆ Back to Top

398. What is babel

Babel is a JavaScript transpiler to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Some of the main features are listed below,

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

⬆ Back to Top

399. Is Node.js completely single threaded

Node is a single thread, but some of the functions included in the Node.js standard library(e.g, fs module functions) are not single threaded. i.e, Their logic runs outside of the Node.js single thread to improve the speed and performance of a program.

⬆ Back to Top

400. What are the common use cases of observables

Some of the most common use cases of observables are web sockets with push notifications, user input changes, repeating intervals, etc

⬆ Back to Top

401. What is RxJS

RxJS (Reactive Extensions for JavaScript) is a library for implementing reactive programming using observables that makes it easier to compose asynchronous or callback-based code. It also provides utility functions for creating and working with observables.

⬆ Back to Top

402. What is the difference between Function constructor and function declaration

The functions which are created with `Function constructor` do not create closures to their creation contexts but they are always created in the global scope. i.e, the function can access its own local variables and global scope variables only. Whereas function declarations can access outer function variables(closures) too.

Let's see this difference with an example,

Function Constructor:

```
var a = 100;
function createFunction() {
  var a = 200;
  return new Function("return a;");
}
console.log(createFunction()()); // 100
```

Function declaration:

```
var a = 100;
function createFunction() {
  var a = 200;
  return function func() {
    return a;
  };
}
console.log(createFunction()()); // 200
```

⬆ Back to Top

403. What is a Short circuit condition

Short circuit conditions are meant for condensed way of writing simple if statements. Let's demonstrate the scenario using an example. If you would like to login to a portal with an authentication condition, the expression would be as below,

```
if (authenticate) {
  loginToPorta();
}
```

Since the javascript logical operators evaluated from left to right, the above expression can be simplified using && logical operator

```
authenticate && loginToPorta();
```

⬆ Back to Top

404. What is the easiest way to resize an array

The length property of an array is useful to resize or empty an array quickly. Let's apply length property on number array to resize the number of elements from 5 to 2,

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5

array.length = 2;
console.log(array.length); // 2
console.log(array); // [1,2]
```

and the array can be emptied too

```
var array = [1, 2, 3, 4, 5];
array.length = 0;
console.log(array.length); // 0
console.log(array); // []
```

⬆ Back to Top

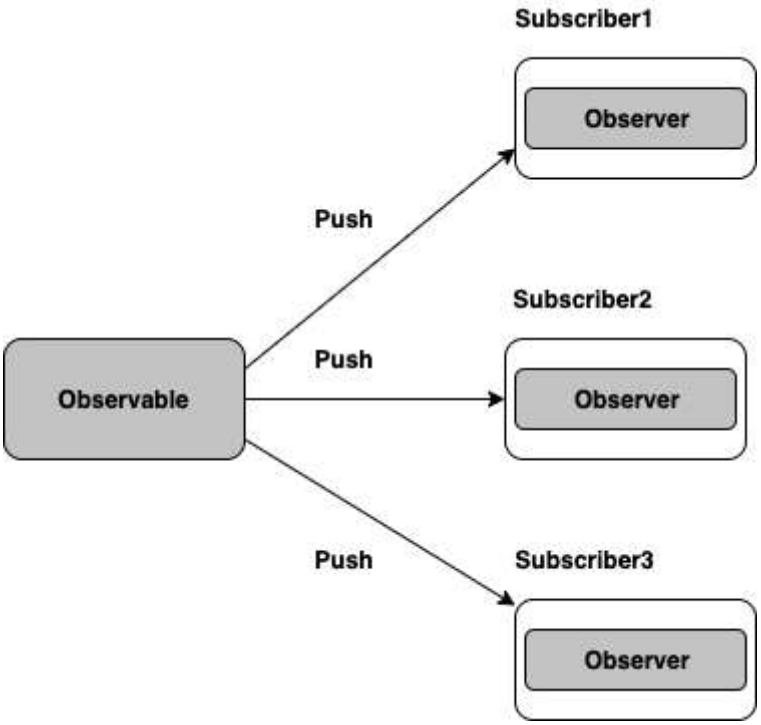
405. What is an observable

An Observable is basically a function that can return a stream of values either synchronously or asynchronously to an observer over time. The consumer can get the value by calling `subscribe()` method. Let's look at a simple example of an Observable

```
import { Observable } from "rxjs";

const observable = new Observable((observer) => {
  setTimeout(() => {
    observer.next("Message from a Observable!");
  }, 3000);
});

observable.subscribe((value) => console.log(value));
```



Note: Observables are not part of the JavaScript language yet but they are being proposed to be added to the language

⬆ Back to Top

406. What is the difference between function and class declarations

The main difference between function declarations and class declarations is `hoisting`. The function declarations are hoisted but not class declarations.

Classes:

```
const user = new User(); // ReferenceError

class User {}
```

Constructor Function:

```
const user = new User(); // No error

function User() {}
```

⬆ Back to Top

407. What is an async function

An async function is a function declared with the `async` keyword which enables asynchronous, promise-based behavior to be written in a cleaner style by avoiding promise chains. These functions can contain zero or more `await` expressions.

Let's take a below async function example,

```
async function logger() {
  let data = await fetch("http://someapi.com/users"); // pause until fetch retur
  console.log(data);
}
logger();
```

It is basically syntax sugar over ES2015 promises and generators.

⬆ Back to Top

408. How do you prevent promises swallowing errors

While using asynchronous code, JavaScript's ES6 promises can make your life a lot easier without having callback pyramids and error handling on every second line. But Promises have some pitfalls and the biggest one is swallowing errors by default.

Let's say you expect to print an error to the console for all the below cases,

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

Promise.reject("error value").catch(function () {
  throw new Error("error");
});

new Promise(function (resolve, reject) {
  throw new Error("error");
});
```

But there are many modern JavaScript environments that won't print any errors. You can fix this problem in different ways,

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

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

But it is quite difficult to type for each promise chain and verbose too.

ii. **Add done method:** You can replace first solution's then and catch blocks with done method

```
Promise.resolve("promised value").done(function () {
  throw new Error("error");
});
```

Let's say you want to fetch data using HTTP and later perform processing on the resulting data asynchronously. You can write `done` block as below,

```
getDataFromHttp()
  .then(function (result) {
    return processDataAsync(result);
  })
  .done(function (processed) {
    displayData(processed);
  });
```

In future, if the processing library API changed to synchronous then you can remove `done` block as below,

```
getDataFromHttp().then(function (result) {
  return displayData(processDataAsync(result));
});
```

and then you forgot to add `done` block to `then` block leads to silent errors.

iii. **Extend ES6 Promises by Bluebird:** Bluebird extends the ES6 Promises API to avoid the issue in the second solution. This library has a "default" onRejection handler which will print all errors from rejected Promises to stderr. After installation, you can process unhandled rejections

```
Promise.onPossiblyUnhandledRejection(function (error) {
  throw error;
});
```

and discard a rejection, just handle it with an empty catch

```
Promise.reject("error value").catch(function () {});
```

⬆ Back to Top

409. What is deno

Deno is a simple, modern and secure runtime for JavaScript and TypeScript that uses V8 JavaScript engine and the Rust programming language.

⬆ Back to Top

410. How do you make an object iterable in javascript

By default, plain objects are not iterable. But you can make the object iterable by defining a `Symbol.iterator` property on it.

Let's demonstrate this with an example,

```
const collection = {
  one: 1,
  two: 2,
  three: 3,
  [Symbol.iterator]() {
    const values = Object.keys(this);
    let i = 0;
    return {
      next: () => {
        return {
          value: this[values[i++]],
          done: i > values.length,
        };
      },
    };
  },
};

const iterator = collection[Symbol.iterator]();
```

```
console.log(iterator.next()); // → {value: 1, done: false}
console.log(iterator.next()); // → {value: 2, done: false}
console.log(iterator.next()); // → {value: 3, done: false}
console.log(iterator.next()); // → {value: undefined, done: true}
```

The above process can be simplified using a generator function,

```
const collection = {
  one: 1,
  two: 2,
  three: 3,
  [Symbol.iterator]: function* () {
    for (let key in this) {
      yield this[key];
    }
  },
};
const iterator = collection[Symbol.iterator]();
console.log(iterator.next()); // {value: 1, done: false}
console.log(iterator.next()); // {value: 2, done: false}
console.log(iterator.next()); // {value: 3, done: false}
console.log(iterator.next()); // {value: undefined, done: true}
```

⬆ Back to Top

411. What is a Proper Tail Call

First, we should know about tail call before talking about "Proper Tail Call". A tail call is a subroutine or function call performed as the final action of a calling function. Whereas **Proper tail call(PTC)** is a technique where the program or code will not create additional stack frames for a recursion when the function call is a tail call.

For example, the below classic or head recursion of factorial function relies on stack for each step. Each step need to be processed upto `n * factorial(n - 1)`

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

But if you use Tail recursion functions, they keep passing all the necessary data it needs down the recursion without relying on the stack.

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

The above pattern returns the same output as the first one. But the accumulator keeps track of total as an argument without using stack memory on recursive calls.

⬆ Back to Top

412. How do you check an object is a promise or not

If you don't know if a value is a promise or not, wrapping the value as `Promise.resolve(value)` which returns a promise

```
function isPromise(object) {
  if (Promise && Promise.resolve) {
    return Promise.resolve(object) == object;
  } else {
    throw "Promise not supported in your environment";
  }
}

var i = 1;
var promise = new Promise(function (resolve, reject) {
  resolve();
});

console.log(isPromise(i)); // false
console.log(isPromise(promise)); // true
```

Another way is to check for `.then()` handler type

```
function isPromise(value) {
  return Boolean(value && typeof value.then === "function");
}
var i = 1;
var promise = new Promise(function (resolve, reject) {
```

```
  resolve();
});

console.log(isPromise(i)); // false
console.log(isPromise(promise)); // true
```

⬆ Back to Top

413. How to detect if a function is called as constructor

You can use `new.target` pseudo-property to detect whether a function was called as a constructor(using the new operator) or as a regular function call.

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

ii. For function calls, new.target is undefined.

```
function Myfunc() {
   if (new.target) {
      console.log('called with new');
   } else {
      console.log('not called with new');
   }
}

new Myfunc(); // called with new
Myfunc(); // not called with new
Myfunc.call({}); not called with new
```

⬆ Back to Top

414. What are the differences between arguments object and rest parameter

There are three main differences between arguments object and rest parameters

i. The arguments object is an array-like but not an array. Whereas the rest parameters are array instances.

ii. The arguments object does not support methods such as sort, map, forEach, or pop. Whereas these methods can be used in rest parameters.

iii. The rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function

⬆ Back to Top

415. What are the differences between spread operator and rest parameter

Rest parameter collects all remaining elements into an array. Whereas Spread operator allows iterables(arrays / objects / strings) to be expanded into single arguments/elements. i.e, Rest parameter is opposite to the spread operator.

⬆ Back to Top

416. What are the different kinds of generators

There are five kinds of generators,

i. **Generator function declaration:**

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

ii. **Generator function expressions:**

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

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

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

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

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

v. **Generator as a computed property:**

```
const SomeObj = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
  },
};

console.log(Array.from(SomeObj)); // [ 1, 2, 3 ]
```

⬆ Back to Top

417. What are the built-in iterables

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

i. Arrays and TypedArrays
ii. Strings: Iterate over each character or Unicode code-points
iii. Maps: iterate over its key-value pairs
iv. Sets: iterates over their elements
v. arguments: An array-like special variable in functions
vi. DOM collection such as NodeList

⬆ Back to Top

418. What are the differences between for...of and for...in statements

Both for...in and for...of statements iterate over js data structures. The only difference is over what they iterate:

i. for..in iterates over all enumerable property keys of an object

ii. for..of iterates over the values of an iterable object.

Let's explain this difference with an example,

```
let arr = ["a", "b", "c"];

arr.newProp = "newVlue";

// key are the property keys
for (let key in arr) {
  console.log(key);
}

// 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.

⬆ Back to Top

419. How do you define instance and non-instance properties

The Instance properties must be defined inside of class methods. For example, name and age properties defined inside constructor as below,

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

But Static(class) and prototype data properties must be defined outside of the ClassBody declaration. Let's assign the age value for Person class as below,

```
Person.staticAge = 30;
Person.prototype.prototypeAge = 40;
```

⬆ Back to Top

420. What is the difference between isNaN and Number.isNaN?

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

ii. **Number.isNaN**: This method does not convert the argument. But it returns true when the type is a Number and value is NaN.

Let's see the difference with an example,

```
isNaN(‘hello’);   // true
Number.isNaN('hello'); // false
```

⬆ Back to Top

421. How to invoke an IIFE without any extra brackets?

Immediately Invoked Function Expressions(IIFE) requires a pair of parenthesis to wrap the function which contains set of statements.

```
(function (dt) {
  console.log(dt.toLocaleTimeString());
})(new Date());
```

Since both IIFE and void operator discard the result of an expression, you can avoid the extra brackets using `void operator` for IIFE as below,

```
void (function (dt) {
  console.log(dt.toLocaleTimeString());
})(new Date());
```

⬆ Back to Top

422. Is that possible to use expressions in switch cases?

You might have seen expressions used in switch condition but it is also possible to use for switch cases by assigning true value for the switch condition. Let's see the weather condition based on 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);
```

⬆ Back to Top

423. What is the easiest way to ignore promise errors?

The easiest and safest way to ignore promise errors is void that error. This approach is ESLint friendly too.

```
await promise.catch((e) => void e);
```

⬆ Back to Top

424. How do style the console output using CSS?

You can add CSS styling to the console output using the CSS format content specifier %c. The console string message can be appended after the specifier and CSS style in another argument. Let's print the red the color text using console.log and CSS specifier as below,

```
console.log("%cThis is a red text", "color:red");
```

It is also possible to add more styles for the content. For example, the font-size can be modified for the above text

```
console.log(
  "%cThis is a red text with bigger font",
  "color:red; font-size:20px"
);
```

⬆ Back to Top

425. What is nullish coalescing operator (??)?

It is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand. This can be contrasted with the logical OR (||) operator, which returns the right-hand side operand if the left operand is any falsy value, not only null or undefined.

```
console.log(null ?? true); // true
console.log(false ?? true); // false
console.log(undefined ?? true); // true
```

⬆ Back to Top

426. How do you group and nest console output?

The `console.group()` can be used to group related log messages to be able to easily read the logs and use console.groupEnd()to close the group. Along with this, you can also nest groups which allows to output message in hierarchical manner.

For example, if you're logging a user's details:

```
console.group("User Details");
console.log("name: Sudheer Jonna");
console.log("job: Software Developer");

// Nested Group
console.group("Address");
console.log("Street: Commonwealth");
console.log("City: Los Angeles");
console.log("State: California");

console.groupEnd();
```

You can also use `console.groupCollapsed()` instead of `console.group()` if you want the groups to be collapsed by default.

⬆ Back to Top

427. What is the difference between dense and sparse arrays?

An array contains items at each index starting from first(0) to last(array.length - 1) is called as Dense array. Whereas if at least one item is missing at any index, the array is called as sparse.

Let's see the below two kind of arrays,

```
const avengers = ["Ironman", "Hulk", "CaptainAmerica"];
console.log(avengers[0]); // 'Ironman'
console.log(avengers[1]); // 'Hulk'
console.log(avengers[2]); // 'CaptainAmerica'
console.log(avengers.length); // 3

const justiceLeague = ["Superman", "Aquaman", , "Batman"];
console.log(justiceLeague[0]); // 'Superman'
console.log(justiceLeague[1]); // 'Aquaman'
console.log(justiceLeague[2]); // undefined
console.log(justiceLeague[3]); // 'Batman'
console.log(justiceLeague.length); // 4
```

⬆ Back to Top

428. What are the different ways to create sparse arrays?

There are 4 different ways to create sparse arrays in JavaScript

i. **Array literal:** Omit a value when using the array literal

```
const justiceLeague = ["Superman", "Aquaman", , "Batman"];
console.log(justiceLeague); // ['Superman', 'Aquaman', empty ,'Batman']
```

ii. **Array() constructor:** Invoking Array(length) or new Array(length)

```
const array = Array(3);
console.log(array); // [empty, empty ,empty]
```

iii. **Delete operator:** Using delete array[index] operator on the array

```
const justiceLeague = ["Superman", "Aquaman", "Batman"];
delete justiceLeague[1];
console.log(justiceLeague); // ['Superman', empty, ,'Batman']
```

iv. **Increase length property:** Increasing length property of an array `js const justiceLeague = ['Superman', 'Aquaman', 'Batman']; justiceLeague.length = 5;`

```
console.log(justiceLeague); // ['Superman', 'Aquaman', 'Batman', empty, empty]
```

⬆ Back to Top

429. What is the difference between setTimeout, setImmediate and process.nextTick?

i. **Set Timeout:** setTimeout() is to schedule execution of a one-time callback after delay milliseconds.

ii. **Set Immediate:** The setImmediate function is used to execute a function right after the current event loop finishes.

iii. **Process NextTick:** If process.nextTick() is called in a given phase, all the callbacks passed to process.nextTick() will be resolved before the event loop continues. This will block the event loop and create I/O Starvation if process.nextTick() is called recursively.

⬆ Back to Top

430. How do you reverse an array without modifying original array?

The `reverse()` method reverses the order of the elements in an array but it mutates the original array. Let's take a simple example to 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.

i. **Using slice and reverse methods:** In this case, just invoke the `slice()` method on the array to create a shallow copy followed by `reverse()` method call on the copy.

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.slice().reverse(); //Slice an array gives a

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

ii. **Using spread and reverse methods:** In this case, let's use the spread syntax (...) to create a copy of the array followed by `reverse()` method call on the copy.

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = [...originalArray].reverse();

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

iii. **Using reduce and spread methods:** Here execute a reducer function on an array elements and append the accumulated array on right side using spread syntax

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reduce((accumulator, value) => {
  return [value, ...accumulator];
}, []);

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

iv. **Using reduceRight and spread methods:** Here execute a right reducer function(i.e. opposite direction of reduce method) on an array elements and append the accumulated array on left side using spread syntax

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reduceRight((accumulator, value) => {
  return [...accumulator, value];
}, []);

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

v. **Using reduceRight and push methods:** Here execute a right reducer function(i.e. opposite direction of reduce method) on an array elements and push the iterated value to the accumulator

```
const originalArray = [1, 2, 3, 4, 5];
const newArray = originalArray.reduceRight((accumulator, value) => {
  accumulator.push(value);
  return accumulator;
}, []);
```

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

⬆ Back to Top

431. How do you create custom HTML element?

The creation of custom HTML elements involves two main steps,

i. **Define your custom HTML element:** First you need to define some custom class by extending HTMLElement class. After that define your component properties (styles,text etc) using `connectedCallback` method. **Note:** The browser exposes a function called `customElements.define` inorder to reuse the element.

```
class CustomElement extends HTMLElement {
  connectedCallback() {
    this.innerHTML = "This is a custom element";
  }
}
customElements.define("custom-element", CustomElement);
```

ii. **Use custome element just like other HTML element:** Declare your custom element as a HTML tag.

```
<body>
     <custom-element>
</body>
```

⬆ Back to Top

432. What is global execution context?

The global execution context is the default or first execution context that is created by the JavaScript engine before any code is executed(i.e, when the file first loads in the browser). All the global code that is not inside a function or object will be executed inside this global execution context. Since JS engine is single threaded there will be only one global environment and there will be only one global execution context.

For example, the below code other than code inside any function or object is executed inside the global execution context.

```
var x = 10;
```

```
function A() {
  console.log("Start function A");

  function B() {
    console.log("In function B");
  }

  B();
}

A();

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

Back to Top

433. What is function execution context?

Whenever a function is invoked, the JavaScript engine creates a different type of Execution Context known as a Function Execution Context (FEC) within the Global Execution Context (GEC) to evaluate and execute the code within that function.

Back to Top

434. What is debouncing?

Debouncing is a programming pattern that allows delaying execution of some piece of code until a specified time to avoid unnecessary *CPU cycles, API calls and improve performance*. The debounce function make sure that your code is only triggered once per user input. The common usecases are Search box suggestions, text-field auto-saves, and eliminating double-button clicks.

Let's say you want to show suggestions for a search query, but only after a visitor has finished typing it. So here you write a debounce function where the user keeps writing the characters with in 500ms then previous timer cleared out using `clearTimeout` and reschedule API call/DB query for a new time—300 ms in the future.

```
function debounce(func, timeout = 500) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => {
      func.apply(this, args);
    }, timeout);
  };
```

```
}
function fetchResults() {
  console.log("Fetching input suggestions");
}
const processChange = debounce(() => fetchResults());
```

The *debounce()* function can be used on input, button and window events

Input:

```
<input type="text" onkeyup="processChange()" />
```

Button:

```
<button onclick="processChange()">Click me</button>
```

Windows event:

```
window.addEventListener("scroll", processChange);
```

↑ Back to Top

435. What is throttling?

Throttling is a technique used to limit the execution of an event handler function, even when this event triggers continuously due to user actions. The common use cases are browser resizing, window scrolling etc.

The below example creates a throttle function to reduce the number of events for each pixel change and trigger scroll event for each 100ms except for the first event.

```
const throttle = (func, limit) => {
  let inThrottle;
  return (...args) => {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => (inThrottle = false), limit);
    }
  };
};
window.addEventListener("scroll", () => {
```

```
    throttle(handleScrollAnimation, 100);
  });
```

⬆ Back to Top

436. What is optional chaining?

According to MDN official docs, the optional chaining operator (?.) permits reading the value of a property located deep within a chain of connected objects without having to expressly validate that each reference in the chain is valid.

The ?. operator is like the . chaining operator, except that instead of causing an error if a reference is nullish (null or undefined), the expression short-circuits with a return value of undefined. When used with function calls, it returns undefined if the given function does not exist.

```
const adventurer = {
  name: 'Alice',
  cat: {
    name: 'Dinah'
  }
};

const dogName = adventurer.dog?.name;
console.log(dogName);
// expected output: undefined

console.log(adventurer.someNonExistentMethod?.());
// expected output: undefined
```

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;
}
```