sudheerj / **javascript-interview-questions**

List of 1000 JavaScript Interview Questions

#javascript #javascript-interview-questions #javascript-applications #vanilla-javascript #core-javascript

13 commits | **1** branch | **0** releases | **1** contributor

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

sudheerj Add hoisting interview questions and answers — Latest commit `4c3f307` 18 days ago

images	let keyword questions and answers	a month ago
.gitignore	Add README	10 months ago
README.md	Add hoisting interview questions and answers	18 days ago

README.md

JavaScript Interview Questions & Answers

> Click ⭐if you like the project. Pull Request are highly appreciated.

Table of Contents

No.	Questions
1	What are the possible ways to create objects in JavaScript?
2	What is prototype chain?
3	What is the difference between Call, Apply and Bind?
4	What is JSON and its common operations
5	What is the purpose of array slice method?
6	What is the purpose of array splice method?
7	What is the difference between slice and splice?
8	How do you compare Object and Map?
9	What is the difference between == and === operators?
10	What are lambda or arrow functions?
11	What is a first class function?
12	What is a first order function?
13	What is a higher order function?
14	What is a unary function?
15	What is currying function?
16	What is a pure function?
17	What is the purpose of let keyword?

No.	Questions
18	What is the difference between let and var?
19	What is the reason to choose the name let as keyword?
20	How do you redeclare variables in switch block without an error?
21	What is Temporal Dead Zone?
22	What is IIFE(Immediately Invoked Function Expression)?
23	What is the benefit of using modules?
24	What is memoization?
25	What is Hoisting?

1. What are the possible ways to create objects in JavaScript?

There are many ways to create objects in javascript as below,

1. **Object constructor:**

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

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

2. **Object's create method:**

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

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

3. **Object literal syntax:** The object literal syntax is equivalent to create method when it passes null as parameter

```
var object = {};
```

4. **Function constructor:** Create any function and apply the new operator to create object instances,

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

5. **Function constructor with prototype:** This is similar to function constructor but it uses prototype for their properties and methods,

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

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

```
function func {};

new func(x, y, z);

**(OR)**

// Create a new instance using function prototype.
var newInstance = Object.create(func.prototype)

// Call the function
var result = func.call(newInstance, x, y, z),

// If the result is a non-null object then use it otherwise just use the new instance.
console.log(result && typeof result === 'object' ? result : newInstance);
```

6. **ES6 Class syntax:** ES6 introduces class feature to create the objects

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

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

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

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

2. What is prototype chain?

Prototype chaining is used to build new types of objects based on existing ones. It is similar to inheritance in a class based language. The prototype on object instance is available through Object.getPrototypeOf(object) or **proto** property whereas prototype on constructors function is available through object.prototype.

3. What is the difference between Call, Apply and Bind?

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

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

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

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

Apply: Invokes the function and allows you to pass in arguments as an array

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

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

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

bind: returns a new function, allowing you to pass in an array and any number of arguments

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

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

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

Call and apply are pretty interchangeable. Both execute the current function immediately. You need to decide whether it's easier to send in an array or a comma separated list of arguments. You can remember by treating Call is for comma (separated list) and Apply is for Array. Whereas Bind creates a new function that will have this set to the first parameter passed to bind().

4. What is JSON and its common operations?

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

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

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

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

5. What is the purpose of array slice method?

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

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

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

6. What is the purpose of array splice method?

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

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

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

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

7. What is the difference between slice and splice?

Some of the major difference in a tabular form

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

8. How do you compare Object and Map?

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

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

7. What is the difference between == and === operators?

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

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

Some of the example which covers the above cases

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

10. What are lambda or arrow functions?

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

11. What is a first class function?

In Javascript, functions are first class objects. First-class functions means when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable. For example, in the below example, handler functions assigned to a listener

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

12. What is a first order function?

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

```
const firstOrder = () => console.log ('Iam a first order functionn!');
```

13. What is a higher order function?

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

```
const increment = x => x+1;
const higherOrder = higherOrder(increment);
console.log(higherOrder(1));
```

14. What is a unary function?

Unary function (i.e. monadic) is a function that accepts exactly one argument. Let us take an example of unary function. It stands for single argument accepted by a function.

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

15. What is currying function?

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

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

Curried functions are great to improve code re-usability and functional composition.

16. What is a pure function?

A **Pure function** is a function where the return value is only determined by its arguments without any side effects. i.e, If you call a function with the same arguments 'n' number of times and 'n' number of places in the application then it will always return the same value. Let's take an example to see the difference between pure and impure functions,

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

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

As per above code snippets, Push function is impure itself by altering the array and returning an push number index which is independent of parameter value. Whereas Concat on the other hand takes the array and concatenates it with the other array producing a whole new array without side effects. Also, the return value is a concatenation of previous array. Remember that Pure functions are important as they simplify unit testing without any side effects and no need for dependency injection. They also avoid tight coupling and makes harder to break your application by not having any side effects. These principles are coming together with **Immutability** concept of ES6 by giving preference to **const** over **let** usage.

17. What is the purpose of let keyword?

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

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

18. What is the difference between let and var?

You can list out the differences in a tabular format

var	let
It is been available from the beginning of JavaScript	Introduced as part of ES6
It has function scope	It has block scope
Variables will be hoisted	Won't get hoisted

Let's take an example to see the difference,

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

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

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

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

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

```
let counter = 1;
switch(x) {
  case 0:
    let name;
    break;

  case 1:
    let name; // SyntaxError for redeclaration.
    break;
}
```

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

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

21. What is Temporal Dead Zone?

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

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

22. What is IIFE(Immediately Invoked Function Expression)?

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

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

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

```
(function ()
        {
          var message = "IIFE";
          console.log(message);
        }
 )
();
console.log(message); //Error: message is not defined
```

23. What is the benefit of using modules?

There are a lot of benefits to using modules in favour of a sprawling. Some of the benefits are,

i. Maintainablity
ii. Reusability
iii. Namespacing

24. What is memoization?

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

```
const memoizAddition = () => {
  let cache = {};
 return (value) => {
  if (value in cache) {
   console.log('Fetching from cache');
   return cache.value;
  }
  else {
   console.log('Calculating result');
   let result = value + 20;
   cache[value] = result;
   return result;
  }
 }
}
// returned function from memoizAddition
const addition = memoizAddition();
console.log(addition(20)); //output: 40 calculated
console.log(addition(20)); //output: 40 cached
```

25. What is Hoisting?

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

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

The above code looks like as below to the interpreter,

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