JS Interview Questions with Answers

Questions:

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

   There are many ways to create objects in javascript as below

   i. **Object constructor:**

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

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

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

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

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

   iii. **Object literal syntax:**

   The object literal syntax is equivalent to create method when it passes null as parameter

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

   iv. **Function constructor:**

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

   ```
   function Person(name){
      var object = {};
   ```

accidentally create multiple instances.

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

2. What is a prototype chain

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

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

 Screenshot

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

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

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

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

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

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

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

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

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

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

v. **Function constructor with prototype:**

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

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

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

```
function func {};

new func(x, y, z);
```

(OR)

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

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

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

vi. **ES6 Class syntax:**

ES6 introduces class feature to create the objects

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

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

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

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

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

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

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

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

4. What is JSON and its common operations

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

Parsing: Converting a string to a native object

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

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

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

5. What is the purpose of the array slice method

Some of the examples of this method are,

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

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

6. What is the purpose of the array splice method

The **splice()** method is used either adds/removes items to/from an array, and then returns the removed item. The first argument specifies the array position for insertion or deletion whereas the 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
let arrayIntegers2 = arrayIntegersOriginal2.splice(3); // returns [4, 5]; original array:
let arrayIntegers3 = arrayIntegersOriginal3.splice(3, 1, "a", "b", "c"); //returns [4]; or
```

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

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

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

9. What is the difference between == and === operators

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

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

Some of the example which covers the above cases,

```
0 == false   // true
```

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

10. What are lambda or arrow functions

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

11. What is a first class function

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

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

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

12. What is a first order function

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

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

13. What is a higher order function

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

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

14. What is a unary function

Let us take an example of unary function,

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

15. What is the currying function

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

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

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

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

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

16. What is a pure function

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

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

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

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

As per 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 the previous array.

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

17. What is the purpose of the let keyword

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

Let's take an example to demonstrate the usage,

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

18. What is the difference between let and var

You can list out the differences in a tabular format

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

Let's take an example to see the difference,

```
function userDetails(username) {
   if(username) {
     console.log(salary); // undefined due to hoisting
```

```
    console.log(salary); //10000 (accessible to due function scope)
    console.log(age); //error: age is not defined(due to block scope)
}
userDetails('John');
```

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

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

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

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

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

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

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

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

21. What is the Temporal Dead Zone

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. Maintainability

24. What is memoization

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

```
const memoizAddition = () => {
  let cache = {};
 return (value) => {
  if (value in cache) {
   console.log('Fetching from cache');
   return cache[value]; // Here, cache.value cannot be used as property name starts with 1
  }
  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';
```

inheritance. For example, the prototype based inheritance written in function expression as below,

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

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

Whereas ES6 classes can be defined as an alternative

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

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

27. What are closures

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

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

Let's take an example of closure concept,

```
function Welcome(name){
  var greetingInfo = function(message){
   console.log(message+' '+name);
  }
return greetingInfo;
}
var myFunction = Welcome('John');
myFunction('Welcome '); //Output: Welcome John
myFunction('Hello Mr.'); //output: Hello Mr.John
```

outer function scope(i.e, Welcome) even after the outer function has returned.

28. What are modules

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

29. Why do you need modules

Below are the list of benefits using modules in javascript ecosystem

i. Maintainability
ii. Reusability
iii. Namespacing

30. What is scope in javascript

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

31. What is a service worker

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

32. How do you manipulate DOM using a service worker

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

33. How do you reuse information across service worker restarts

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

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

35. What is web storage

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

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

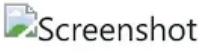
36. What is a post message

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

37. What is a Cookie

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

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



38. Why do you need a Cookie

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

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

39. What are the options in a cookie

There are few below options available for a cookie,

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

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

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

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

40. How do you delete a cookie

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

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

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

41. What are the differences between cookie, local storage and session storage

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

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

42. What is the main difference between localStorage and sessionStorage

LocalStorage is the same as SessionStorage but it persists the data even when the browser is

43. How do you access web storage

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

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

44. What are the methods available on session storage

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

```
// Save data to sessionStorage
sessionStorage.setItem('key', 'value');

// Get saved data from sessionStorage
let data = sessionStorage.getItem('key');

// Remove saved data from sessionStorage
sessionStorage.removeItem('key');

// Remove all saved data from sessionStorage
sessionStorage.clear();
```

45. What is a storage event and its event handler

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

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

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

```
window.onstorage = function(e) {
  console.log('The ' + e.key +
    ' key has been changed from ' + e.oldValue +
```

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

47. How do you check web storage browser support

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

```
if (typeof(Storage) !== "undefined") {
  // Code for localStorage/sessionStorage.
} else {
  // Sorry! No Web Storage support..
}
```

48. How do you check web workers browser support

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

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

49. Give an example of a web worker

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

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

```
let i = 0;

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

timedCount();
```

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

```
  w = new Worker("counter.js");
}
```

and we can receive messages from web worker

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

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

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

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

```
w = undefined;
```

50. What are the restrictions of web workers on DOM

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

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

51. What is a promise

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

The syntax of Promise creation looks like below,

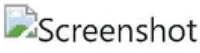
```
const promise = new Promise(function(resolve, reject) {
  // promise description
})
```

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

});

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

The action flow of a promise will be as below,



52. Why do you need a promise

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

53. What are the three states of promise

Promises have three states:

i. **Pending:** This is an initial state of the Promise before an operation begins
ii. **Fulfilled:** This state indicates that the specified operation was completed.
iii. **Rejected:** This state indicates that the operation did not complete. In this case an error value will be thrown.

54. What is a callback function

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

```
function callbackFunction(name) {
  console.log('Hello ' + name);
}

function outerFunction(callback) {
  let name = prompt('Please enter your name.');
  callback(name);
}

outerFunction(callbackFunction);
```