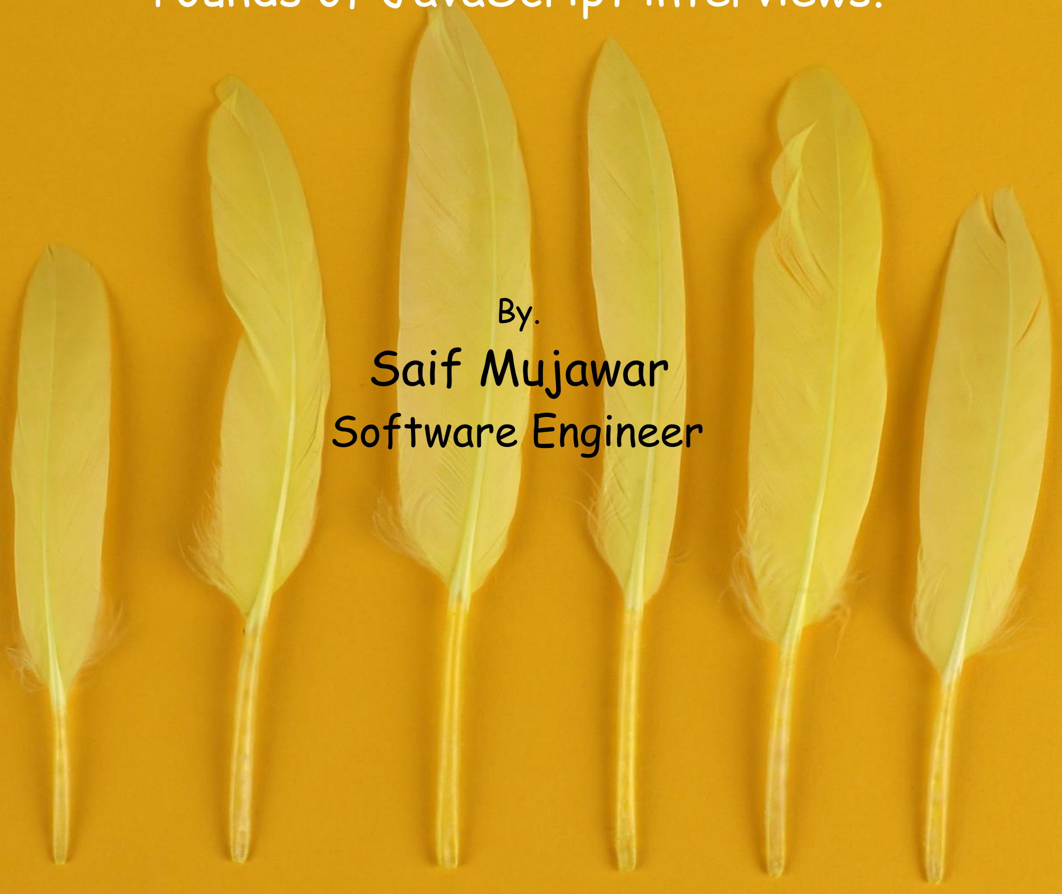
JavaSript Interview QnA

I've written this guide to answer frequently asked JavaScript questions.

Your confidence will increase as you prepare for the initial rounds of JavaScript interviews.



By.
Saif Mujawar
Software Engineer

Q 1 – What is “use strict” and why?

1. The "use strict" directive was new in ECMAScript version 5.
2. It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.
3. The purpose of "use strict" is to indicate that the code should be executed in "strict mode".
4. With strict mode, you cannot, for example, use undeclared variables.
5. All modern browsers support "use strict" except Internet Explorer 9 and lower

Why Strict Mode?

- Strict mode makes it easier to write "secure" JavaScript.
- Strict mode changes previously accepted "bad syntax" into real errors.
- As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.
- In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.
- In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

Q 2 – What is “this” keyword?

- In JavaScript, the this keyword refers to an **object**.
- **Which** object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:

1. In an object method, this refers to the object.
2. Alone, this refers to the global object.
3. In a function, this refers to the global object.
4. In a function, in strict mode, this is undefined.
5. In an event, this refers to the element that received the event. Methods like call(), apply(), and bind() can refer this to any object.

Q 3 – What are different types in JavaScript?

1. Null
2. Undefined
3. Boolean
4. Number
5. BigInt
6. String
7. Symbol

Saif Mujawar

Q 4 - What is == and === in JavaScript?

The main difference between the == and === operator in JavaScript is that

The == operator does the type conversion of the operands before comparison,
whereas the === operator compares the values as well as the data types of the operands.

Q 5 - What is NaN and how can we check for it?

So, NaN is JavaScript stands for Not a Number and **typeof(NaN)** is a "**number**"

There are some side effects checking if NaN is equal to NaN, so to avoid that we can make use of Object.is()
Method to check if the two values are same.

Object.is(NaN,NaN)

It is helpfull because it covers even the corner cases like -0 === 0 // true (which should be false in this particular case) and it covers the NaN equality quirk as well.

Q 6 - What are Scopes in JavaScript?

Scope Determines the accessibility(visibility) of variables.

JavaScript has 3 types of scope:

1. Block Scope
2. Function Scope
3. Global Scope

Block Scope:

- Before ES6 (2015), JavaScript had only Global Scope and Function Scope.
- ES6 introduced two important new JavaScript keywords: let and const.
- These two keywords provide Block Scope in JavaScript.
- Variables declared inside a { } block cannot be accessed from outside the block:

Example:

```
{
 let x = 2;
}
// x can NOT be used here
```

Variables declared with the var keyword can NOT have block scope. Variables declared inside a { } block can be accessed from outside the block.

Example:

```
{
```

Saif Mujawar

```
  var x = 2;
}
// x CAN be used here
```

Function Scope:

- JavaScript has function scope: Each function creates a new scope.
- Variables defined inside a function are not accessible (visible) from outside the function.
- Variables declared with var, let and const are quite similar when declared inside a function.

They all have **Function Scope**:

```
function myFunction() {
  var carName = "Volvo";   // Function Scope
}

function myFunction() {
  let carName = "Volvo";   // Function Scope
}

function myFunction() {
  const carName = "Volvo";   // Function Scope
}
```

Global Scope:

- Variables declared **Globally** (outside any function) have **Global Scope**.
- **Global** variables can be accessed from anywhere in a JavaScript program.
- Variables declared with var, let and const are quite similar when declared outside a block.

They all have **Global Scope**:

```
var x = 2;       // Global scope
let x = 2;       // Global scope
const x = 2;       // Global scope
```

Q 7 - What is hoisting and how to avoid it?

Hoisting is **a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution**. Inevitably, this means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local

Saif Mujawar

Some ways to avoid hoisting are:

- Use let or const — As explained above, using let or const instead of var would throw an exception and not let the program run, hence helping catch the issue earlier.
- Use function expressions instead of function declarations

An example of a function declaration:

```
console.log(printRandom());
function printRandom() {
  return "Print Random";
}
```

The above function declaration will be hoisted and will print out **Print Random**. However, a function expression like the below snippet will not be hoisted and will throw an error:

```
console.log(printRandom());
var printRandom = function () {
  return "Print Random";
}
```

Hoisting is a weird concept in JavaScript. It makes code unreadable and unpredictable. Hence, we should refrain from using it whenever possible by using the above methods.

Q 8 - Explain closures in JavaScript with example?

A closure is **the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment)**. In other words, a closure gives you access to an outer function's scope from an inner function.

When to use Closure?

Closure is useful in hiding implementation detail in JavaScript. In other words, it can be useful to create private variables or functions.

The following example shows how to create private functions & variable.

```
let counter = (function() {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
```

Saif Mujawar

```
     changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  };
})();

alert(counter.value()); // 0
counter.increment();
counter.increment();
alert(counter.value()); // 2
counter.decrement();
alert(counter.value()); // 1
```

In the above example, increment(), decrement() and value() becomes public function because they are included in the return object, whereas changeBy() function becomes private function because it is not returned and only used internally by increment() and decrement().

Q 9 – What is prototype chaining in JavaScript?

If an object tries to access the same property that is in the constructor function and the prototype object, the object takes the property from the constructor function. For example,

```
function Person() {
    this.name = 'John'
}

// adding property
Person.prototype.name = 'Peter';
Person.prototype.age = 23

const person1 = new Person();

console.log(person1.name); // John
console.log(person1.age); // 23
```

In the above program, a property name is declared in the constructor function and in the prototype property of the constructor function.

When the program executes, person1.name looks in the constructor function to see if there is a property named name. Since the constructor function has the name property with value 'John', the object takes value from that property.

When the program executes, person1.age looks in the constructor function to see if there is a property named age. Since the constructor function doesn't have age property, the program looks

Saif Mujawar

into the prototype object of the constructor function and the object inherits property from the prototype object (if available).

Note: You can also access the prototype property of a constructor function from an object.

```
function Person () {
   this.name = 'John'
}

// adding a prototype
Person.prototype.age = 24;

// creating object
const person = new Person();

// accessing prototype property
console.log(person.__proto__);   // { age: 24 }
```

In the above example, a person object is used to access the prototype property using **__proto__**. However**, __proto__** has been deprecated and you should avoid using it.

Q 10 – What is Currying and Advance Currying with Example?

Currying simply means **evaluating functions with multiple arguments and decomposing them into a sequence of functions with a single argument**.

In other words, currying is when a function — instead of taking all arguments at one time — takes the first one and returns a new function, which takes the second one and returns a new function, which takes the third one, etc. until all arguments are completed.

Example:

```
 const addCurry =(a) => {
   return (b)=>{
      return (c)=>{
         return a+b+c
      }
   }
}
console.log(addCurry(2)(4)(5)) // 11
```

Advanced currying:

```
 const curry =(fn) =>{
   return curried = (...args) => {
      if (fn.length !== args.length){
         return curried.bind(null, ...args)
```

Saif Mujawar

```
      }
   return fn(...args);
   };
 }
 const totalNum=(x,y,z) => {
   return x+y+z
 }
 const currTotal = curry(totalNum);

 console.log(currTotal(10) (20) (30));
```

In the example above, we created a function that requires a fixed number of parameters.

It receives a function curry as the outer function. This function is a wrapper function. It returns another function named curried, which receives an argument with the spread operator (...args), and it compares the function length fn length.

The function length means that whatever the number of parameters we pass here, it will reflect in the length property of the function.

But the argument will increase every time. If the number of the parameters we need is not equal, it is going to return curried. If we call bind, this creates a new function and we pass the (...args).

Q 11 – What is memoization and How does it work?

Memoization is an optimization technique that makes applications more efficient and hence faster. It does this by storing computation results in cache, and retrieving that same information from the cache the next time it's needed instead of computing it again.

A **cache** is simply a temporary data store that holds data so that future requests for that data can be served faster.

Memoization is a simple but powerful trick that can help speed up our code, especially when dealing with repetitive and heavy computing functions.

The concept of memoization in JavaScript relies on two concepts:
Closures: The combination of a function and the lexical environment within which that function was declared.
Higher Order Functions: Functions that operate on other functions, either by taking them as arguments or by returning them.

Example:

Saif Mujawar

Let's say we need to write a function that returns the nth element in the Fibonacci sequence. Knowing that each element is the sum of the previous two, a recursive solution could be the following:

```
const fib = n => {
  if (n <= 1) return 1
  return fib(n - 1) + fib(n - 2)
}
```

If you're not familiar with recursion, it's simply the concept of a function that calls itself, with some sort of base case to avoid an infinite loop (in our case if (n <= 1)).

See that we're executing fib(0), fib(1), fib(2) and fib(3) multiple times. Well, that's exactly the kind of problem memoization helps to solve.

With memoization, there's no need to recalculate the same values once and again – we just store each computation and return the same value when required again.

Implementing memoization, our function would look like this:

```
const fib = (n, memo) => {
   memo = memo || {}

   if (memo[n]) return memo[n]

   if (n <= 1) return 1
   return memo[n] = fib(n-1, memo) + fib(n-2, memo)
}
```

Q 12 – What is Event Loop?

Before understanding Event Loop lets see the concept of Call Stack and Call-back Queue.

Call stack is a place where all our code is executed, whenever we try to run a piece of code, it goes to call stack and then executed. It works in **LIFO** style. **Last In First Out.**

Call Back Queue:

1. It's a guard who monitors the stack of asynchronous call-back functions who just completed the task of waiting and passed the gate of Web API.

Saif Mujawar

2. Call-back Queue works using **FIFO** (**First In First Out**) method. And now, they waits here to go back to Call Stack.
3. But how will Call Stack know that there's some call-back functions waiting in Call-back Queue?

Here come's the concept of **Event Loop!!**

Event loop is just a guardian who keeps a good **communication** with **Call Stack** and **Call-back Queue**. It checks if the call stack is free, then let's know the call-back queue. Then Call-back queue passes the call-back function to Call stack to be executed. When all the call-back functions are executed, the call stack is out and global execution context is free.

Q 13 – What is Callback?

A callback is a plain JavaScript function passed to some method as an argument or option. It is a function that is to be executed after another function has finished executing, hence the name 'call back'. In JavaScript, functions are objects. Because of this, functions can take functions as arguments, and can be returned by other functions.

Q 14 – What is Callback Hell and how to avoid it?

Callback Hell is essentially **nested callbacks stacked below one another forming a pyramid structure**. Every callback depends/waits for the previous callback, thereby making a pyramid structure that affects the readability and maintainability of the code.

With four easy ways you can manage Callback Hell:
1. Write comments
2. Split functions into smaller functions
3. Using Promises
4. Using Async/await

Q 15 - What are Promises in JavaScript?

Promises are **used to handle asynchronous operations** in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.

The syntax of Promise creation looks like below:

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

Saif Mujawar

The usage of a promise would be as below:

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

Q 16 – What are the state's of promise?

Promises have three states:

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

Q 17 – What is Callbacks in Callbacks?

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

Example:

```
loadScript('/my/script.js', function(script) {
  loadScript('/my/script2.js', function(script) {
    loadScript('/my/script3.js', function(script) {
      // ...continue after all scripts are loaded
    });
  });
});
```

Q 18 - What is Promise Chaining?

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

Saif Mujawar

Example:

```
new Promise(function (resolve, reject) {
 setTimeout(() => resolve(2), 2000);
})
 .then(function (result) {
   console.log(result); // 1
   return result * 2;
 })
 .then(function (result) {
   console.log(result); // 2
   return result * 4;
 })
 .then(function (result) {
   console.log(result); // 8
   return result * 8;
 });
```

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

1. The initial promise resolves in 2 second,
2. After that. then handler is called by logging the result(2) and then return a promise with the value of result * 2.
3. After that the value passed to the next. then handler by logging the result(4) and return a promise with result * 4.
4. Finally the value passed to the last. then handler by logging the result(16) and return a promise with result * 8.

Q 19 – What is async and await in JavaScript?

- The **async** keyword is used to define an asynchronous function, which returns a AsyncFunction object.

- The **await** keyword is used to pause async function execution until a Promise is fulfilled, that is resolved or rejected, and to resume execution of the async function after fulfilment.

Q 20 – Write a function to generate Unique Id using JavaScript?

```
function UIdGenerator(){
 let left,right, userName="saif";

 const getFormat = (e) => {
        left = (new Date().getTime() + Math.random()*16)%16 | 0;
        right = Math.floor(right/16);
        return (e == 'x' ? left : (left&0x8|0x8)).toString(16);
  }
```

Saif Mujawar

```
  const UID = `xxxxxxxx-xxxx-${userName}-yxxx- xxxxxxxxxxxx`.replace(/[xy]/g, getFormat);

  return UID;
}
```

Q 21 – What is Debounce Function in JavaScript?

Debouncing is a technique to restrict the calling of a time-consuming function frequently, by delaying the execution of the function until a specified time to avoid unnecessary CPU cycles, and API calls and improve performance.

Example: Suppose there is an online E-Book store, and I want to search for some book, So When I'll type something inside the search/input box, on every keystroke an API call will be made. If I go to the network tab on Chrome, and type results in the search/input box, I'll see that on every letter we type, a network/API call is being made.

This is not good from a performance point of view because if the user is typing, let's say, 30 or 100 characters in the input box, then 30-100 API calls will be made.

So to fix this we make use of Debounce Technique!!

```
const debounce = (func, delay = 500) => {
    let timer;
    return function (...args) {
      const context = this;
      if (timer) clearTimeout(timer);
      timer = setTimeout(() => {
        timer = null;
        func.apply(context, args);
      }, delay);
    };
  };
const Usage = debounce(()=> functionName());
```

Q 22 – What is Throttling In JavaScript?

Throttling is a technique in which, no matter how many times the user fires the event, the attached function will be executed only once in a given time interval.

Example:

Saif Mujawar

```
function throttle(fn, delay = 1000) {
  let timer = false;
  let waitingArgs;
  const timeoutFunc = () => {
    if (waitingArgs == null) {
      timer = false;
    } else {
      fn(...waitingArgs);
      waitingArgs = null;
      setTimeout(timeoutFunc, delay);
    }
  };

  return (...args) => {
    if (timer) {
      waitingArgs = args;
      return;
    }

    fn(...args);
    timer = true;
    setTimeout(timeoutFunc, delay);
  };
}
```

Here we are storing the previous args in a variable called **waitingArgs** if throttle is called during the delay. Then when our delay ends we check to see if we have any **waitingArgs**. If we do not we just do everything as normal and set **timer** to false so we can wait for the next trigger. If we do have **waitingArgs**, though, that means we called throttle during the delay and we want to trigger our function with those **waitingArgs** and then reset our timer.

Q 23 – What is Event Capturing and Event Bubbling in JavaScript?

Basically, There are three phases in a JavaScript event:

- **Capture Phase:** Event propagates starting from ancestors towards the parent of the target. Propagation starts from Window object.
- **Target Phase:** The event reaches the target element or the element which started the event.
- **Bubble Phase:** This is the reverse of capture. Event is propagated towards ancestors until Window object.

Saif Mujawar

Event capturing:

Event capturing is the scenario in which the events will propagate, starting from the wrapper elements down to the target element that initiated the event cycle.

Event Bubbling:

Event bubbling will start from a child element and propagate up the DOM tree until the topmost ancestor's event is handled.

Q 24 - In How many ways you can select elements in DOM using selectors?

There are 5 ways in which you can select elements in a DOM using selectors.

1. getElementsByTagName()
2. getElementsByClassName()
3. getElementById()
4. querySelector()
5. querySelectorAll()

Q 25 – Explain Call(), apply() and bind() methods.

Call is a function that helps you change the context of the invoking function. In leyman language, it helps you replace the value of this inside a function with whatever value you want.

```
Syntax : func.call(thisObj, args1, args2, ...)
```

Apply is very similar to the call function. The only difference is that in apply you can pass an array as an argument list.

```
Syntax: func.apply(thisObj, argumentsArray);
```

Bind is a function that helps you create another function that you can execute later with the new context of this that is provided.

```
Syntax : func.bind(thisObj, arg1, arg2, ..., argN);
```

Saif Mujawar