Interview Questions

1. What will be the output and why?

```
console.log(a); // First Output
var a = 10;

function foo() {
    console.log(a); // Second Output
    var a = 20;
}

foo();
console.log(a); // Third Output

Output
undefined
undefined
10
```

First Output

- Since a is declared with 'var' it is hoisted to the top, As it is hoisted it is declared to the top of the global scope
- While variable 'a' is logged the variable 'a' exists and returns undefined

Second Output

- Inside the function the variable 'a' is redeclared and it is hoisted to the top of the function scope
- So when variable 'a' is logged it refers to the variable within the function scope and returns undefined

Third Output

- The value assigned to variable 'a' in function scope doesn't affect the variable 'a' assigned with a value in global scope
- The global variable 'a' was assigned 10, so it prints 10

Key Takeaways

Hoisting: Variables declared with 'var' are hoisted to the top of their respective scope, but the initialization happens where the data is scripted

Hoisting Inside Function: Variable declared with 'var' inside a function will hoisted to top within the function scope

2. What will be the output?

```
for (var i=0; i<5; i++) {
    setTimeout(function () {
        console.log(i);
    }, 1000);
}

Output
5 5 5 5 5
```

- The variable 'i' is declared with 'var' and is hoisted to the top and ensures that the variable 'i' exists throughout the entire loop and after it ends
- setTimeout is asynchronous and it schedules to run after specified delay and it doesn't block the execution of the next lines
- Here the loop finishes first before the setTimeout callbacks and 'i' has already been incremented to 5, so all callbacks log 5

How can you modify the code so that it logs 0, 1, 2, 3, 4 instead?

```
for (let i=0; i<5; i++) {
    setTimeout(function () {
        console.log(i);
    }, 1000);
}

Output
0 1 2 3 4
```

Using 'let' creates a new block scope for each iteration, so each setTimeout gets the correct value of 'i' at that iteration

Key Takeaways

- 'var’ is function scoped or global scoped, so it doesn't create a new scope for each loop iteration
- 'let' creates a new scope for each iteration
- setTimeout is asynchronous and doesn't block the rest of the code
- setTimeout functions are scheduled quickly but their execution is delayed

3. What will be the order of the logs and why?

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

setTimeout(() => {
    console.log("Middle");
}, 0);

console.log("End");

Output
Start
End
Middle
```

- console.log("Start"); and console.log("End"); are synchronous, they run immediately in the order they are scripted
- setTimeout doesn’t execute immediately even when it is set to 0 milliseconds and it doesn’t block execution of next lines
- setTimeout is asynchronous and it schedules the callback, therefore it executes after the execution of all the synchronous code

Key Takeaways

- Synchronous code runs first
- setTimeout doesn't run immediately. It waits for the current execution to complete and then runs in the next event loop iteration
- Therefore 'Middle' is logged last even though setTimeout was set 0ms

4. What will be the output and why?

```
let arr = [1, 2, 3];
arr[10] = 5;
console.log(arr.length);
console.log(arr);

Output
11
[ 1, 2, 3, <7 empty items>, 5 ]
```

- When arr[10] = 5 assigned, the element is setted at index 10 and indices 3 through 9 remain empty and values haven't assigned to those indices
- These indices are considered empty and retrieving those elements will return undefined as they doesn't contain actual values
- The length of the array is determined by the highest index plus one and since the array has an element at index 10, the length becomes 11

- Arrays have a special way of representing empty slots
- Instead of showing undefined in empty slots, JavaScript uses <empty> marker representation

Key Takeaways

- The empty slots don’t hold any values but counted toward the array's length
- The length of the array is updated to highest index plus one, even if there are empty slots in between

5. What will be the output of these expressions and why?

```
console.log([] + []);
console.log([] + {});
console.log({} + []);

Output
// An empty string
[object Object]
[object Object]
```

The arrays and objects are converted to string representations implicitly when tried to concatenate

First Output
[] + [] = ""

Second Output
"" + "[object Object]" = "[object Object]"

Third Output
"[object Object]" + "" = "[object Object]"

Key Takeaways

- The '+' operator triggers the toString() method to convert values to strings when it tries to concatenate or perform string addition
- JavaScript coerces arrays into string
- JavaScript coerces objects into '[object Object]'
- Coercion means implicit type conversion

6. What will be the output and why?

```
let a = { football: 'barcelona' };
let b = { football: 'barcelona' };

console.log(a == b);
console.log(a === b);

Output
false
false
```

- In JavaScript objects are reference types, meaning objects are stored at specific memory address and variables holding object stores the reference, not the actual value
- When compared using == or === it compares memory references and obviously returns false as they are not considered equal while comparing

How to compare the contents of objects?

- Check whether both the objects have same length
- Loop through the keys of the objects and then compare their values
- Use JSON Serialization to convert objects to JSON strings and compare those strings using === operator

```
let a = { football: 'barcelona' };
let b = { football: 'barcelona' };

function compareObjects(a, b) {
    if (Object.keys(a).length !== Object.keys(b).length)
        return false;
    for (let key of Object.keys(a))
        if (!b.hasOwnProperty(key) || a[key] !== b[key])
            return false;
    return true;
}
console.log(compareObjects(a, b));
```

```
let a = { football: 'barcelona' };
let b = { football: 'barcelona' };
console.log(JSON.stringify(a) === JSON.stringify(b));
```

7. What will happen when you run this code and why?

```
foo();
var foo = function () {
    console.log("Function Expression");
}

Output
TypeError: foo() is not a function
Compilation Error
```

- JavaScript hoists the declaration to the top of the current scope
- The variable foo is undefined before the function expression is assigned
- As the function was called before the function assignment and it is undefined at that point, the compilation results in TypeError

Key Takeaways

- Hoisting moves only the declaration of variables declared with 'var' to the top and is undefined as it is not initialized at top
- A function should only be called after it has been initialized, as it cannot be accessed before initialization when assigned to a variable

Correct Code

```
var foo = function () {
    console.log("Function Expression");
}
foo()

Output
Function Expression
```

8. What will be the output and why?

```
const person = {
    name: 'Alice',
    age: 25,
    address: {
        city: 'Wonderland'
    }
};
const { name, address: { city }, country = 'Unknown' } = person;
console.log(name, city, country);
```

Output

```
Alice Wonderland Unknown
```

- This code uses destructuring assignment, a feature that allows you to unpack values from arrays or properties from objects and directly assign them into variables
- name: extracts name property from object person which is 'Alice'
- address: { city } extracts the city property from the address object which is 'Wonderland'
- country = 'Unknown' sets default value of 'Unknown' for country variable in case the person object doesn't have country property

Key Takeaways

- Destructuring allows to pull values out of an object directly into variables
- Default values are used if the property doesn’t exist in the object

9. What will be the output and why?

```
const promise = new Promise((resolve, reject) => {
    console.log('Promise Started');
    resolve('Success');
});
promise.then(res => {
    console.log(res);
});
console.log('Promise Created');
```

Output

```
Promise Started
Promise Created
Success
```

- When a promise is created the function passed to the promise is executed immediately, so 'Promise Started' was logged in console immediately
- After the promise is created and resolved, the synchronous code lines are executed, so 'Promise Created' was logged in the console
- The then() method attaches a callback that will run when a promise is resolved
- Since the promise is resolved asynchronously, it waits until current call stack to clear for executing the callback, therefore 'Success' is logged after 'Promise Created'

Key Takeaways

- Synchronous code runs first as they aren't involved in any promises
- The .then() callback executes asynchronously and runs after all synchronous code is done, even if the promise is resolved earlier

10. What will be the output and why?

```
(function () {
    var x = 10;
    (function () {
        console.log(x);
        var x = 20;
        console.log(x);
    })();
})();

Output
undefined
20
```

First Output
Inside the inner function, variable 'x' is hoisted to the top but its not initialized in that respective function scope
Therefore as it not initialized at this point it logs undefined

Second Output
At this point the value of 'x' has been assigned with the value 20, so it logs 20

Key Takeaways
Hoisting causes 'x' to be undefined initially in the inner function as outer function doesn't affect as they have separate scopes