1. What will be the output and why?

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

Answer:
undefined
undefined
10

Explanation

The output is undefined, undefined, 10. This happens due to **hoisting**. In the global scope, a is declared but not yet initialized when first logged, so it outputs undefined. Inside the foo() function, there's a separate local a, which is also hoisted and initially undefined when logged. After foo finishes, the global a, which was set to 10, is logged, producing the final output of 10.

2. What will be the output? How can you modify the code so that it logs 0, 1, 2, 3, 4 instead?

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

Answer:

5
5
5
5
5

Explanation:

The variable i is declared with var, so it is function-scoped and shared across all iterations of the loop. By the time the setTimeout callbacks execute (after 1 second), the loop has already completed, and i has been incremented to 5. As a result, each console.log(i); logs the final value of i, which is 5.

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

```
console.log('Start');
setTimeout(()
=> { console.log('Middle'); }, 0);
```

Answer:

Start
Middle

Explanation:
JavaScript executes code synchronously by default, but setTimeout is asynchronous and moves the callback function to the **event queue**. Even though the delay is 0, the callback (console.log('Middle')) waits until the synchronous code finishes executing. Thus, 'Start' logs first, followed by 'Middle' after the synchronous code completes.

4. What will be the output and why?

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

Answer:

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

Explanation:

When arr[10] = 5; is assigned, JavaScript expands the array to accommodate index 10. This creates "empty" slots (sparse array) from index 3 to 9, increasing the array's

length to 11. Therefore, arr.length outputs 11. The console.log(arr); shows the array with the existing values [1, 2, 3], followed by 7 empty slots, and finally 5 at index 10.

5. What will be the output of these expressions and why?
   console.log([] + []);
   console.log([] + {});
   console.log({} + []);

   Answer:
   ""
   "[object Object]"
   0

   Explanation:
   The first expression results in an empty string because both arrays are converted to strings. The second expression outputs "[object Object]" due to the concatenation of an empty string and the object's string representation. The third expression evaluates to 0 because the object is treated as a block, leaving only the addition of the empty array, which converts to 0.

6. What will be the output and why? How can you compare the content of two objects?
   let a = { foo: 'bar' };
   let b = { foo: 'bar' };
   console.log(a == b);
   console.log(a === b);

   Answer:
   false
   false

   Explanation:
   Both expressions output false because in JavaScript, objects are compared by reference, not by value. Even though a and b have identical content ({ foo: 'bar' }), they are distinct objects in memory with different references. Therefore, both the loose equality operator (==) and the strict equality operator (===) evaluate to false. To compare the content of two objects, you can convert them to JSON strings using

JSON.stringify() and compare the resulting strings, like so: JSON.stringify(a) === JSON.stringify(b), which will return true for identical object.

7. What will happen when you run this code and why?
   foo();
   var foo = function() {
   console.log('Function expression');
   };
   Answer:
   TypeError: foo is not a function

   Explanation:
   When the code is executed, it results in a TypeError because foo is declared as a variable using var, but the assignment of the function occurs after the call to foo(). Due to **hoisting**, the declaration var foo; is processed before any code runs, but the assignment of the function expression does not occur until that line of code is executed. As a result, when foo() is called, foo is undefined, leading to the error since you cannot invoke undefined as a function.

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);

   Answer:
   Alice Wonderland Unknown

   Explanation:
   In this code, destructuring is used to extract properties from the person object. The variable name is assigned the value 'Alice', and the variable city is assigned the value 'Wonderland' from the nested address object. The country variable is also declared during destructuring with a default value of 'Unknown'. Since country is not a property of the person object, it takes the default value. Therefore, when logged, the output shows Alice, Wonderland, and Unknown.

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');
```

Answer:
Promise started
Promise created
Success

Explanation:

When the promise is created, the executor function runs immediately, logging 'Promise started' to the console. It then calls resolve('Success'), which schedules the resolution of the promise. However, the resolution does not occur immediately; it is added to the microtask queue and will execute after the current stack of synchronous code is completed. Next, 'Promise created' is logged. Finally, once the synchronous code has finished executing, the resolved value 'Success' is processed, triggering the .then() method, which logs 'Success' to the console. Thus, the output appears in that order.

10. What will be the output and why?

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

Answer:
undefined
20

Explanation:
When the code runs, the outer function is invoked first, creating a scope where var x = 10; is declared. Inside the nested inner function, var x = 20; is also declared, which hoists the variable declaration x to the top of the inner function's scope. Due to

hoisting, the first console.log(x); tries to log x before it is assigned a value, leading to undefined. After this, the inner function sets x to 20, and the second console.log(x); outputs 20. Thus, the final output is undefined followed by 20.