1) 1 point

What will be the output when the following code is executed? Explain.

const a = 1

const b = "1"

const c = true

console.log(a == b)

console.log(c == b)

console.log(c === b)

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

Ans:

* “true”, variable “a” whose value is 1 (a NUMBER) and variable “b” whose value is 1 but as string and the “==” operator does not perform strict type comparison, meaning it converts both sides to the same type before comparison.
* “true”, same scenario above, true is coerced to number 1 so it becomes 1 == 1 which equals to true.
* “false”, because the “===” operator performs a strict comparison of types that’s why variable “c” which is a Boolean and variable “b” which is a String does not equal.
* “true”, because variable “a” == variable “b”, which again does not perform a strict comparison, which is true so it becomes (true === true (value of variable “c”)) is equal to true.

2) 1 point

What’s the output? Explain.

let number = 0;

console.log(number++);

console.log(++number);

console.log(number);

Ans:

0 because “number++” is a post-increment operator which means the value of variable “number” is used before incrementing it.

2 because the value of variable “number” is incremented by “number++” in the first line which becomes 1, then used another operator “++number” which is a pre-increment operator, which in this case the variable “number” is incremented before use which becomes 2.

2 because there were no operations performed, so the current value of variable “number” is 2.

3) 1 point

What will the code below output to the console and why ?

const object = {

'name': 'person',

'age': '24'

}

object.age = 40

console.log(object.age)

Ans: 40

We declared an object with two properties; “name”; and “age”. As you can see in line 5, the “age” property of the object is being changed from a string value “24” to a number 40. Even though we used “const” the objects in Javascript are still mutable.

4) 2 points

What’s the output? Explain.

class Dog {

constructor() {

console.log("I'm a dog.");

}

}

class Bulldog extends Dog {

constructor() {

console.log("I'm cute.");

super();

}

}

const pet = new Bulldog();

Ans:

“I’m cute”

“I’m a dog”

The Dog class has a constructor that logs “I’m a dog” to the console when a Dog instance is created.

The Bulldog has a constructor that logs “I’m cute” to the console and after that, the function super() is also called which calls the constructor of its parent class which is the Dog class since Bulldog extends the Dog class.

5) 5 points

What will be the output of this code and why?

var x = 21;

var girl = function () {

console.log(x);

var x = 20;

};

girl ();

Ans:

Undefined - (because of hoisting)

The var x declaration and not the initialization inside the girl function is moved to the top of function scope. Because of hoisting, the var x is undefined because console.log(x) is executed before assigning 20 to the var x is executed.

6) 5 points

What will be the output of the following code: (explain your answer)

for (var i = 1; i < 5; i++) {

setTimeout( () => console.log(i), i \* 100000 );

}

setTimeout( () => console.log(‘done’) );

Explain your answer. How could the use of closures help here?

Ans:

“done”

5

5

5

5

Simply because var has function scope and not block scope. Since variable “i” is declared using “var”, it is shared across the whole iteration that is why by the time the setTimeout callbacks has been completed, it will always log 5 since the loop has already been completed and the final value of variable “i” is always 5. The log “done” has no delay so it executes immediately.

Closure or block scope (let) will help to ensure that each iteration will create a new scope or instance of variable “i”.

6) 5 points

What will be the output? Explain your answer.

const myPromise = () => Promise.resolve('I have resolved!');

const myOtherPromise = () => Promise.resolve('I have resolved again!')

function firstFunction() {

myPromise().then(res => console.log(res));

console.log('first');

}

async function second Function() {

console.log('second')

console.log(await myOtherPromise());

console.log('second and a half');

}

firstFunction();

secondFunction();

Ans:

* "first" This is logged immediately from the firstFunction() call.
* "second" This is logged immediately from the secondFunction() call.
* "I have resolved!" This is logged when the promise from myPromise() resolves (as a microtask).
* "I have resolved again!" This is logged when the promise from myOtherPromise() resolves (after the await).
* "second and a half” This is logged after the await completes and execution resumes in secondFunction().

8) 1 point

What’s the output, and why?

class Person {

constructor(name) {

this.name = name;

}

}

const member = new Person('John');

console.log(typeof member);

Ans:

“object”

A class in Javascript is essentially a blueprint of creating an object. Here we declare a class Person with a constructor that has a parameter “name” which later on will be assigned to the instance property this.name  
  
After that, we declare “member” a new instance of class Person and set a parameter ‘John’ to assign to the instance property this.name

And after that we log the type of “member” which becomes an object that is an instance of Person class.  
In javascript, the type of an object will always be an “object”.

9) 3 points

What do the following lines output, and why?

// counter.js

let counter = 10;

export default counter;

// index.js

import myCounter from './counter';

myCounter += 1;

console.log(myCounter);

Ans:

Error

Import variable myCounter is treated as const (read-only) and not writable. You can read the value of myCounter, but you cannot directly modify it in the importing module.

7) 2 points

What is the value of:

typeof undefined == typeof NULL

Ans:

true because NULL is undefined. But if we write NULL in lowercase, it will log false because typeof null is an object.