**Asynchronous Execution & Microtasks**

async function test() {  
 console.log(1);  
 await Promise.resolve();  
 console.log(2);  
}  
console.log(3);  
test();  
console.log(4);

**Answer:** The output is: 3, 1, 4, 2

**Explanation:**The function test logs 1 synchronously. The await pauses execution and schedules the subsequent log (2) as a microtask, which runs after the current synchronous code completes. Thus, 3 and 4 print first.

**2. Promise Chain with Error Handling**

Promise.resolve(1)  
 .then(val => {  
 console.log(val);  
 return val \* 2;  
 })  
 .then(val => {  
 console.log(val);  
 throw new Error('oops');  
 })  
 .catch(err => {  
 console.log(err.message);  
 return 10;  
 })  
 .then(val => console.log(val));

**Answer:**The output is: 1, 2, oops, 10

**Explanation:**After printing 1 and 2, an error is thrown, caught by the .catch block (logging “oops”), which then returns 10 to the next .then.

**3. Prototype Modification Impact**

function A() {}  
A.prototype = { foo: function() { return 1; } };  
  
const a = new A();  
A.prototype.foo = function() { return 2; };  
  
console.log(a.foo());

**Answer:**The output is: 2

**Explanation:**Even though a was created before modifying the prototype, the lookup for foo occurs dynamically. Changing A.prototype.foo affects all instances that reference that method.

**4. Constructor Function with Explicit Return**

function Foo() {  
 this.value = 1;  
 return { value: 2 };  
}  
  
const obj = new Foo();  
console.log(obj.value);

**Answer:**The output is: 2

**Explanation:**When a constructor explicitly returns an object, that object becomes the result of the new expression — overriding the usual this binding.

**5. “this” Binding in Arrow vs. Regular Functions**

const obj = {  
 num: 100,  
 regular: function() { return this.num; },  
 arrow: () => this.num,  
};  
  
console.log(obj.regular());  
console.log(obj.arrow());

**Answer:**The output is: 100 undefined

**Explanation:**The regular function binds this to obj, while the arrow function does not have its own this and thus inherits from the surrounding (global) scope, which doesn’t have num.

**6. Implicit Type Coercion Quirk**

console.log([] == ![]);

**Answer:**The output is: true

**Explanation:**![] evaluates to false because an empty array is truthy, and then the loose equality causes [] (which coerces to an empty string and then to 0) to equal false (which coerces to 0).

**7. Variable Hoisting and Shadowing**

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

**Answer:**The output is: undefined

**Explanation:**Inside the function, var a is hoisted, which means the local variable a is declared but not yet initialized at the time of the console.log, resulting in undefined.

**8. Object Destructuring with Default Values**

const { a: x = 5 } = { a: 10 };  
console.log(x);  
  
const { b: y = 5 } = { a: 10 };  
console.log(y);

**Answer:**The outputs are: 10 5

**Explanation:**For x, property a exists and its value (10) is used. For y, property b is missing, so the default value (5) is assigned.

**9. For-In Loop and Inherited Properties**

const obj = Object.create({ a: 1 });  
obj.b = 2;  
for (let key in obj) {  
 console.log(key);  
}

**Answer:** The output is: b a

**Explanation:**The for…in loop iterates over both own and inherited enumerable properties, so it prints b (own property) followed by a (inherited).

**10. Typeof Operator Precedence**

console.log(typeof typeof 123);

**Answer:**The output is: “string”

**Explanation:**typeof 123 evaluates to “number”, and then typeof “number” evaluates to “string”.

**Bonus Tip**

When preparing for advanced JavaScript interviews, always remember to:

* **Experiment:**Use your browser’s developer console to test code snippets.
* **Understand Underlying Concepts:**Delve into the nuances of scope, hoisting, prototypes, and asynchronous operations.
* **Practice Regularly:**Keep solving challenging problems to sharpen your problem-solving skills.