

# JavaScript's unusual nature

---

## Case study 1 - Type Coercion

```
console.log([] = ![]); // Output: true
```



At first glance, you might expect this comparison to be `false` since an empty array (`[]`) is truthy, and `![]` would be `false`. However, the actual output is `true`.

This seemingly strange behavior is due to the type coercion rules in JavaScript. When you use the `=` operator, JavaScript tries to convert the operands to the same type before making the comparison. In this case:

- The empty array (`[]`) is coerced to an empty string (`''`).
- The `![]` is coerced to a boolean, so it becomes `false`.
- Now, you have `'' = false`.

In the process of comparison, the empty string (`''`) is then coerced to a number, and an empty string as a number is `0`. So, you end up with `0 = false`, which evaluates to `true`.

To avoid such unexpected behavior and ensure strict equality, it's recommended to use the `===` operator, which checks both value and type:

```
console.log([] === ![]); // Output: false
```



With strict equality, the output is as expected: `false`. This case underscores the importance of understanding type coercion in JavaScript and being cautious when using loose equality comparisons.

## References:

 [Freecodecamp](#)

### Case study 2 - Closure, loop & IIFE

```
// Example 1
for (var i = 1; i ≤ 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, 1000);
}
```

At first glance, you might expect this code to log the numbers 1 through 5 to the console with a delay of 1 second between each log. However, the actual output will be:

```
6
6
6
6
6
```

This unexpected behavior is due to the fact that JavaScript has function-level scope, and the `setTimeout` callback function inside the loop creates a closure over the variable `i`. By the time the callback functions are executed, the loop has already completed, and `i` is equal to 6.

To fix this issue, you can use an immediately-invoked function expression (IIFE) to create a new scope for each iteration of the loop:

```
// Example 2
for (var i = 1; i ≤ 5; i++) {
  (function(j) {
    setTimeout(function() {
      console.log(j);
    }, 1000);
  })(i);
}
```

In this example, the IIFE is used to capture the current value of `i` for each iteration of the loop and pass it as an argument to the function inside. As a result, the output will be:

## JavaScript's unusual nature

```
1  
2  
3  
4  
5
```



This case study highlights the importance of understanding closures and how they interact with variables in different scopes. It also demonstrates the need to be mindful of asynchronous operations like `setTimeout` and the behavior of the event loop in JavaScript.

### Case Study 3 - Hoisting

Here's another example that showcases an unusual behavior in JavaScript related to variable hoisting:

```
function example() {  
  console.log(x); // undefined  
  var x = 5;  
  console.log(x); // 5  
}  
  
example();
```



In most programming languages, attempting to access the value of a variable before it's declared and assigned a value would result in an error. However, in JavaScript, due to hoisting, variable and function declarations are moved to the top of their containing scope during the compilation phase. This means that the above code is actually interpreted like this:

```
function example() {  
  var x; // declaration is hoisted  
  console.log(x); // undefined  
  x = 5; // assignment is not hoisted  
  console.log(x); // 5  
}
```



## JavaScript's unusual nature

```
}  
  
example();
```

As a result, even though the `console.log(x)` appears before the `var x = 5;` line, it doesn't throw an error. Instead, it logs `undefined` because the variable `x` is hoisted to the top of the function, but its assignment is not hoisted.

Understanding hoisting and the execution order in JavaScript is crucial to avoiding unexpected behaviours in your code. It's generally a good practice to declare and initialize variables at the beginning of their scope to improve code readability and avoid potential pitfalls related to hoisting.

## Case study 4 - this keyword Implicit binding

What will be the output of the following code snippet and why?

```
function greeting(obj) {  
  obj.logMessage = function() {  
    console.log(`${this.name} is ${this.age} years old!`);  
  }  
};  
  
const tom = {  
  name: 'Tom',  
  age: 7  
};  
  
const jerry = {  
  name: 'jerry',  
  age: 3  
};  
  
greeting(tom);  
greeting(jerry);  
  
tom.logMessage ();  
jerry.logMessage ();
```

Implicit binding covers most of the use-cases for dealing with the `this` keyword.

When we invoke a method of an object, we use the dot(.) notation to access it. In implicit binding, you need to check the object adjacent to the method at the invocation time. This determines what `this` is binding to.

## JavaScript's unusual nature

In this example, we have two objects, `tom` and `jerry`. We have decorated (enhanced) these objects by attaching a method called `logMessage()`.

Notice that when we invoke `tom.logMessage()`, it was invoked on the `tom` object. So `this` is bound to the `tom` object, and it logs the value *tom and 7* (`this.name` is equal to `tom` and `this.age` is 7 here). The same applies when `jerry.logMessage()` is invoked.

## Case Study 5 - Promise Chain

What's the output?

```
// Create a Promise
let promise = new Promise(function (resolve, reject) {
  resolve(10);
});

promise.then(function (value) {
  value++;
  return value;
});
promise.then(function (value) {
  value = value + 10;
  return value;
});
promise.then(function (value) {
  value = value + 20;
  console.log(value);
  return value;
});
```

The Options are:


- 10
- 41
- 30
- None of the above.

Rule: Calling the `.then()` handler method multiple times on a single promise is *NOT* chaining.

A Promise chain starts with a promise, a sequence of handlers methods to pass the value/error down in the chain. But calling the handler methods multiple times on the same promise doesn't create the chain.

Ok, the answer is `30`. It is because we do not have a promise chain here. Each of the `.then()` methods gets called individually. They do not pass down any result to the other

## JavaScript's unusual nature

.then() methods. We have kept the console log inside the last .then() method alone. Hence the only log will be `30` (10 + 20). You interviewers love asking questions like this .

*In the case of a promise chain, the answer will be, `41`. Please try it out.*

## Case Study 6 - Asynchronous JavaScript

Let's consider a scenario involving parallel asynchronous operations and handling the results. Imagine you have an array of tasks, each representing a different asynchronous operation, and you want to perform these tasks concurrently. However, you need to aggregate the results and proceed with additional processing only when all tasks are completed.

```
// Example 6
function performAsyncTask(taskId) {
  return new Promise((resolve, reject) => {
    // Simulating an asynchronous task
    const randomDelay = Math.random() * 2000;
    setTimeout(() => {
      console.log(`Task ${taskId} completed after ${randomDelay.toFixed(2)} ms`);
      resolve(`Result of Task ${taskId}`);
    }, randomDelay);
  });
}

async function performParallelTasks(taskIds) {
  const taskPromises = taskIds.map(performAsyncTask);

  try {
    const results = await Promise.all(taskPromises);
    console.log('All tasks completed successfully.');
```

```
    console.log('Results:', results);
    // Additional processing with the aggregated results
  } catch (error) {
    console.error('Error performing tasks:', error);
  }
}

const taskIds = [1, 2, 3, 4, 5];

performParallelTasks(taskIds);
```

In this example:

## JavaScript's unusual nature

- The `performAsyncTask` function simulates an asynchronous task with a random delay. It returns a Promise that resolves with the result of the task.
- The `performParallelTasks` function takes an array of task IDs, maps each ID to a Promise returned by `performAsyncTask`, and then uses `Promise.all` to execute all promises concurrently. The `await` keyword is used to wait for all promises to settle.
- The `Promise.all` method returns an array of results in the same order as the input promises. These results are then logged to the console, indicating the completion of all tasks.

## Case Study 7 - instanceof and typeof

Consider the example below

```
"lws"; // → 'lws'  
typeof "lws"; // → 'string'  
"lws" instanceof String; // → false
```

Typeof operator checks the primitive type of a value, whereas instanceof checks if the value is an instance of a class or class function.

```
new String("lws") instanceof String; // true
```

## Case Study 8 - Tagged Template Literals

A `Tagged Template Literal` is usually a function that precedes a `template literal` to help you manipulate the output.

Define a function which logs all params into the console:

```
function f(...args) {  
  return args;  
}
```

```
f`true is ${true}, false is ${false}, array is ${[1, 2, 3]}`;
```

What's the output of the above weird-looking function call, and why?

# JavaScript's unusual nature

Well, this is not magic at all if you're familiar with *Tagged template literals*. In the example above, `f` function is a tag for template literal. Tags before template literal allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions. Example:

```
function template(strings, ...keys) {  
  // do something with strings and keys...  
}
```

This is the [magic behind](#) a famous library called [styled-components](#), which is popular in the React community.

## Case Study 9 - A Well-Known Joke(Precision of 0.1 + 0.2)

With unusuality, this is a big joke in the developer community. How heck on the earth,the

```
0.1 + 0.2 === 0.3; // → false
```

First of all, if you do `0.1 + 0.2` is JavaScript, you do not get an exact `0.3` . Rather, you get a deadly precise number `0.30000000000000004` .

It is really not a JavaScript problem. What is happening here is the foating-point math. The constants `0.2` and `0.3` in your program will also be approximations to their true values. It happens that the closest `double` to `0.2` is larger than the rational number `0.2` but that the closest `double` to `0.3` is smaller than the rational number `0.3` . The sum of `0.1` and `0.2` winds up being larger than the rational number `0.3` and hence disagreeing with the constant in your code.

Refer: [0.30000000000000004.com/](#)

## Case Study 10 - Comparison of three numbers

```
1 < 2 < 3; // → true  
3 > 2 > 1; // → false
```

Let's break down -



## JavaScript's unusual nature

1. ``1 < 2 < 3;``

- The comparison `1 < 2` evaluates to `true`.
- In the second part of the expression, `true` is treated as the number `1`.
- Therefore, `true < 3` is equivalent to `1 < 3`, which evaluates to `true`.

2. ``3 > 2 > 1;``

- The comparison `3 > 2` evaluates to `true`.
- `True` is treated as the number `1` in the next comparison.
- Hence, `true > 1` is equivalent to `1 > 1`, which evaluates to `false`.

The unexpected behavior arises because JavaScript doesn't perform chained comparisons as we might intuitively expect. Instead, it evaluates each comparison independently and performs implicit type coercion.

To obtain the desired result, where all comparisons are evaluated correctly, we can use the greater than or equal (`>=`) operator: `3 > 2 >= 1;`