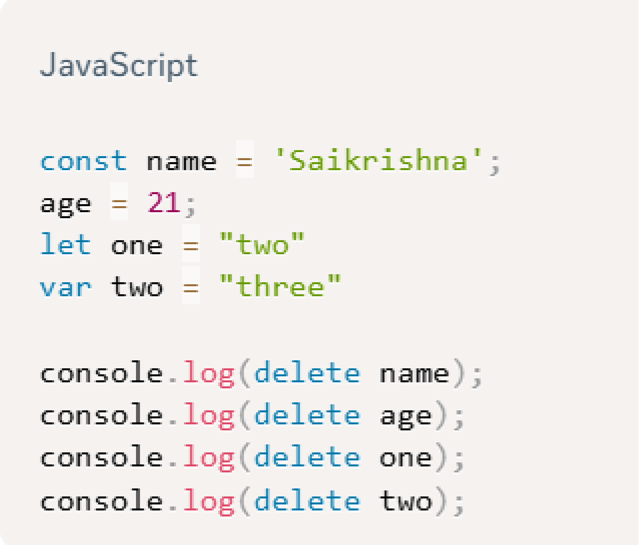
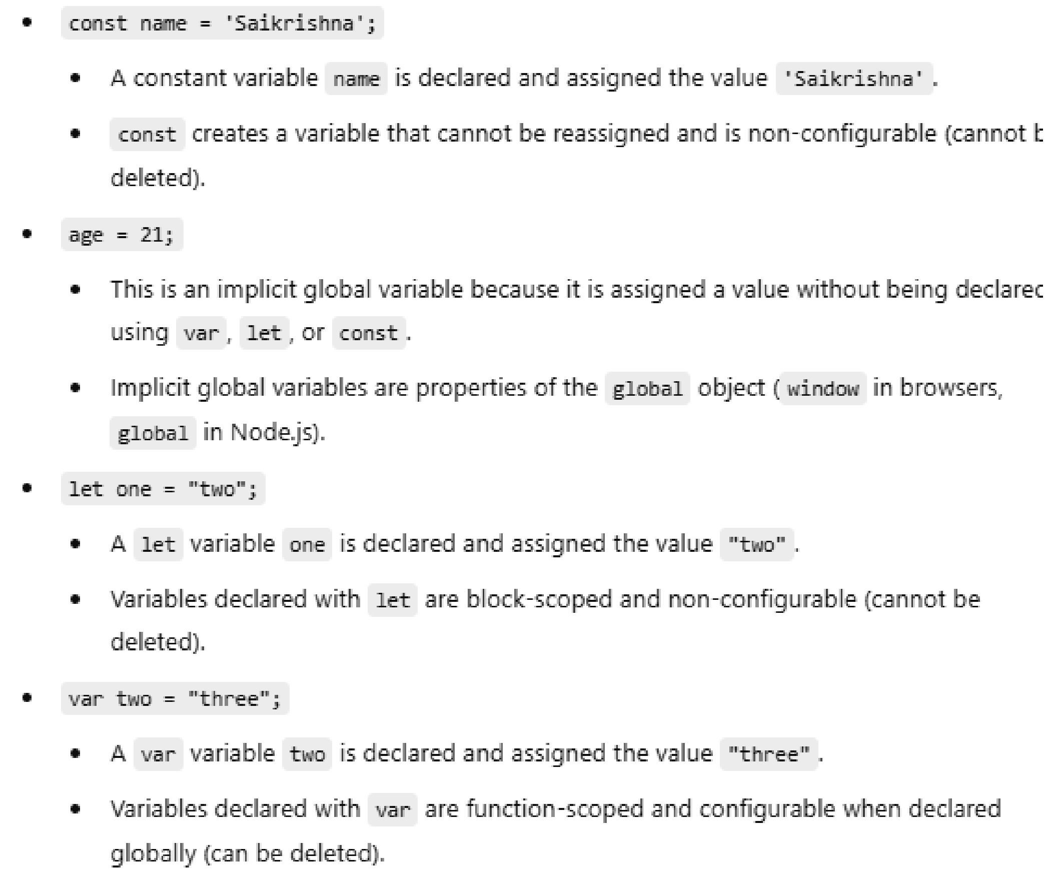
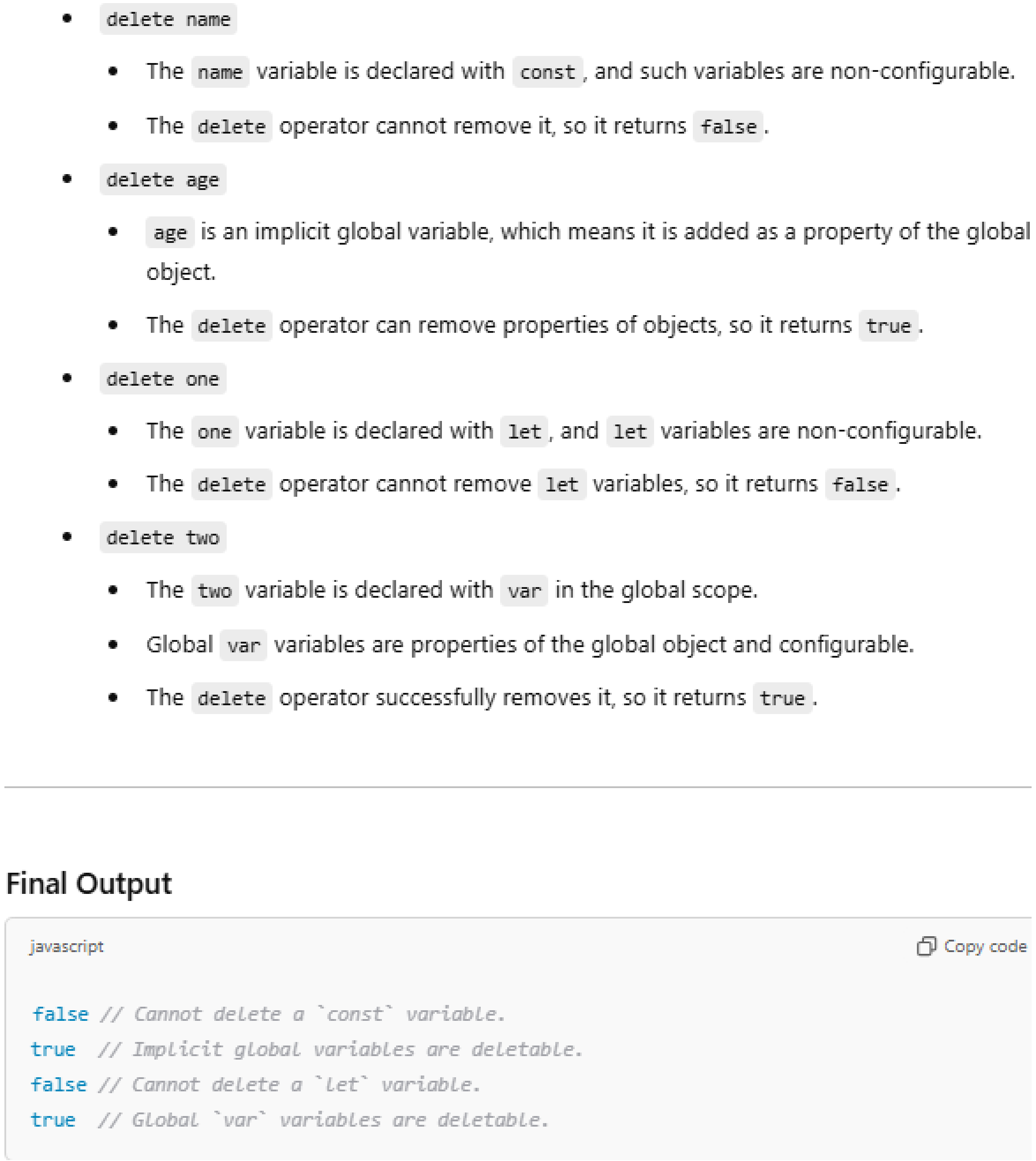
PROGRAMMING QUESTIONS



```javascript
JavaScript

const name = 'Saikrishna';
age = 21;
let one = "two"
var two = "three"

console.log(delete name);
console.log(delete age);
console.log(delete one);
console.log(delete two);
```



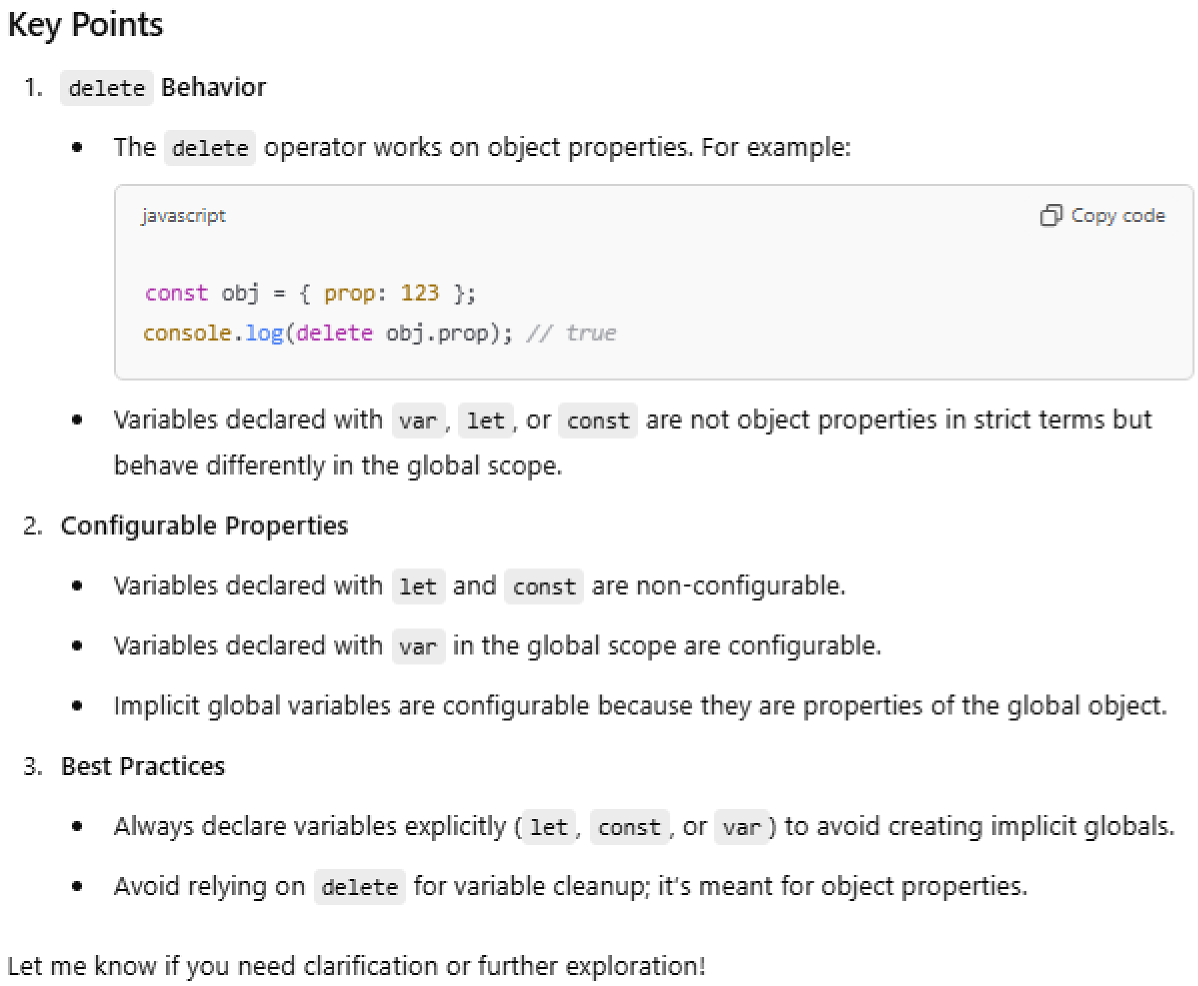
- `const name = 'Saikrishna';`
  - A constant variable `name` is declared and assigned the value `'Saikrishna'`.
  - `const` creates a variable that cannot be reassigned and is non-configurable (cannot b deleted).
- `age = 21;`
  - This is an implicit global variable because it is assigned a value without being declarec using `var`, `let`, or `const`.
  - Implicit global variables are properties of the `global` object (`window` in browsers, `global` in Node.js).
- `let one = "two";`
  - A `let` variable `one` is declared and assigned the value `"two"`.
  - Variables declared with `let` are block-scoped and non-configurable (cannot be deleted).
- `var two = "three";`
  - A `var` variable `two` is declared and assigned the value `"three"`.
  - Variables declared with `var` are function-scoped and configurable when declared globally (can be deleted).



- `delete name`
  - The `name` variable is declared with `const`, and such variables are non-configurable.
  - The `delete` operator cannot remove it, so it returns `false`.
- `delete age`
  - `age` is an implicit global variable, which means it is added as a property of the global object.
  - The `delete` operator can remove properties of objects, so it returns `true`.
- `delete one`
  - The `one` variable is declared with `let`, and `let` variables are non-configurable.
  - The `delete` operator cannot remove `let` variables, so it returns `false`.
- `delete two`
  - The `two` variable is declared with `var` in the global scope.
  - Global `var` variables are properties of the global object and configurable.
  - The `delete` operator successfully removes it, so it returns `true`.

Final Output

```javascript
false // Cannot delete a `const` variable.
true  // Implicit global variables are deletable.
false // Cannot delete a `let` variable.
true  // Global `var` variables are deletable.
```



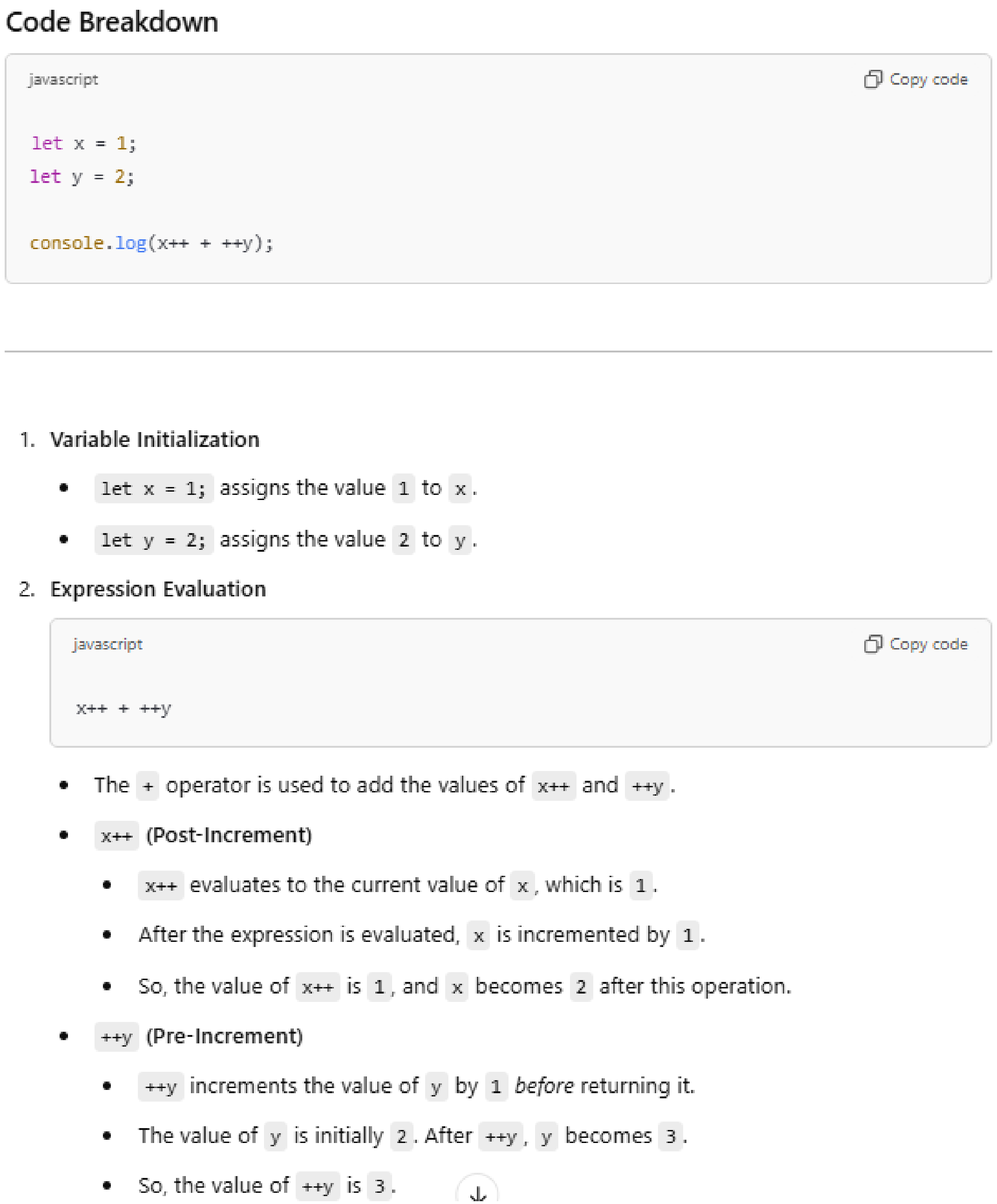
Key Points

1. `delete` **Behavior**
   - The `delete` operator works on object properties. For example:

     ```javascript
     const obj = { prop: 123 };
     console.log(delete obj.prop); // true
     ```

   - Variables declared with `var`, `let`, or `const` are not object properties in strict terms but behave differently in the global scope.
2. **Configurable Properties**
   - Variables declared with `let` and `const` are non-configurable.
   - Variables declared with `var` in the global scope are configurable.
   - Implicit global variables are configurable because they are properties of the global object.
3. **Best Practices**
   - Always declare variables explicitly (`let`, `const`, or `var`) to avoid creating implicit globals.
   - Avoid relying on `delete` for variable cleanup; it's meant for object properties.

Let me know if you need clarification or further exploration!



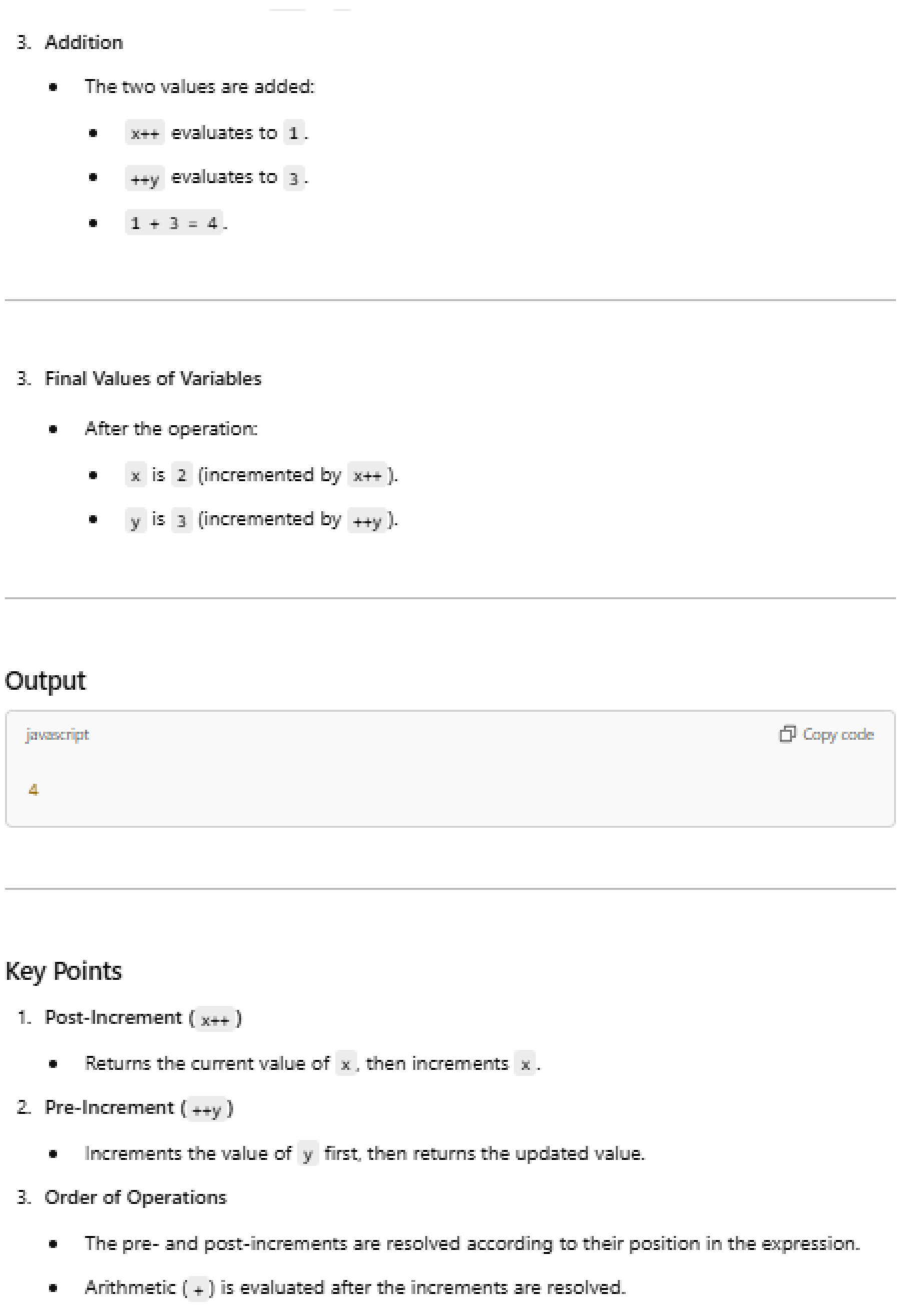
Code Breakdown

```javascript
let x = 1;
let y = 2;

console.log(x++ + ++y);
```

1. **Variable Initialization**
   - `let x = 1;` assigns the value `1` to `x`.
   - `let y = 2;` assigns the value `2` to `y`.
2. **Expression Evaluation**

   ```javascript
   x++ + ++y
   ```

   - The `+` operator is used to add the values of `x++` and `++y`.
   - `x++` **(Post-Increment)**
     - `x++` evaluates to the current value of `x`, which is `1`.
     - After the expression is evaluated, `x` is incremented by `1`.
     - So, the value of `x++` is `1`, and `x` becomes `2` after this operation.
   - `++y` **(Pre-Increment)**
     - `++y` increments the value of `y` by `1` *before* returning it.
     - The value of `y` is initially `2`. After `++y`, `y` becomes `3`.
     - So, the value of `++y` is `3`.



3. **Addition**
   - The two values are added:
     - `x++` evaluates to `1`.
     - `++y` evaluates to `3`.
     - `1 + 3 = 4`.

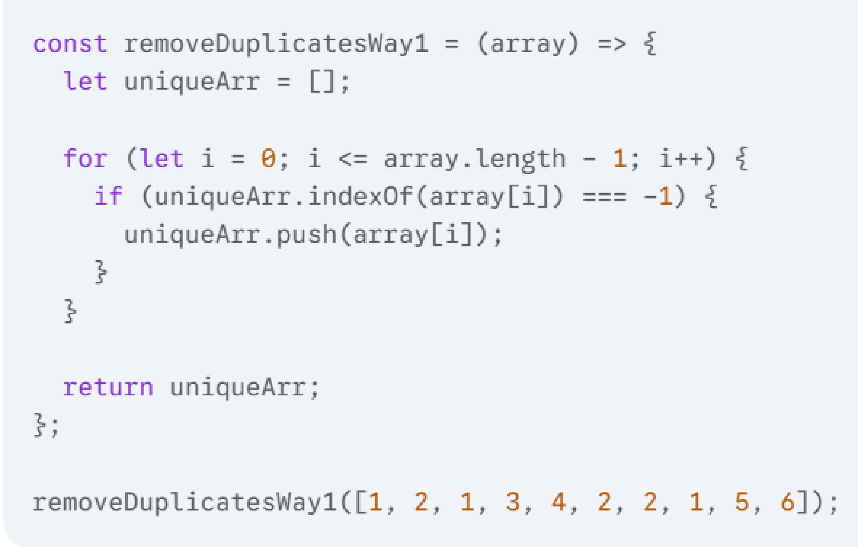
3. **Final Values of Variables**
   - After the operation:
     - `x` is `2` (incremented by `x++`).
     - `y` is `3` (incremented by `++y`).

Output

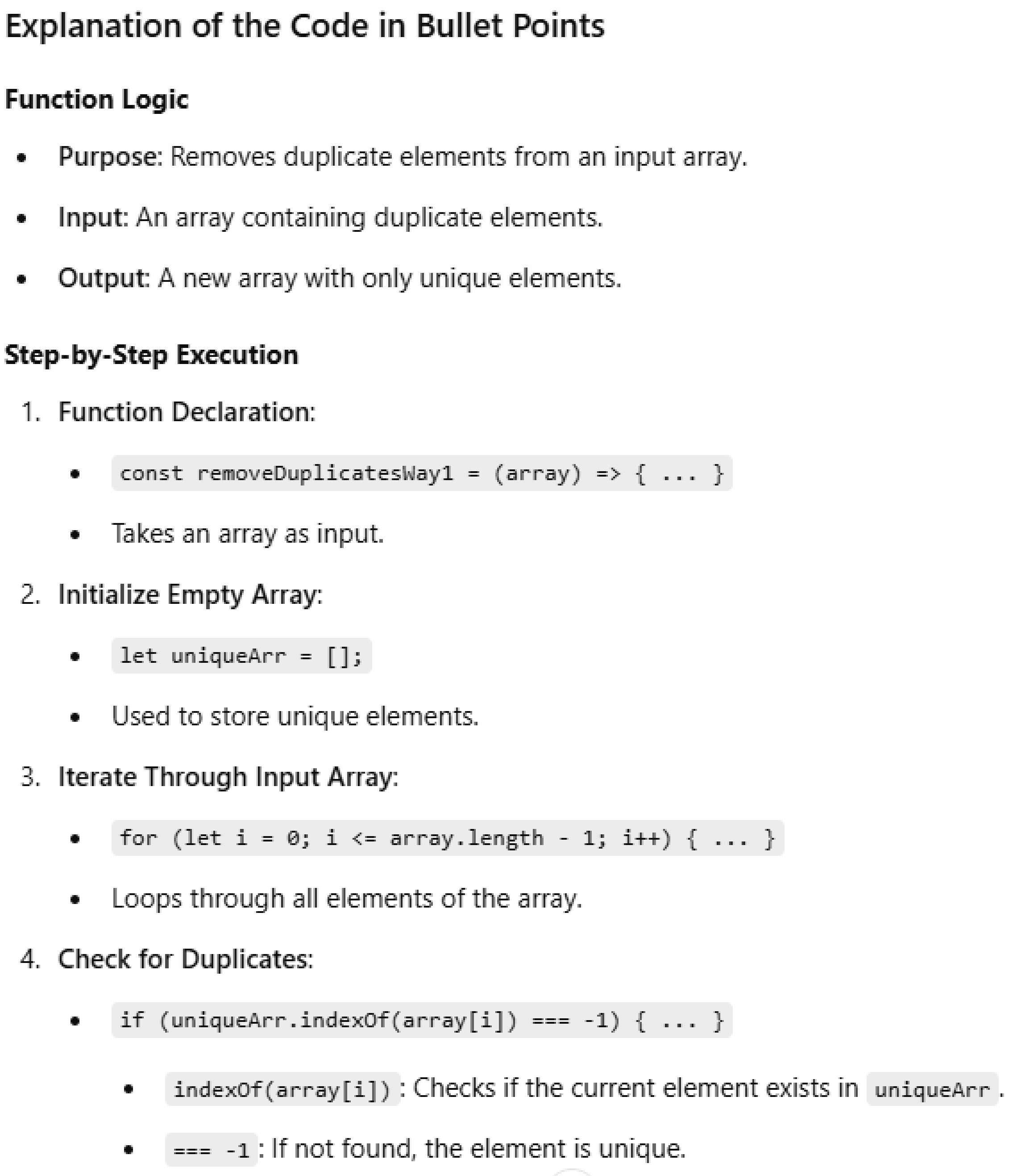
```javascript
4
```

Key Points

1. **Post-Increment (`x++`)**
   - Returns the current value of `x`, then increments `x`.
2. **Pre-Increment (`++y`)**
   - Increments the value of `y` first, then returns the updated value.
3. **Order of Operations**
   - The pre- and post-increments are resolved according to their position in the expression.
   - Arithmetic (`+`) is evaluated after the increments are resolved.



```
const removeDuplicatesWay1 = (array) => {
  let uniqueArr = [];

  for (let i = 0; i <= array.length - 1; i++) {
    if (uniqueArr.indexOf(array[i]) === -1) {
      uniqueArr.push(array[i]);
    }
  }

  return uniqueArr;
};

removeDuplicatesWay1([1, 2, 1, 3, 4, 2, 2, 1, 5, 6]);
```



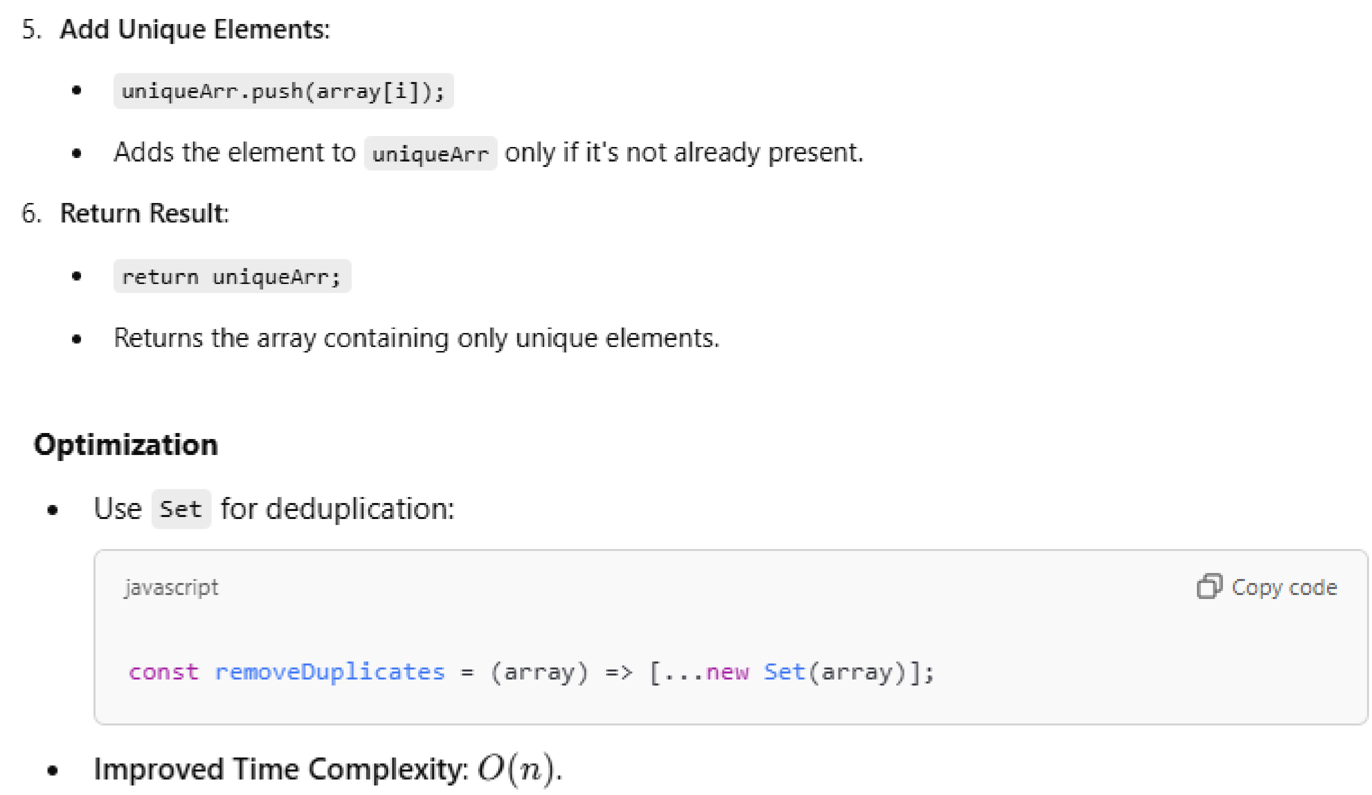
Explanation of the Code in Bullet Points

Function Logic

- **Purpose**: Removes duplicate elements from an input array.
- **Input**: An array containing duplicate elements.
- **Output**: A new array with only unique elements.

Step-by-Step Execution

1. **Function Declaration**:
   - `const removeDuplicatesWay1 = (array) => { ... }`
   - Takes an array as input.
2. **Initialize Empty Array**:
   - `let uniqueArr = [];`
   - Used to store unique elements.
3. **Iterate Through Input Array**:
   - `for (let i = 0; i <= array.length - 1; i++) { ... }`
   - Loops through all elements of the array.
4. **Check for Duplicates**:
   - `if (uniqueArr.indexOf(array[i]) === -1) { ... }`
     - `indexOf(array[i])`: Checks if the current element exists in `uniqueArr`.
     - `=== -1`: If not found, the element is unique.



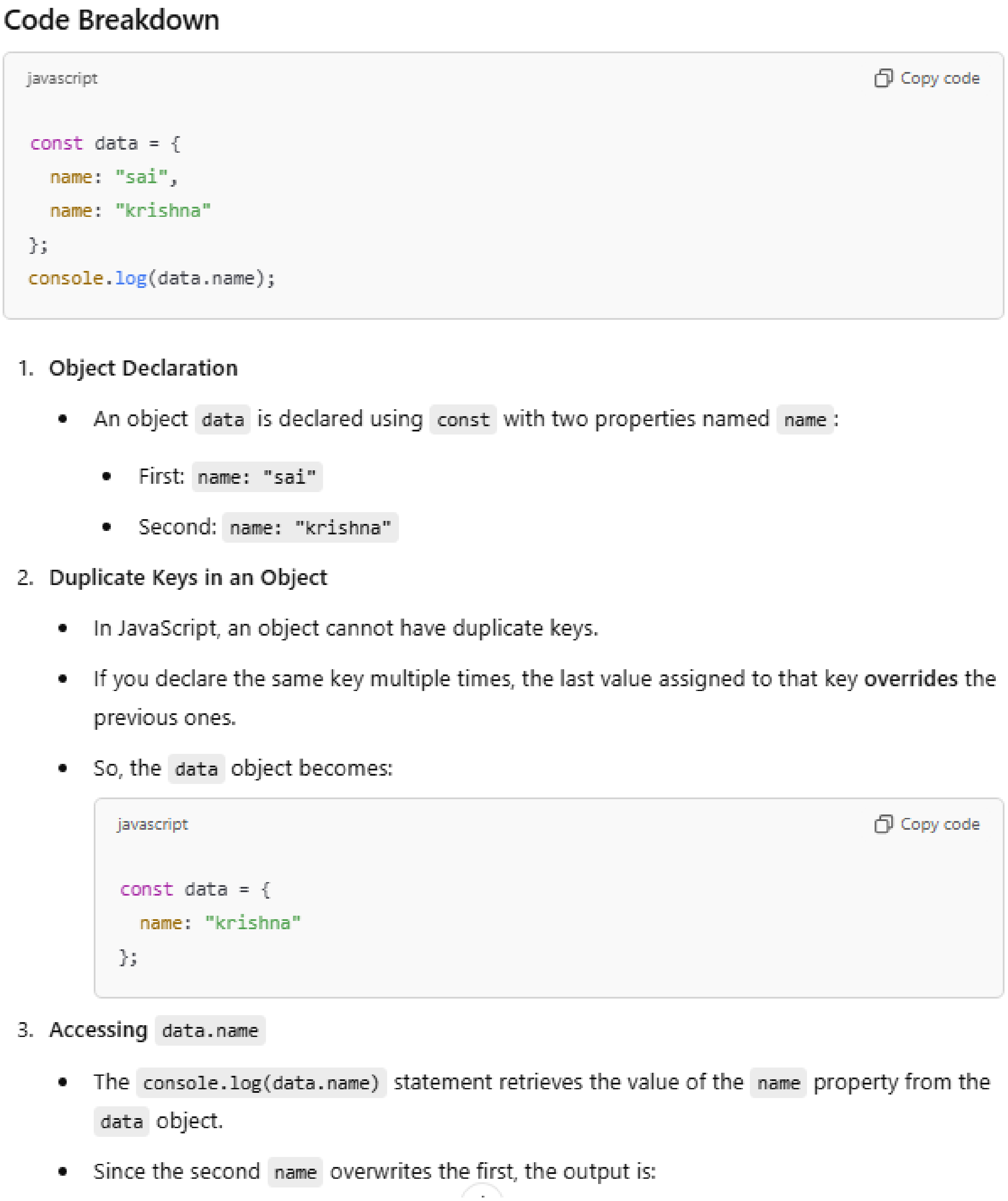
5. **Add Unique Elements**:
   - `uniqueArr.push(array[i]);`
   - Adds the element to `uniqueArr` only if it's not already present.
6. **Return Result**:
   - `return uniqueArr;`
   - Returns the array containing only unique elements.

Optimization

- Use `Set` for deduplication:

```javascript
const removeDuplicates = (array) => [...new Set(array)];
```

- **Improved Time Complexity**: $O(n)$.



Code Breakdown

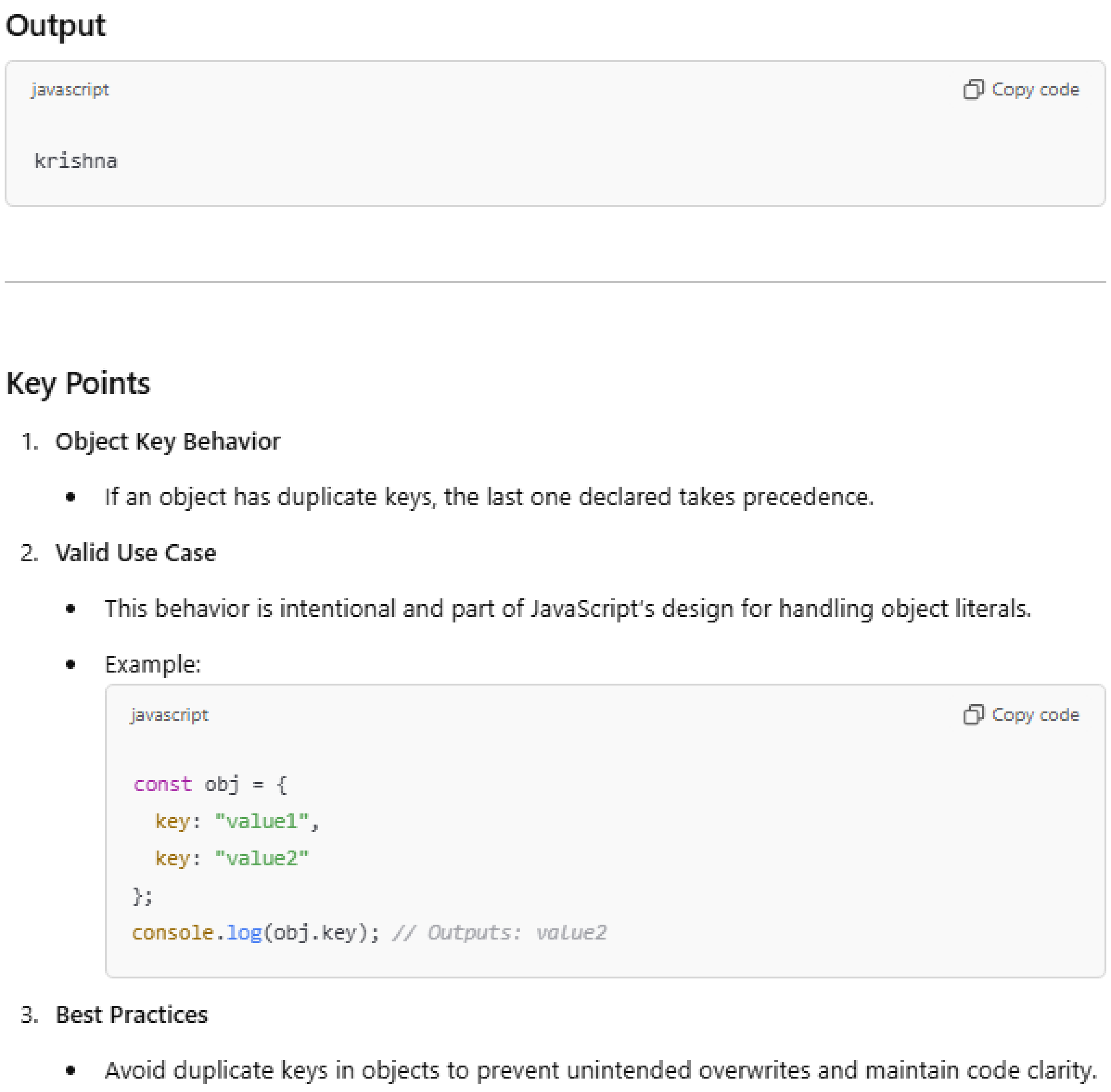
javascript

```javascript
const data = {
  name: "sai",
  name: "krishna"
};
console.log(data.name);
```

1. **Object Declaration**
   - An object `data` is declared using `const` with two properties named `name`:
     - First: `name: "sai"`
     - Second: `name: "krishna"`
2. **Duplicate Keys in an Object**
   - In JavaScript, an object cannot have duplicate keys.
   - If you declare the same key multiple times, the last value assigned to that key **overrides** the previous ones.
   - So, the `data` object becomes:

     javascript

     ```javascript
     const data = {
       name: "krishna"
     };
     ```
3. **Accessing `data.name`**
   - The `console.log(data.name)` statement retrieves the value of the `name` property from the `data` object.
   - Since the second `name` overwrites the first, the output is:



Output

javascript

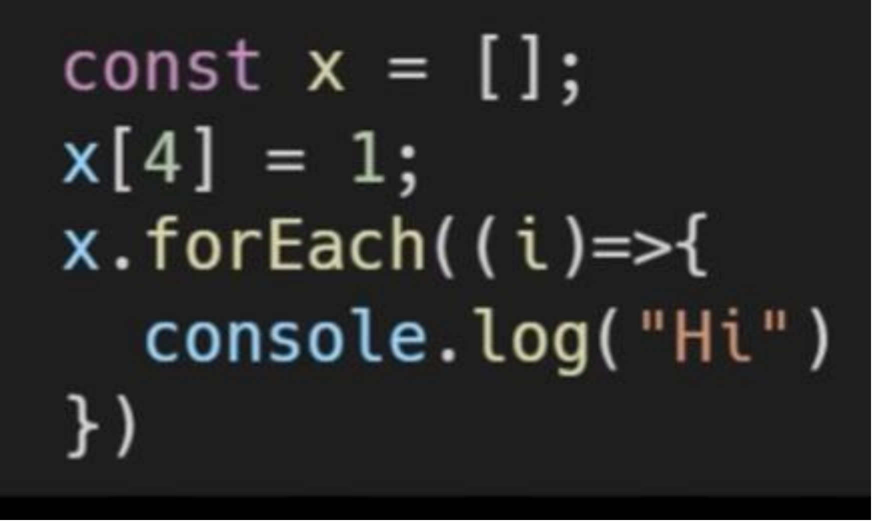
```
krishna
```

Key Points

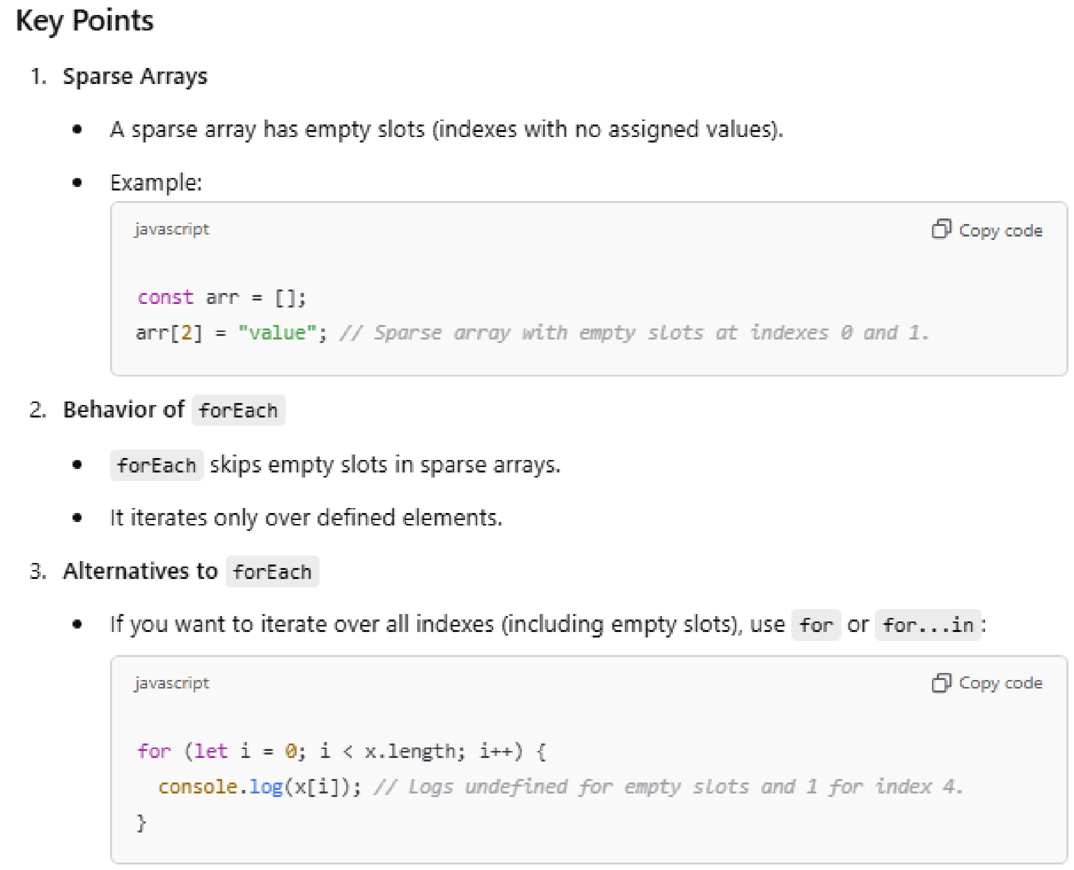
1. **Object Key Behavior**
   - If an object has duplicate keys, the last one declared takes precedence.
2. **Valid Use Case**
   - This behavior is intentional and part of JavaScript's design for handling object literals.
   - Example:

     javascript

     ```javascript
     const obj = {
       key: "value1",
       key: "value2"
     };
     console.log(obj.key); // Outputs: value2
     ```
3. **Best Practices**
   - Avoid duplicate keys in objects to prevent unintended overwrites and maintain code clarity.



```
const x = [];
x[4] = 1;
x.forEach((i)=>{
  console.log("Hi")
})
```



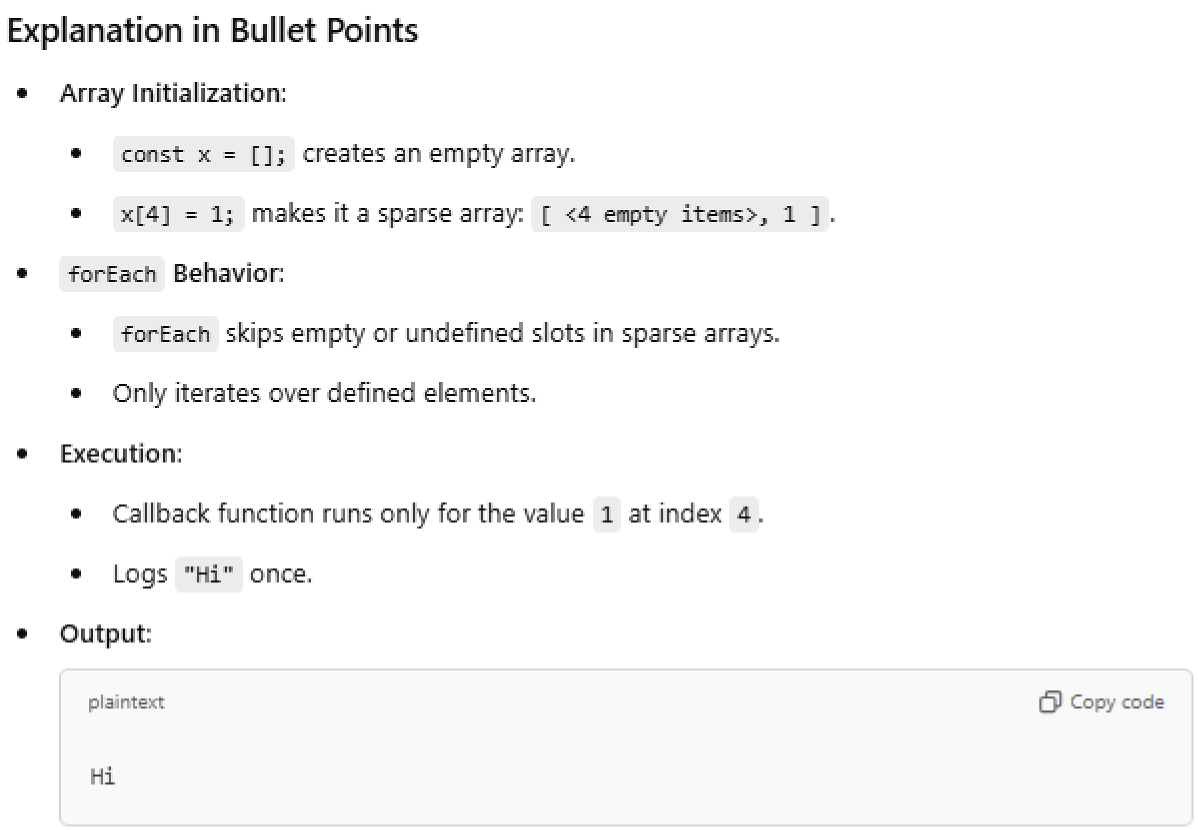
Key Points

1. Sparse Arrays
   - A sparse array has empty slots (indexes with no assigned values).
   - Example:

     ```javascript
     const arr = [];
     arr[2] = "value"; // Sparse array with empty slots at indexes 0 and 1.
     ```

2. **Behavior of `forEach`**
   - `forEach` skips empty slots in sparse arrays.
   - It iterates only over defined elements.
3. **Alternatives to `forEach`**
   - If you want to iterate over all indexes (including empty slots), use `for` or `for...in`:

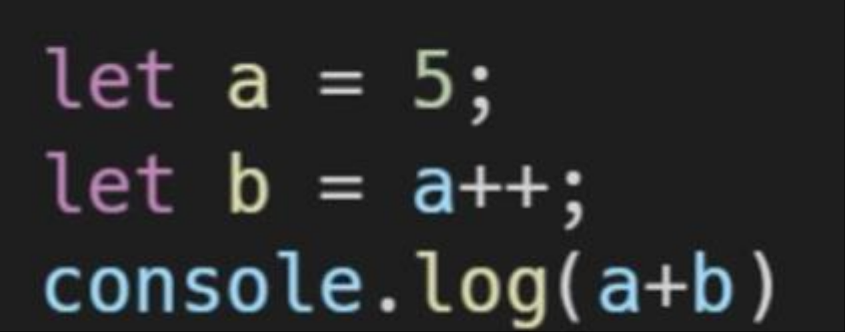
     ```javascript
     for (let i = 0; i < x.length; i++) {
       console.log(x[i]); // Logs undefined for empty slots and 1 for index 4.
     }
     ```



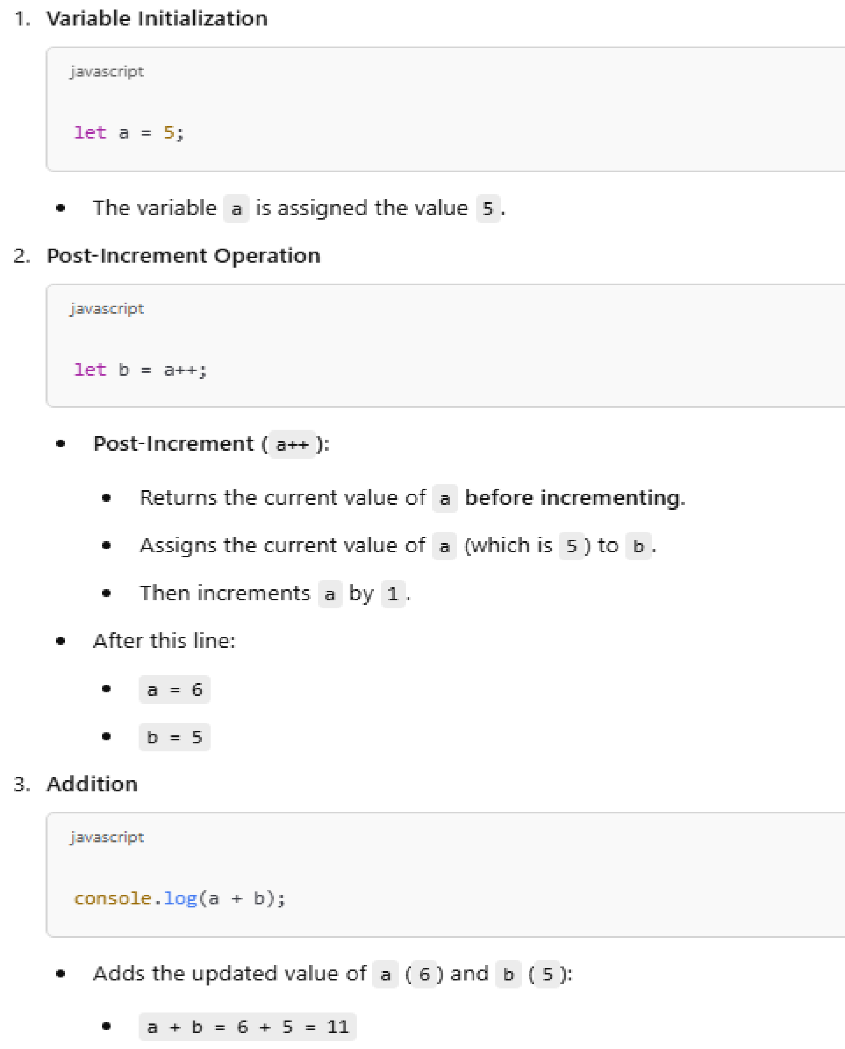
Explanation in Bullet Points

- **Array Initialization:**
  - `const x = [];` creates an empty array.
  - `x[4] = 1;` makes it a sparse array: `[ <4 empty items>, 1 ]`.
- **`forEach` Behavior:**
  - `forEach` skips empty or undefined slots in sparse arrays.
  - Only iterates over defined elements.
- **Execution:**
  - Callback function runs only for the value `1` at index `4`.
  - Logs `"Hi"` once.
- **Output:**

  ```plaintext
  Hi
  ```



```javascript
let a = 5;
let b = a++;
console.log(a+b)
```



1. **Variable Initialization**

   javascript

   ```javascript
   let a = 5;
   ```

   - The variable `a` is assigned the value `5`.

2. **Post-Increment Operation**

   javascript

   ```javascript
   let b = a++;
   ```

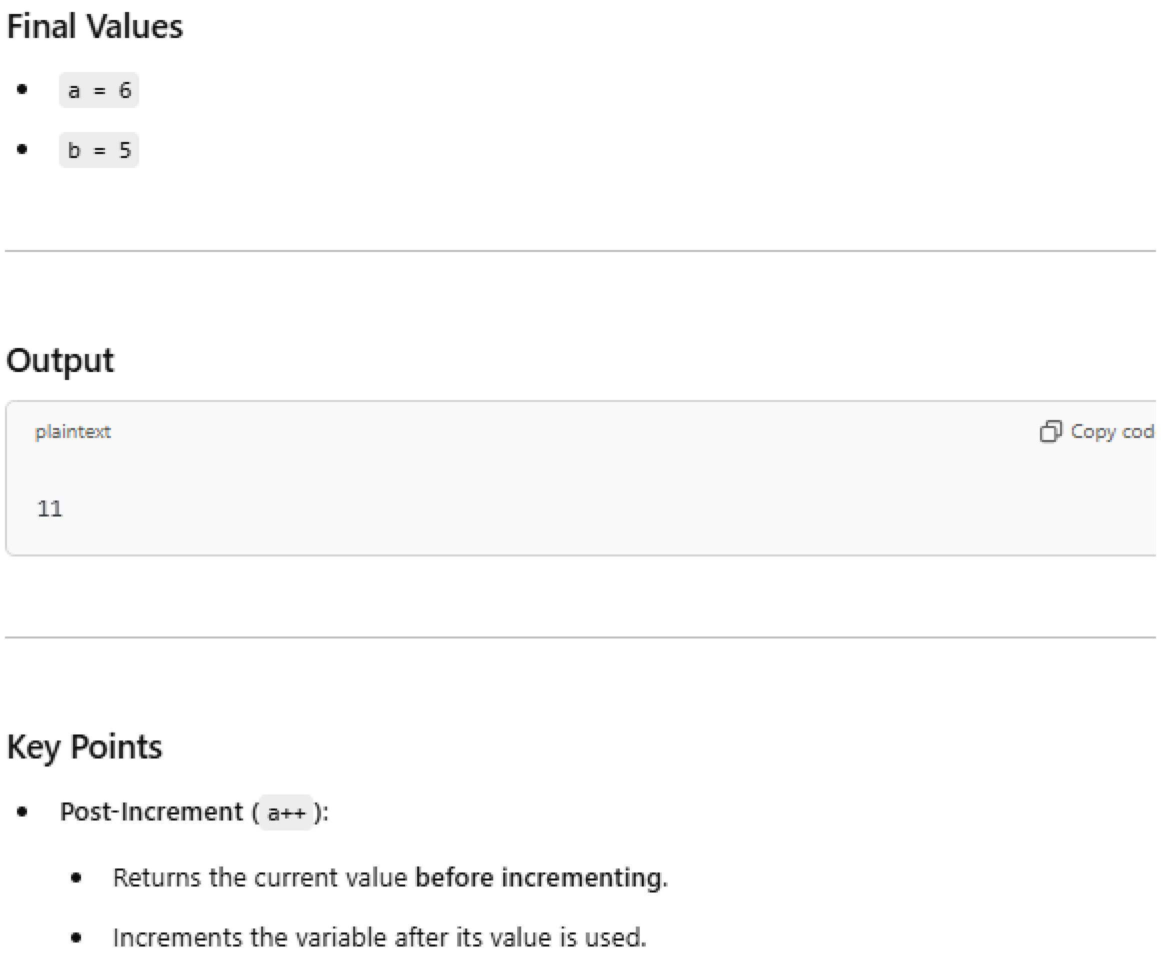
   - **Post-Increment** (`a++`):
     - Returns the current value of `a` **before incrementing.**
     - Assigns the current value of `a` (which is `5`) to `b`.
     - Then increments `a` by `1`.
   - After this line:
     - `a = 6`
     - `b = 5`

3. **Addition**

   javascript

   ```javascript
   console.log(a + b);
   ```

   - Adds the updated value of `a` (`6`) and `b` (`5`):
     - `a + b = 6 + 5 = 11`



Final Values

- `a = 6`
- `b = 5`

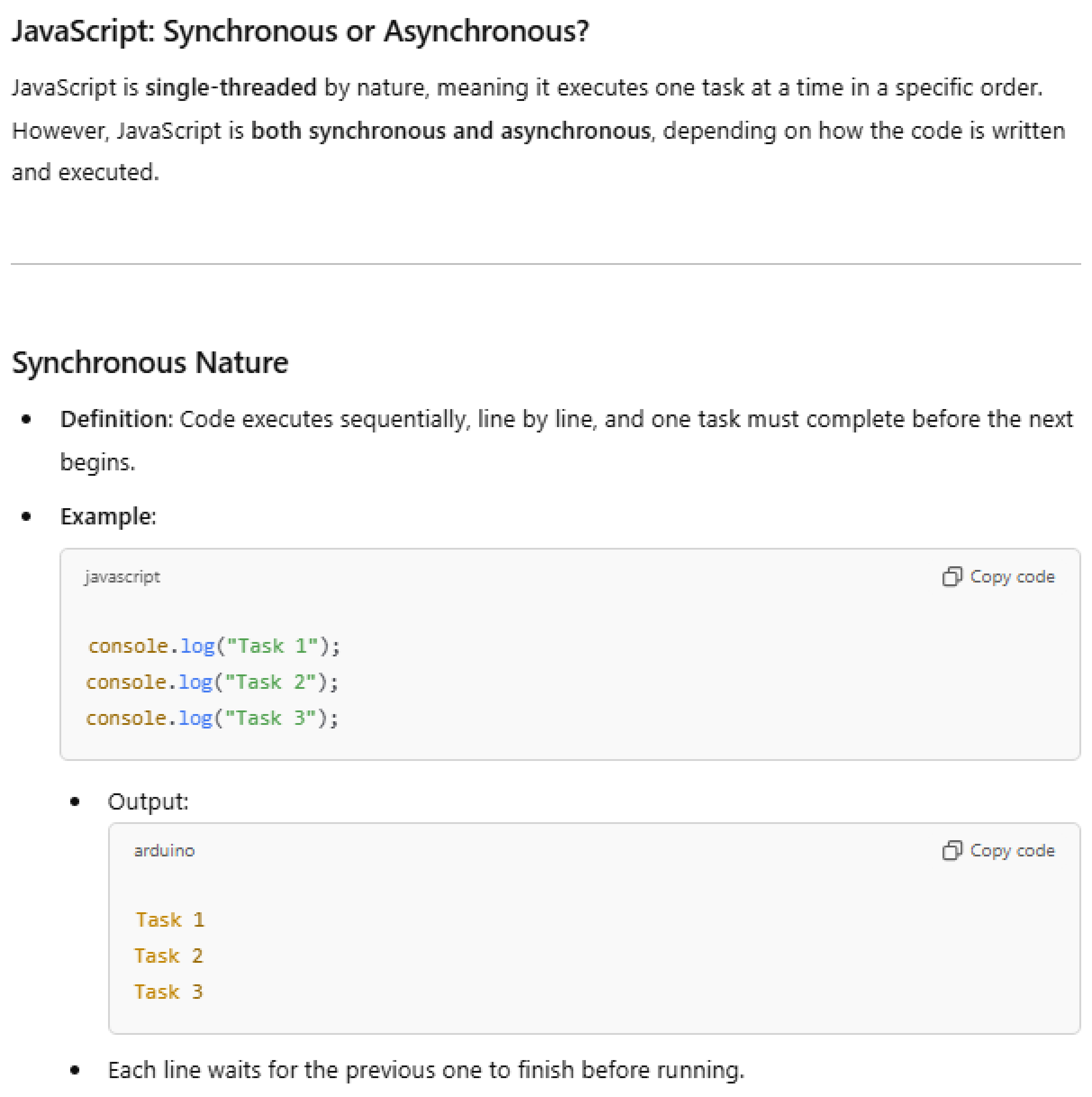
Output

plaintext

```
11
```

Key Points

- **Post-Increment** (`a++`):
  - Returns the current value **before incrementing.**
  - Increments the variable after its value is used.



JavaScript: Synchronous or Asynchronous?

JavaScript is **single-threaded** by nature, meaning it executes one task at a time in a specific order. However, JavaScript is **both synchronous and asynchronous**, depending on how the code is written and executed.

Synchronous Nature

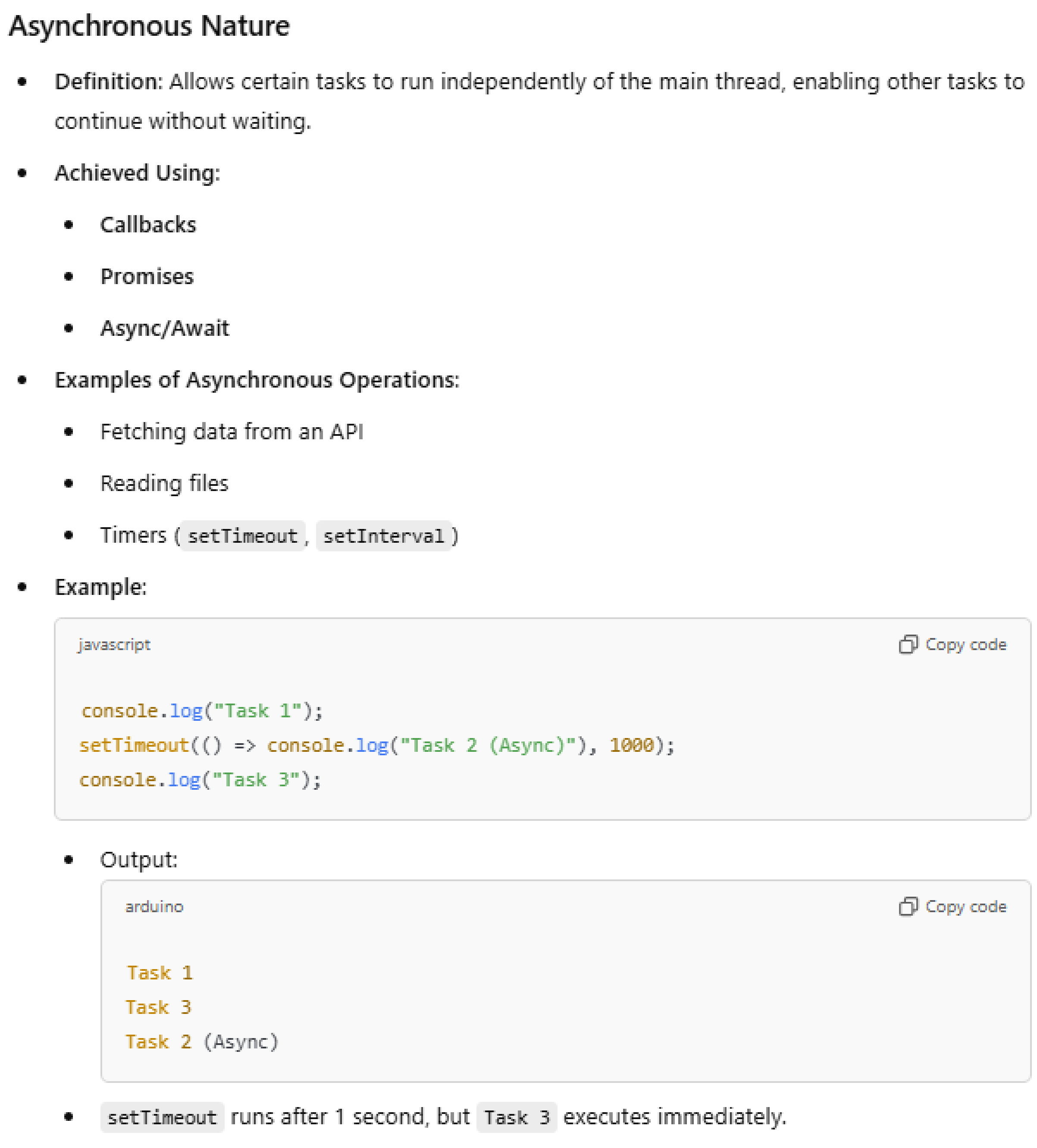
- **Definition:** Code executes sequentially, line by line, and one task must complete before the next begins.
- **Example:**

```javascript
console.log("Task 1");
console.log("Task 2");
console.log("Task 3");
```

  - Output:

```arduino
Task 1
Task 2
Task 3
```

  - Each line waits for the previous one to finish before running.



Asynchronous Nature

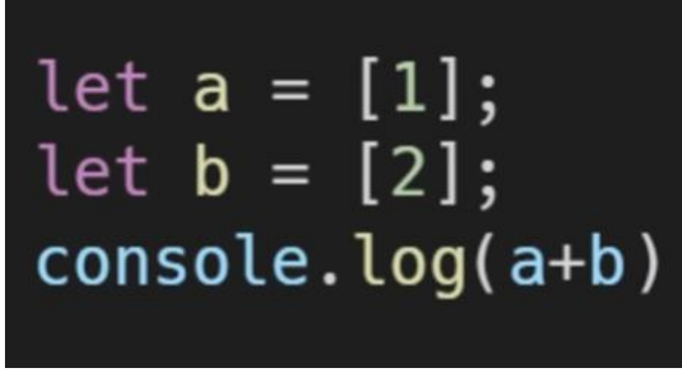
- **Definition:** Allows certain tasks to run independently of the main thread, enabling other tasks to continue without waiting.
- **Achieved Using:**
  - **Callbacks**
  - **Promises**
  - **Async/Await**
- **Examples of Asynchronous Operations:**
  - Fetching data from an API
  - Reading files
  - Timers (`setTimeout`, `setInterval`)
- **Example:**

```javascript
console.log("Task 1");
setTimeout(() => console.log("Task 2 (Async)"), 1000);
console.log("Task 3");
```

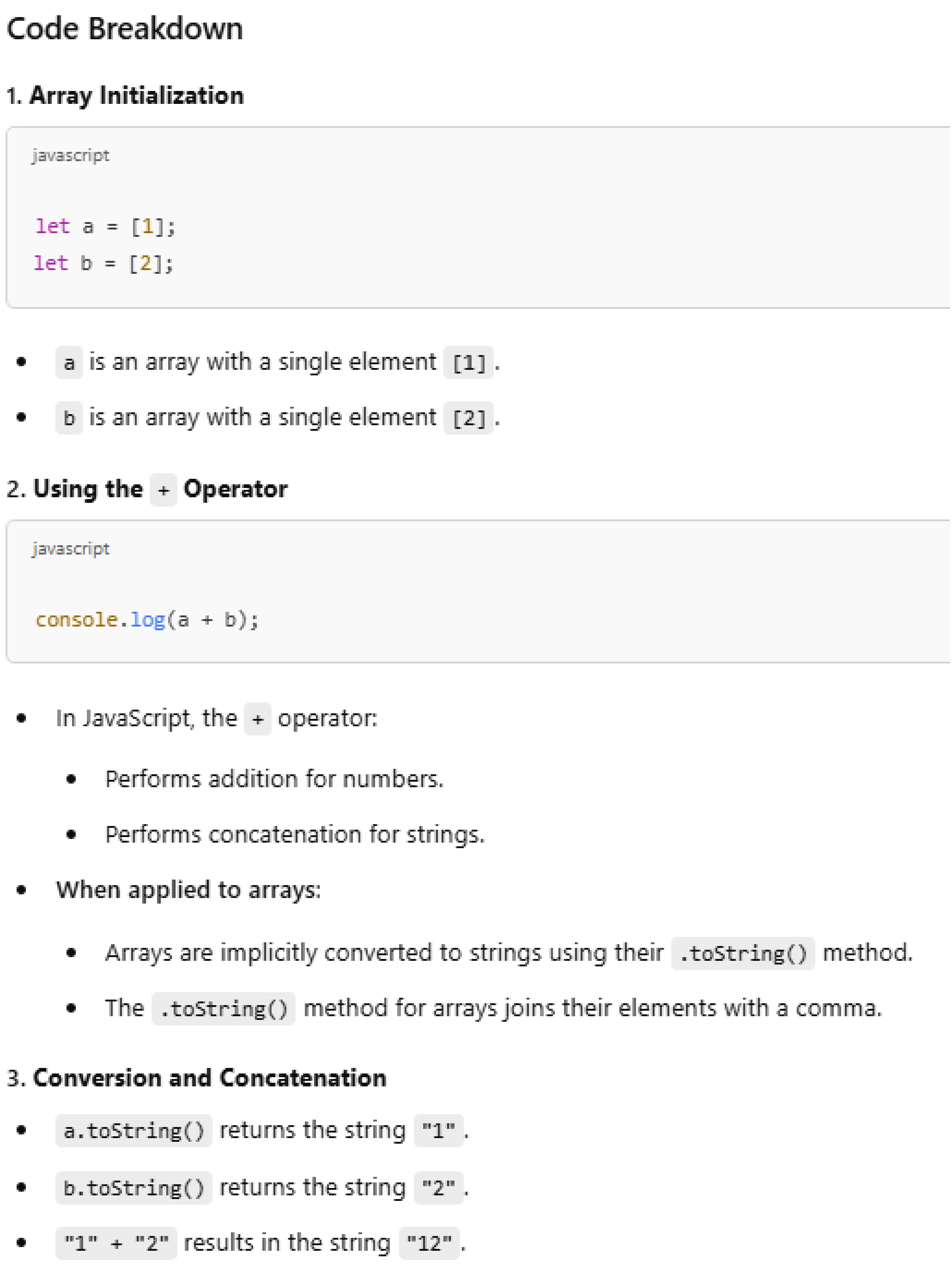
  - Output:

```arduino
Task 1
Task 3
Task 2 (Async)
```

  - `setTimeout` runs after 1 second, but `Task 3` executes immediately.



```
let a = [1];
let b = [2];
console.log(a+b)
```



Code Breakdown

1. Array Initialization

```javascript
let a = [1];
let b = [2];
```

- `a` is an array with a single element `[1]`.
- `b` is an array with a single element `[2]`.

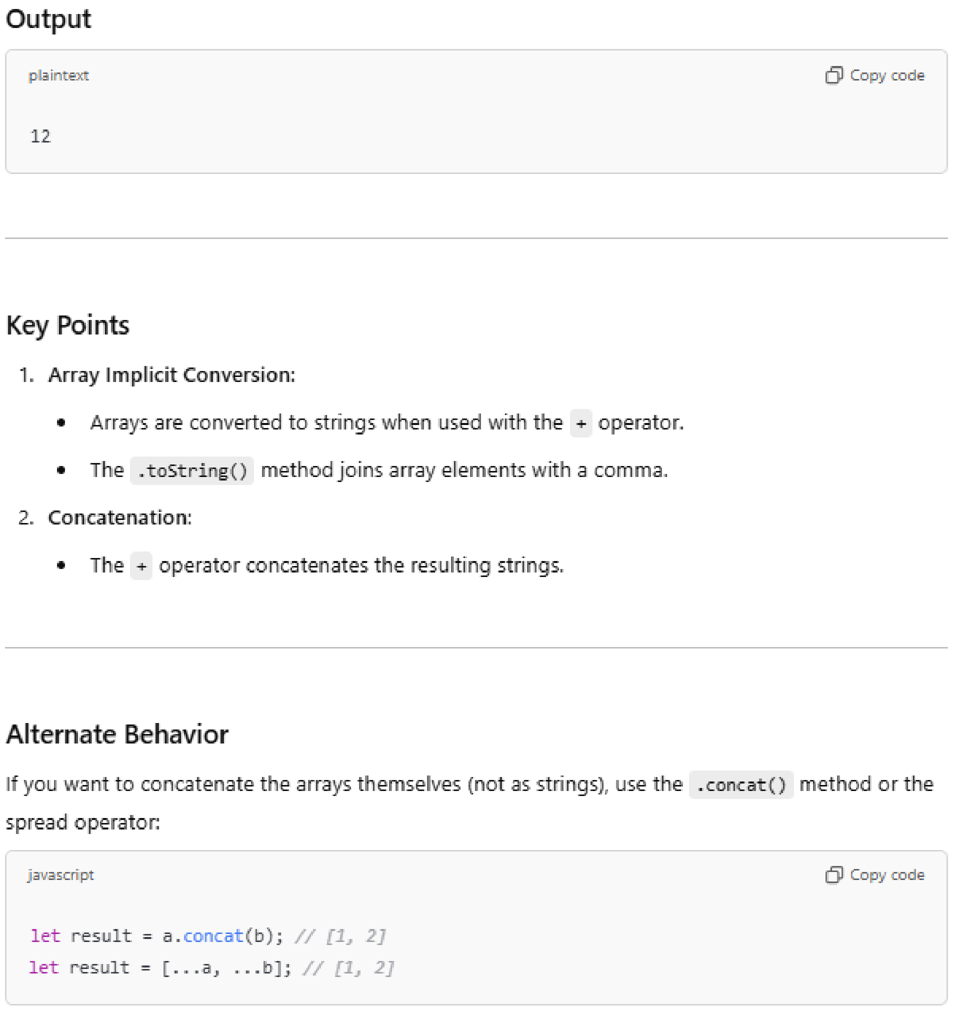
2. Using the `+` Operator

```javascript
console.log(a + b);
```

- In JavaScript, the `+` operator:
  - Performs addition for numbers.
  - Performs concatenation for strings.
- When applied to arrays:
  - Arrays are implicitly converted to strings using their `.toString()` method.
  - The `.toString()` method for arrays joins their elements with a comma.

3. Conversion and Concatenation

- `a.toString()` returns the string `"1"`.
- `b.toString()` returns the string `"2"`.
- `"1" + "2"` results in the string `"12"`.



Output

```plaintext
12
```

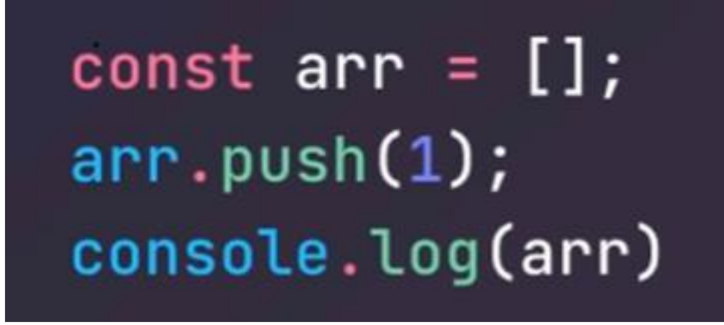
Key Points

1. **Array Implicit Conversion:**
   - Arrays are converted to strings when used with the `+` operator.
   - The `.toString()` method joins array elements with a comma.
2. **Concatenation:**
   - The `+` operator concatenates the resulting strings.

Alternate Behavior

If you want to concatenate the arrays themselves (not as strings), use the `.concat()` method or the spread operator:

```javascript
let result = a.concat(b); // [1, 2]
let result = [...a, ...b]; // [1, 2]
```



```javascript
const arr = [];
arr.push(1);
console.log(arr)
```

Code Breakdown

1. Array Initialization

   ```javascript
   const arr = [];
   ```

   - `arr` is declared as a constant and initialized as an empty array (`[]`).

2. Using `push()` Method

   ```javascript
   arr.push(1);
   ```

   - The `push()` method adds an element (`1`) to the **end** of the array.
   - After this operation, the array becomes `[1]`.

3. Logging the Array

   ```javascript
   console.log(arr);
   ```

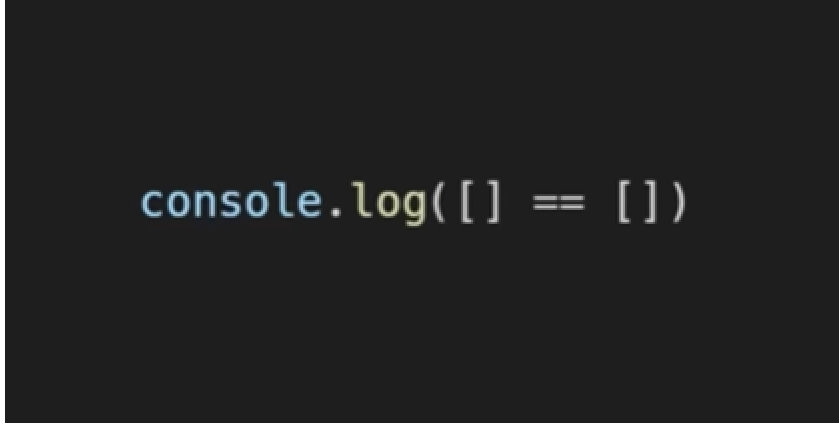
   - Prints the updated array to the console.

Output

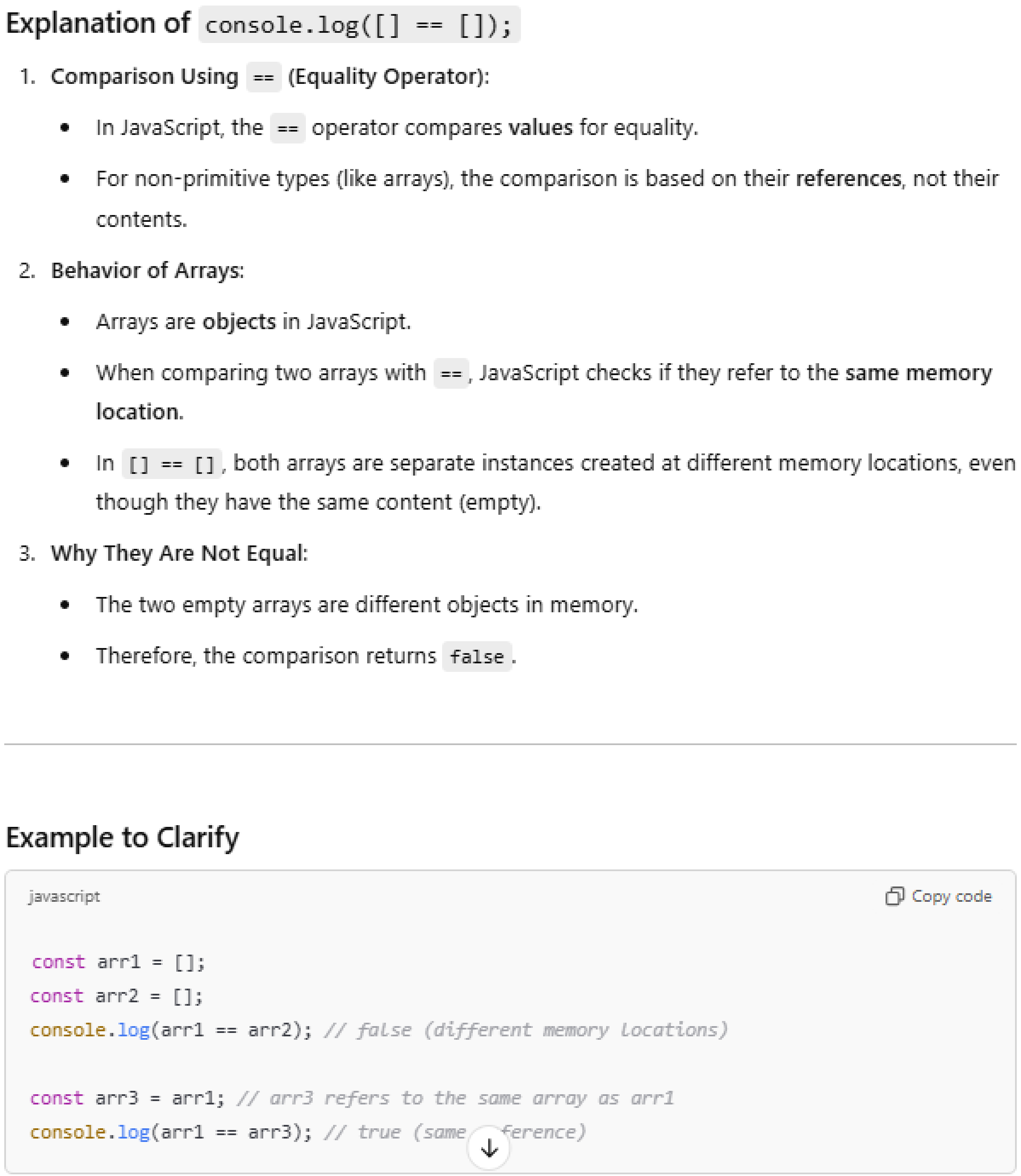
```plaintext
[1]
```

Key Points

1. `const` **with Arrays:**
   - The `const` keyword means the reference to the array (`arr`) cannot change.
   - **You can still modify the contents** of the array (e.g., adding, removing elements).
2. `push()` **Method:**
   - Adds elements to the end of the array.
   - Modifies the array in place and returns the new length of the array.
3. **Final State of** `arr`**:**
   - The array now contains one element: `[1]`.



```
console.log([] == [])
```

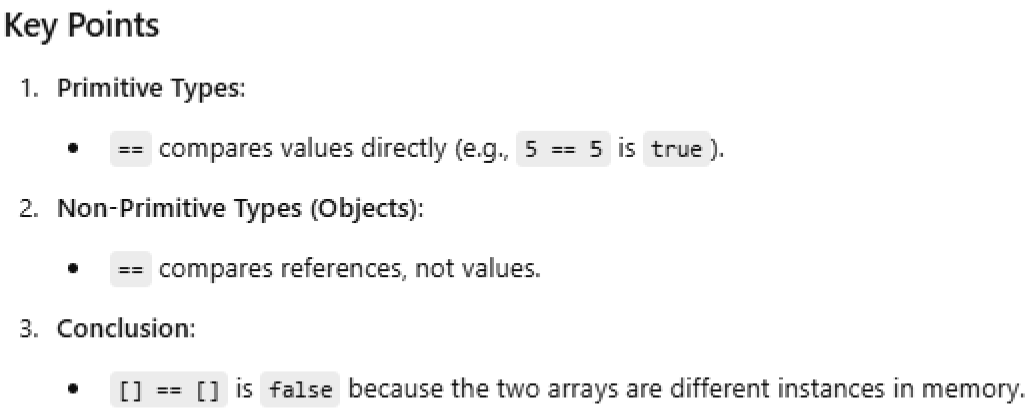


Explanation of `console.log([] == []);`

1. **Comparison Using `==` (Equality Operator):**
   - In JavaScript, the `==` operator compares **values** for equality.
   - For non-primitive types (like arrays), the comparison is based on their **references**, not their contents.
2. **Behavior of Arrays:**
   - Arrays are **objects** in JavaScript.
   - When comparing two arrays with `==`, JavaScript checks if they refer to the **same memory location.**
   - In `[] == []`, both arrays are separate instances created at different memory locations, even though they have the same content (empty).
3. **Why They Are Not Equal:**
   - The two empty arrays are different objects in memory.
   - Therefore, the comparison returns `false`.

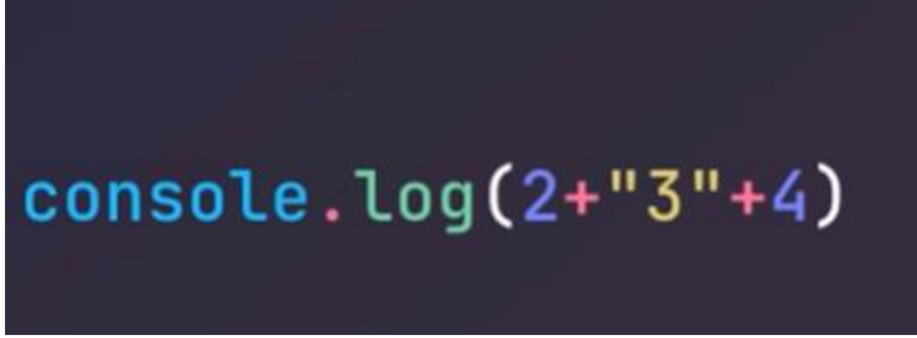
Example to Clarify

```javascript
const arr1 = [];
const arr2 = [];
console.log(arr1 == arr2); // false (different memory locations)

const arr3 = arr1; // arr3 refers to the same array as arr1
console.log(arr1 == arr3); // true (same ference)
```



Key Points

1. **Primitive Types:**
   - `==` compares values directly (e.g., `5 == 5` is `true`).
2. **Non-Primitive Types (Objects):**
   - `==` compares references, not values.
3. **Conclusion:**
   - `[] == []` is `false` because the two arrays are different instances in memory.



```javascript
console.log(2+"3"+4)
```

Code Analysis

1. **Expression:** `2 + "3" + 4`
   - This is a combination of numbers and a string, and JavaScript uses **type coercion** to determine how to handle the operation.
   - The `+` operator:
     - Adds numbers if both operands are numbers.
     - Concatenates (joins) if either operand is a string.
2. **Step-by-Step Evaluation**
   - **Step 1:** `2 + "3"`
     - `2` (a number) and `"3"` (a string) are combined.
     - JavaScript converts `2` to a string (`"2"`) and concatenates it with `"3"`.
     - Result: `"23"` (a string).
   - **Step 2:** `"23" + 4`
     - `"23"` (a string) and `4` (a number) are combined.
     - JavaScript converts `4` to a string (`"4"`) and concatenates it with `"23"`.
     - Result: `"234"` (a string).
3. **Final Output**

```javascript
console.log("234");
```

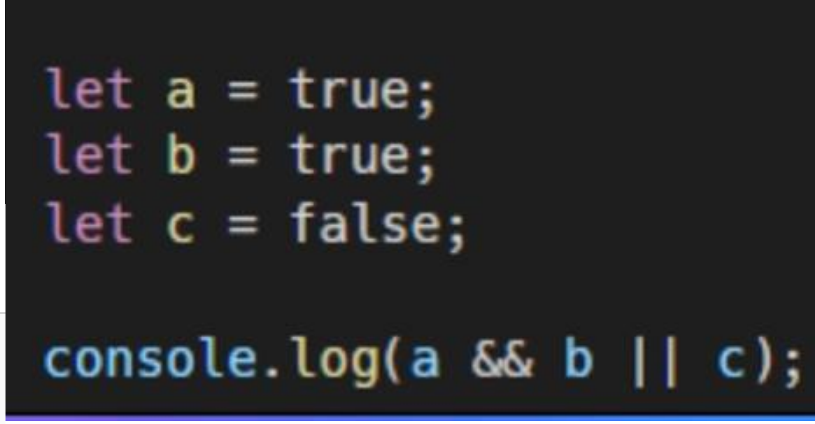
Output

```plaintext
234
```

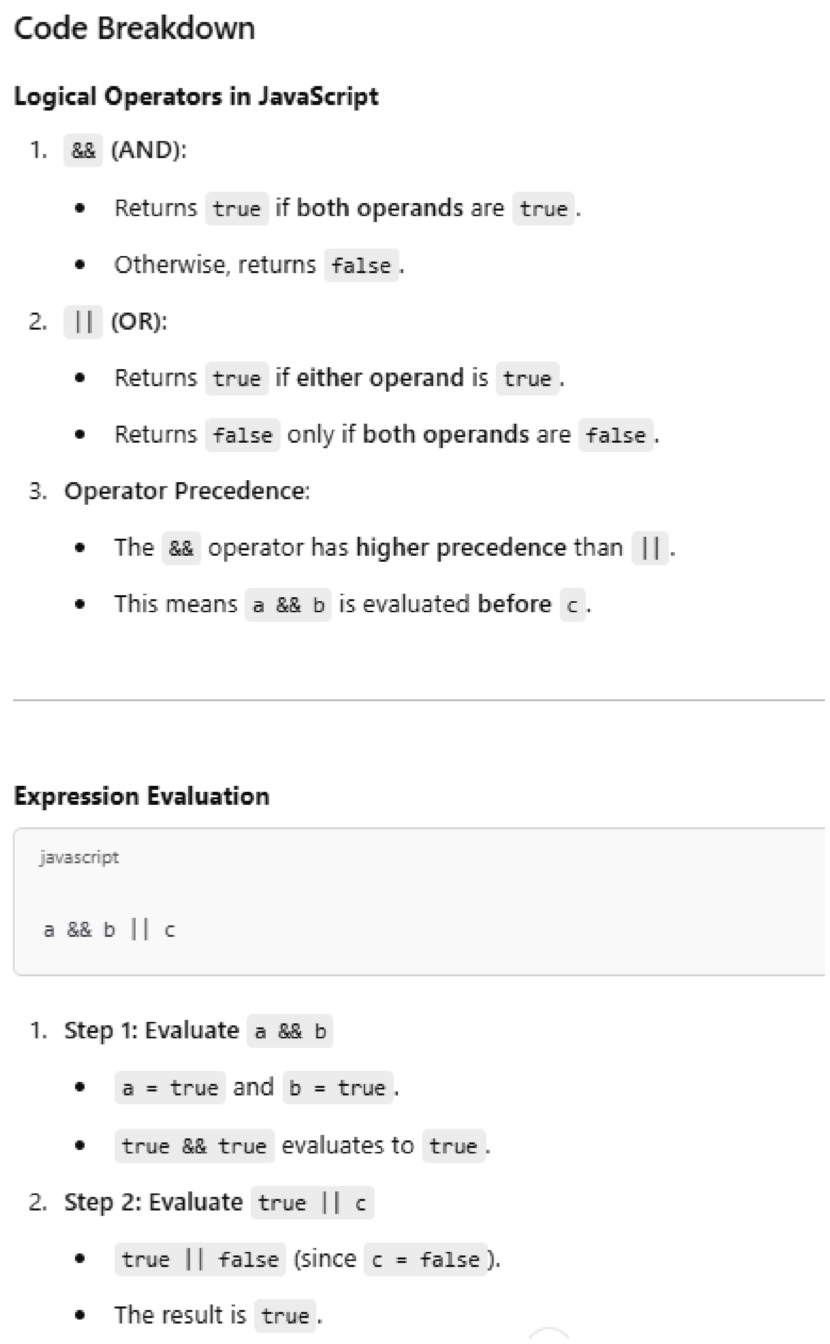
Key Points

1. **Type Coercion:**
   - JavaScript automatically converts types when using the `+` operator.
   - If one operand is a string, the other is converted to a string, and concatenation occurs.
2. **Execution Order:**
   - The `+` operator is evaluated from left to right.
   - Parentheses can change the evaluation order.
3. **Example with Parentheses:**

```javascript
console.log(2 + (3 + 4)); // Outputs: 9 (because 3 + 4 is evaluated first)
```



```
let a = true;
let b = true;
let c = false;

console.log(a && b || c);
```



Code Breakdown

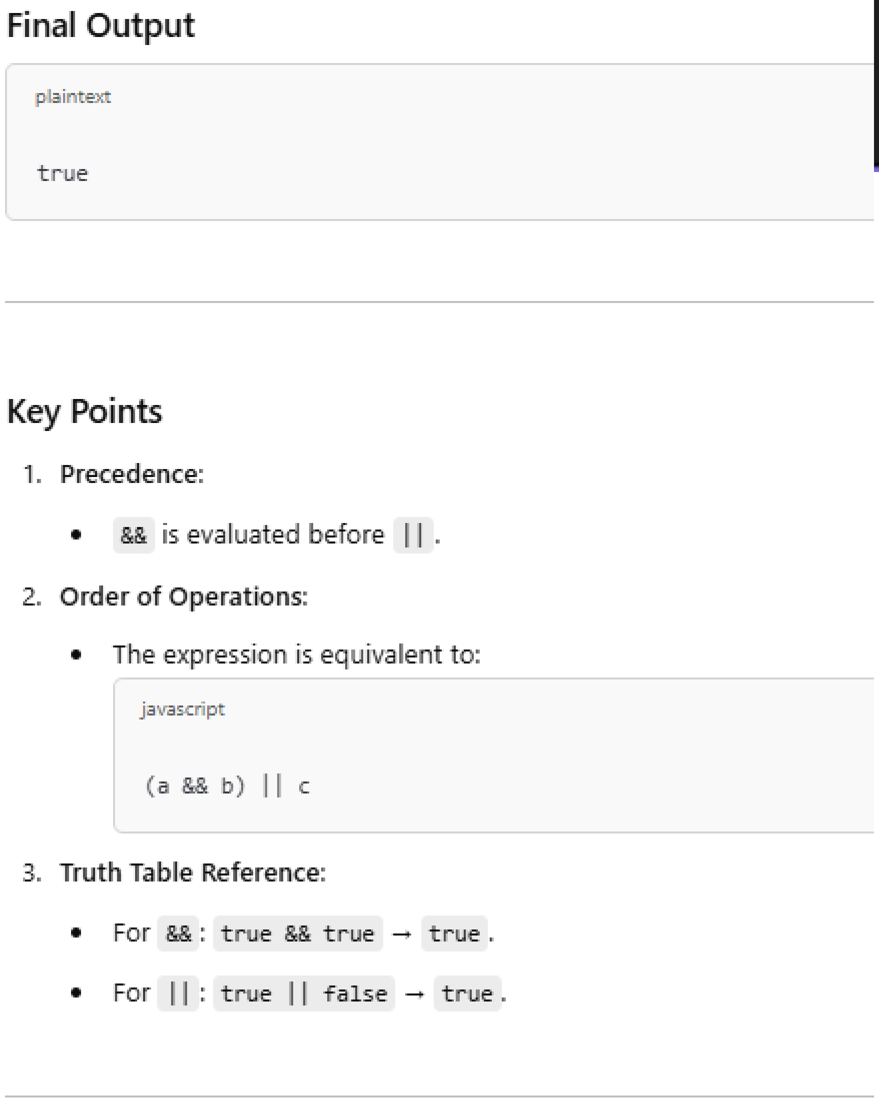
Logical Operators in JavaScript

1. `&&` (AND):
   - Returns `true` if **both operands** are `true`.
   - Otherwise, returns `false`.
2. `||` (OR):
   - Returns `true` if **either operand** is `true`.
   - Returns `false` only if **both operands** are `false`.
3. **Operator Precedence:**
   - The `&&` operator has **higher precedence** than `||`.
   - This means `a && b` is evaluated **before** `c`.

Expression Evaluation

```javascript
a && b || c
```

1. **Step 1: Evaluate `a && b`**
   - `a = true` and `b = true`.
   - `true && true` evaluates to `true`.
2. **Step 2: Evaluate `true || c`**
   - `true || false` (since `c = false`).
   - The result is `true`.



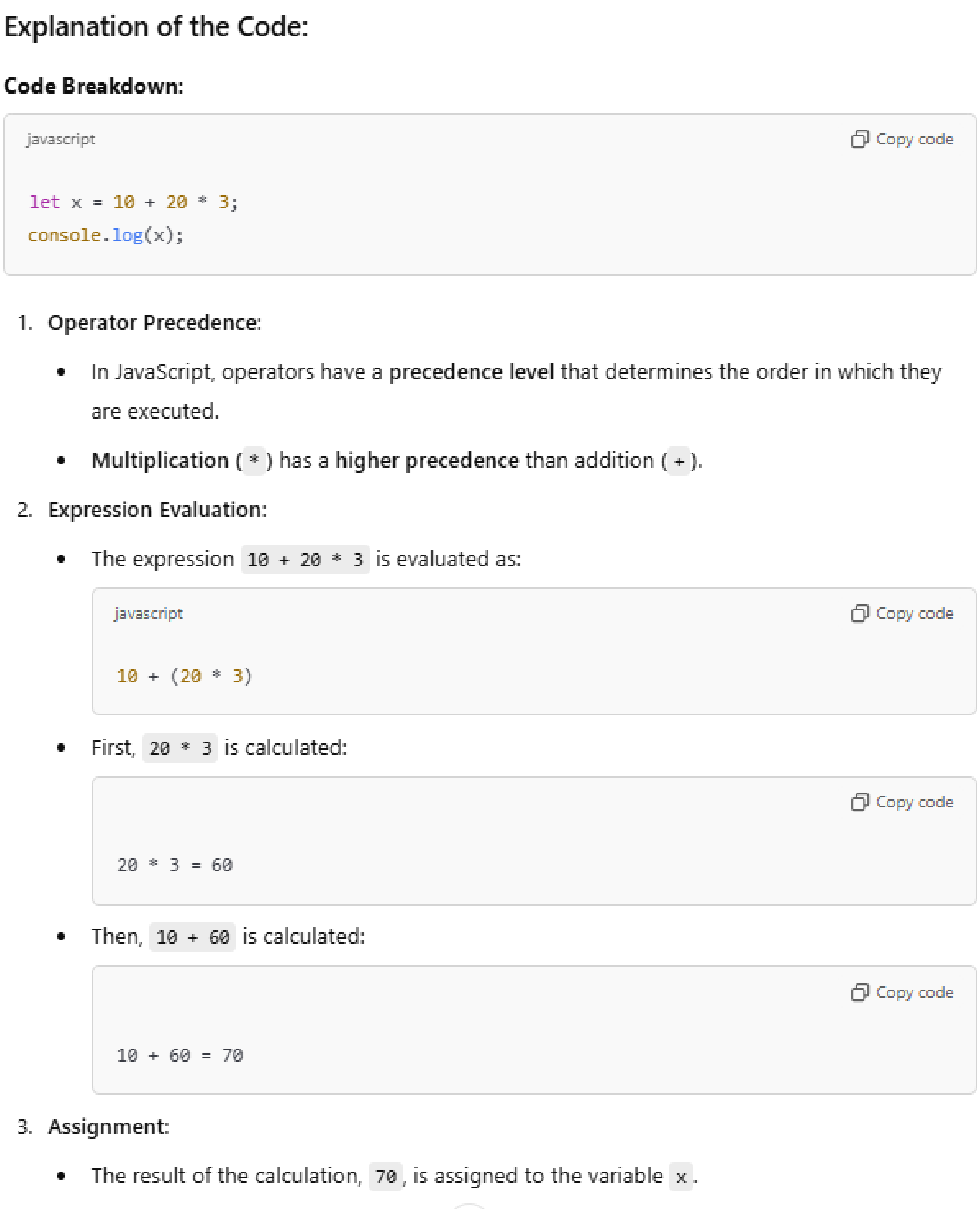
Final Output

```plaintext
true
```

Key Points

1. **Precedence:**
   - `&&` is evaluated before `||`.
2. **Order of Operations:**
   - The expression is equivalent to:

     ```javascript
     (a && b) || c
     ```
3. **Truth Table Reference:**
   - For `&&`: `true && true` → `true`.
   - For `||`: `true || false` → `true`.



Explanation of the Code:

Code Breakdown:

```javascript
let x = 10 + 20 * 3;
console.log(x);
```

1. **Operator Precedence:**
   - In JavaScript, operators have a **precedence level** that determines the order in which they are executed.
   - **Multiplication** (`*`) has a **higher precedence** than addition (`+`).
2. **Expression Evaluation:**
   - The expression `10 + 20 * 3` is evaluated as:

     ```javascript
     10 + (20 * 3)
     ```

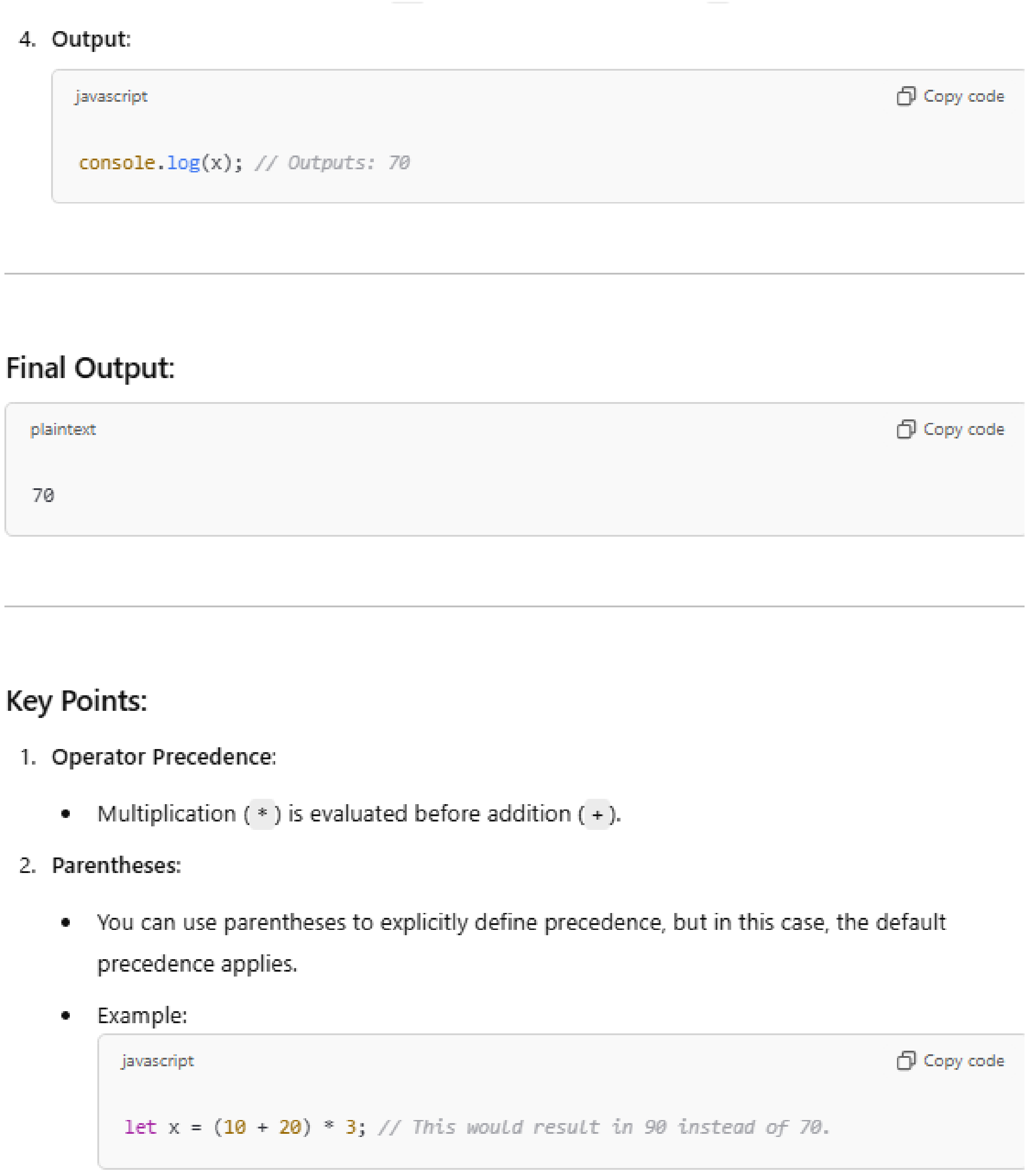
   - First, `20 * 3` is calculated:

     ```
     20 * 3 = 60
     ```

   - Then, `10 + 60` is calculated:

     ```
     10 + 60 = 70
     ```

3. **Assignment:**
   - The result of the calculation, `70`, is assigned to the variable `x`.



4. **Output:**

   ```javascript
   console.log(x); // Outputs: 70
   ```

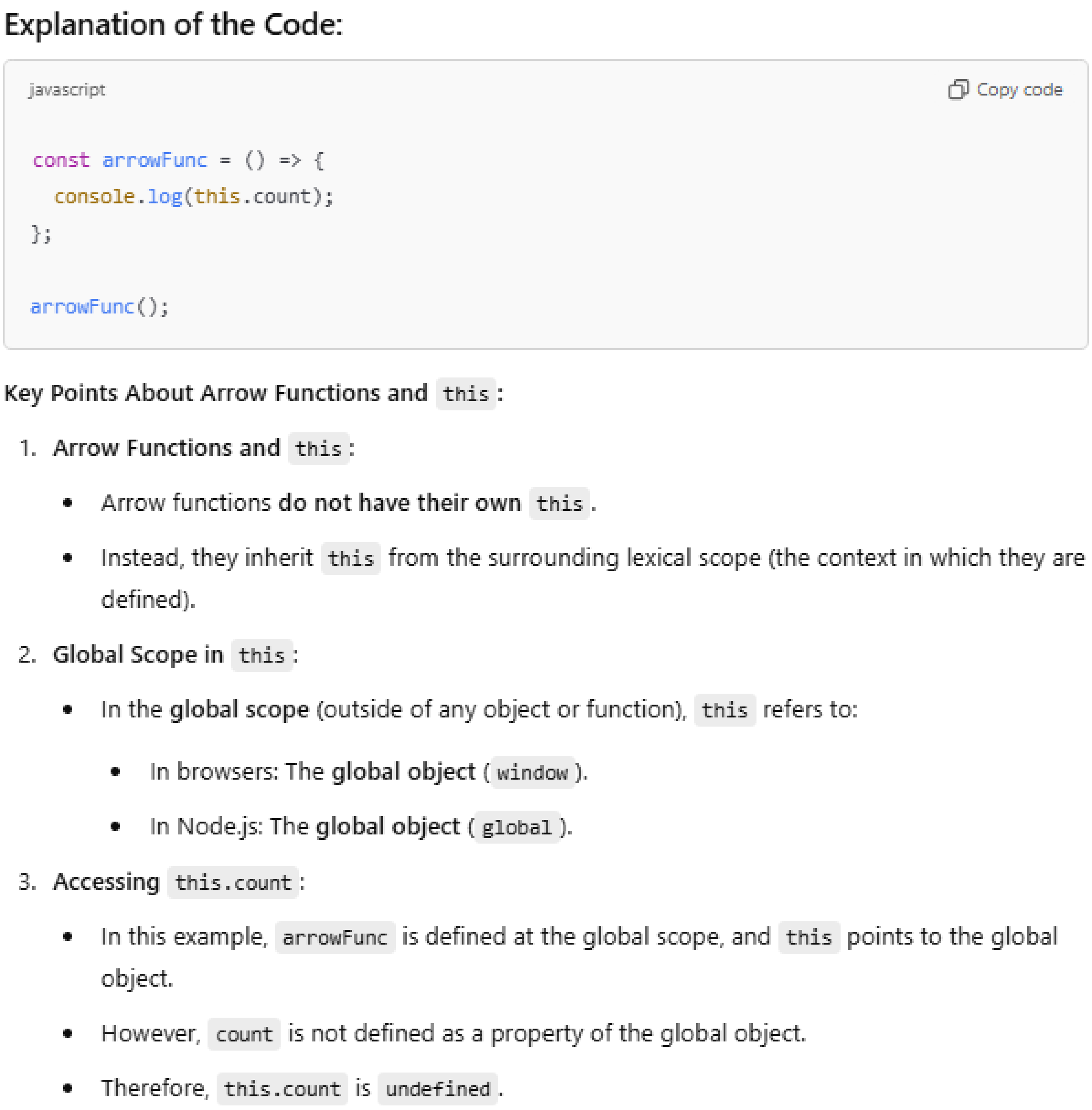
Final Output:

```plaintext
70
```

Key Points:

1. **Operator Precedence:**
   - Multiplication (`*`) is evaluated before addition (`+`).
2. **Parentheses:**
   - You can use parentheses to explicitly define precedence, but in this case, the default precedence applies.
   - Example:

     ```javascript
     let x = (10 + 20) * 3; // This would result in 90 instead of 70.
     ```

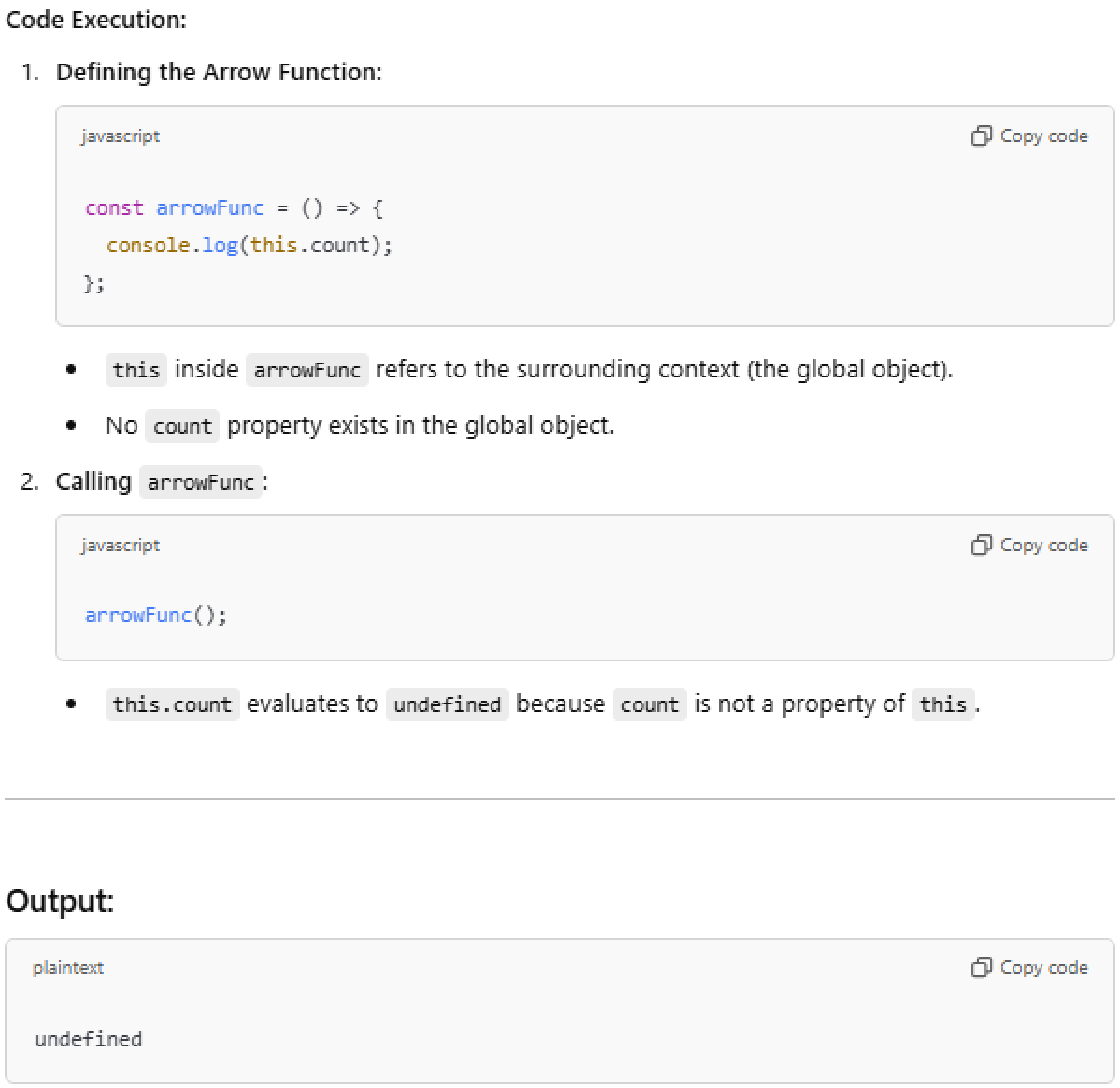


Explanation of the Code:

```javascript
const arrowFunc = () => {
  console.log(this.count);
};

arrowFunc();
```

Key Points About Arrow Functions and `this`:

1. **Arrow Functions and `this`:**
   - Arrow functions **do not have their own `this`**.
   - Instead, they inherit `this` from the surrounding lexical scope (the context in which they are defined).
2. **Global Scope in `this`:**
   - In the **global scope** (outside of any object or function), `this` refers to:
     - In browsers: The **global object** (`window`).
     - In Node.js: The **global object** (`global`).
3. **Accessing `this.count`:**
   - In this example, `arrowFunc` is defined at the global scope, and `this` points to the global object.
   - However, `count` is not defined as a property of the global object.
   - Therefore, `this.count` is `undefined`.



Code Execution:

1. **Defining the Arrow Function:**

```javascript
const arrowFunc = () => {
  console.log(this.count);
};
```

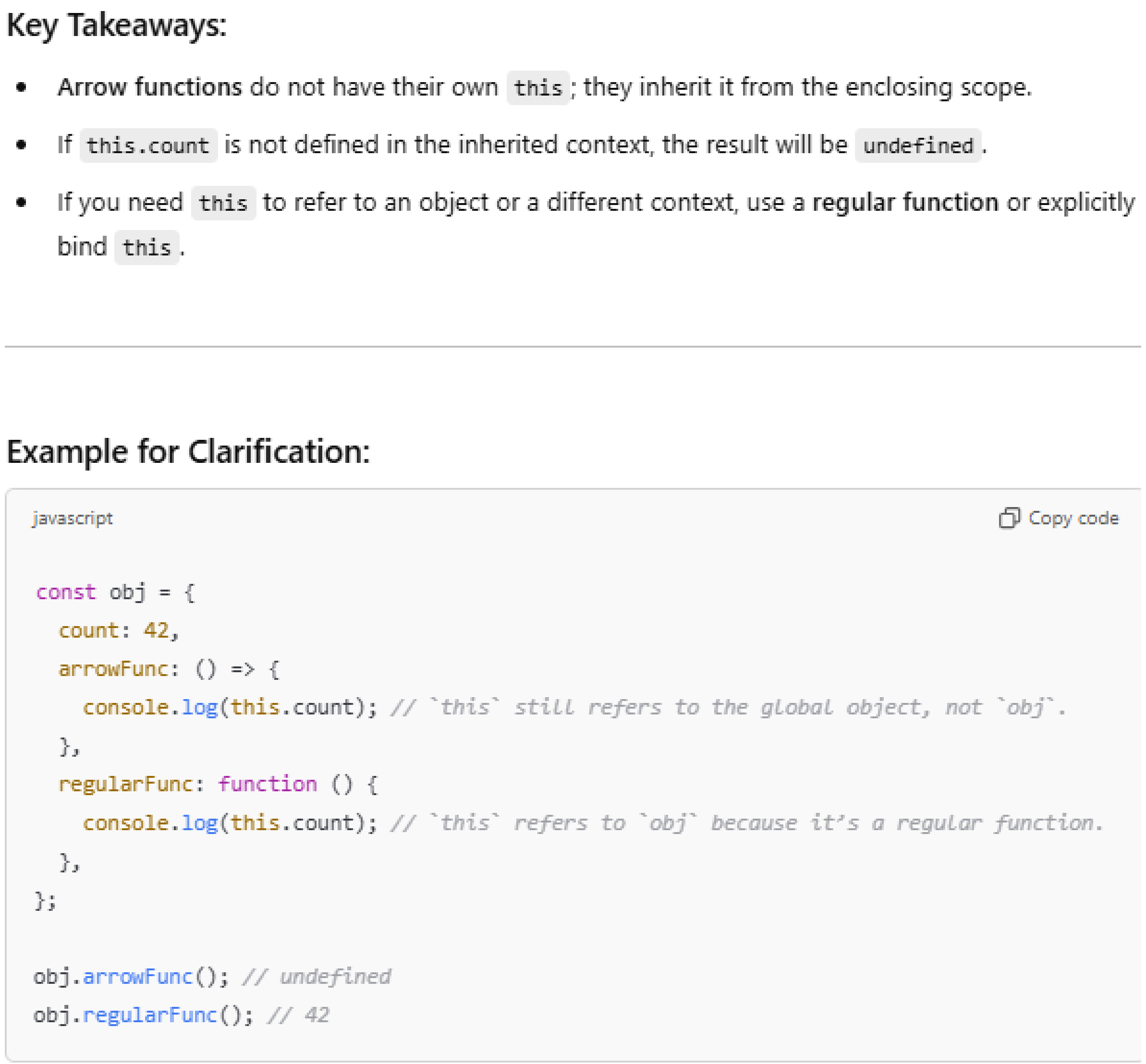
   - `this` inside `arrowFunc` refers to the surrounding context (the global object).
   - No `count` property exists in the global object.
2. **Calling `arrowFunc`:**

```javascript
arrowFunc();
```

   - `this.count` evaluates to `undefined` because `count` is not a property of `this`.

Output:

```plaintext
undefined
```

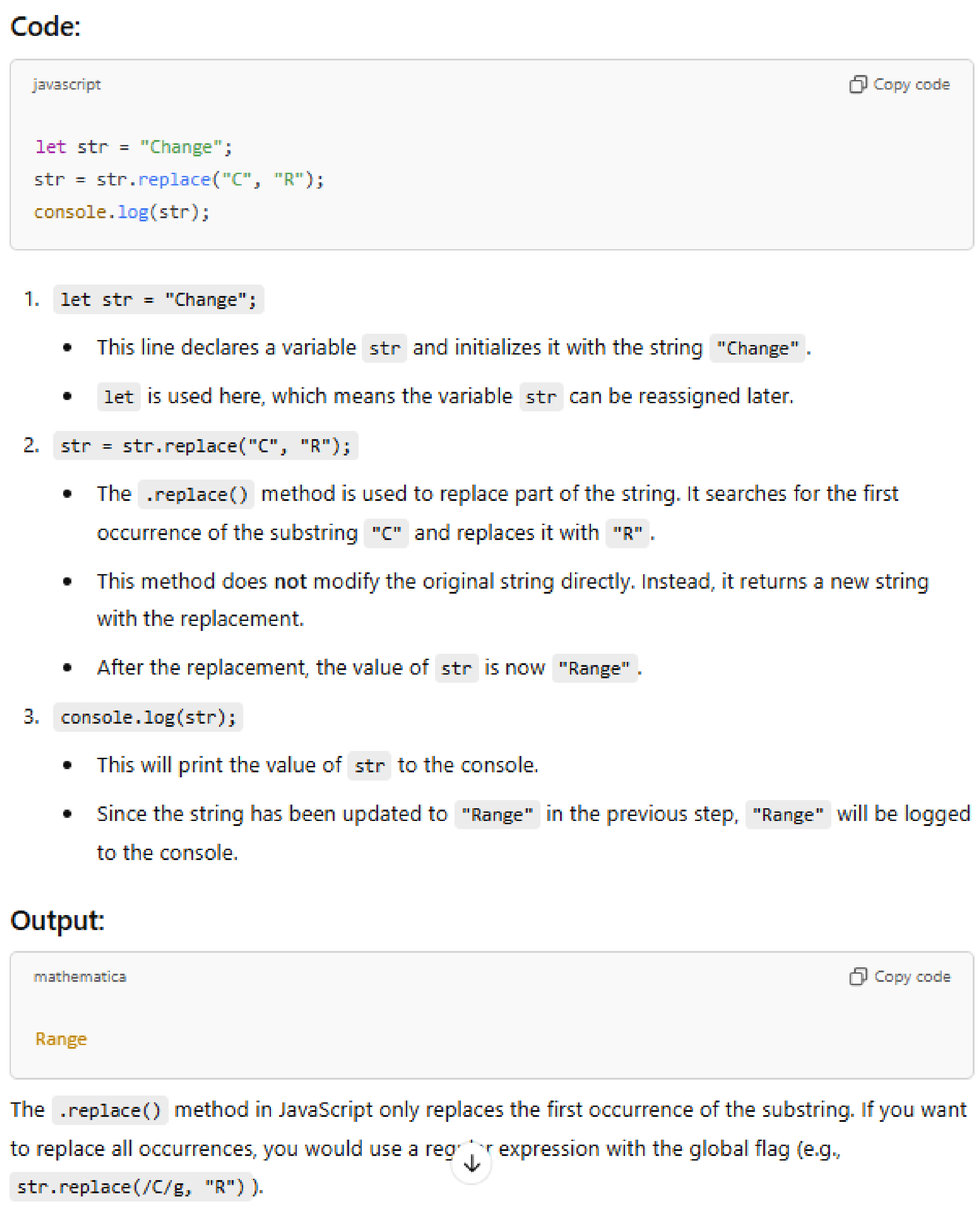


Key Takeaways:

- **Arrow functions** do not have their own `this`; they inherit it from the enclosing scope.
- If `this.count` is not defined in the inherited context, the result will be `undefined`.
- If you need `this` to refer to an object or a different context, use a **regular function** or explicitly bind `this`.

Example for Clarification:

```javascript
const obj = {
  count: 42,
  arrowFunc: () => {
    console.log(this.count); // `this` still refers to the global object, not `obj`.
  },
  regularFunc: function () {
    console.log(this.count); // `this` refers to `obj` because it’s a regular function.
  },
};

obj.arrowFunc(); // undefined
obj.regularFunc(); // 42
```



Code:

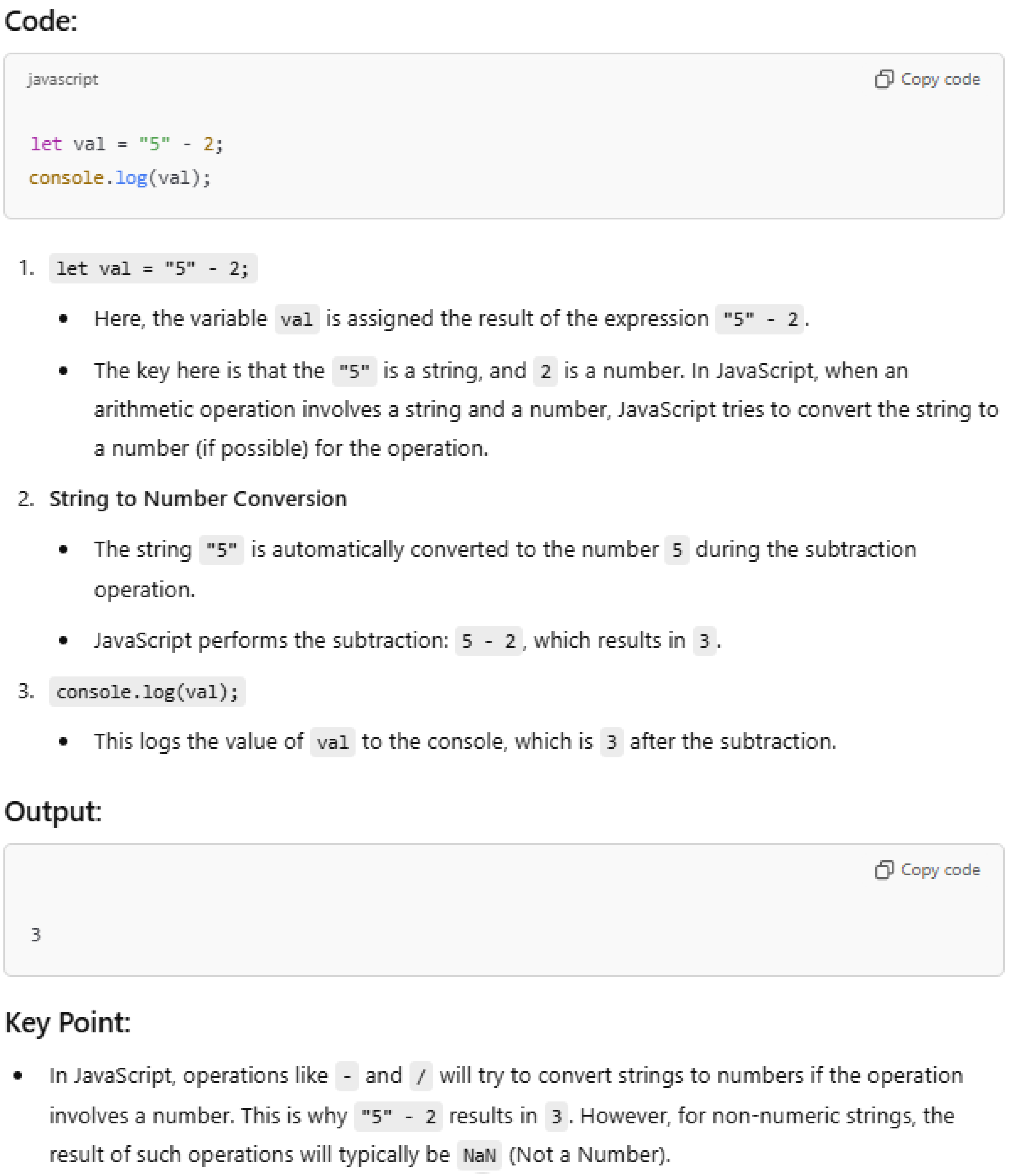
```javascript
let str = "Change";
str = str.replace("C", "R");
console.log(str);
```

1. `let str = "Change";`
   - This line declares a variable `str` and initializes it with the string `"Change"`.
   - `let` is used here, which means the variable `str` can be reassigned later.
2. `str = str.replace("C", "R");`
   - The `.replace()` method is used to replace part of the string. It searches for the first occurrence of the substring `"C"` and replaces it with `"R"`.
   - This method does **not** modify the original string directly. Instead, it returns a new string with the replacement.
   - After the replacement, the value of `str` is now `"Range"`.
3. `console.log(str);`
   - This will print the value of `str` to the console.
   - Since the string has been updated to `"Range"` in the previous step, `"Range"` will be logged to the console.

Output:

```mathematica
Range
```

The `.replace()` method in JavaScript only replaces the first occurrence of the substring. If you want to replace all occurrences, you would use a regular expression with the global flag (e.g., `str.replace(/C/g, "R")`).



Code:

```javascript
let val = "5" - 2;
console.log(val);
```

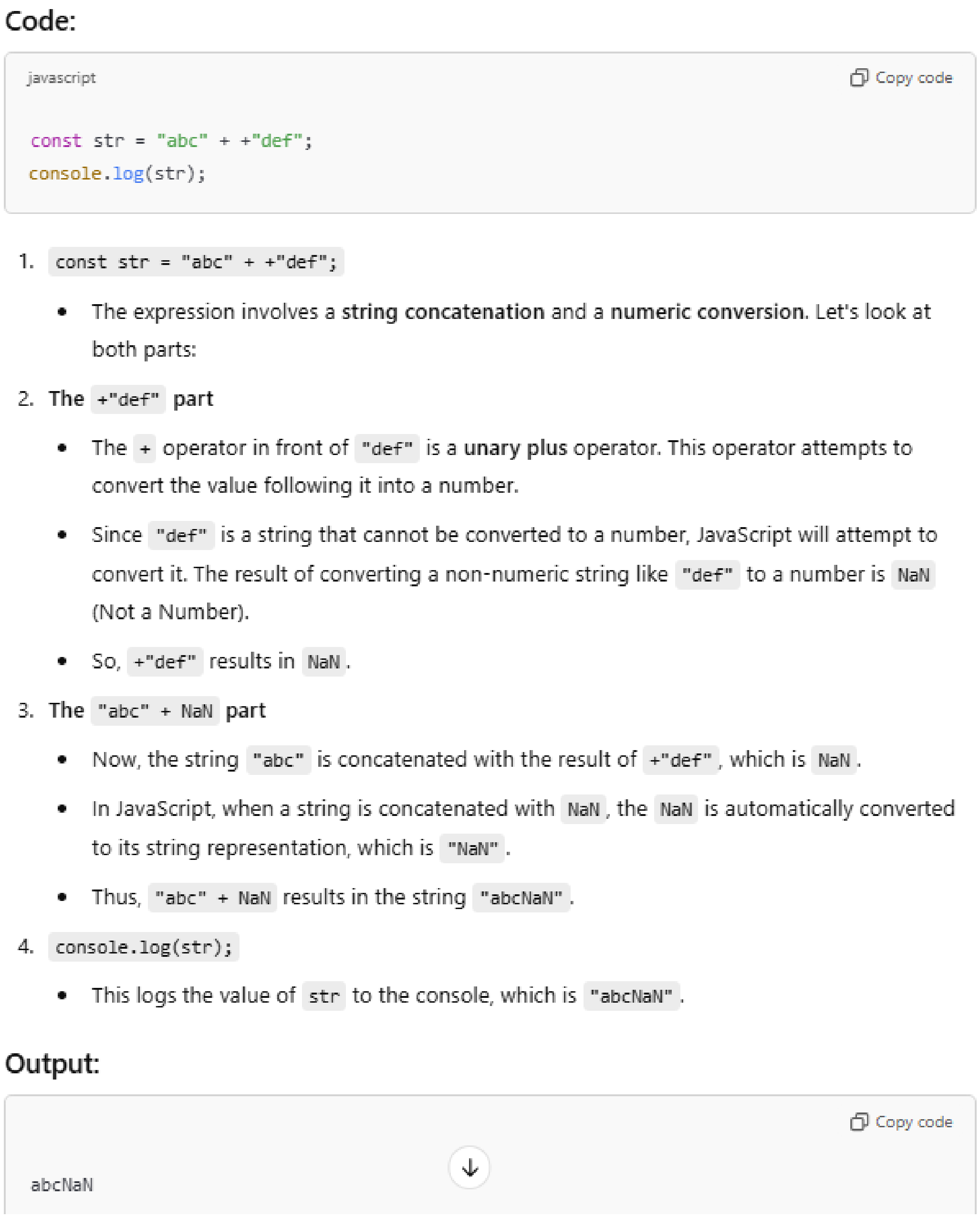
1. `let val = "5" - 2;`
   - Here, the variable `val` is assigned the result of the expression `"5" - 2`.
   - The key here is that the `"5"` is a string, and `2` is a number. In JavaScript, when an arithmetic operation involves a string and a number, JavaScript tries to convert the string to a number (if possible) for the operation.
2. **String to Number Conversion**
   - The string `"5"` is automatically converted to the number `5` during the subtraction operation.
   - JavaScript performs the subtraction: `5 - 2`, which results in `3`.
3. `console.log(val);`
   - This logs the value of `val` to the console, which is `3` after the subtraction.

Output:

```
3
```

Key Point:

- In JavaScript, operations like `-` and `/` will try to convert strings to numbers if the operation involves a number. This is why `"5" - 2` results in `3`. However, for non-numeric strings, the result of such operations will typically be `NaN` (Not a Number).



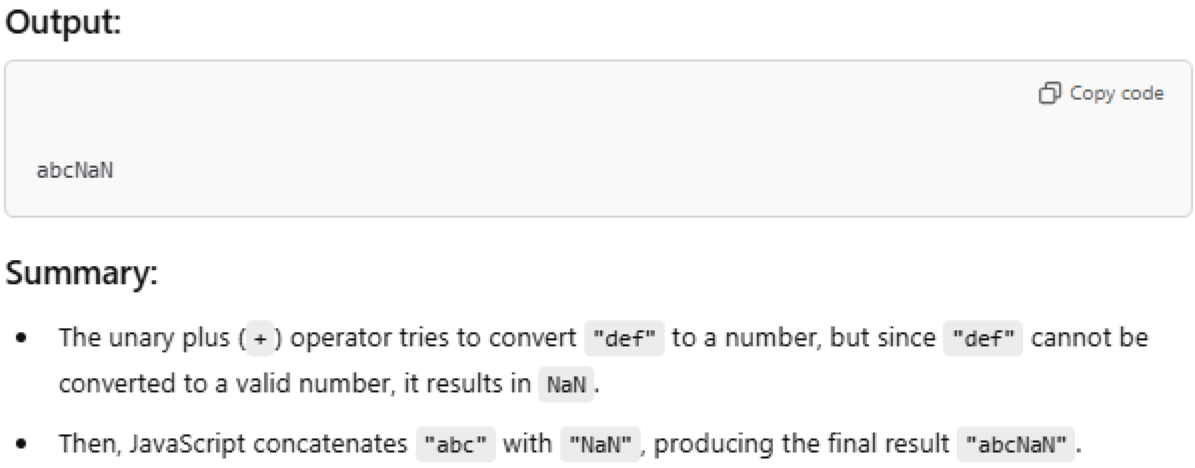
Code:

```javascript
const str = "abc" + +"def";
console.log(str);
```

1. `const str = "abc" + +"def";`
   - The expression involves a **string concatenation** and a **numeric conversion**. Let's look at both parts:
2. **The `+"def"` part**
   - The `+` operator in front of `"def"` is a **unary plus** operator. This operator attempts to convert the value following it into a number.
   - Since `"def"` is a string that cannot be converted to a number, JavaScript will attempt to convert it. The result of converting a non-numeric string like `"def"` to a number is `NaN` (Not a Number).
   - So, `+"def"` results in `NaN`.
3. **The `"abc" + NaN` part**
   - Now, the string `"abc"` is concatenated with the result of `+"def"`, which is `NaN`.
   - In JavaScript, when a string is concatenated with `NaN`, the `NaN` is automatically converted to its string representation, which is `"NaN"`.
   - Thus, `"abc" + NaN` results in the string `"abcNaN"`.
4. `console.log(str);`
   - This logs the value of `str` to the console, which is `"abcNaN"`.

Output:

```
abcNaN
```

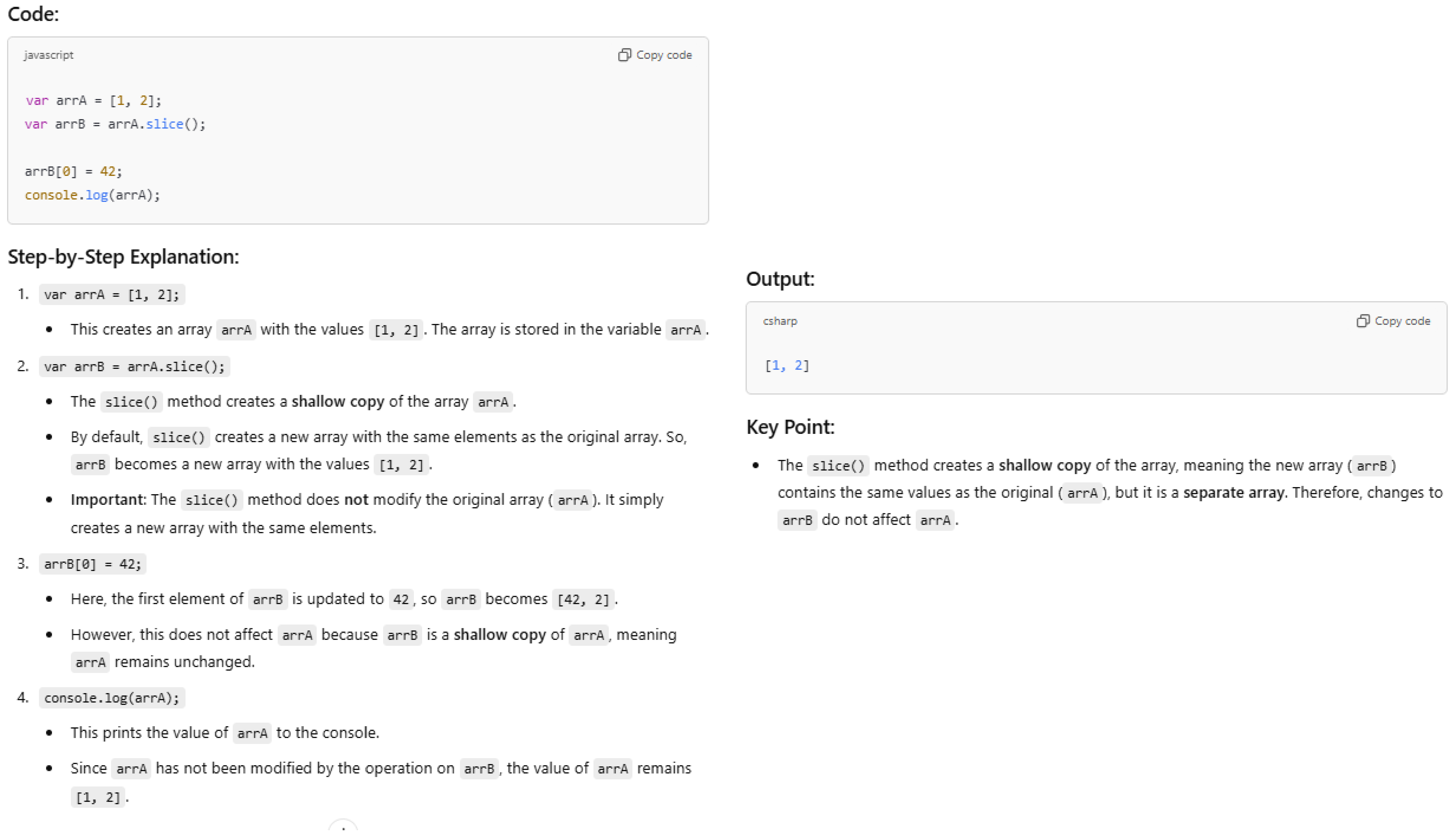


Output:

```
abcNaN
```

Summary:

- The unary plus (`+`) operator tries to convert `"def"` to a number, but since `"def"` cannot be converted to a valid number, it results in `NaN`.
- Then, JavaScript concatenates `"abc"` with `"NaN"`, producing the final result `"abcNaN"`.



Code:

```javascript
var arrA = [1, 2];
var arrB = arrA.slice();

arrB[0] = 42;
console.log(arrA);
```

Step-by-Step Explanation:

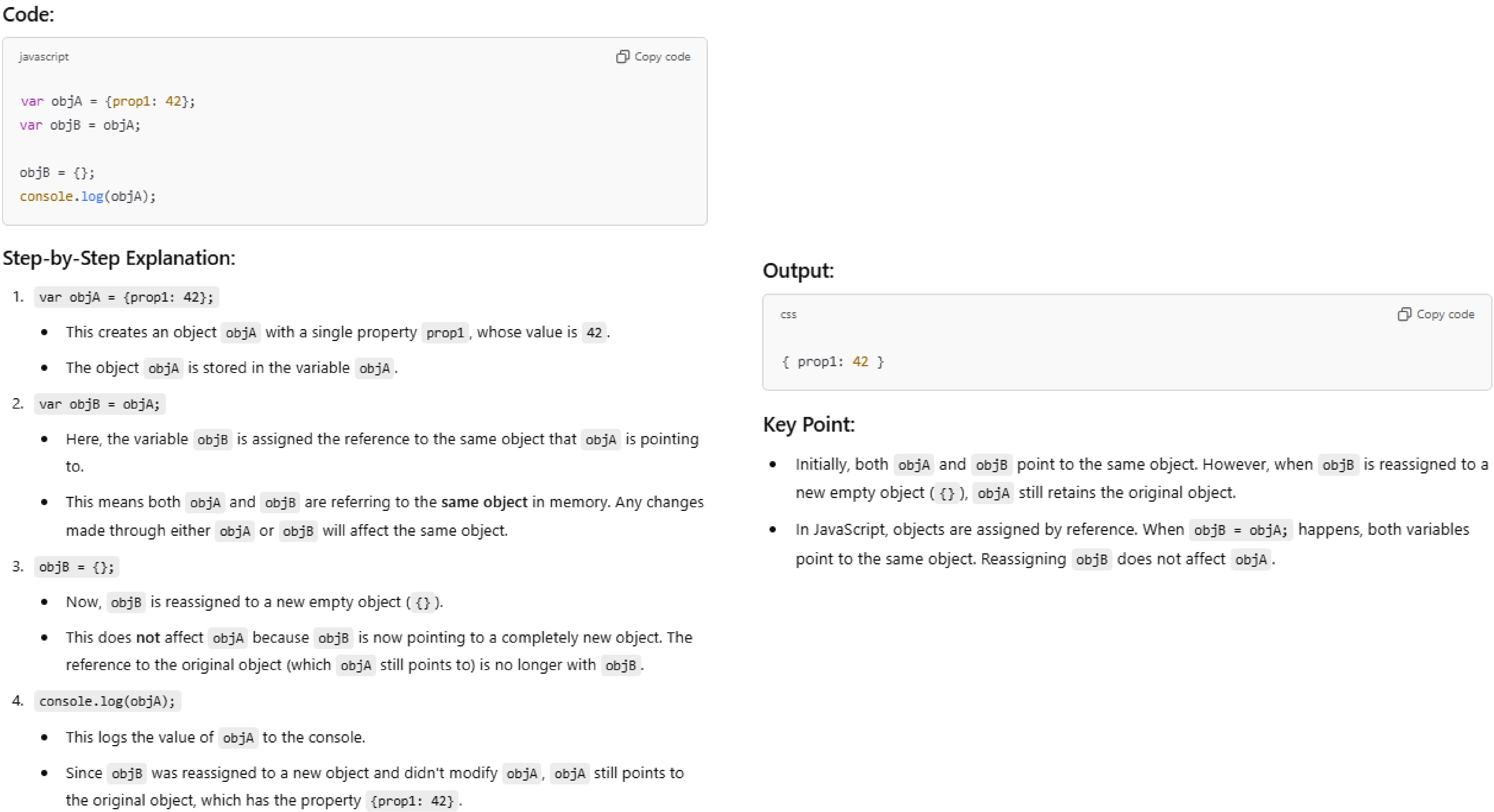
1. `var arrA = [1, 2];`
   - This creates an array `arrA` with the values `[1, 2]`. The array is stored in the variable `arrA`.
2. `var arrB = arrA.slice();`
   - The `slice()` method creates a **shallow copy** of the array `arrA`.
   - By default, `slice()` creates a new array with the same elements as the original array. So, `arrB` becomes a new array with the values `[1, 2]`.
   - **Important**: The `slice()` method does **not** modify the original array (`arrA`). It simply creates a new array with the same elements.
3. `arrB[0] = 42;`
   - Here, the first element of `arrB` is updated to `42`, so `arrB` becomes `[42, 2]`.
   - However, this does not affect `arrA` because `arrB` is a **shallow copy** of `arrA`, meaning `arrA` remains unchanged.
4. `console.log(arrA);`
   - This prints the value of `arrA` to the console.
   - Since `arrA` has not been modified by the operation on `arrB`, the value of `arrA` remains `[1, 2]`.

Output:

```csharp
[1, 2]
```

Key Point:

- The `slice()` method creates a **shallow copy** of the array, meaning the new array (`arrB`) contains the same values as the original (`arrA`), but it is a **separate array**. Therefore, changes to `arrB` do not affect `arrA`.



Code:

```javascript
var objA = {prop1: 42};
var objB = objA;

objB = {};
console.log(objA);
```

Step-by-Step Explanation:

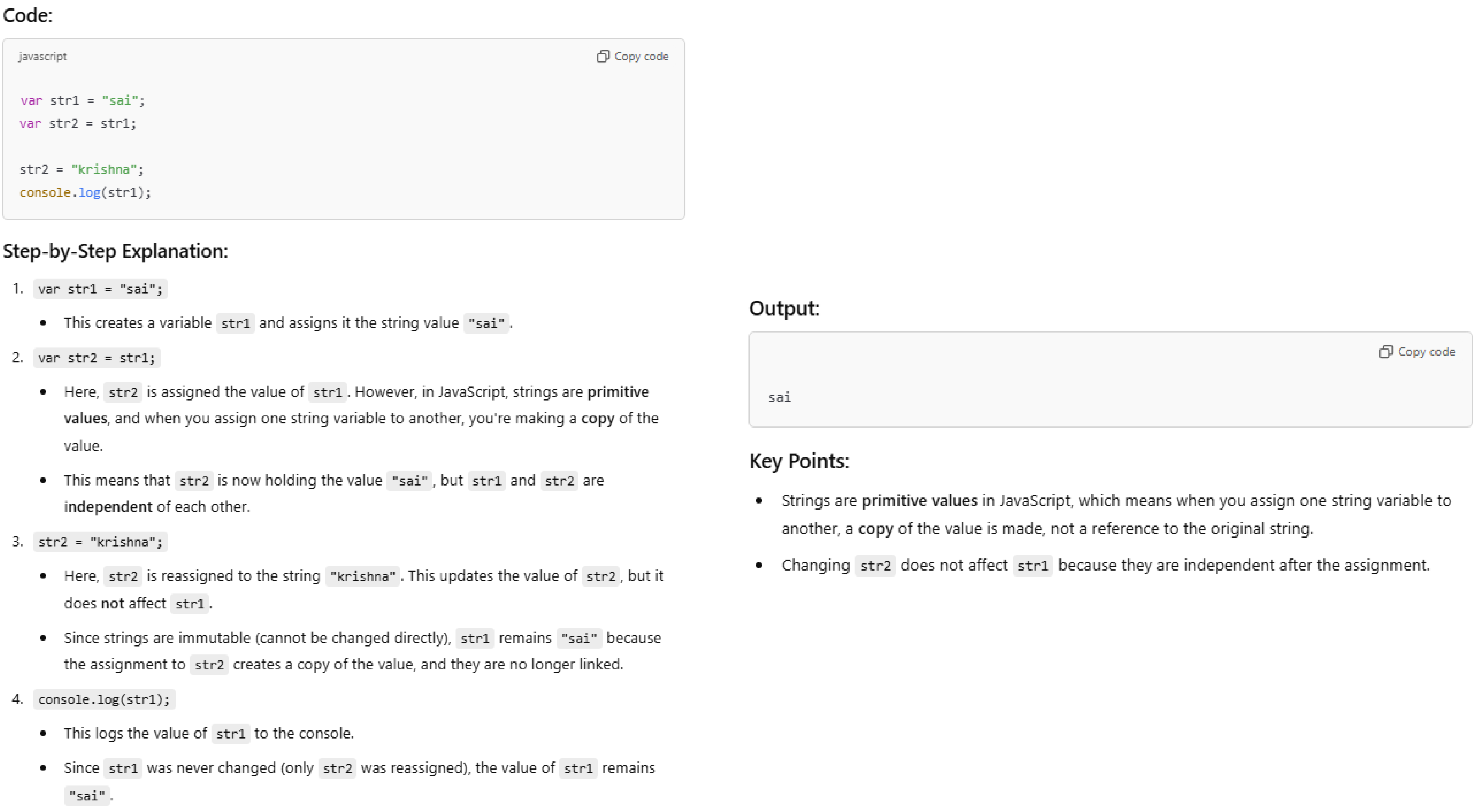
1. `var objA = {prop1: 42};`
   - This creates an object `objA` with a single property `prop1`, whose value is `42`.
   - The object `objA` is stored in the variable `objA`.
2. `var objB = objA;`
   - Here, the variable `objB` is assigned the reference to the same object that `objA` is pointing to.
   - This means both `objA` and `objB` are referring to the **same object** in memory. Any changes made through either `objA` or `objB` will affect the same object.
3. `objB = {};`
   - Now, `objB` is reassigned to a new empty object (`{}`).
   - This does **not** affect `objA` because `objB` is now pointing to a completely new object. The reference to the original object (which `objA` still points to) is no longer with `objB`.
4. `console.log(objA);`
   - This logs the value of `objA` to the console.
   - Since `objB` was reassigned to a new object and didn't modify `objA`, `objA` still points to the original object, which has the property `{prop1: 42}`.

Output:

```css
{ prop1: 42 }
```

Key Point:

- Initially, both `objA` and `objB` point to the same object. However, when `objB` is reassigned to a new empty object (`{}`), `objA` still retains the original object.
- In JavaScript, objects are assigned by reference. When `objB = objA;` happens, both variables point to the same object. Reassigning `objB` does not affect `objA`.



Code:

```javascript
var str1 = "sai";
var str2 = str1;

str2 = "krishna";
console.log(str1);
```

Step-by-Step Explanation:

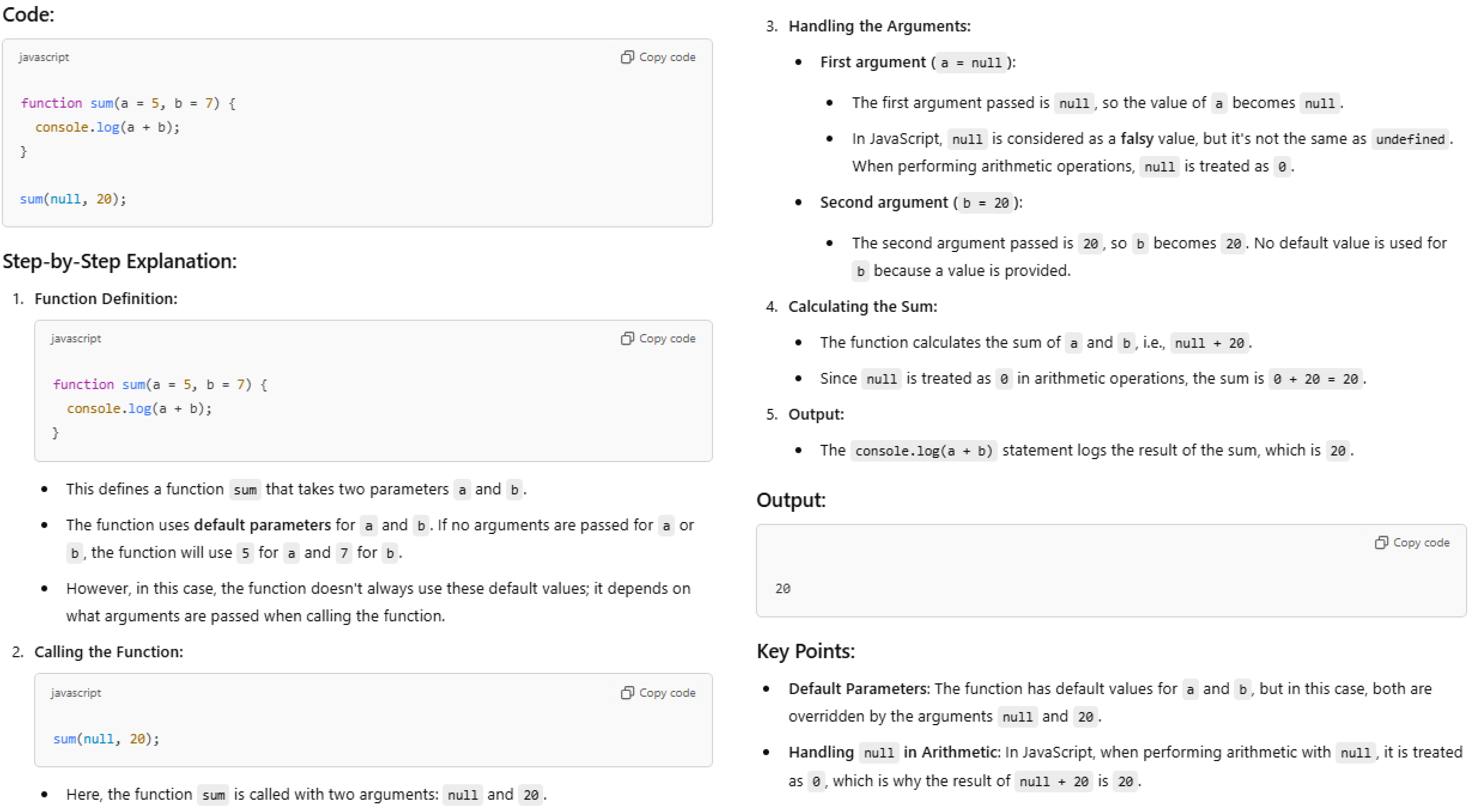
1. `var str1 = "sai";`
   - This creates a variable `str1` and assigns it the string value `"sai"`.
2. `var str2 = str1;`
   - Here, `str2` is assigned the value of `str1`. However, in JavaScript, strings are **primitive values**, and when you assign one string variable to another, you're making a **copy** of the value.
   - This means that `str2` is now holding the value `"sai"`, but `str1` and `str2` are **independent** of each other.
3. `str2 = "krishna";`
   - Here, `str2` is reassigned to the string `"krishna"`. This updates the value of `str2`, but it does **not** affect `str1`.
   - Since strings are immutable (cannot be changed directly), `str1` remains `"sai"` because the assignment to `str2` creates a copy of the value, and they are no longer linked.
4. `console.log(str1);`
   - This logs the value of `str1` to the console.
   - Since `str1` was never changed (only `str2` was reassigned), the value of `str1` remains `"sai"`.

Output:

```
sai
```

Key Points:

- Strings are **primitive values** in JavaScript, which means when you assign one string variable to another, a **copy** of the value is made, not a reference to the original string.
- Changing `str2` does not affect `str1` because they are independent after the assignment.



Code:

```javascript
function sum(a = 5, b = 7) {
  console.log(a + b);
}

sum(null, 20);
```

Step-by-Step Explanation:

1. **Function Definition:**

   ```javascript
   function sum(a = 5, b = 7) {
     console.log(a + b);
   }
   ```

   - This defines a function `sum` that takes two parameters `a` and `b`.
   - The function uses **default parameters** for `a` and `b`. If no arguments are passed for `a` or `b`, the function will use `5` for `a` and `7` for `b`.
   - However, in this case, the function doesn't always use these default values; it depends on what arguments are passed when calling the function.

2. **Calling the Function:**

   ```javascript
   sum(null, 20);
   ```

   - Here, the function `sum` is called with two arguments: `null` and `20`.

3. **Handling the Arguments:**
   - **First argument (`a = null`):**
     - The first argument passed is `null`, so the value of `a` becomes `null`.
     - In JavaScript, `null` is considered as a **falsy** value, but it's not the same as `undefined`. When performing arithmetic operations, `null` is treated as `0`.
   - **Second argument (`b = 20`):**
     - The second argument passed is `20`, so `b` becomes `20`. No default value is used for `b` because a value is provided.

4. **Calculating the Sum:**
   - The function calculates the sum of `a` and `b`, i.e., `null + 20`.
   - Since `null` is treated as `0` in arithmetic operations, the sum is `0 + 20 = 20`.

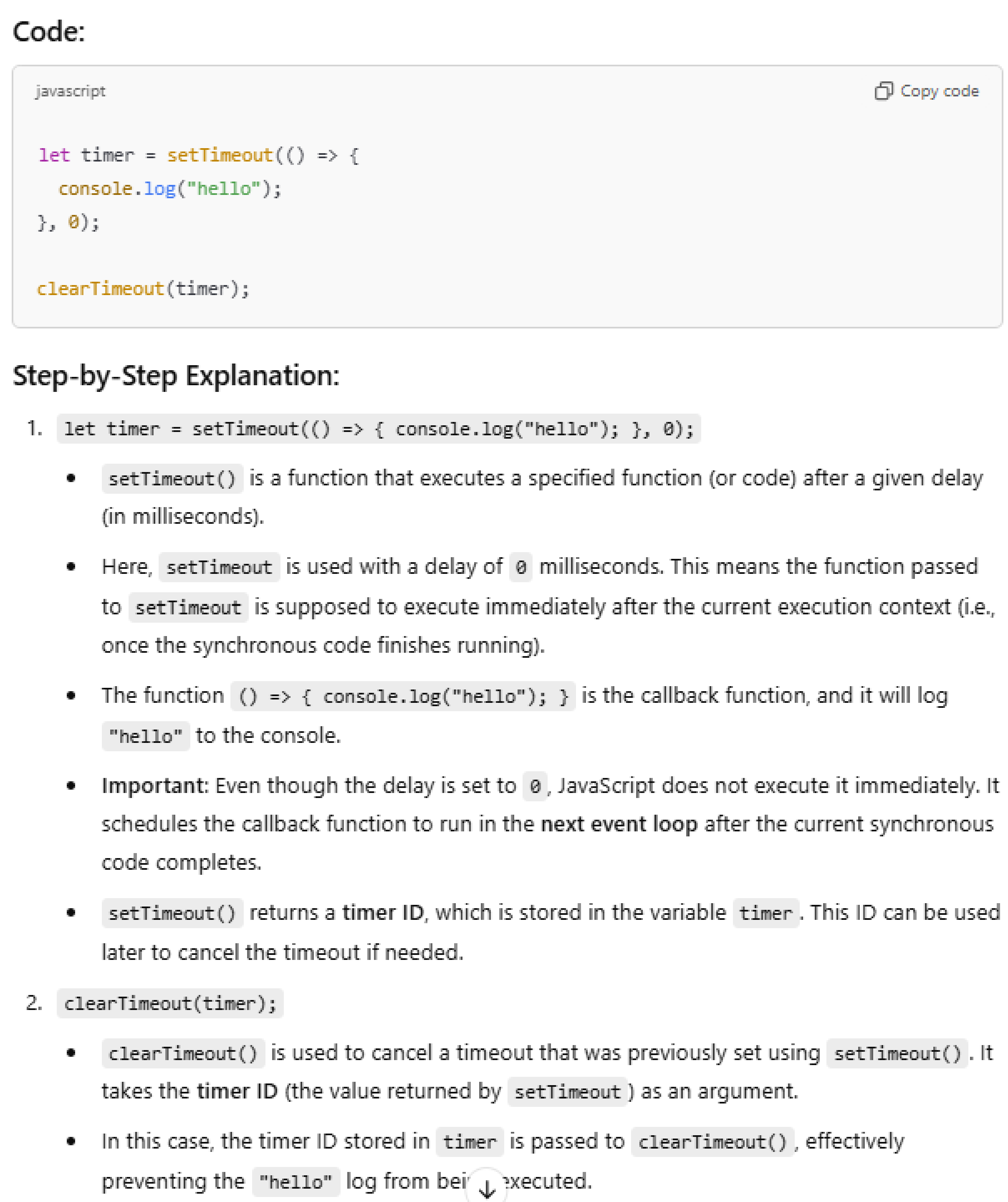
5. **Output:**
   - The `console.log(a + b)` statement logs the result of the sum, which is `20`.

Output:

```
20
```

Key Points:

- **Default Parameters:** The function has default values for `a` and `b`, but in this case, both are overridden by the arguments `null` and `20`.
- **Handling `null` in Arithmetic:** In JavaScript, when performing arithmetic with `null`, it is treated as `0`, which is why the result of `null + 20` is `20`.

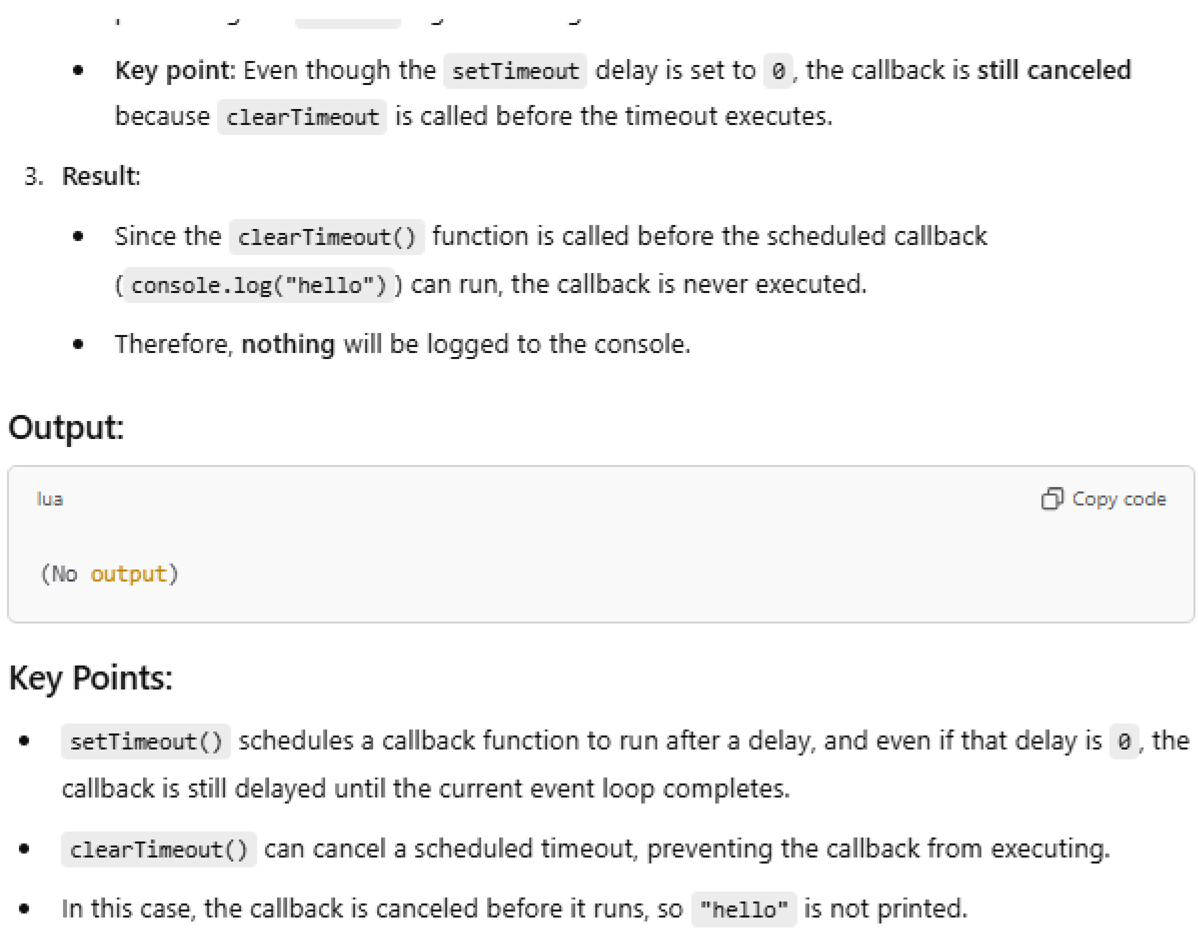


Code:

```javascript
let timer = setTimeout(() => {
  console.log("hello");
}, 0);

clearTimeout(timer);
```

Step-by-Step Explanation:

1. `let timer = setTimeout(() => { console.log("hello"); }, 0);`
   - `setTimeout()` is a function that executes a specified function (or code) after a given delay (in milliseconds).
   - Here, `setTimeout` is used with a delay of `0` milliseconds. This means the function passed to `setTimeout` is supposed to execute immediately after the current execution context (i.e., once the synchronous code finishes running).
   - The function `() => { console.log("hello"); }` is the callback function, and it will log `"hello"` to the console.
   - **Important**: Even though the delay is set to `0`, JavaScript does not execute it immediately. It schedules the callback function to run in the **next event loop** after the current synchronous code completes.
   - `setTimeout()` returns a **timer ID**, which is stored in the variable `timer`. This ID can be used later to cancel the timeout if needed.
2. `clearTimeout(timer);`
   - `clearTimeout()` is used to cancel a timeout that was previously set using `setTimeout()`. It takes the **timer ID** (the value returned by `setTimeout`) as an argument.
   - In this case, the timer ID stored in `timer` is passed to `clearTimeout()`, effectively preventing the `"hello"` log from bei xecuted.



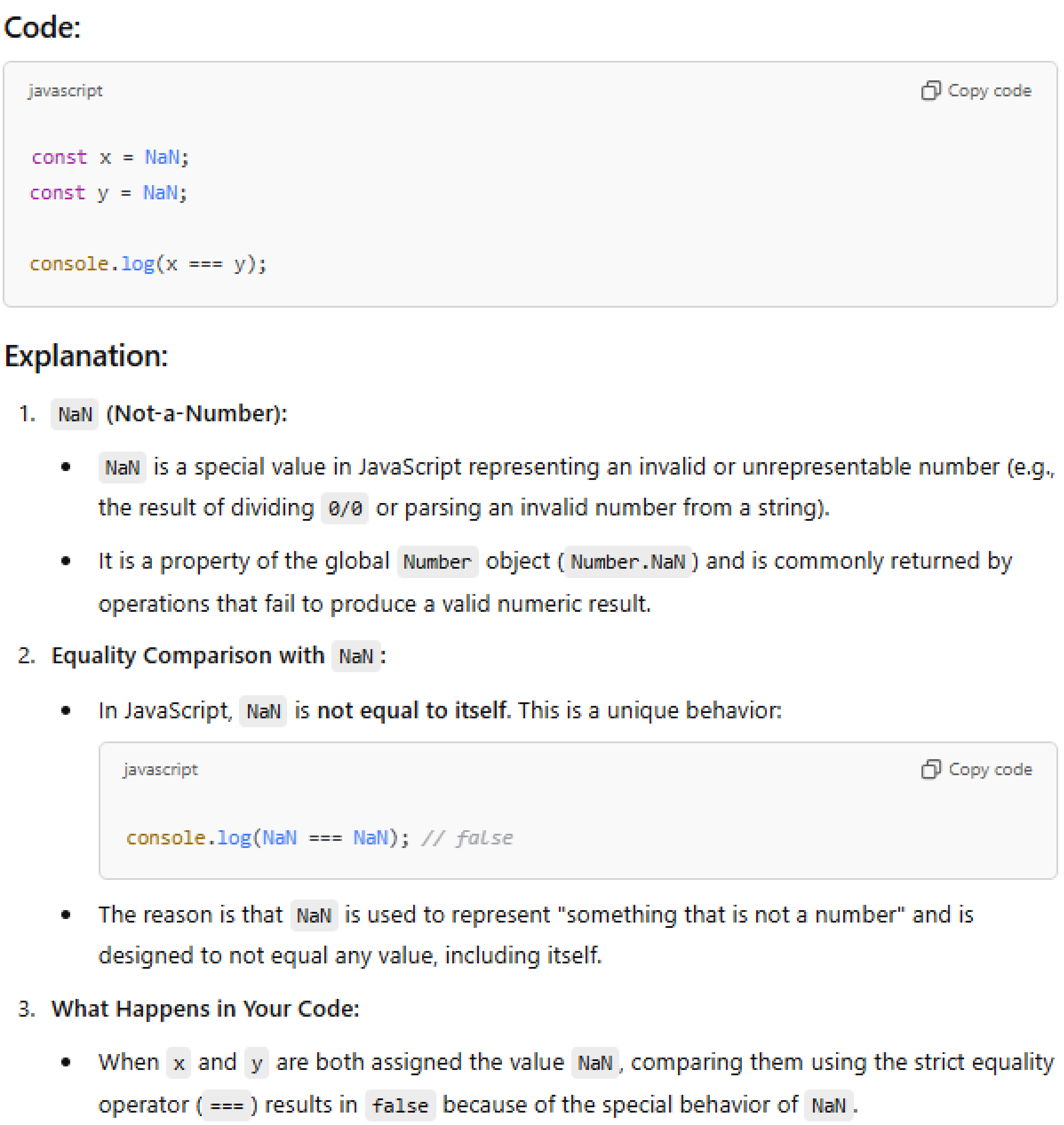
   - **Key point**: Even though the `setTimeout` delay is set to `0`, the callback is **still canceled** because `clearTimeout` is called before the timeout executes.
3. **Result**:
   - Since the `clearTimeout()` function is called before the scheduled callback (`console.log("hello")`) can run, the callback is never executed.
   - Therefore, **nothing** will be logged to the console.

Output:

```lua
(No output)
```

Key Points:

- `setTimeout()` schedules a callback function to run after a delay, and even if that delay is `0`, the callback is still delayed until the current event loop completes.
- `clearTimeout()` can cancel a scheduled timeout, preventing the callback from executing.
- In this case, the callback is canceled before it runs, so `"hello"` is not printed.



Code:

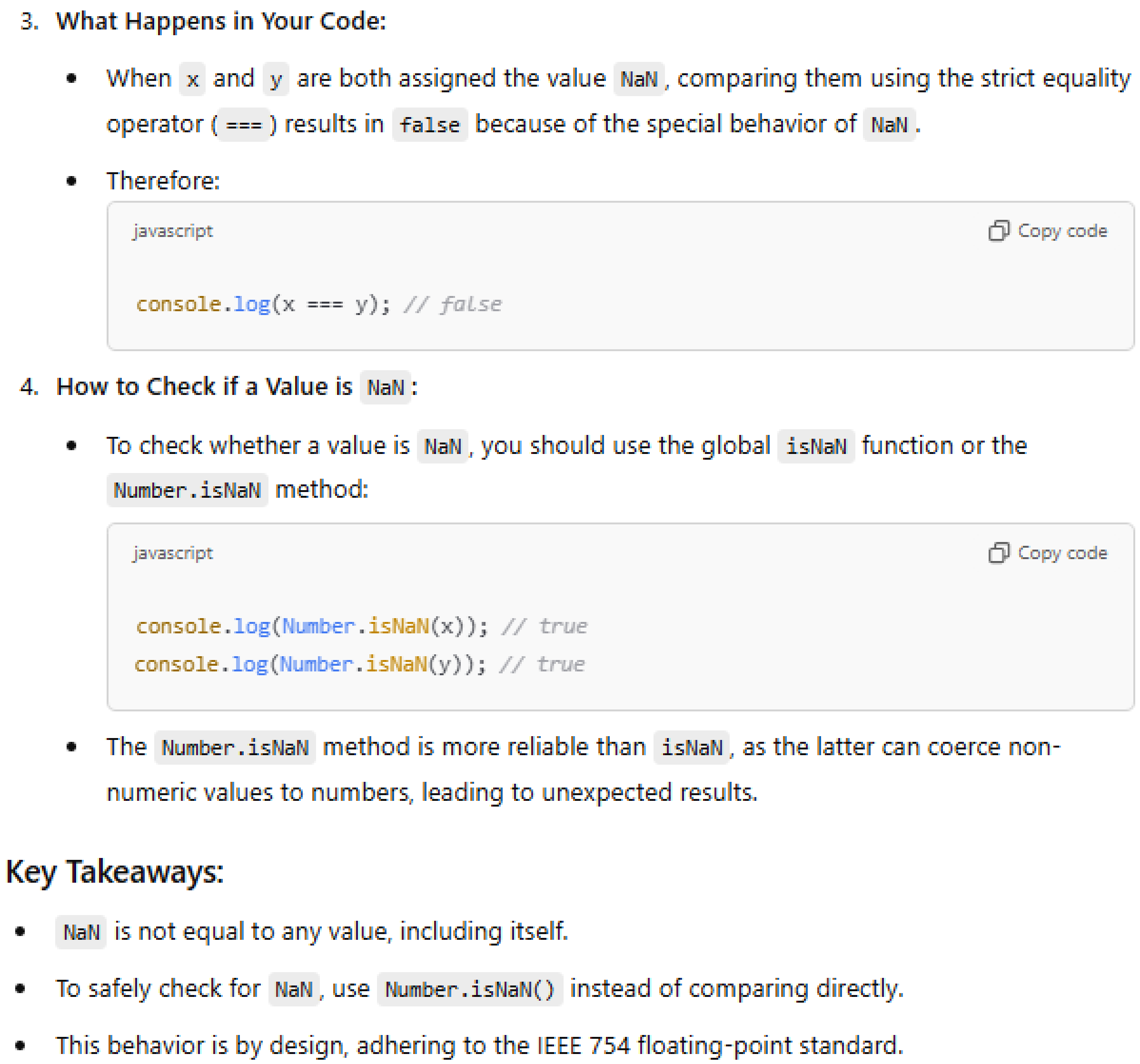
```javascript
const x = NaN;
const y = NaN;

console.log(x === y);
```

Explanation:

1. **`NaN` (Not-a-Number):**
   - `NaN` is a special value in JavaScript representing an invalid or unrepresentable number (e.g., the result of dividing `0/0` or parsing an invalid number from a string).
   - It is a property of the global `Number` object (`Number.NaN`) and is commonly returned by operations that fail to produce a valid numeric result.
2. **Equality Comparison with `NaN`:**
   - In JavaScript, `NaN` is **not equal to itself.** This is a unique behavior:

```javascript
console.log(NaN === NaN); // false
```

   - The reason is that `NaN` is used to represent "something that is not a number" and is designed to not equal any value, including itself.
3. **What Happens in Your Code:**
   - When `x` and `y` are both assigned the value `NaN`, comparing them using the strict equality operator (`===`) results in `false` because of the special behavior of `NaN`.



3. **What Happens in Your Code:**
   - When `x` and `y` are both assigned the value `NaN`, comparing them using the strict equality operator (`===`) results in `false` because of the special behavior of `NaN`.

   - Therefore:

```javascript
console.log(x === y); // false
```

4. **How to Check if a Value is `NaN`:**
   - To check whether a value is `NaN`, you should use the global `isNaN` function or the `Number.isNaN` method:

```javascript
console.log(Number.isNaN(x)); // true
console.log(Number.isNaN(y)); // true
```

   - The `Number.isNaN` method is more reliable than `isNaN`, as the latter can coerce non-numeric values to numbers, leading to unexpected results.

Key Takeaways:

- `NaN` is not equal to any value, including itself.
- To safely check for `NaN`, use `Number.isNaN()` instead of comparing directly.
- This behavior is by design, adhering to the IEEE 754 floating-point standard.

Let's analyze the code step by step:

Code:

```javascript
let person = { name: 'sai' };
const members = [person];
person = null;

console.log(members);
```

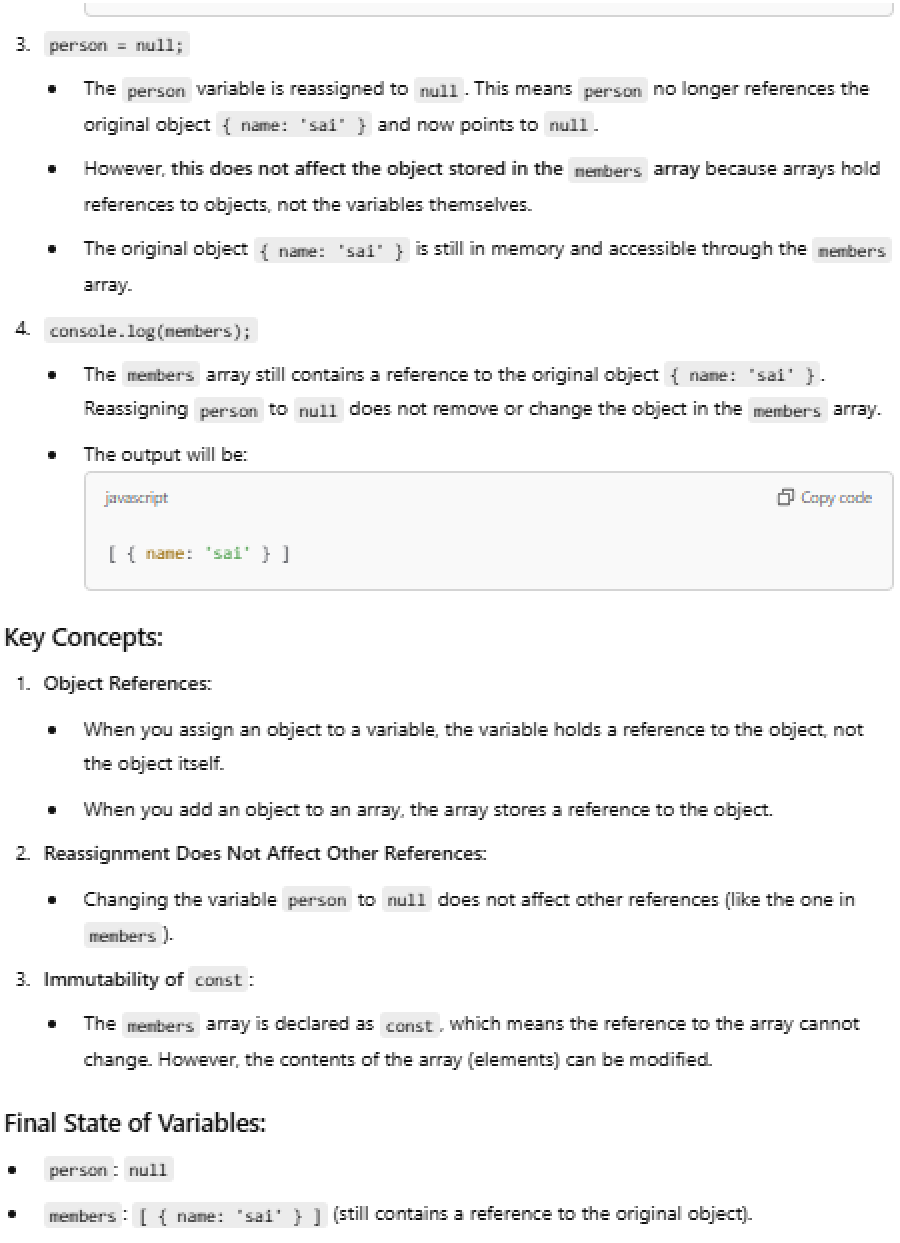
Step-by-Step Explanation:

1. `let person = { name: 'sai' };`
   - A variable `person` is declared and assigned an object with a property `name` having the value `'sai'`.
   - At this point:

```javascript
person = { name: 'sai' };
```

2. `const members = [person];`
   - A constant array `members` is declared and initialized with `person` as its only element. Since `person` references the object `{ name: 'sai' }`, the array now contains a reference to this object.
   - `members` looks like this:

```javascript
members = [ { name: 'sai' } ];
```



3. `person = null;`
   - The `person` variable is reassigned to `null`. This means `person` no longer references the original object `{ name: 'sai' }` and now points to `null`.
   - However, this does not affect the object stored in the `members` array because arrays hold references to objects, not the variables themselves.
   - The original object `{ name: 'sai' }` is still in memory and accessible through the `members` array.
4. `console.log(members);`
   - The `members` array still contains a reference to the original object `{ name: 'sai' }`. Reassigning `person` to `null` does not remove or change the object in the `members` array.
   - The output will be:

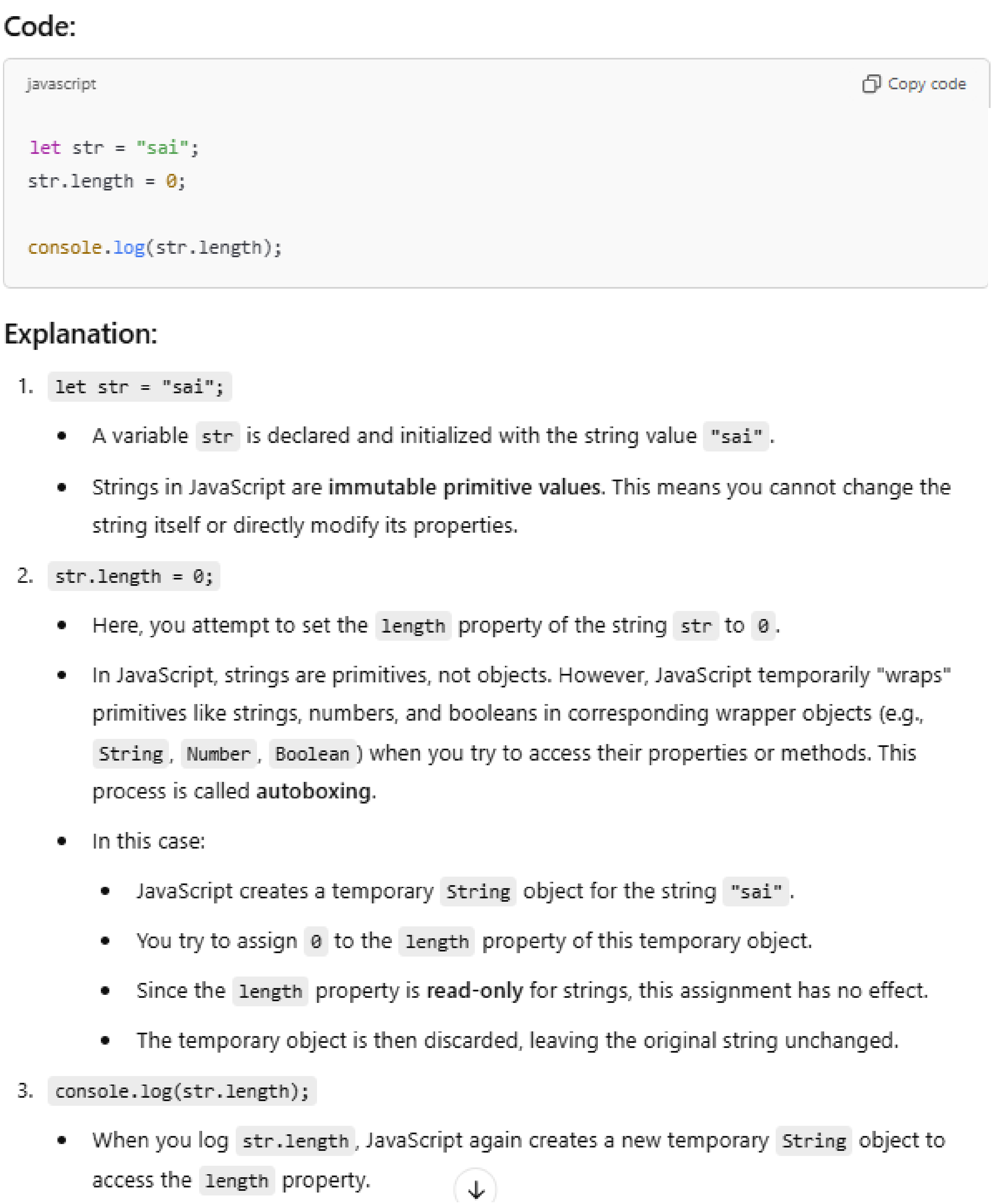
```javascript
[ { name: 'sai' } ]
```

Key Concepts:

1. **Object References:**
   - When you assign an object to a variable, the variable holds a reference to the object, not the object itself.
   - When you add an object to an array, the array stores a reference to the object.
2. **Reassignment Does Not Affect Other References:**
   - Changing the variable `person` to `null` does not affect other references (like the one in `members`).
3. **Immutability of `const`:**
   - The `members` array is declared as `const`, which means the reference to the array cannot change. However, the contents of the array (elements) can be modified.

Final State of Variables:

- `person`: `null`
- `members`: `[ { name: 'sai' } ]` (still contains a reference to the original object).

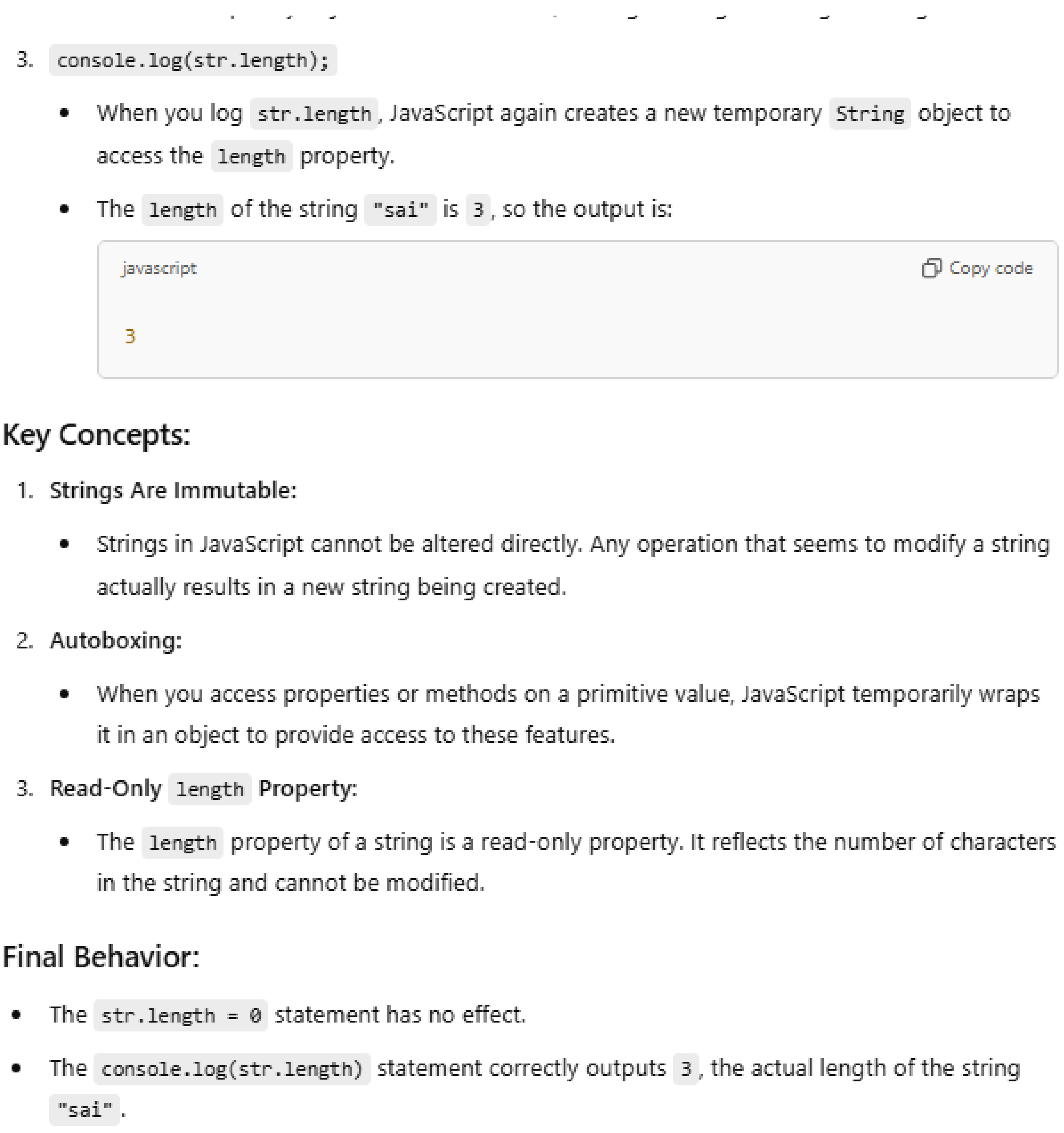


Code:

```javascript
let str = "sai";
str.length = 0;

console.log(str.length);
```

Explanation:

1. `let str = "sai";`
   - A variable `str` is declared and initialized with the string value `"sai"`.
   - Strings in JavaScript are **immutable primitive values.** This means you cannot change the string itself or directly modify its properties.
2. `str.length = 0;`
   - Here, you attempt to set the `length` property of the string `str` to `0`.
   - In JavaScript, strings are primitives, not objects. However, JavaScript temporarily "wraps" primitives like strings, numbers, and booleans in corresponding wrapper objects (e.g., `String`, `Number`, `Boolean`) when you try to access their properties or methods. This process is called **autoboxing**.
   - In this case:
     - JavaScript creates a temporary `String` object for the string `"sai"`.
     - You try to assign `0` to the `length` property of this temporary object.
     - Since the `length` property is **read-only** for strings, this assignment has no effect.
     - The temporary object is then discarded, leaving the original string unchanged.
3. `console.log(str.length);`
   - When you log `str.length`, JavaScript again creates a new temporary `String` object to access the `length` property.



3. `console.log(str.length);`
   - When you log `str.length`, JavaScript again creates a new temporary `String` object to access the `length` property.
   - The `length` of the string `"sai"` is `3`, so the output is:

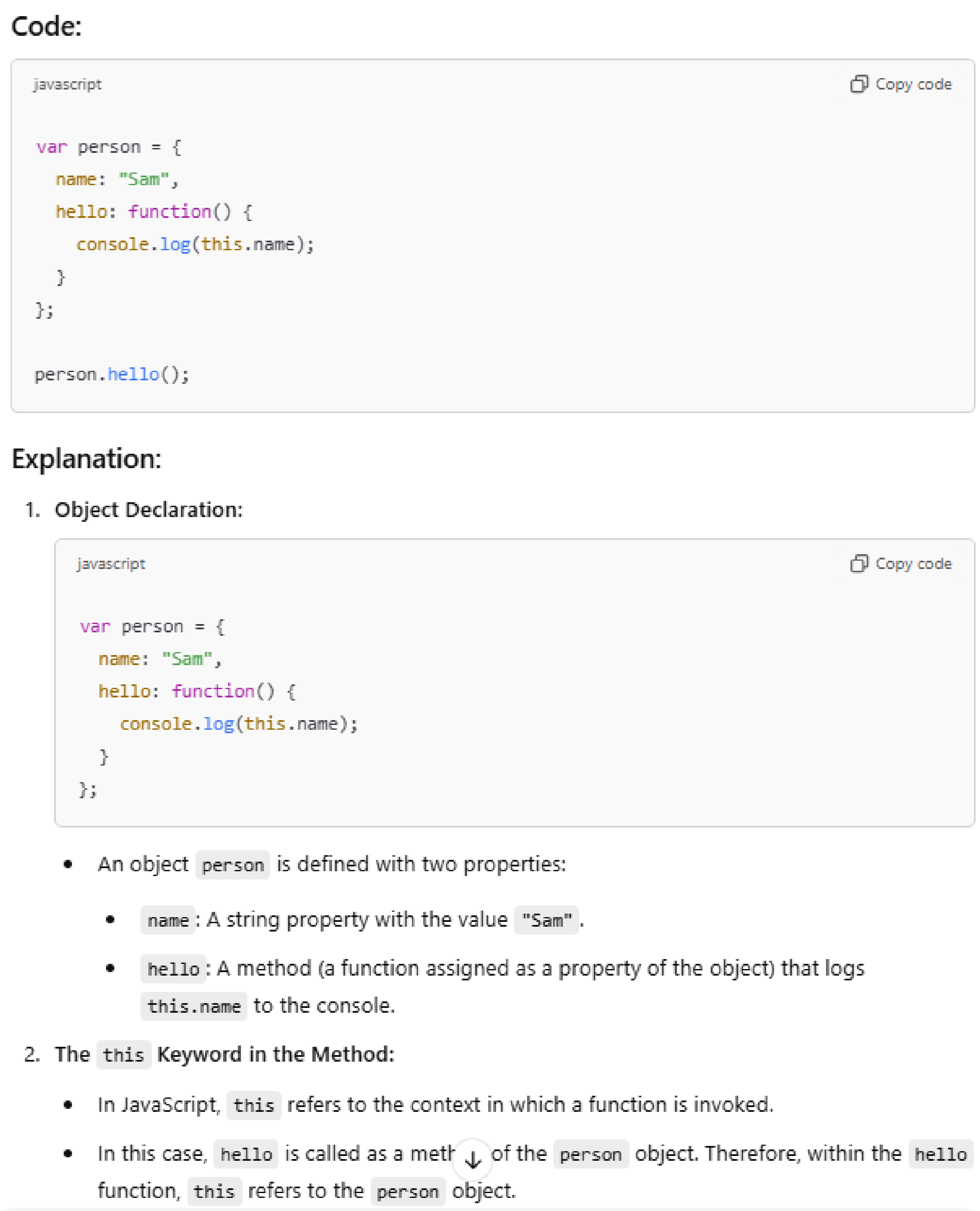
```javascript
3
```

Key Concepts:

1. **Strings Are Immutable:**
   - Strings in JavaScript cannot be altered directly. Any operation that seems to modify a string actually results in a new string being created.
2. **Autoboxing:**
   - When you access properties or methods on a primitive value, JavaScript temporarily wraps it in an object to provide access to these features.
3. **Read-Only `length` Property:**
   - The `length` property of a string is a read-only property. It reflects the number of characters in the string and cannot be modified.

Final Behavior:

- The `str.length = 0` statement has no effect.
- The `console.log(str.length)` statement correctly outputs `3`, the actual length of the string `"sai"`.



Code:

```javascript
var person = {
  name: "Sam",
  hello: function() {
    console.log(this.name);
  }
};

person.hello();
```

Explanation:

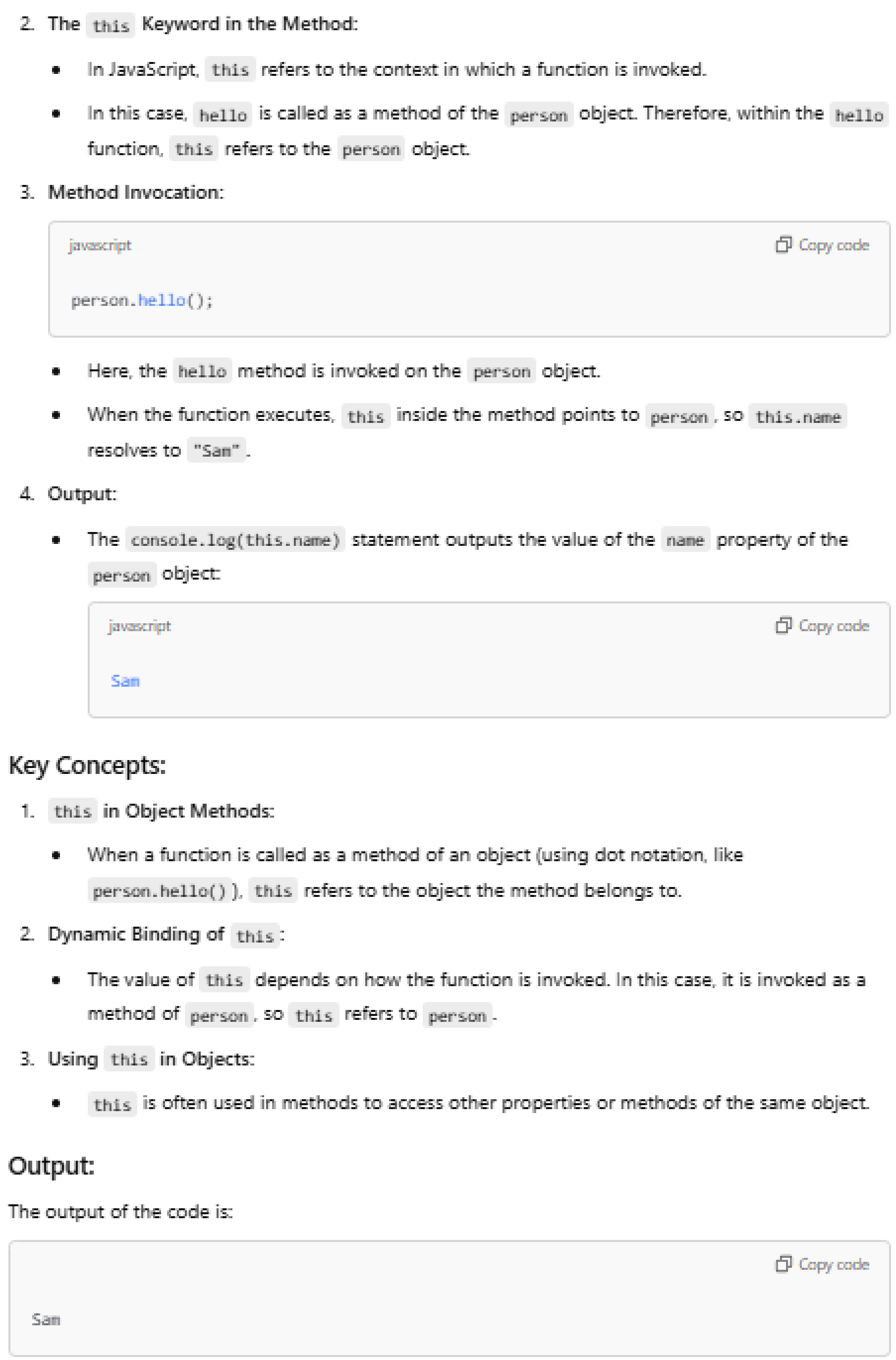
1. **Object Declaration:**

```javascript
var person = {
  name: "Sam",
  hello: function() {
    console.log(this.name);
  }
};
```

- An object `person` is defined with two properties:
  - `name`: A string property with the value `"Sam"`.
  - `hello`: A method (a function assigned as a property of the object) that logs `this.name` to the console.

2. **The `this` Keyword in the Method:**

- In JavaScript, `this` refers to the context in which a function is invoked.
- In this case, `hello` is called as a meth of the `person` object. Therefore, within the `hello` function, `this` refers to the `person` object.



2. **The `this` Keyword in the Method:**

- In JavaScript, `this` refers to the context in which a function is invoked.
- In this case, `hello` is called as a method of the `person` object. Therefore, within the `hello` function, `this` refers to the `person` object.

3. **Method Invocation:**

```javascript
person.hello();
```

- Here, the `hello` method is invoked on the `person` object.
- When the function executes, `this` inside the method points to `person`, so `this.name` resolves to `"Sam"`.

4. **Output:**

- The `console.log(this.name)` statement outputs the value of the `name` property of the `person` object:

```javascript
Sam
```

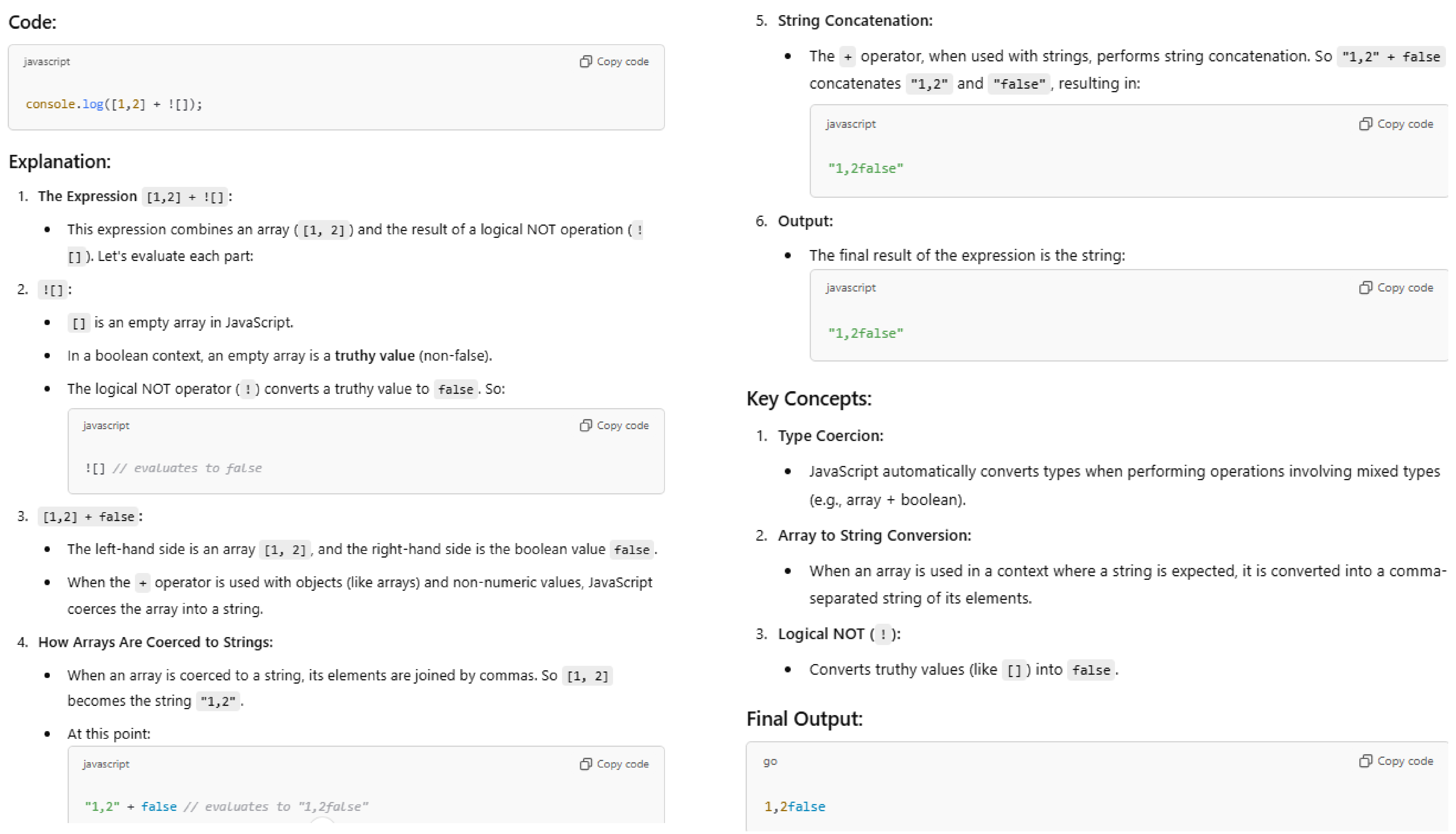
Key Concepts:

1. **`this` in Object Methods:**
   - When a function is called as a method of an object (using dot notation, like `person.hello()`), `this` refers to the object the method belongs to.
2. **Dynamic Binding of `this`:**
   - The value of `this` depends on how the function is invoked. In this case, it is invoked as a method of `person`, so `this` refers to `person`.
3. **Using `this` in Objects:**
   - `this` is often used in methods to access other properties or methods of the same object.

Output:

The output of the code is:

```
Sam
```



Code:

```javascript
console.log([1,2] + ![]);
```

Explanation:

1. **The Expression `[1,2] + ![]`:**
   - This expression combines an array (`[1, 2]`) and the result of a logical NOT operation (`![]`). Let's evaluate each part:
2. **`![]`:**
   - `[]` is an empty array in JavaScript.
   - In a boolean context, an empty array is a **truthy value** (non-false).
   - The logical NOT operator (`!`) converts a truthy value to `false`. So:

     ```javascript
     ![] // evaluates to false
     ```
3. **`[1,2] + false`:**
   - The left-hand side is an array `[1, 2]`, and the right-hand side is the boolean value `false`.
   - When the `+` operator is used with objects (like arrays) and non-numeric values, JavaScript coerces the array into a string.
4. **How Arrays Are Coerced to Strings:**
   - When an array is coerced to a string, its elements are joined by commas. So `[1, 2]` becomes the string `"1,2"`.
   - At this point:

     ```javascript
     "1,2" + false // evaluates to "1,2false"
     ```
5. **String Concatenation:**
   - The `+` operator, when used with strings, performs string concatenation. So `"1,2" + false` concatenates `"1,2"` and `"false"`, resulting in:

     ```javascript
     "1,2false"
     ```
6. **Output:**
   - The final result of the expression is the string:

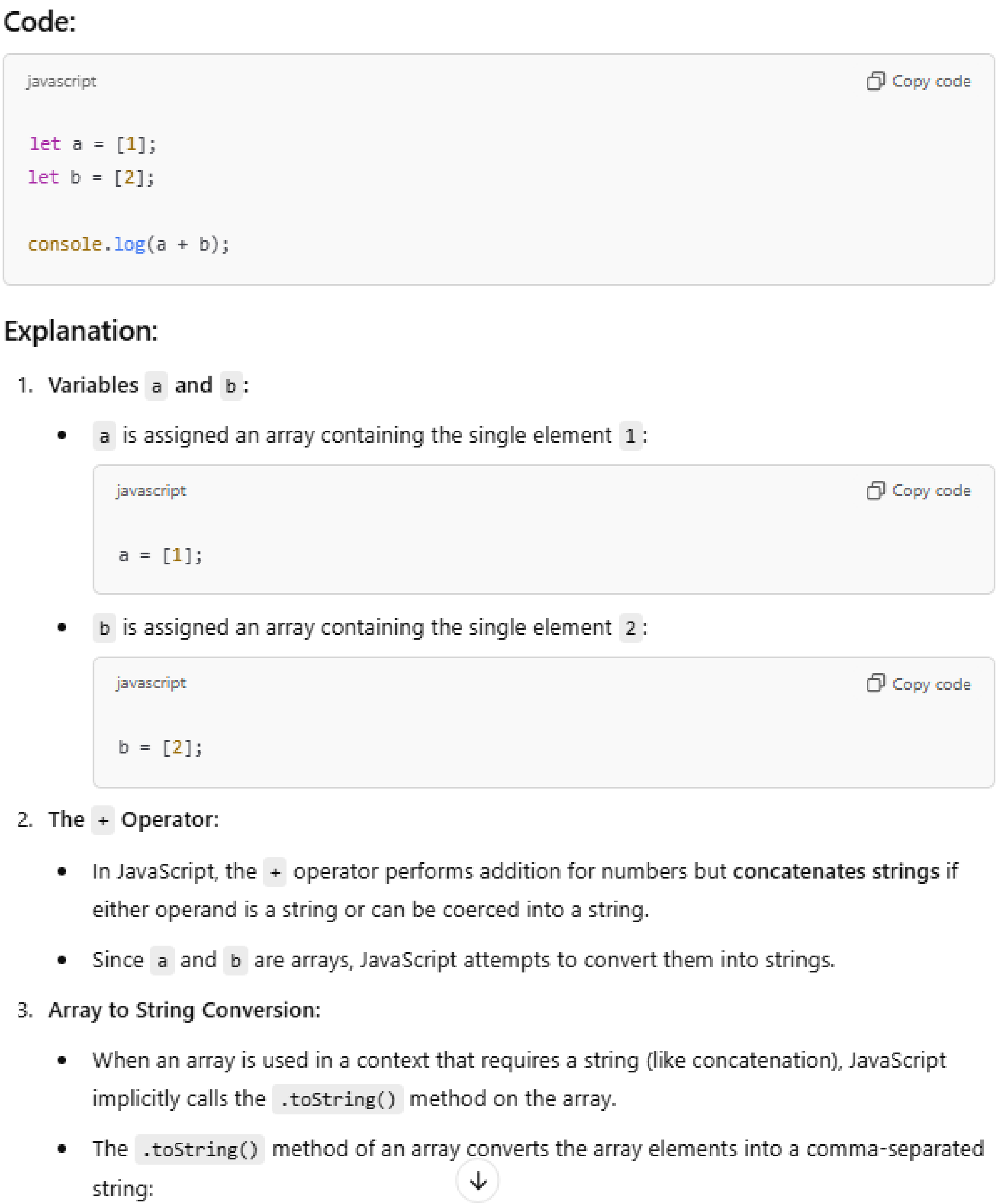
     ```javascript
     "1,2false"
     ```

Key Concepts:

1. **Type Coercion:**
   - JavaScript automatically converts types when performing operations involving mixed types (e.g., array + boolean).
2. **Array to String Conversion:**
   - When an array is used in a context where a string is expected, it is converted into a comma-separated string of its elements.
3. **Logical NOT (`!`):**
   - Converts truthy values (like `[]`) into `false`.

Final Output:

```go
1,2false
```



Code:

```javascript
let a = [1];
let b = [2];

console.log(a + b);
```

Explanation:

1. **Variables `a` and `b`:**
   - `a` is assigned an array containing the single element `1`:

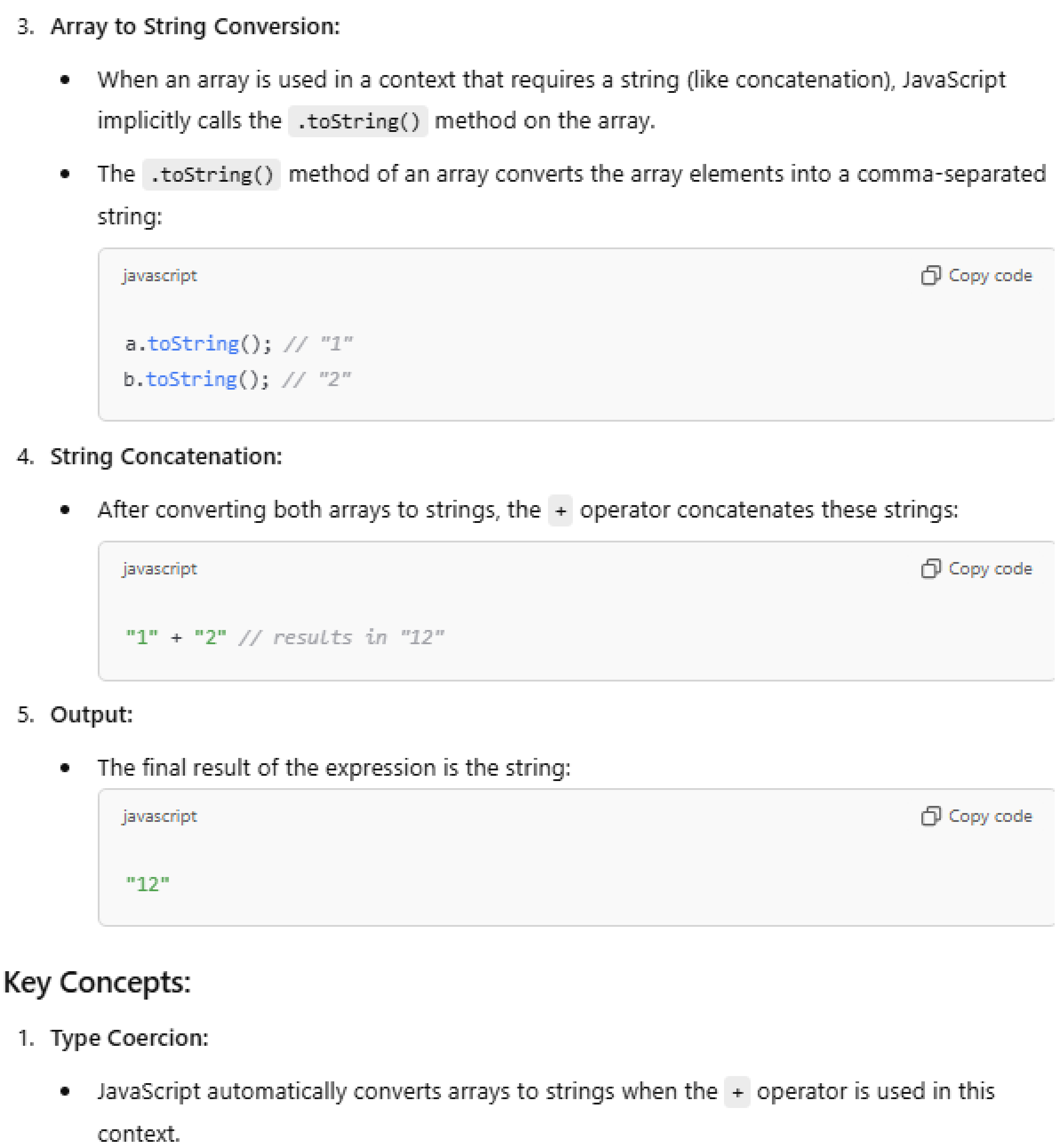
     ```javascript
     a = [1];
     ```

   - `b` is assigned an array containing the single element `2`:

     ```javascript
     b = [2];
     ```

2. **The `+` Operator:**
   - In JavaScript, the `+` operator performs addition for numbers but **concatenates strings** if either operand is a string or can be coerced into a string.
   - Since `a` and `b` are arrays, JavaScript attempts to convert them into strings.

3. **Array to String Conversion:**
   - When an array is used in a context that requires a string (like concatenation), JavaScript implicitly calls the `.toString()` method on the array.
   - The `.toString()` method of an array converts the array elements into a comma-separated string:



3. **Array to String Conversion:**
   - When an array is used in a context that requires a string (like concatenation), JavaScript implicitly calls the `.toString()` method on the array.
   - The `.toString()` method of an array converts the array elements into a comma-separated string:

```javascript
a.toString(); // "1"
b.toString(); // "2"
```

4. **String Concatenation:**
   - After converting both arrays to strings, the `+` operator concatenates these strings:

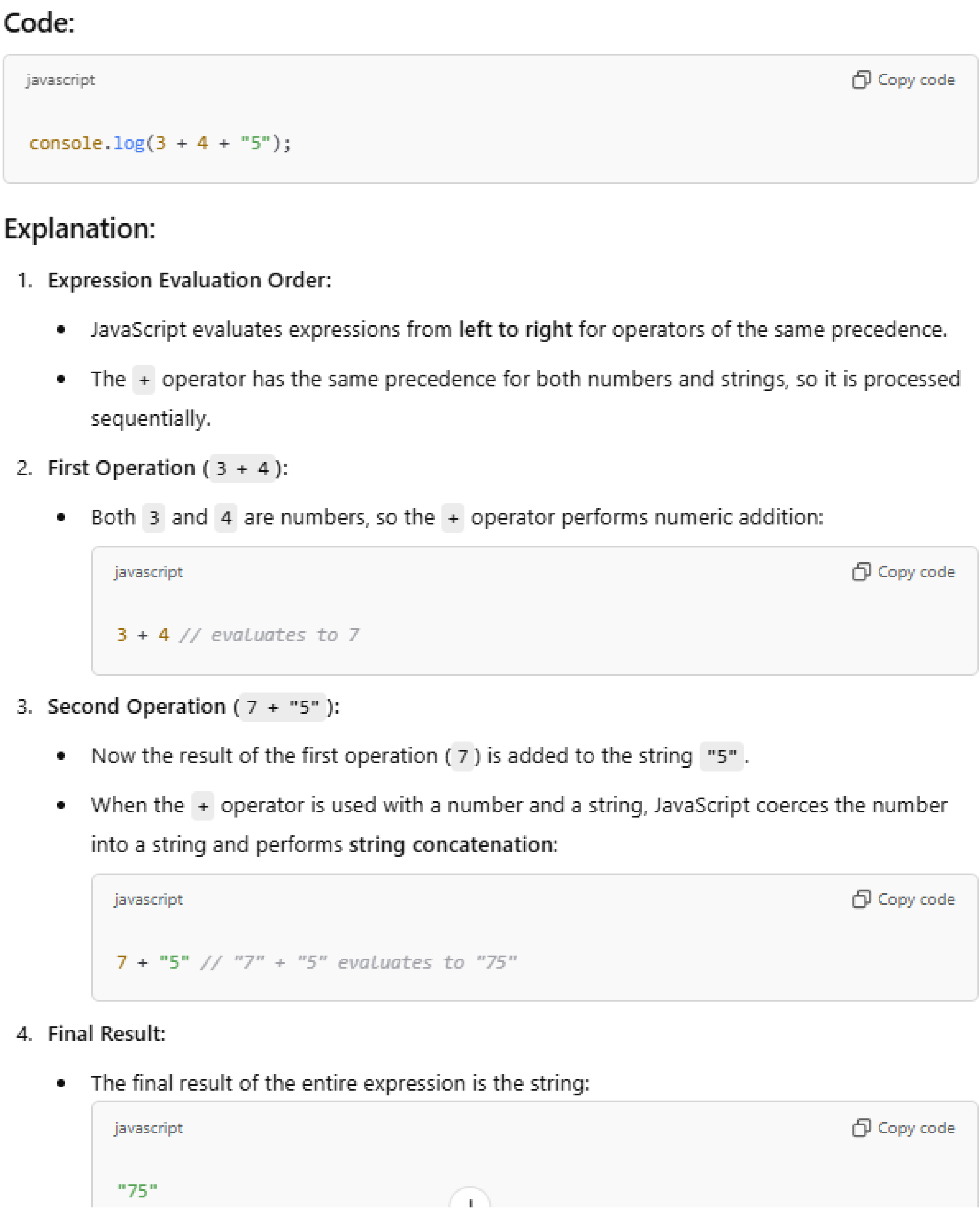
     ```javascript
     "1" + "2" // results in "12"
     ```

5. **Output:**
   - The final result of the expression is the string:

     ```javascript
     "12"
     ```

Key Concepts:

1. **Type Coercion:**
   - JavaScript automatically converts arrays to strings when the `+` operator is used in this context.



Code:

```javascript
console.log(3 + 4 + "5");
```

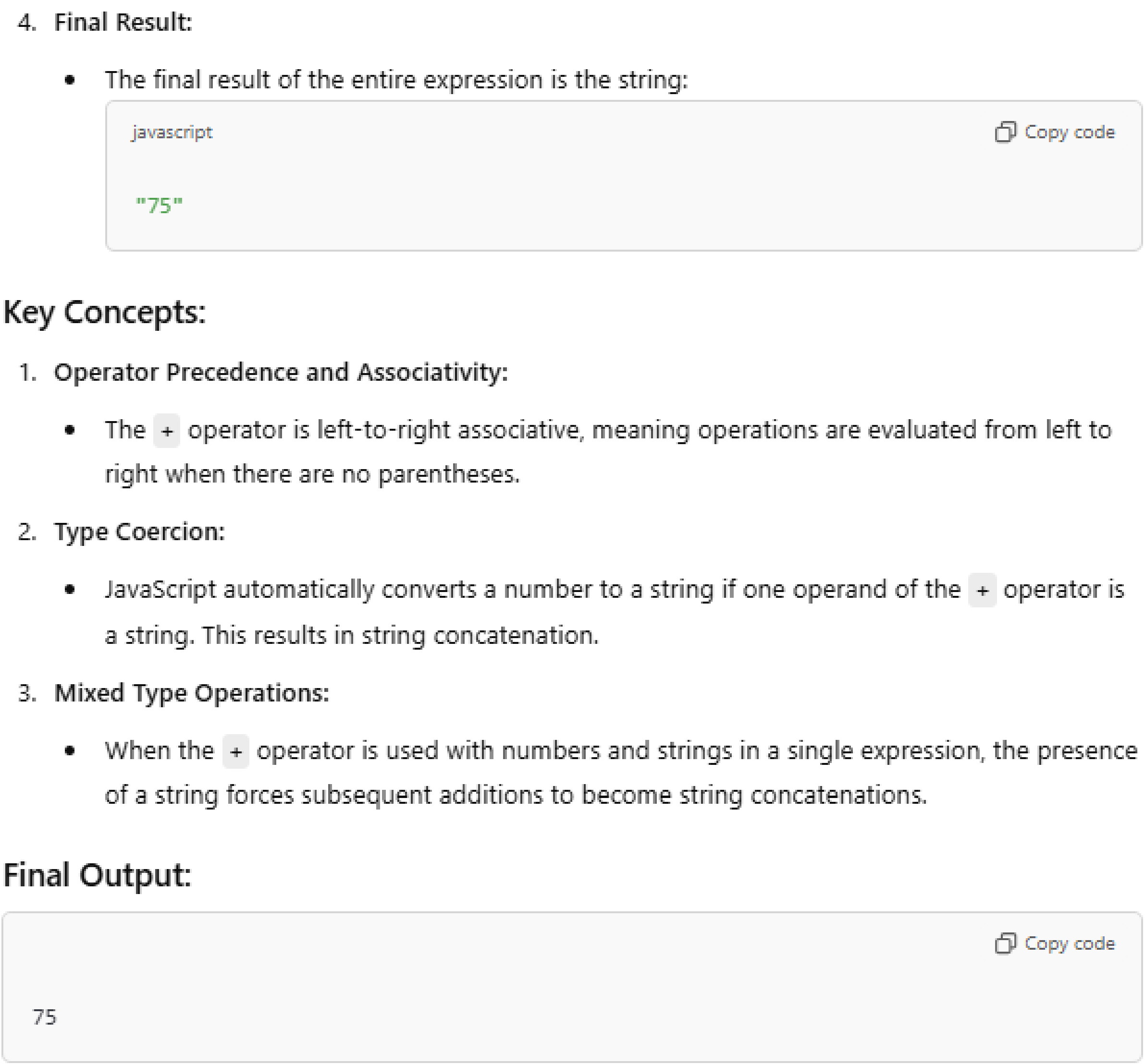
Explanation:

1. **Expression Evaluation Order:**
   - JavaScript evaluates expressions from **left to right** for operators of the same precedence.
   - The `+` operator has the same precedence for both numbers and strings, so it is processed sequentially.
2. **First Operation (`3 + 4`):**
   - Both `3` and `4` are numbers, so the `+` operator performs numeric addition:

     ```javascript
     3 + 4 // evaluates to 7
     ```
3. **Second Operation (`7 + "5"`):**
   - Now the result of the first operation (`7`) is added to the string `"5"`.
   - When the `+` operator is used with a number and a string, JavaScript coerces the number into a string and performs **string concatenation**:

     ```javascript
     7 + "5" // "7" + "5" evaluates to "75"
     ```
4. **Final Result:**
   - The final result of the entire expression is the string:

     ```javascript
     "75"
     ```



4. **Final Result:**
   - The final result of the entire expression is the string:

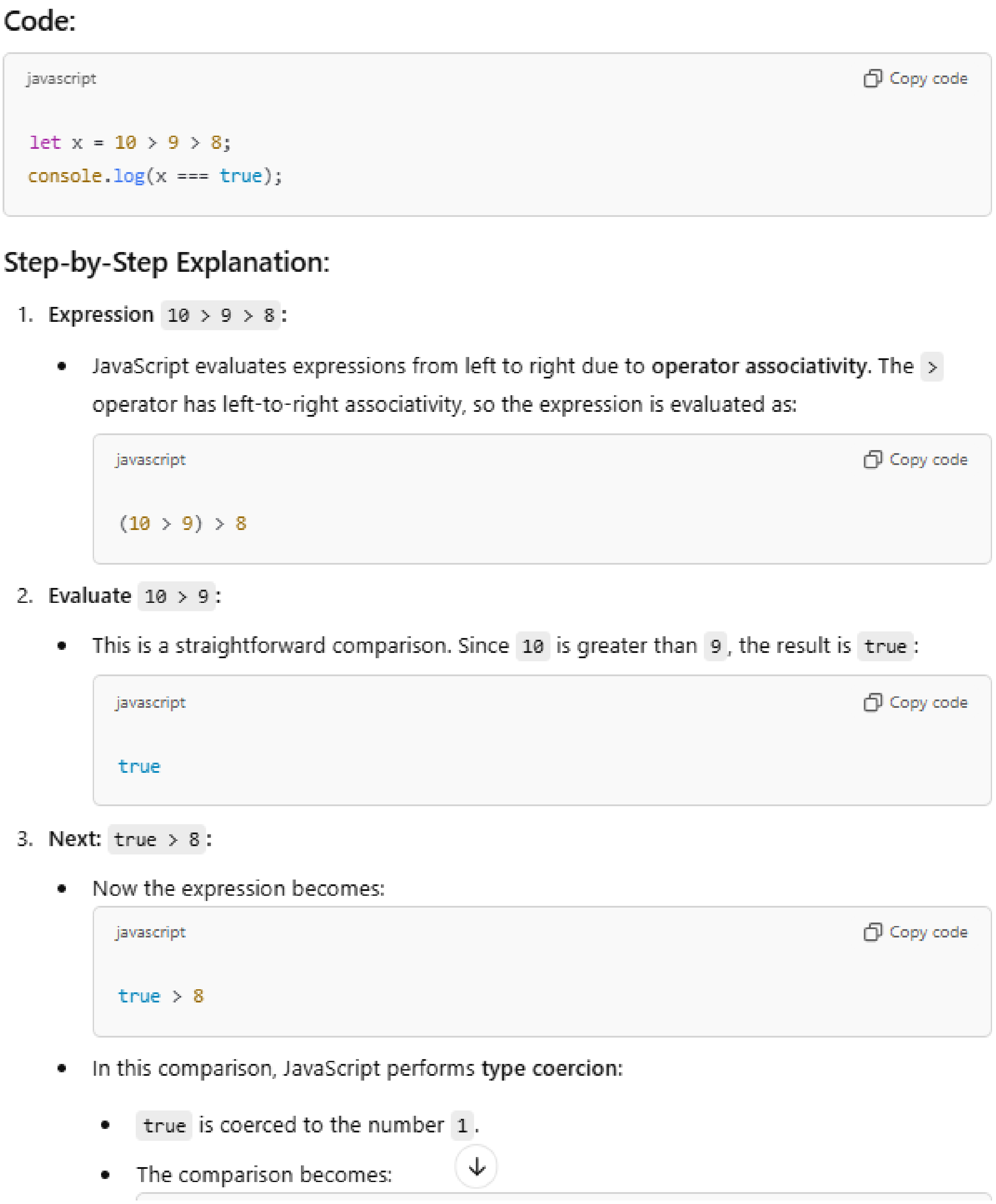
     ```javascript
     "75"
     ```

Key Concepts:

1. **Operator Precedence and Associativity:**
   - The `+` operator is left-to-right associative, meaning operations are evaluated from left to right when there are no parentheses.
2. **Type Coercion:**
   - JavaScript automatically converts a number to a string if one operand of the `+` operator is a string. This results in string concatenation.
3. **Mixed Type Operations:**
   - When the `+` operator is used with numbers and strings in a single expression, the presence of a string forces subsequent additions to become string concatenations.

Final Output:

```
75
```



Code:

```javascript
let x = 10 > 9 > 8;
console.log(x === true);
```

Step-by-Step Explanation:

1. **Expression `10 > 9 > 8`:**
   - JavaScript evaluates expressions from left to right due to **operator associativity**. The `>` operator has left-to-right associativity, so the expression is evaluated as:

     ```javascript
     (10 > 9) > 8
     ```

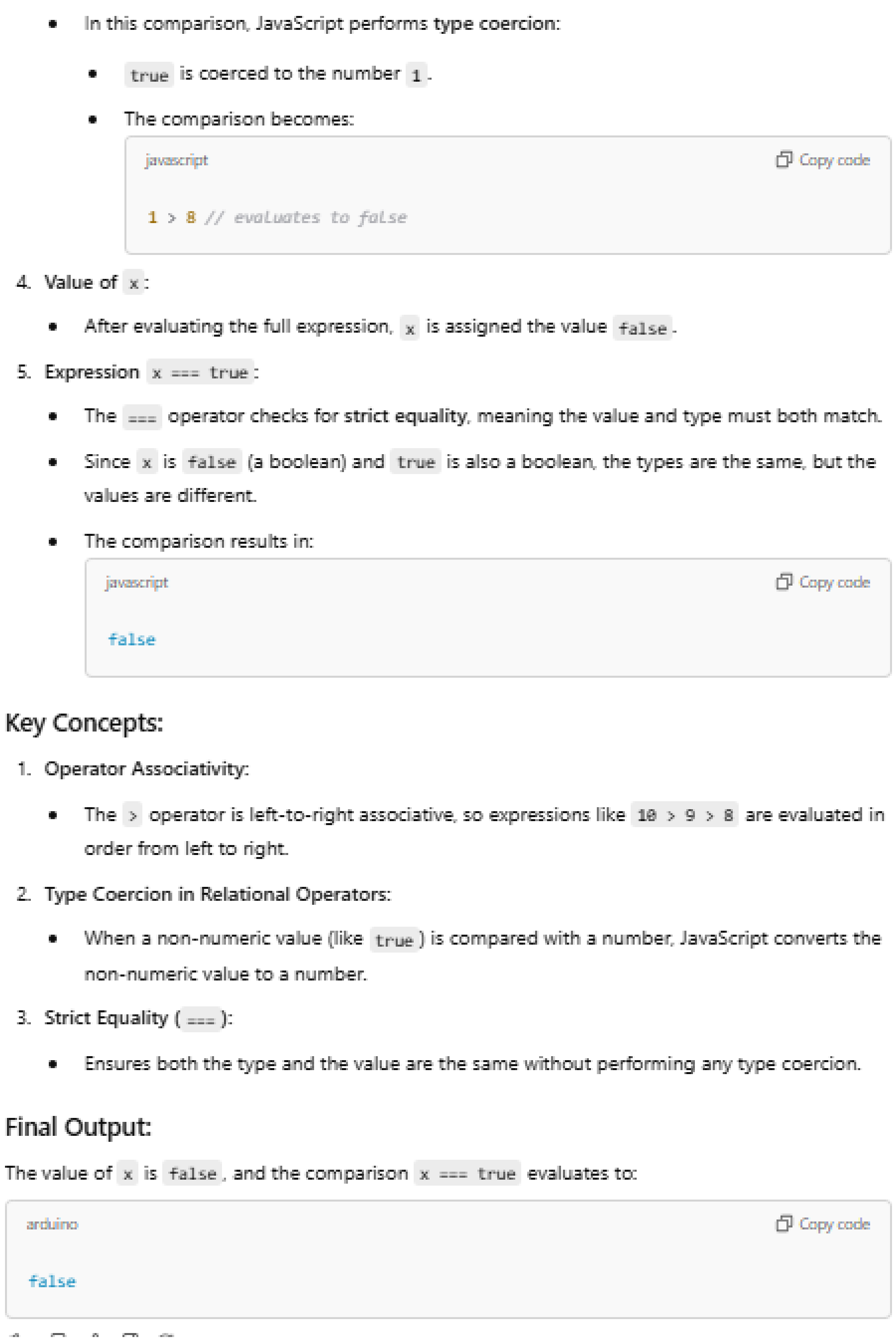
2. **Evaluate `10 > 9`:**
   - This is a straightforward comparison. Since `10` is greater than `9`, the result is `true`:

     ```javascript
     true
     ```

3. **Next: `true > 8`:**
   - Now the expression becomes:

     ```javascript
     true > 8
     ```

   - In this comparison, JavaScript performs **type coercion**:
     - `true` is coerced to the number `1`.
     - The comparison becomes:



   - In this comparison, JavaScript performs type coercion:
     - `true` is coerced to the number `1`.
     - The comparison becomes:

       ```javascript
       1 > 8 // evaluates to false
       ```

4. **Value of `x`:**
   - After evaluating the full expression, `x` is assigned the value `false`.

5. **Expression `x === true`:**
   - The `===` operator checks for strict equality, meaning the value and type must both match.
   - Since `x` is `false` (a boolean) and `true` is also a boolean, the types are the same, but the values are different.
   - The comparison results in:

     ```javascript
     false
     ```

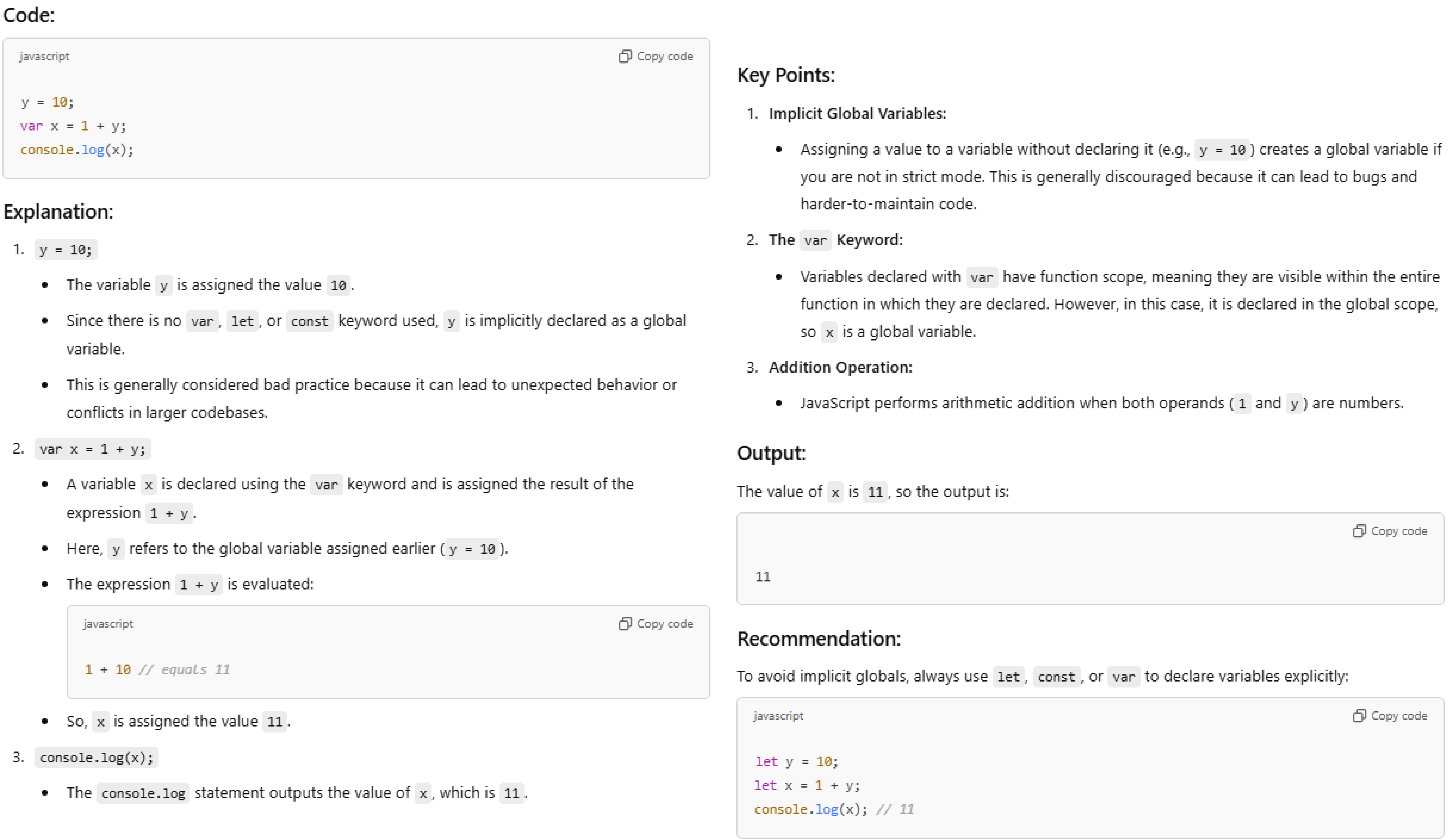
Key Concepts:

1. **Operator Associativity:**
   - The `>` operator is left-to-right associative, so expressions like `10 > 9 > 8` are evaluated in order from left to right.
2. **Type Coercion in Relational Operators:**
   - When a non-numeric value (like `true`) is compared with a number, JavaScript converts the non-numeric value to a number.
3. **Strict Equality (`===`):**
   - Ensures both the type and the value are the same without performing any type coercion.

Final Output:

The value of `x` is `false`, and the comparison `x === true` evaluates to:

```arduino
false
```



Code:

```javascript
y = 10;
var x = 1 + y;
console.log(x);
```

Explanation:

1. `y = 10;`
   - The variable `y` is assigned the value `10`.
   - Since there is no `var`, `let`, or `const` keyword used, `y` is implicitly declared as a global variable.
   - This is generally considered bad practice because it can lead to unexpected behavior or conflicts in larger codebases.
2. `var x = 1 + y;`
   - A variable `x` is declared using the `var` keyword and is assigned the result of the expression `1 + y`.
   - Here, `y` refers to the global variable assigned earlier (`y = 10`).
   - The expression `1 + y` is evaluated:

     ```javascript
     1 + 10 // equals 11
     ```

   - So, `x` is assigned the value `11`.
3. `console.log(x);`
   - The `console.log` statement outputs the value of `x`, which is `11`.

Key Points:

1. **Implicit Global Variables:**
   - Assigning a value to a variable without declaring it (e.g., `y = 10`) creates a global variable if you are not in strict mode. This is generally discouraged because it can lead to bugs and harder-to-maintain code.
2. **The `var` Keyword:**
   - Variables declared with `var` have function scope, meaning they are visible within the entire function in which they are declared. However, in this case, it is declared in the global scope, so `x` is a global variable.
3. **Addition Operation:**
   - JavaScript performs arithmetic addition when both operands (`1` and `y`) are numbers.

Output:

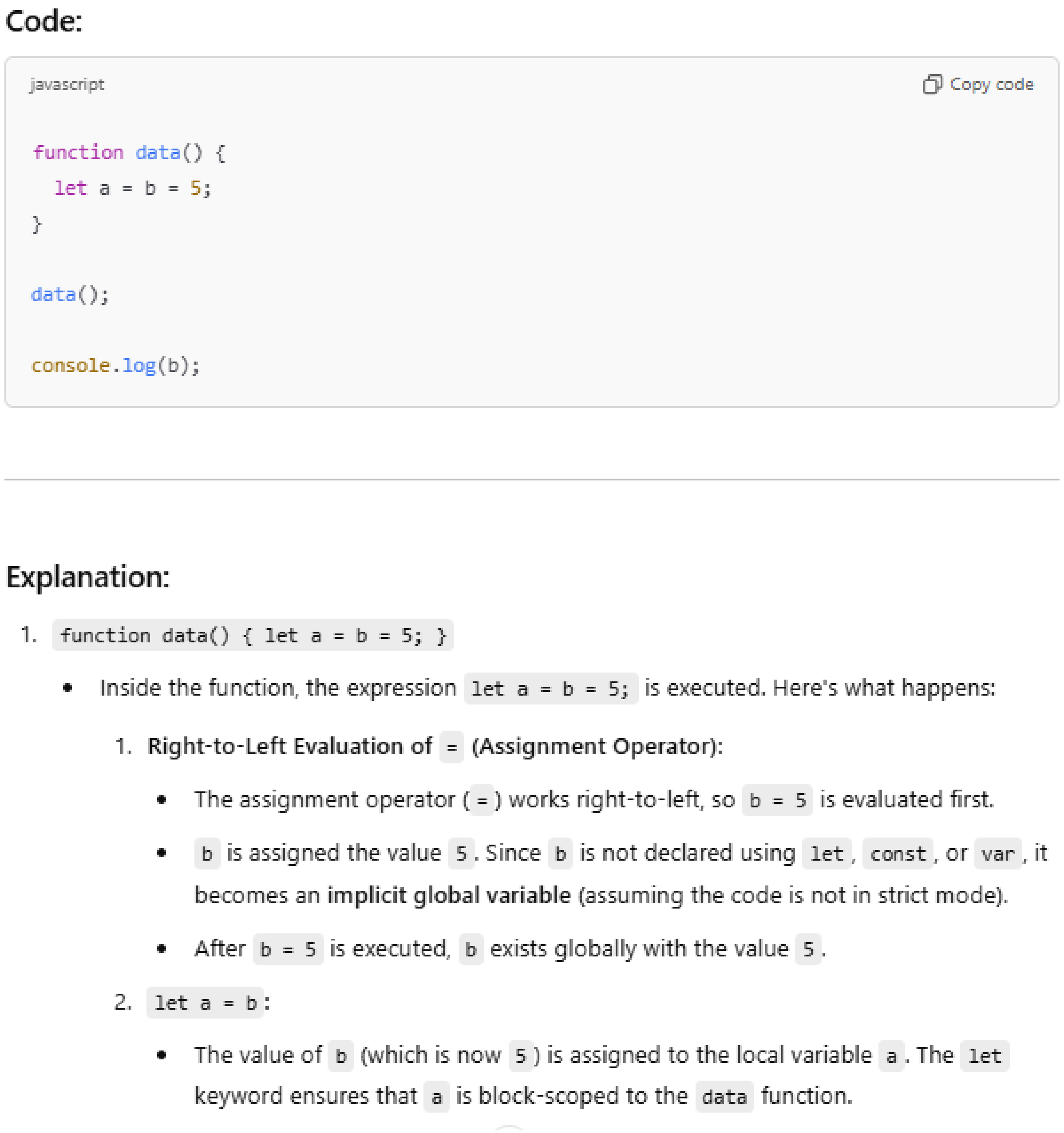
The value of `x` is `11`, so the output is:

```
11
```

Recommendation:

To avoid implicit globals, always use `let`, `const`, or `var` to declare variables explicitly:

```javascript
let y = 10;
let x = 1 + y;
console.log(x); // 11
```



Code:

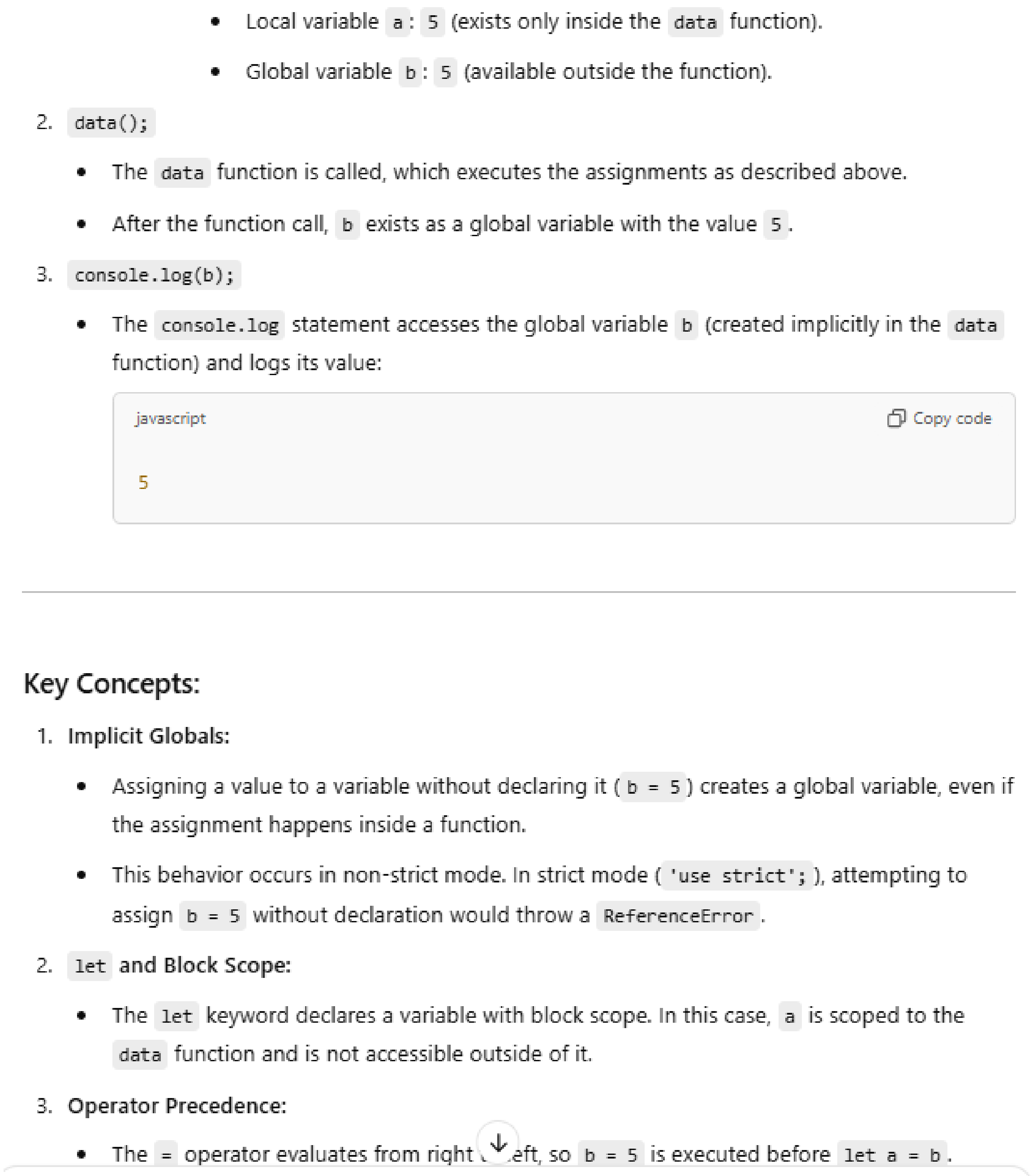
javascript

Copy code

```javascript
function data() {
  let a = b = 5;
}

data();

console.log(b);
```

Explanation:

1. `function data() { let a = b = 5; }`
   - Inside the function, the expression `let a = b = 5;` is executed. Here's what happens:
     1. **Right-to-Left Evaluation of `=` (Assignment Operator):**
        - The assignment operator (`=`) works right-to-left, so `b = 5` is evaluated first.
        - `b` is assigned the value `5`. Since `b` is not declared using `let`, `const`, or `var`, it becomes an **implicit global variable** (assuming the code is not in strict mode).
        - After `b = 5` is executed, `b` exists globally with the value `5`.
     2. `let a = b`:
        - The value of `b` (which is now `5`) is assigned to the local variable `a`. The `let` keyword ensures that `a` is block-scoped to the `data` function.



- Local variable `a` : `5` (exists only inside the `data` function).
- Global variable `b` : `5` (available outside the function).

2. `data();`
   - The `data` function is called, which executes the assignments as described above.
   - After the function call, `b` exists as a global variable with the value `5`.
3. `console.log(b);`
   - The `console.log` statement accesses the global variable `b` (created implicitly in the `data` function) and logs its value:

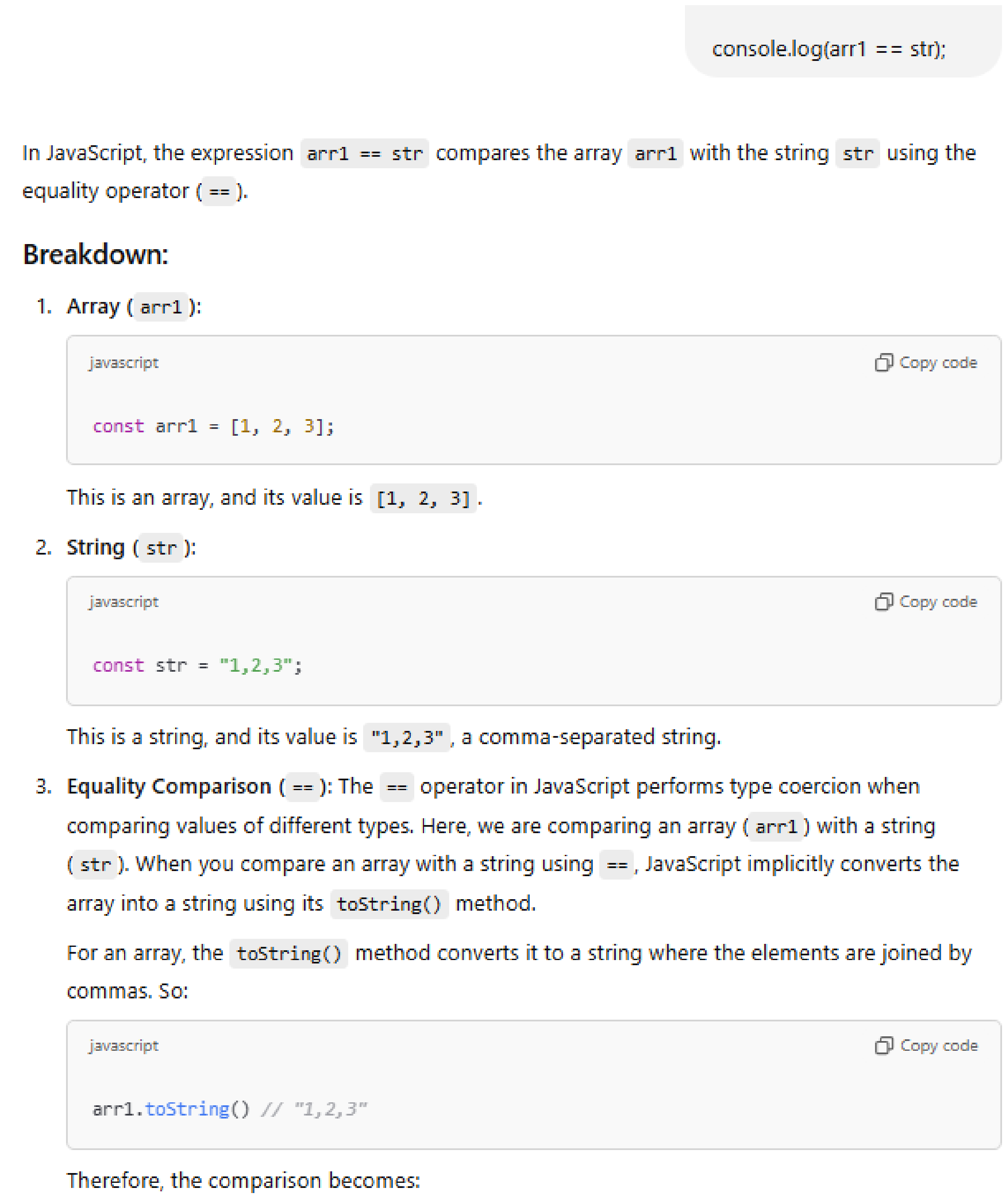
javascript

Copy code

```javascript
5
```

Key Concepts:

1. **Implicit Globals:**
   - Assigning a value to a variable without declaring it (`b = 5`) creates a global variable, even if the assignment happens inside a function.
   - This behavior occurs in non-strict mode. In strict mode (`'use strict';`), attempting to assign `b = 5` without declaration would throw a `ReferenceError`.
2. `let` **and Block Scope:**
   - The `let` keyword declares a variable with block scope. In this case, `a` is scoped to the `data` function and is not accessible outside of it.
3. **Operator Precedence:**
   - The `=` operator evaluates from right to left, so `b = 5` is executed before `let a = b`.



console.log(arr1 == str);

In JavaScript, the expression `arr1 == str` compares the array `arr1` with the string `str` using the equality operator (`==`).

Breakdown:

1. **Array (`arr1`):**

   ```javascript
   const arr1 = [1, 2, 3];
   ```

   This is an array, and its value is `[1, 2, 3]`.

2. **String (`str`):**

   ```javascript
   const str = "1,2,3";
   ```

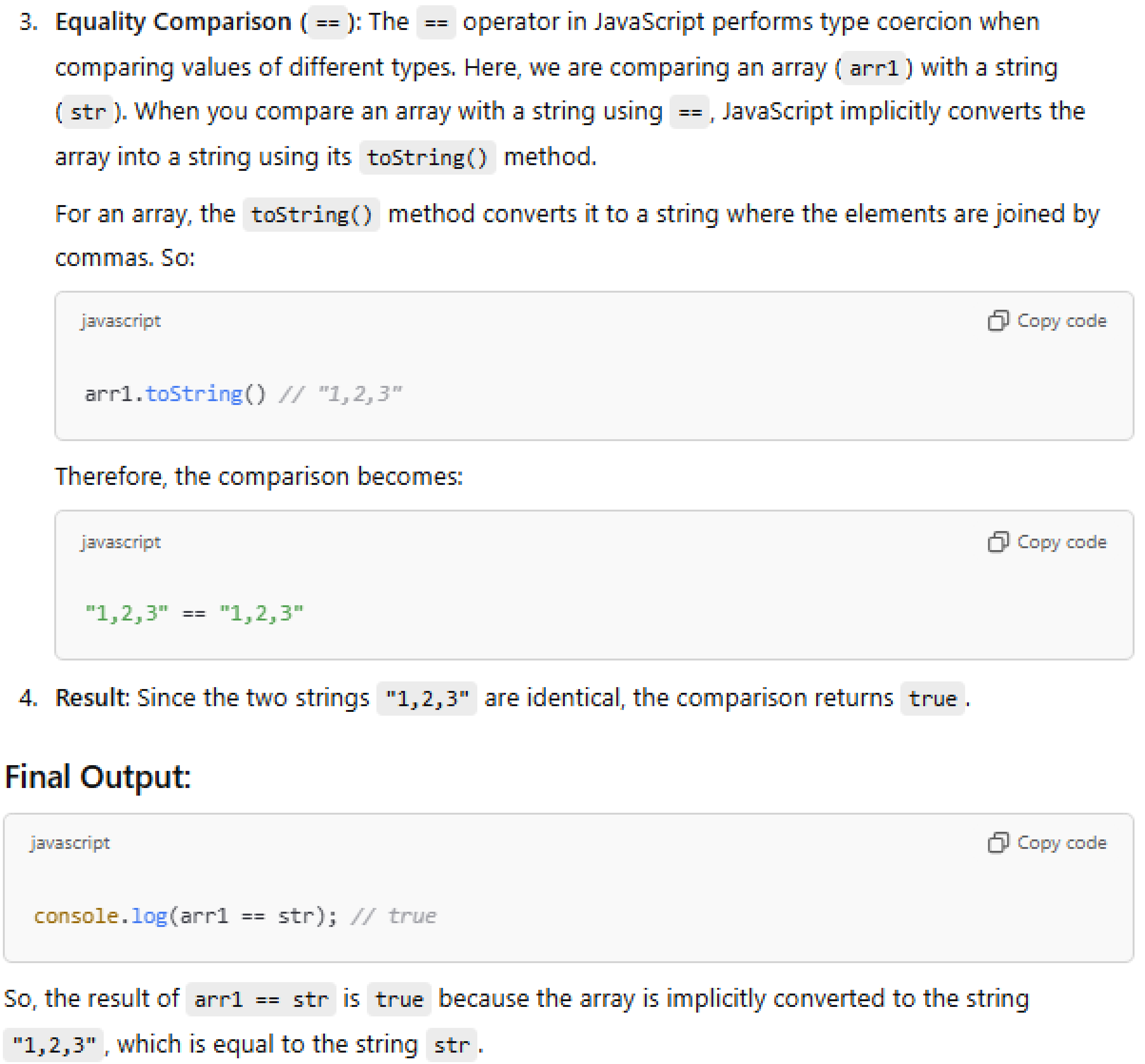
   This is a string, and its value is `"1,2,3"`, a comma-separated string.

3. **Equality Comparison (`==`):** The `==` operator in JavaScript performs type coercion when comparing values of different types. Here, we are comparing an array (`arr1`) with a string (`str`). When you compare an array with a string using `==`, JavaScript implicitly converts the array into a string using its `toString()` method.

   For an array, the `toString()` method converts it to a string where the elements are joined by commas. So:

   ```javascript
   arr1.toString() // "1,2,3"
   ```

   Therefore, the comparison becomes:



3. **Equality Comparison (`==`):** The `==` operator in JavaScript performs type coercion when comparing values of different types. Here, we are comparing an array (`arr1`) with a string (`str`). When you compare an array with a string using `==`, JavaScript implicitly converts the array into a string using its `toString()` method.

For an array, the `toString()` method converts it to a string where the elements are joined by commas. So:

arr1.toString() // "1,2,3"

Therefore, the comparison becomes:

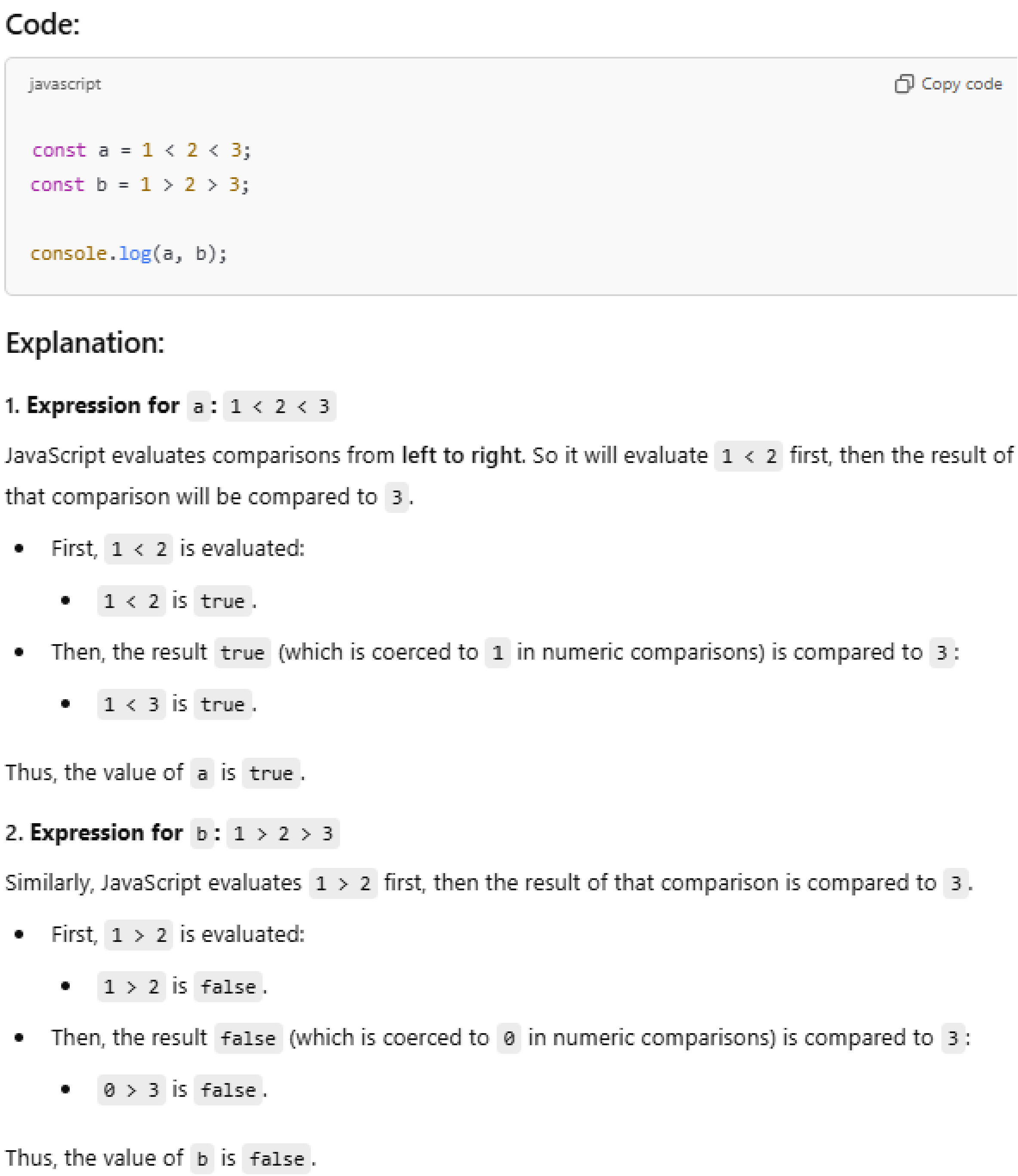
   ```javascript
   "1,2,3" == "1,2,3"
   ```

4. **Result**: Since the two strings `"1,2,3"` are identical, the comparison returns `true`.

Final Output:

```javascript
console.log(arr1 == str); // true
```

So, the result of `arr1 == str` is `true` because the array is implicitly converted to the string `"1,2,3"`, which is equal to the string `str`.



Code:

```javascript
const a = 1 < 2 < 3;
const b = 1 > 2 > 3;

console.log(a, b);
```

Explanation:

1. **Expression for `a`: `1 < 2 < 3`**

JavaScript evaluates comparisons from **left to right**. So it will evaluate `1 < 2` first, then the result of that comparison will be compared to `3`.

- First, `1 < 2` is evaluated:
  - `1 < 2` is `true`.
- Then, the result `true` (which is coerced to `1` in numeric comparisons) is compared to `3`:
  - `1 < 3` is `true`.

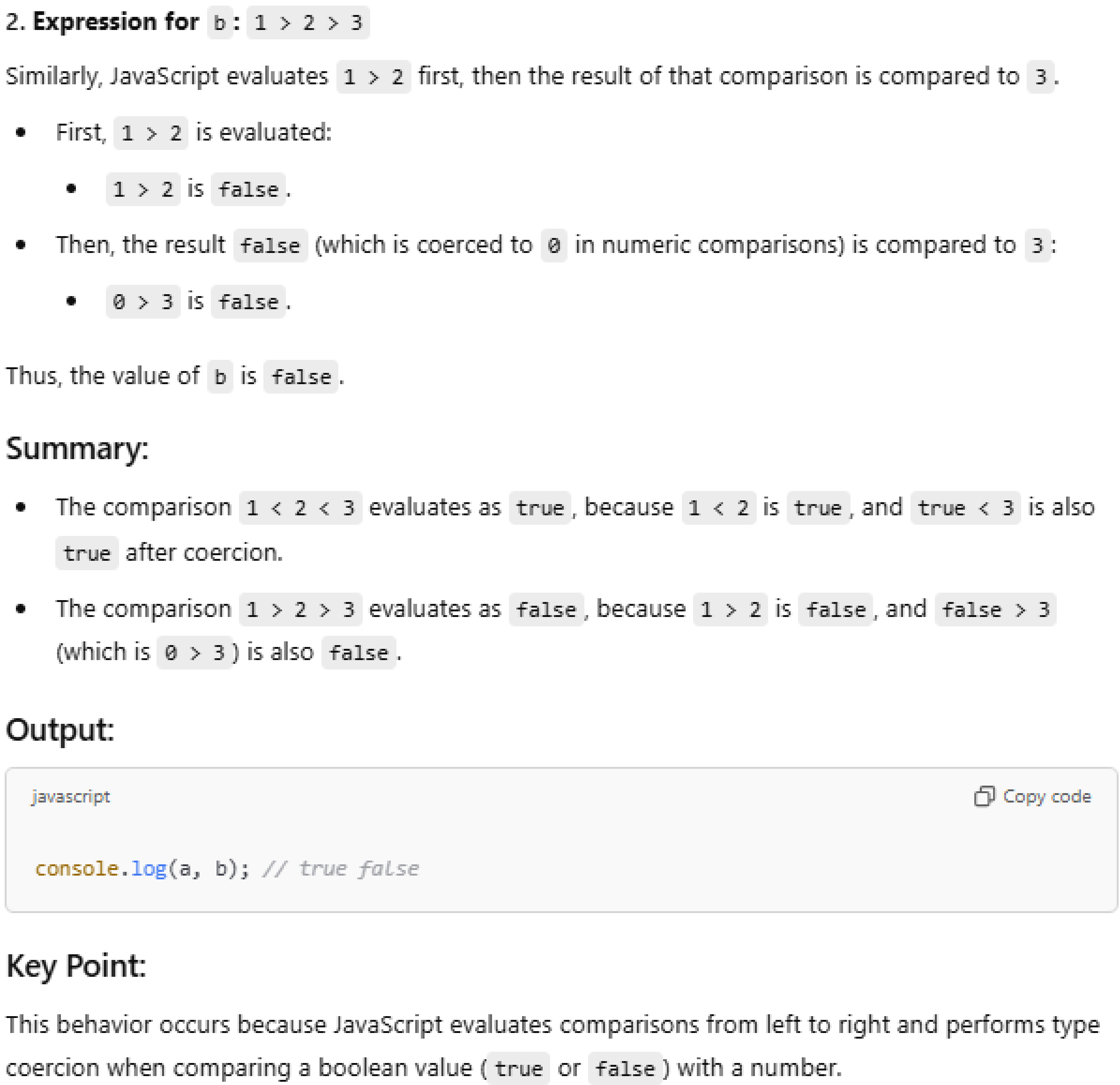
Thus, the value of `a` is `true`.

2. **Expression for `b`: `1 > 2 > 3`**

Similarly, JavaScript evaluates `1 > 2` first, then the result of that comparison is compared to `3`.

- First, `1 > 2` is evaluated:
  - `1 > 2` is `false`.
- Then, the result `false` (which is coerced to `0` in numeric comparisons) is compared to `3`:
  - `0 > 3` is `false`.

Thus, the value of `b` is `false`.



2. **Expression for `b`: `1 > 2 > 3`**

Similarly, JavaScript evaluates `1 > 2` first, then the result of that comparison is compared to `3`.

- First, `1 > 2` is evaluated:
  - `1 > 2` is `false`.
- Then, the result `false` (which is coerced to `0` in numeric comparisons) is compared to `3`:
  - `0 > 3` is `false`.

Thus, the value of `b` is `false`.

Summary:

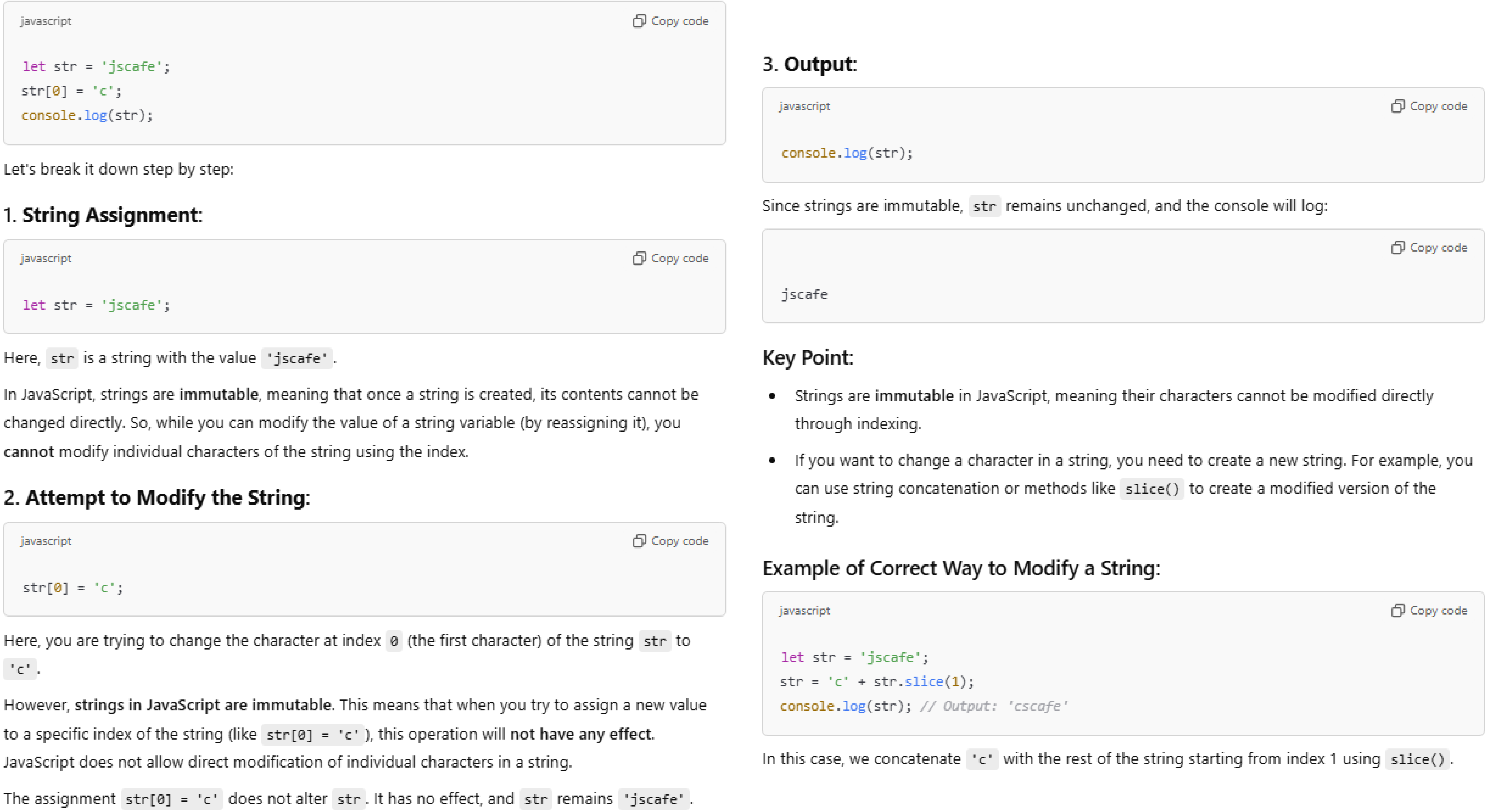
- The comparison `1 < 2 < 3` evaluates as `true`, because `1 < 2` is `true`, and `true < 3` is also `true` after coercion.
- The comparison `1 > 2 > 3` evaluates as `false`, because `1 > 2` is `false`, and `false > 3` (which is `0 > 3`) is also `false`.

Output:

```javascript
console.log(a, b); // true false
```

Key Point:

This behavior occurs because JavaScript evaluates comparisons from left to right and performs type coercion when comparing a boolean value (`true` or `false`) with a number.



```javascript
let str = 'jscafe';
str[0] = 'c';
console.log(str);
```

Let's break it down step by step:

1. **String Assignment**:

```javascript
let str = 'jscafe';
```

Here, `str` is a string with the value `'jscafe'`.

In JavaScript, strings are **immutable**, meaning that once a string is created, its contents cannot be changed directly. So, while you can modify the value of a string variable (by reassigning it), you **cannot** modify individual characters of the string using the index.

2. **Attempt to Modify the String**:

```javascript
str[0] = 'c';
```

Here, you are trying to change the character at index `0` (the first character) of the string `str` to `'c'`.

However, **strings in JavaScript are immutable.** This means that when you try to assign a new value to a specific index of the string (like `str[0] = 'c'`), this operation will **not have any effect**. JavaScript does not allow direct modification of individual characters in a string.

The assignment `str[0] = 'c'` does not alter `str`. It has no effect, and `str` remains `'jscafe'`.

3. **Output**:

```javascript
console.log(str);
```

Since strings are immutable, `str` remains unchanged, and the console will log:

```
jscafe
```

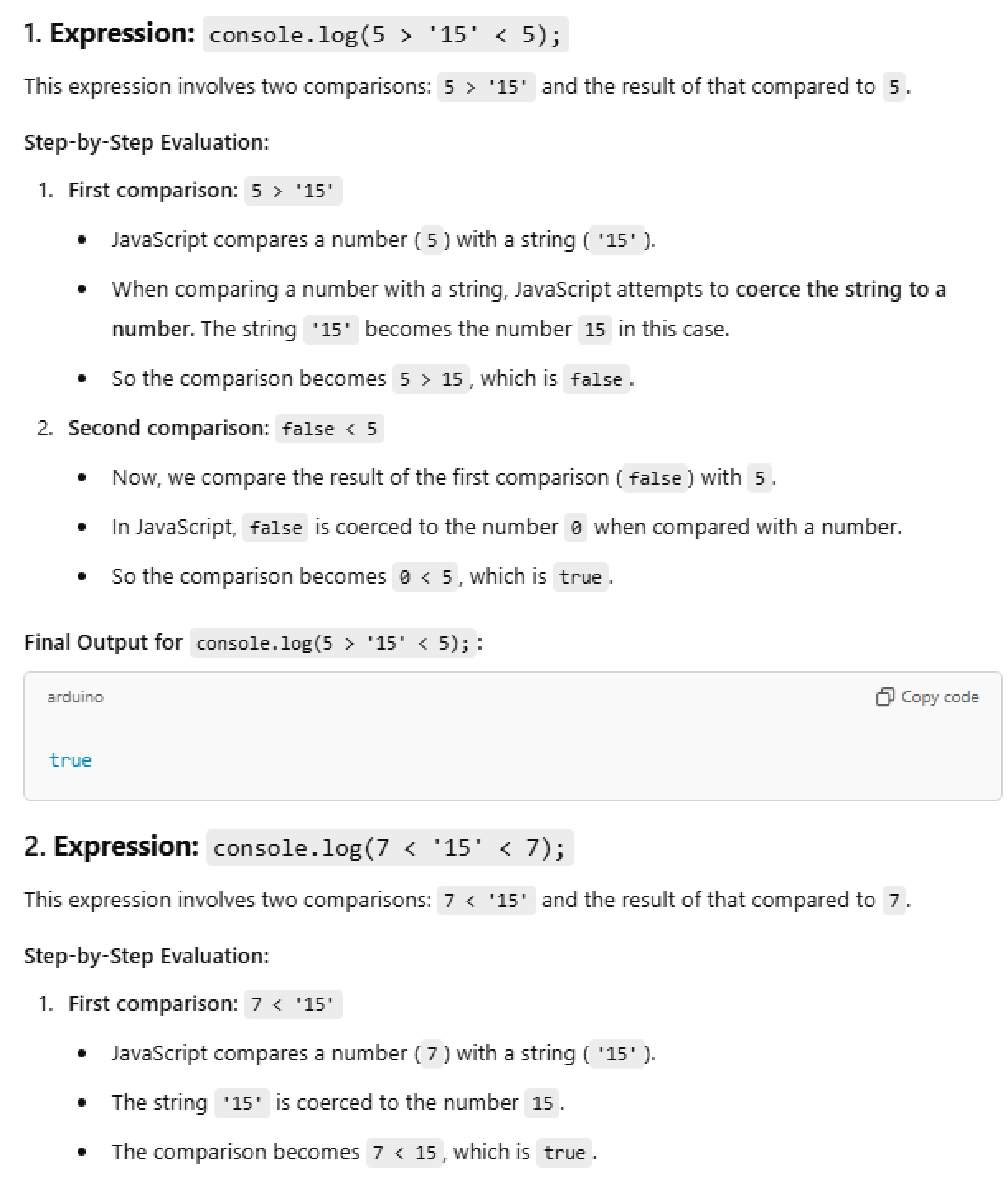
Key Point:

- Strings are **immutable** in JavaScript, meaning their characters cannot be modified directly through indexing.
- If you want to change a character in a string, you need to create a new string. For example, you can use string concatenation or methods like `slice()` to create a modified version of the string.

Example of Correct Way to Modify a String:

```javascript
let str = 'jscafe';
str = 'c' + str.slice(1);
console.log(str); // Output: 'cscafe'
```

In this case, we concatenate `'c'` with the rest of the string starting from index 1 using `slice()`.



1. Expression: `console.log(5 > '15' < 5);`

This expression involves two comparisons: `5 > '15'` and the result of that compared to `5`.

Step-by-Step Evaluation:

1. **First comparison:** `5 > '15'`
   - JavaScript compares a number (`5`) with a string (`'15'`).
   - When comparing a number with a string, JavaScript attempts to **coerce the string to a number**. The string `'15'` becomes the number `15` in this case.
   - So the comparison becomes `5 > 15`, which is `false`.
2. **Second comparison:** `false < 5`
   - Now, we compare the result of the first comparison (`false`) with `5`.
   - In JavaScript, `false` is coerced to the number `0` when compared with a number.
   - So the comparison becomes `0 < 5`, which is `true`.

Final Output for `console.log(5 > '15' < 5);`:

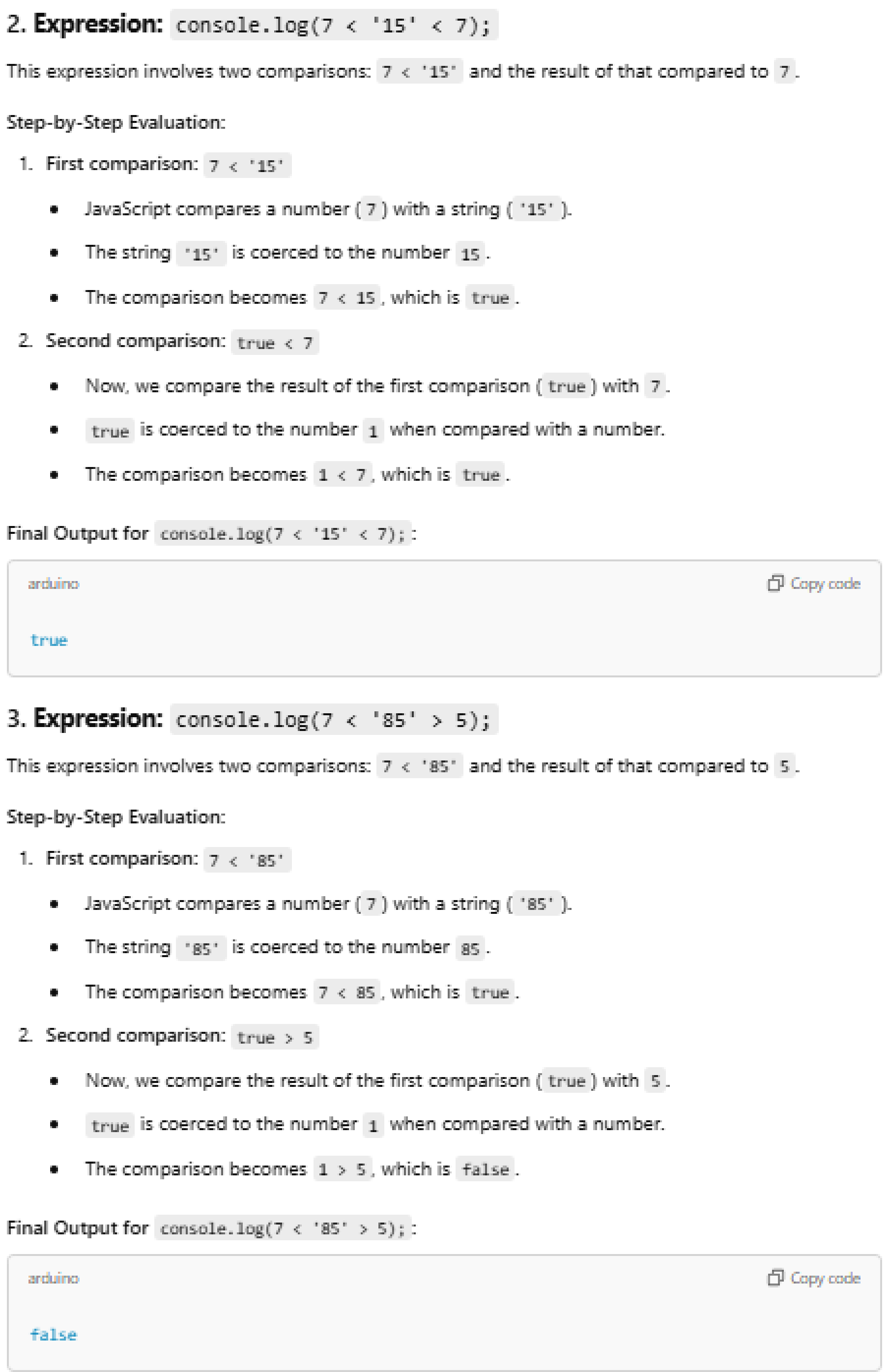
```arduino
true
```

2. Expression: `console.log(7 < '15' < 7);`

This expression involves two comparisons: `7 < '15'` and the result of that compared to `7`.

Step-by-Step Evaluation:

1. **First comparison:** `7 < '15'`
   - JavaScript compares a number (`7`) with a string (`'15'`).
   - The string `'15'` is coerced to the number `15`.
   - The comparison becomes `7 < 15`, which is `true`.



2. Expression: `console.log(7 < '15' < 7);`

This expression involves two comparisons: `7 < '15'` and the result of that compared to `7`.

Step-by-Step Evaluation:

1. First comparison: `7 < '15'`
   - JavaScript compares a number (`7`) with a string (`'15'`).
   - The string `'15'` is coerced to the number `15`.
   - The comparison becomes `7 < 15`, which is `true`.
2. Second comparison: `true < 7`
   - Now, we compare the result of the first comparison (`true`) with `7`.
   - `true` is coerced to the number `1` when compared with a number.
   - The comparison becomes `1 < 7`, which is `true`.

Final Output for `console.log(7 < '15' < 7);`:

```arduino
true
```

3. Expression: `console.log(7 < '85' > 5);`

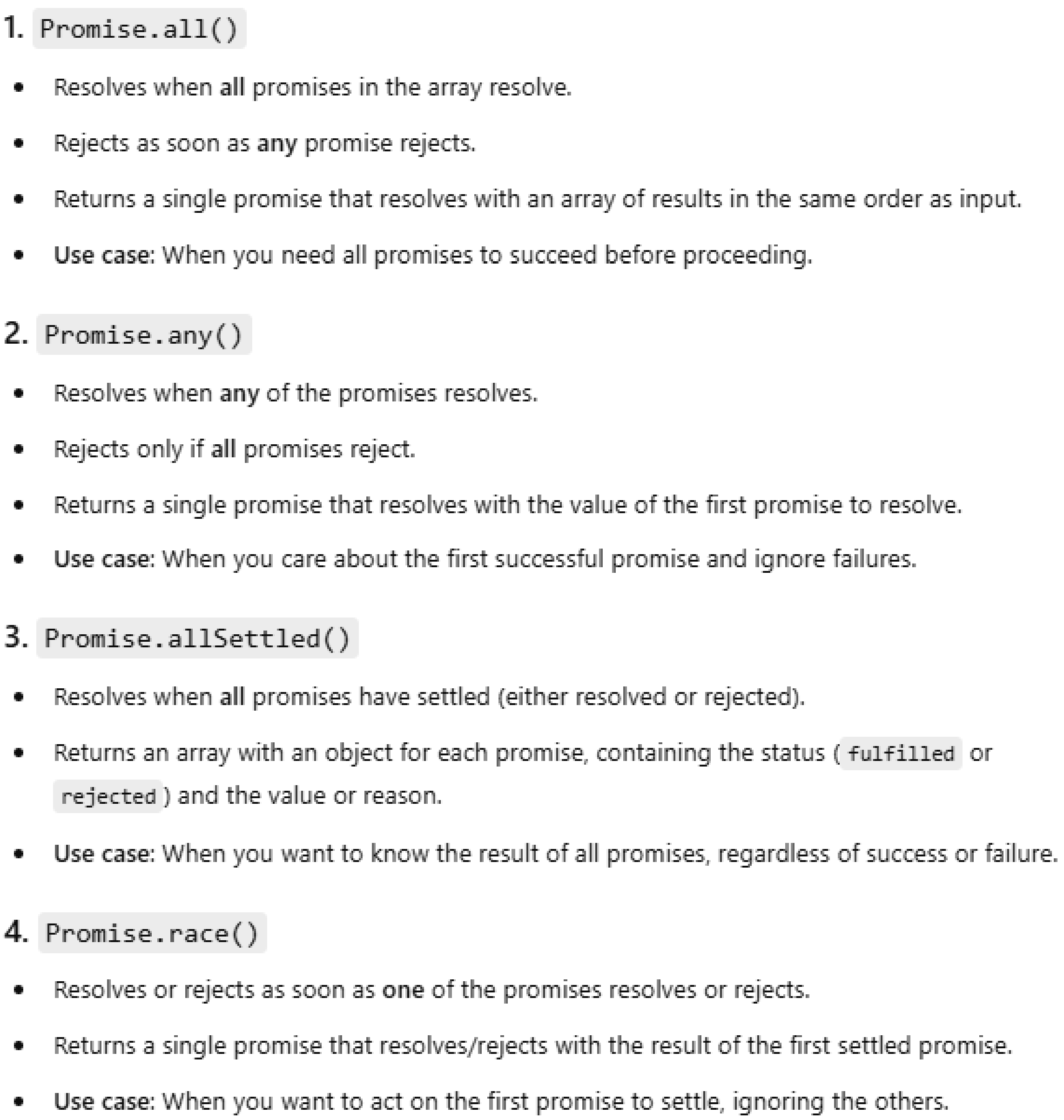
This expression involves two comparisons: `7 < '85'` and the result of that compared to `5`.

Step-by-Step Evaluation:

1. First comparison: `7 < '85'`
   - JavaScript compares a number (`7`) with a string (`'85'`).
   - The string `'85'` is coerced to the number `85`.
   - The comparison becomes `7 < 85`, which is `true`.
2. Second comparison: `true > 5`
   - Now, we compare the result of the first comparison (`true`) with `5`.
   - `true` is coerced to the number `1` when compared with a number.
   - The comparison becomes `1 > 5`, which is `false`.

Final Output for `console.log(7 < '85' > 5);`:

```arduino
false
```



1. `Promise.all()`

- Resolves when **all** promises in the array resolve.
- Rejects as soon as **any** promise rejects.
- Returns a single promise that resolves with an array of results in the same order as input.
- **Use case**: When you need all promises to succeed before proceeding.

2. `Promise.any()`

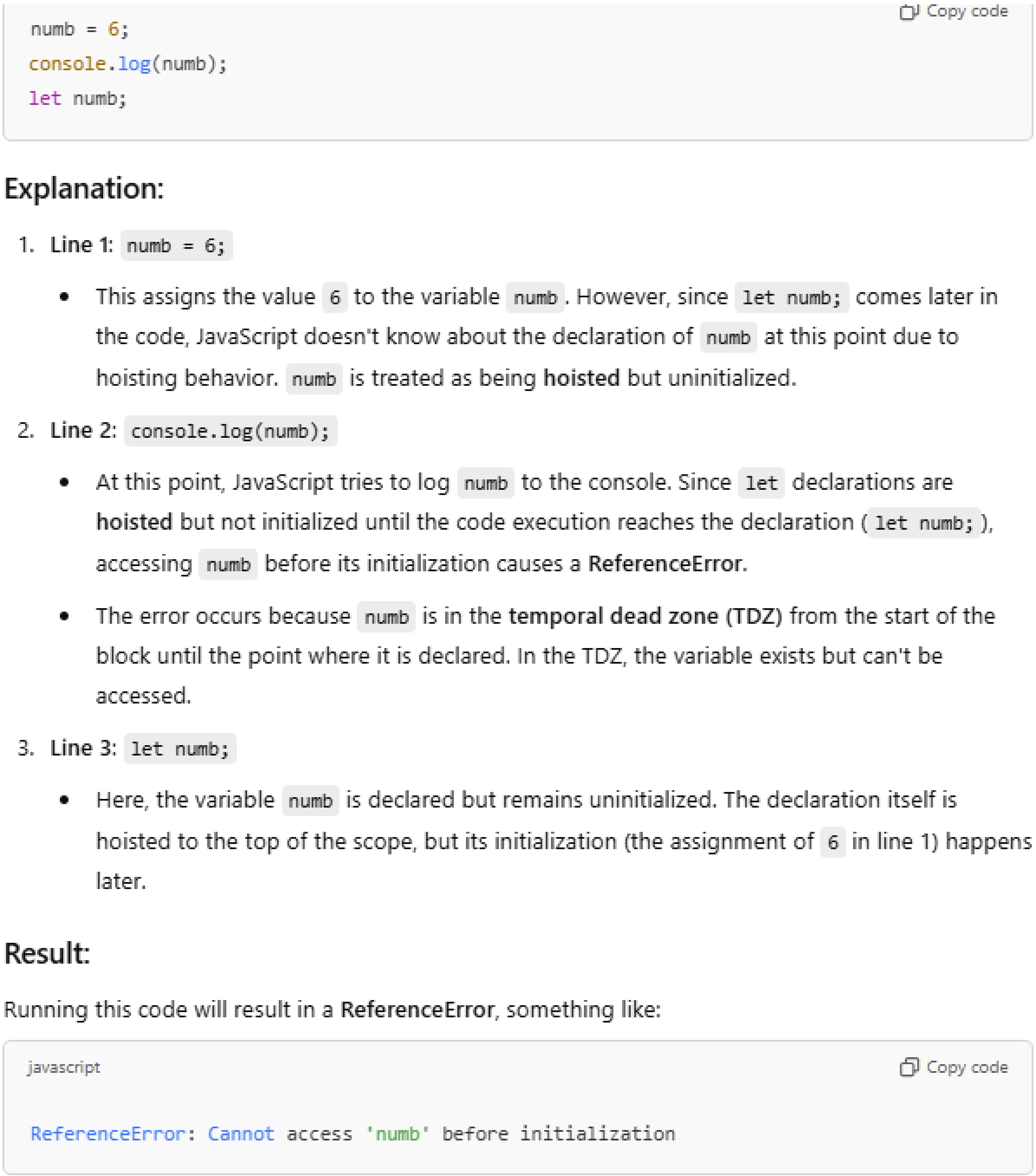
- Resolves when **any** of the promises resolves.
- Rejects only if **all** promises reject.
- Returns a single promise that resolves with the value of the first promise to resolve.
- **Use case**: When you care about the first successful promise and ignore failures.

3. `Promise.allSettled()`

- Resolves when **all** promises have settled (either resolved or rejected).
- Returns an array with an object for each promise, containing the status (`fulfilled` or `rejected`) and the value or reason.
- **Use case**: When you want to know the result of all promises, regardless of success or failure.

4. `Promise.race()`

- Resolves or rejects as soon as **one** of the promises resolves or rejects.
- Returns a single promise that resolves/rejects with the result of the first settled promise.
- **Use case**: When you want to act on the first promise to settle, ignoring the others.



```
numb = 6;
console.log(numb);
let numb;
```

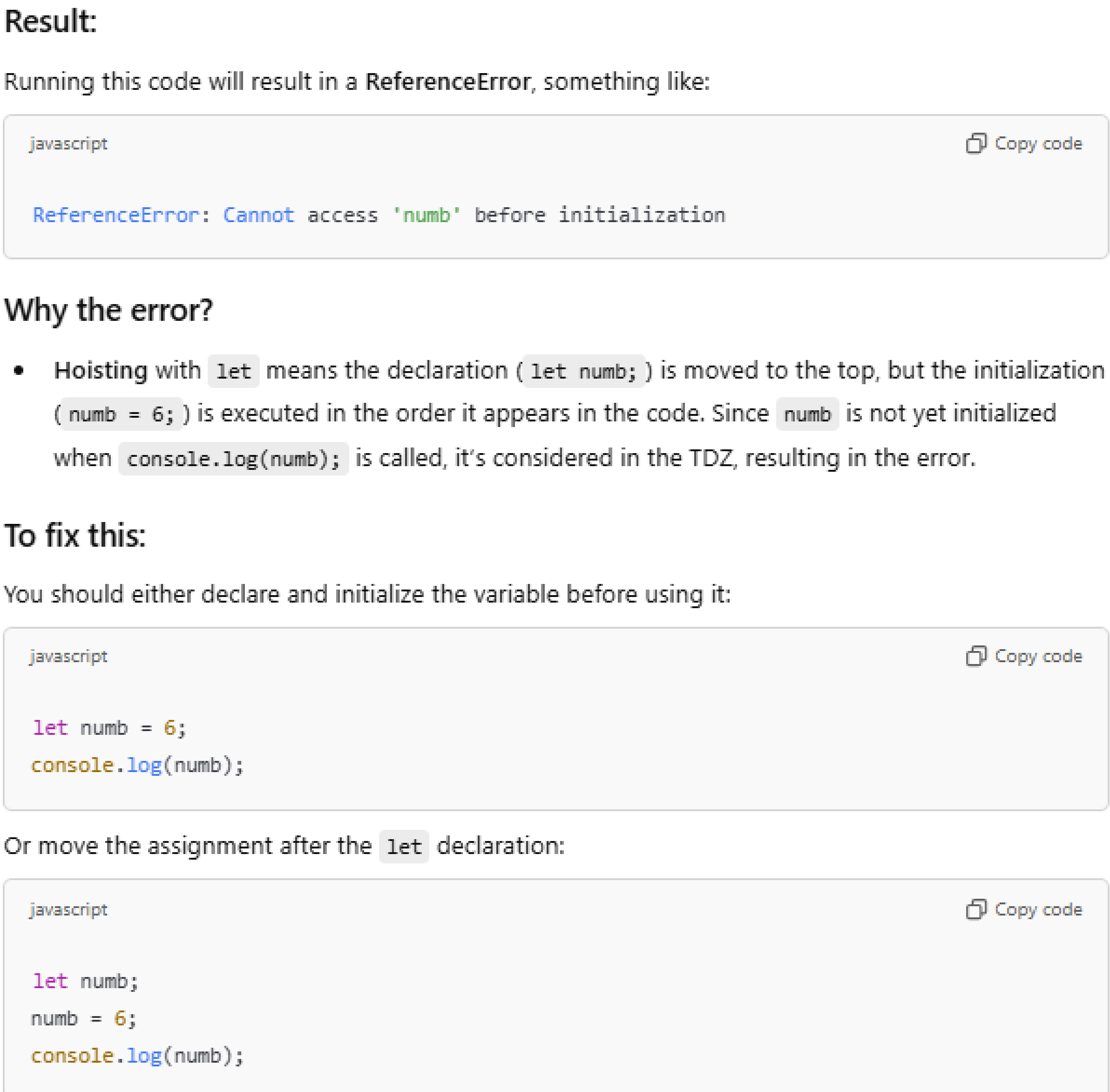
Explanation:

1. **Line 1:** `numb = 6;`
   - This assigns the value `6` to the variable `numb`. However, since `let numb;` comes later in the code, JavaScript doesn't know about the declaration of `numb` at this point due to hoisting behavior. `numb` is treated as being **hoisted** but uninitialized.
2. **Line 2:** `console.log(numb);`
   - At this point, JavaScript tries to log `numb` to the console. Since `let` declarations are **hoisted** but not initialized until the code execution reaches the declaration (`let numb;`), accessing `numb` before its initialization causes a **ReferenceError**.
   - The error occurs because `numb` is in the **temporal dead zone (TDZ)** from the start of the block until the point where it is declared. In the TDZ, the variable exists but can't be accessed.
3. **Line 3:** `let numb;`
   - Here, the variable `numb` is declared but remains uninitialized. The declaration itself is hoisted to the top of the scope, but its initialization (the assignment of `6` in line 1) happens later.

Result:

Running this code will result in a **ReferenceError**, something like:

```
ReferenceError: Cannot access 'numb' before initialization
```



Result:

Running this code will result in a **ReferenceError**, something like:

```
ReferenceError: Cannot access 'numb' before initialization
```

Why the error?

- **Hoisting** with `let` means the declaration (`let numb;`) is moved to the top, but the initialization (`numb = 6;`) is executed in the order it appears in the code. Since `numb` is not yet initialized when `console.log(numb);` is called, it's considered in the TDZ, resulting in the error.

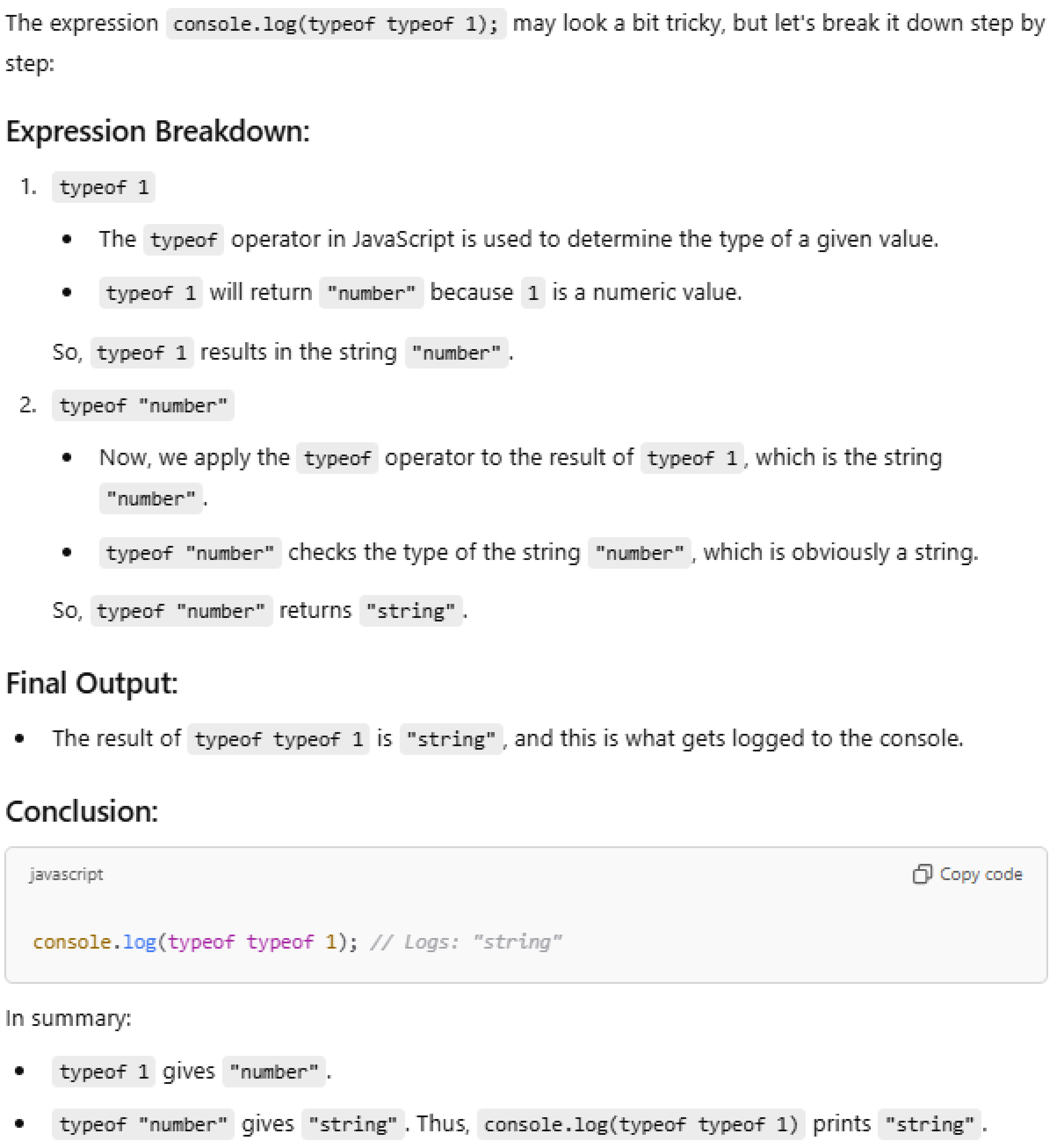
To fix this:

You should either declare and initialize the variable before using it:

```
let numb = 6;
console.log(numb);
```

Or move the assignment after the `let` declaration:

```
let numb;
numb = 6;
console.log(numb);
```



The expression `console.log(typeof typeof 1);` may look a bit tricky, but let's break it down step by step:

Expression Breakdown:

1. `typeof 1`

   - The `typeof` operator in JavaScript is used to determine the type of a given value.
   - `typeof 1` will return `"number"` because `1` is a numeric value.

   So, `typeof 1` results in the string `"number"`.

2. `typeof "number"`

   - Now, we apply the `typeof` operator to the result of `typeof 1`, which is the string `"number"`.
   - `typeof "number"` checks the type of the string `"number"`, which is obviously a string.

   So, `typeof "number"` returns `"string"`.

Final Output:

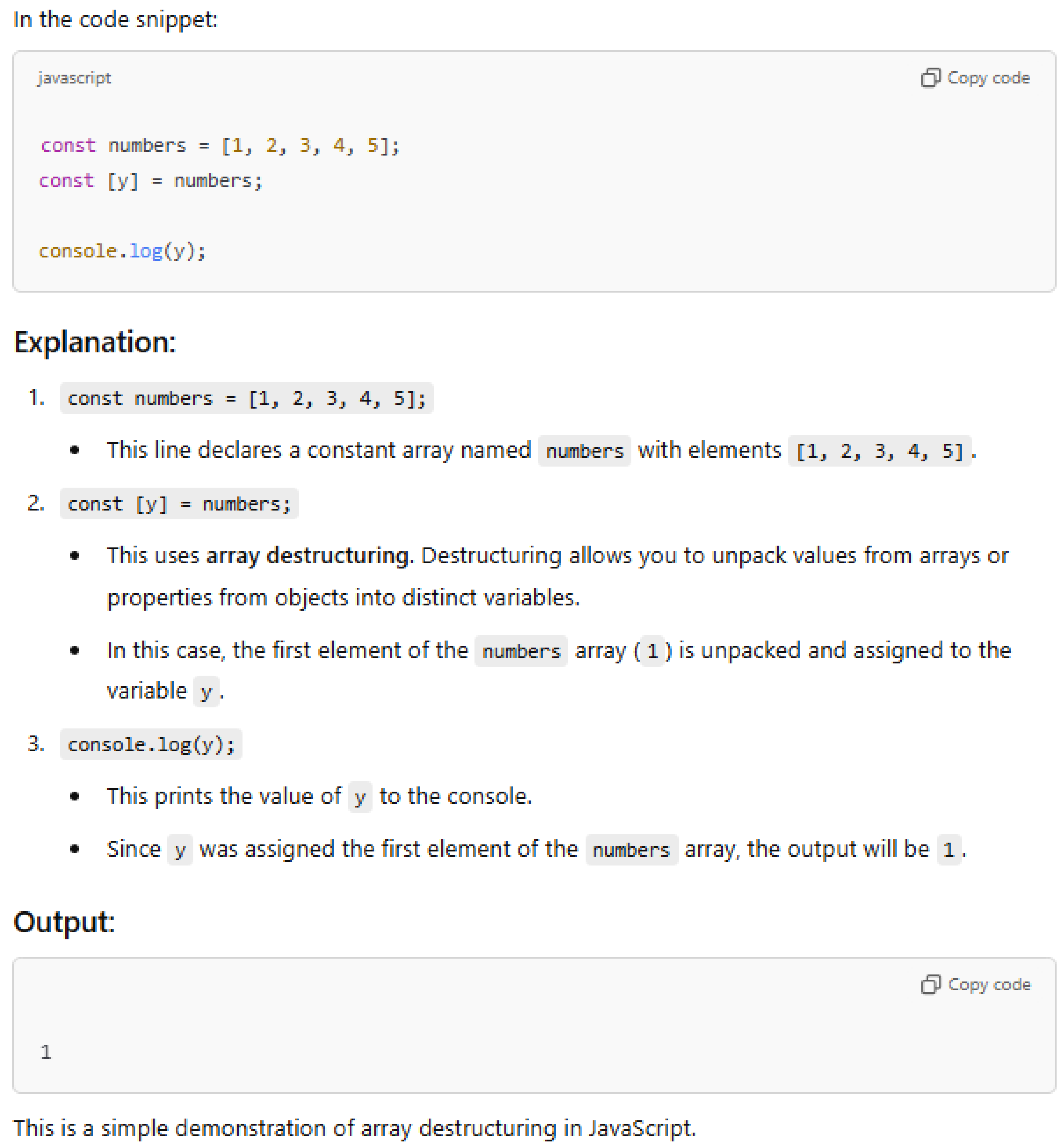
- The result of `typeof typeof 1` is `"string"`, and this is what gets logged to the console.

Conclusion:

```javascript
console.log(typeof typeof 1); // Logs: "string"
```

In summary:

- `typeof 1` gives `"number"`.
- `typeof "number"` gives `"string"`. Thus, `console.log(typeof typeof 1)` prints `"string"`.



In the code snippet:

```javascript
const numbers = [1, 2, 3, 4, 5];
const [y] = numbers;

console.log(y);
```

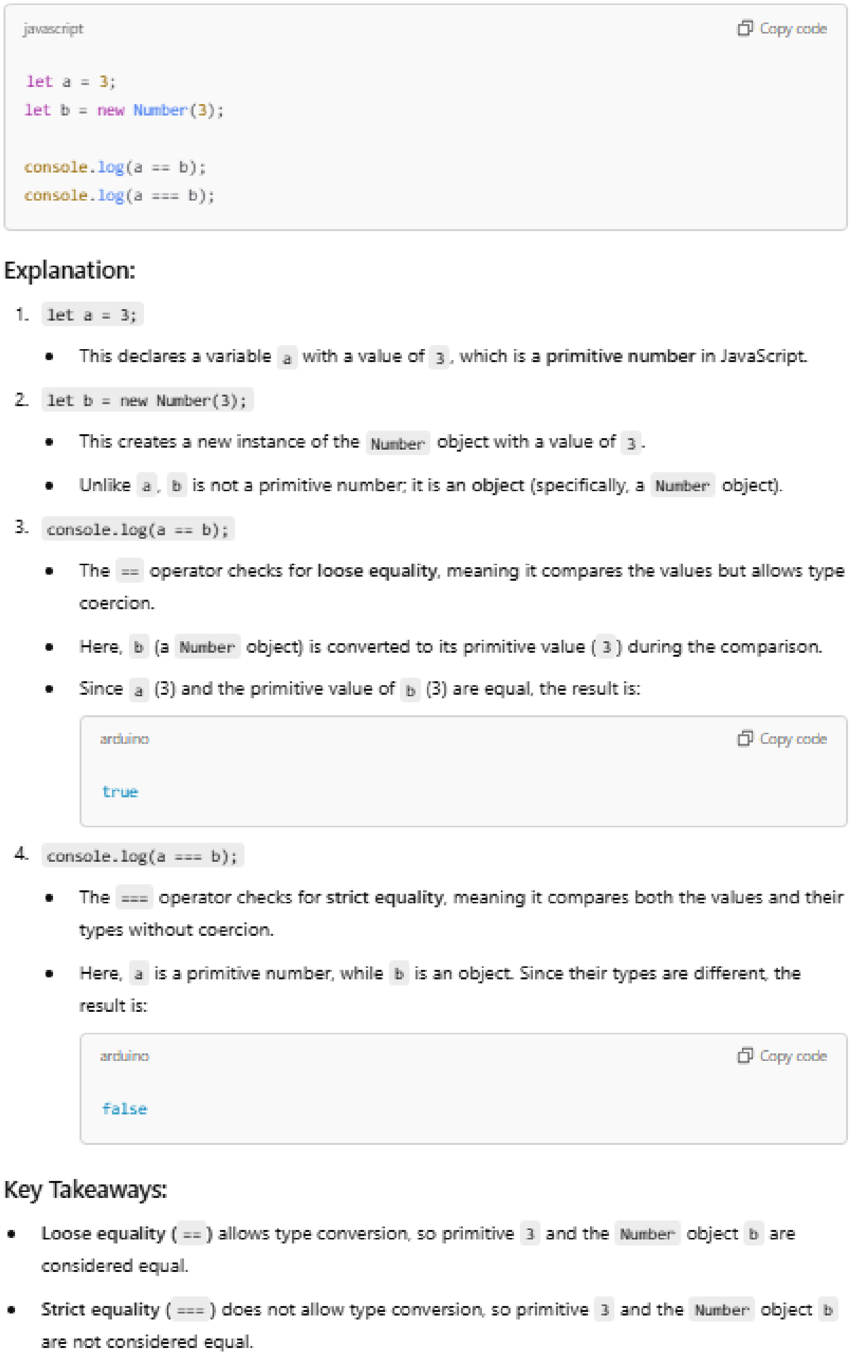
Explanation:

1. `const numbers = [1, 2, 3, 4, 5];`
   - This line declares a constant array named `numbers` with elements `[1, 2, 3, 4, 5]`.
2. `const [y] = numbers;`
   - This uses **array destructuring**. Destructuring allows you to unpack values from arrays or properties from objects into distinct variables.
   - In this case, the first element of the `numbers` array (`1`) is unpacked and assigned to the variable `y`.
3. `console.log(y);`
   - This prints the value of `y` to the console.
   - Since `y` was assigned the first element of the `numbers` array, the output will be `1`.

Output:

```
1
```

This is a simple demonstration of array destructuring in JavaScript.



```javascript
let a = 3;
let b = new Number(3);

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

Explanation:

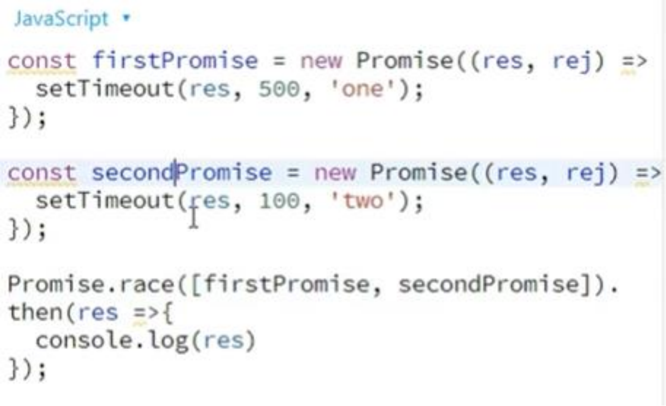
1. `let a = 3;`
   - This declares a variable `a` with a value of `3`, which is a **primitive number** in JavaScript.
2. `let b = new Number(3);`
   - This creates a new instance of the `Number` object with a value of `3`.
   - Unlike `a`, `b` is not a primitive number; it is an **object** (specifically, a `Number` object).
3. `console.log(a == b);`
   - The `==` operator checks for **loose equality**, meaning it compares the values but allows type coercion.
   - Here, `b` (a `Number` object) is converted to its primitive value (`3`) during the comparison.
   - Since `a` (3) and the primitive value of `b` (3) are equal, the result is:

     ```arduino
     true
     ```
4. `console.log(a === b);`
   - The `===` operator checks for **strict equality**, meaning it compares both the values and their types without coercion.
   - Here, `a` is a primitive number, while `b` is an object. Since their types are different, the result is:

     ```arduino
     false
     ```

Key Takeaways:

- **Loose equality** (`==`) allows type conversion, so primitive `3` and the `Number` object `b` are considered equal.
- **Strict equality** (`===`) does not allow type conversion, so primitive `3` and the `Number` object `b` are not considered equal.



```
const firstPromise = new Promise((res, rej) =>
  setTimeout(res, 500, 'one');
});

const secondPromise = new Promise((res, rej) =>
  setTimeout(res, 100, 'two');
});

Promise.race([firstPromise, secondPromise]).
then(res =>{
  console.log(res)
});
```

This code demonstrates the use of `Promise.race` in JavaScript, which resolves or rejects as soon as the first promise in the array settles (resolves or rejects), regardless of the outcome of the other promises.

Code Breakdown:

1. `const firstPromise = new Promise((res, rej) => { ... })`

- A new promise, `firstPromise`, is created.
- Inside the promise:
  - `setTimeout` is used to simulate an asynchronous operation.
  - After 500 milliseconds, the promise is resolved with the value `'one'`.

2. `const secondPromise = new Promise((res, rej) => { ... })`

- A new promise, `secondPromise`, is created.
- Inside the promise:
  - `setTimeout` is used to simulate an asynchronous operation.
  - After 100 milliseconds, the promise is resolved with the value `'two'`.

3. `Promise.race([firstPromise, secondPromise])`

- The `Promise.race` method takes an array of promises as input and returns a new promise.
- This new promise resolves or rejects as soon as the first promise in the array settles (either resolves or rejects).

4. `.then(res => { console.log(res); })`

- When the first promise settles (resolves or rejects), the `.then()` handler is triggered.
- The resolved value of the first settled promise is passed as an argument to the callback function.

4. `.then(res => { console.log(res); })`

- When the first promise settles (resolves or rejects), the `.then()` handler is triggered.
- The resolved value of the first settled promise is passed as an argument to the callback function.

What Happens in This Code:

- Both `firstPromise` and `secondPromise` start running simultaneously.
- `secondPromise` resolves first because it is set to resolve after 100 milliseconds, whereas `firstPromise` resolves after 500 milliseconds.
- Since `Promise.race` resolves with the first settled promise, the value `'two'` (from `secondPromise`) is returned.

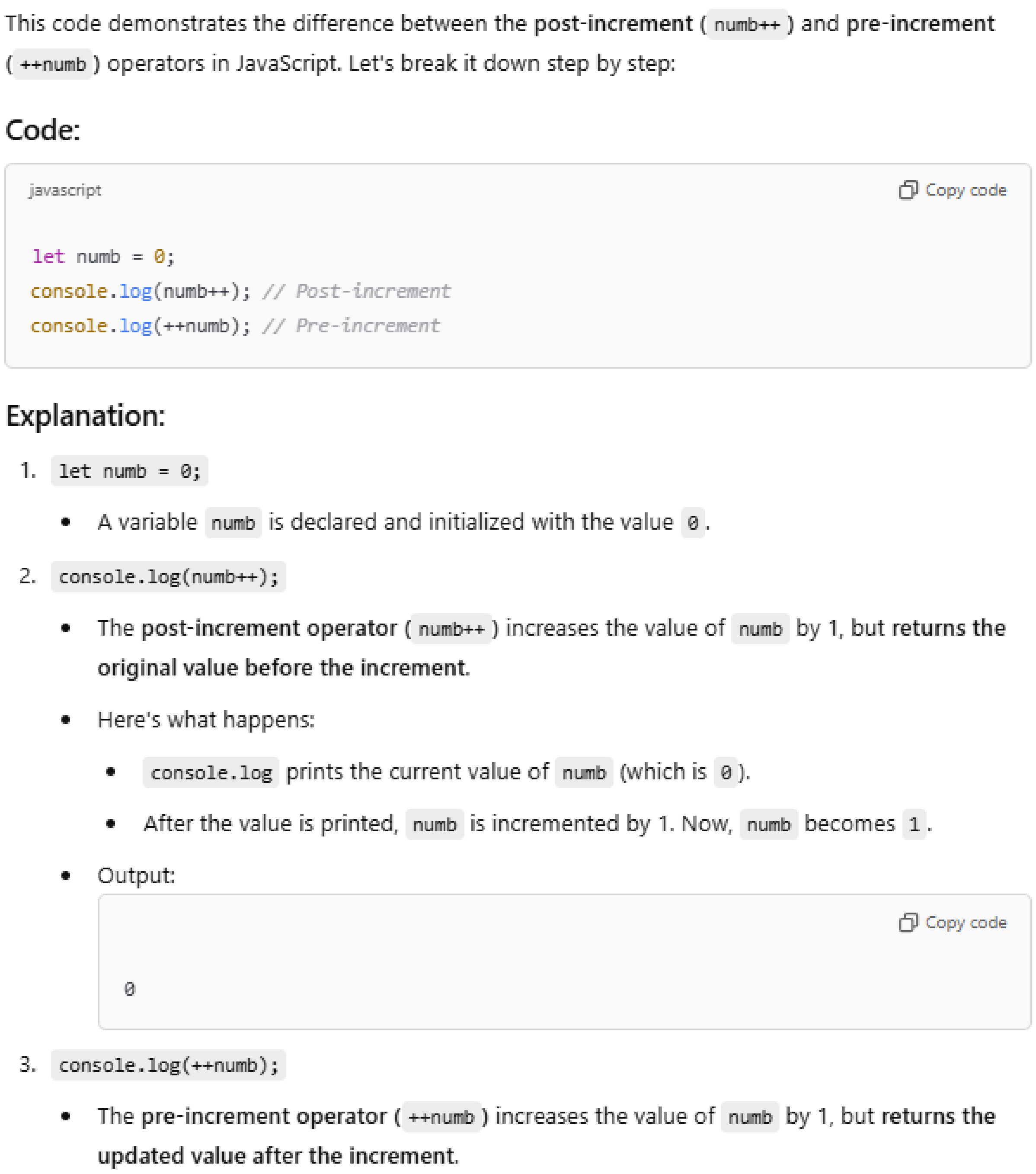
Output:

Copy code

```
two
```

Key Points:

1. `Promise.race` resolves or rejects with the first settled promise in the array.
2. The order of resolution/rejection depends on the time taken by the individual promises to settle.
3. Even though `firstPromise` resolves later, it doesn't affect the result because `Promise.race` only cares about the first settled promise.



This code demonstrates the difference between the **post-increment** (`numb++`) and **pre-increment** (`++numb`) operators in JavaScript. Let's break it down step by step:

Code:

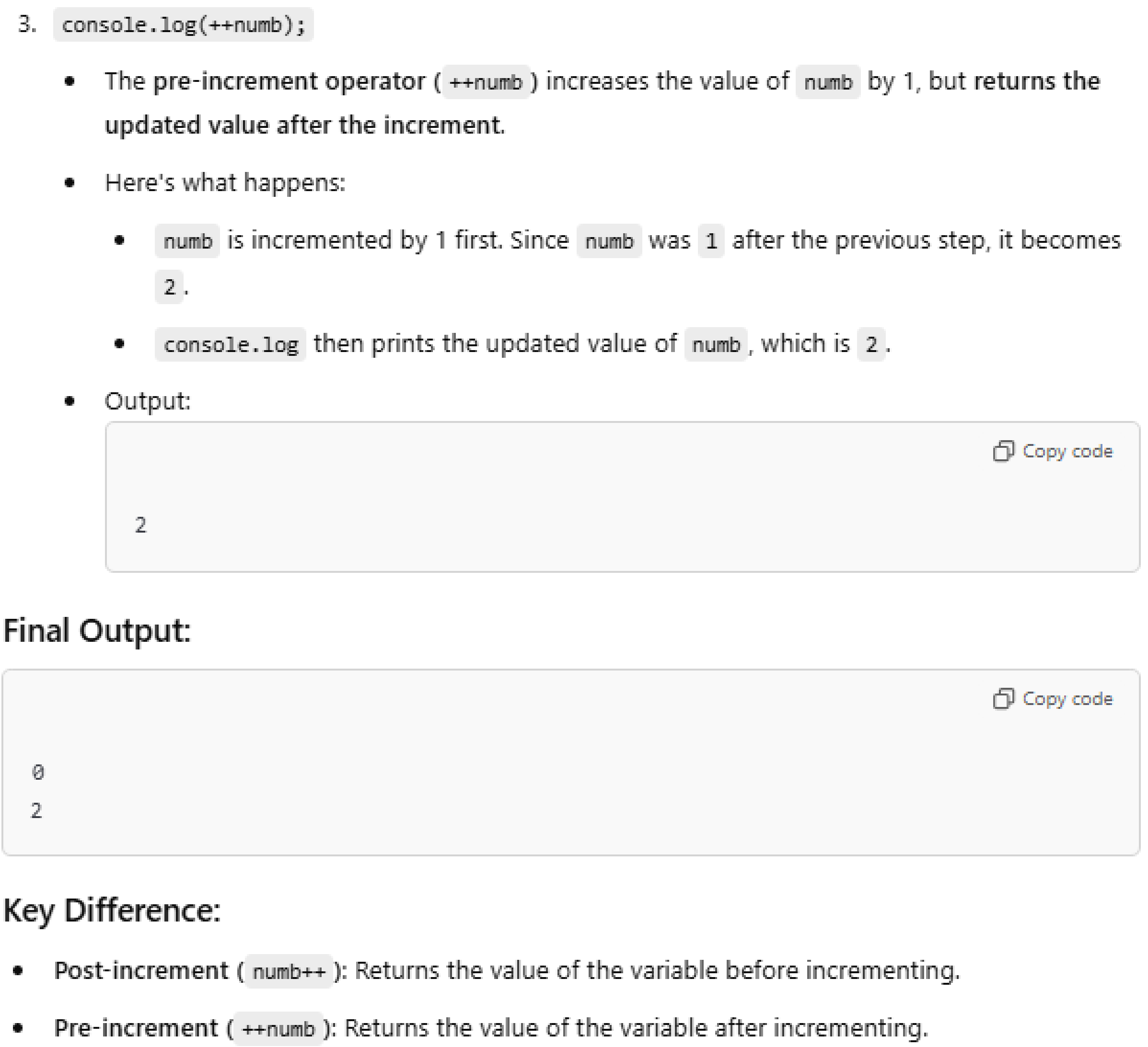
```javascript
let numb = 0;
console.log(numb++); // Post-increment
console.log(++numb); // Pre-increment
```

Explanation:

1. `let numb = 0;`
   - A variable `numb` is declared and initialized with the value `0`.
2. `console.log(numb++);`
   - The **post-increment operator** (`numb++`) increases the value of `numb` by 1, but **returns the original value before the increment.**
   - Here's what happens:
     - `console.log` prints the current value of `numb` (which is `0`).
     - After the value is printed, `numb` is incremented by 1. Now, `numb` becomes `1`.
   - Output:

```
0
```

3. `console.log(++numb);`
   - The **pre-increment operator** (`++numb`) increases the value of `numb` by 1, but **returns the updated value after the increment.**



3. `console.log(++numb);`
   - The **pre-increment operator** (`++numb`) increases the value of `numb` by 1, but **returns the updated value after the increment.**

   - Here's what happens:
     - `numb` is incremented by 1 first. Since `numb` was `1` after the previous step, it becomes `2`.
     - `console.log` then prints the updated value of `numb`, which is `2`.
   - Output:

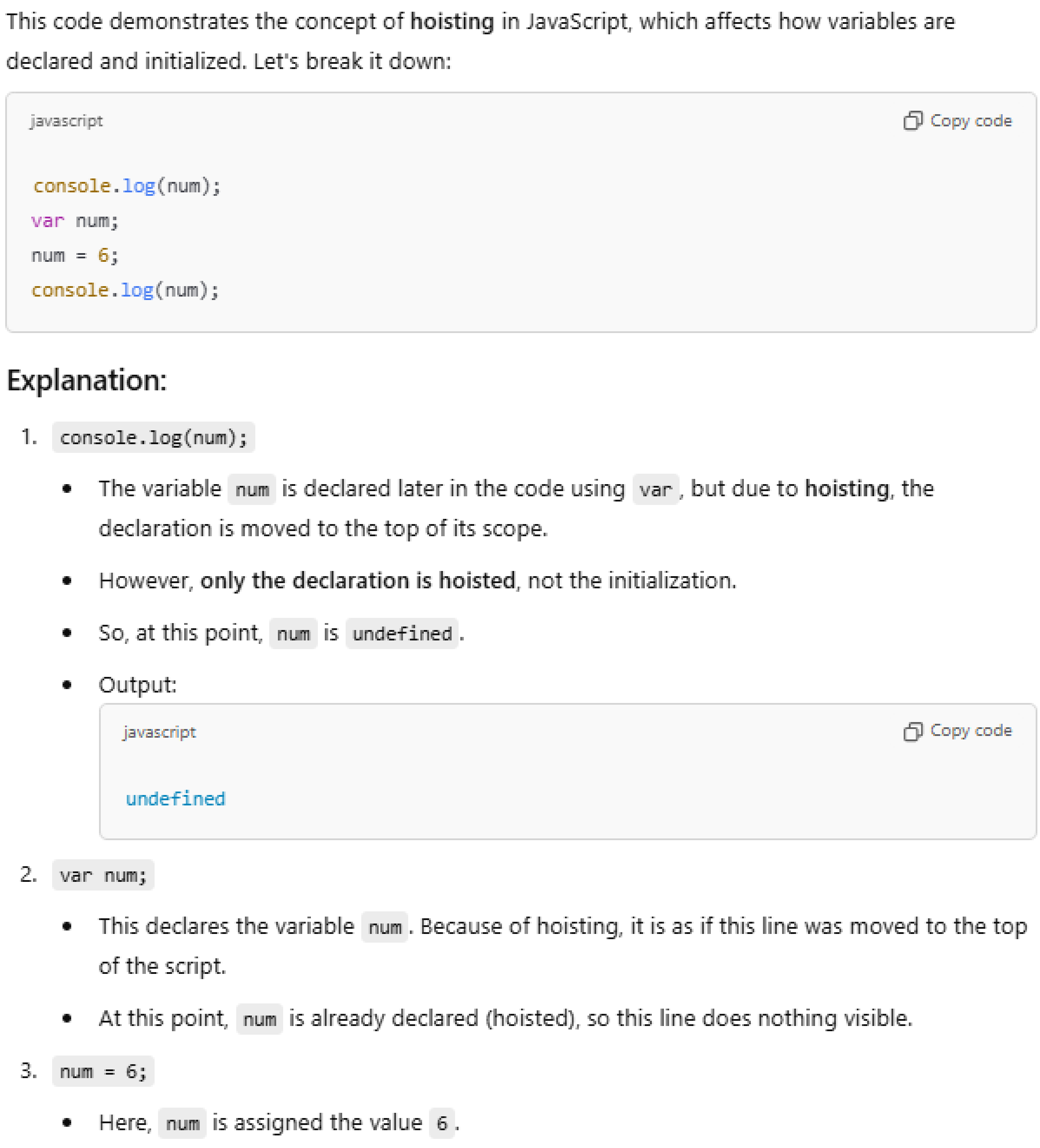
```
2
```

Final Output:

```
0
2
```

Key Difference:

- **Post-increment** (`numb++`): Returns the value of the variable before incrementing.
- **Pre-increment** (`++numb`): Returns the value of the variable after incrementing.



This code demonstrates the concept of **hoisting** in JavaScript, which affects how variables are declared and initialized. Let's break it down:

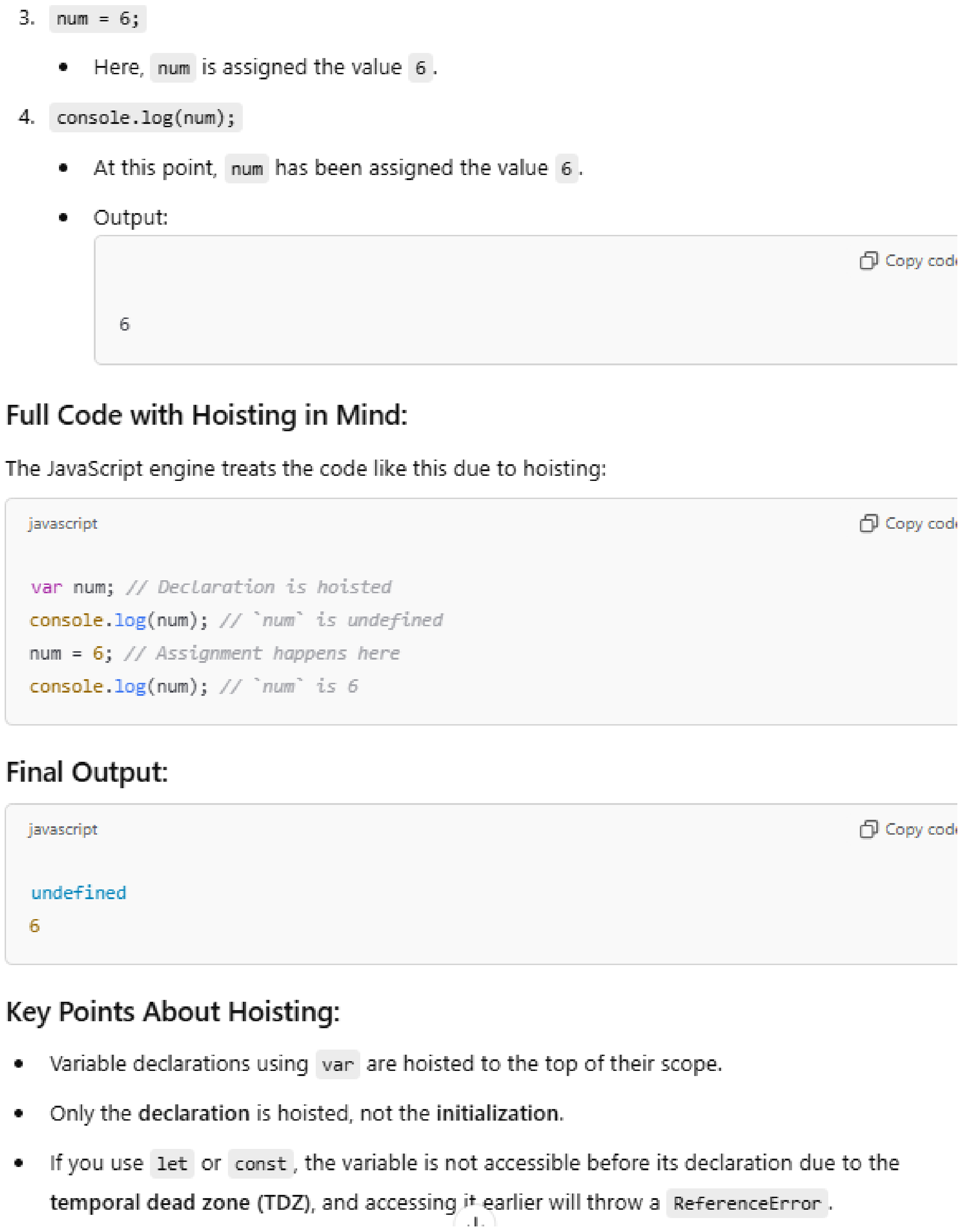
```javascript
console.log(num);
var num;
num = 6;
console.log(num);
```

Explanation:

1. `console.log(num);`
   - The variable `num` is declared later in the code using `var`, but due to **hoisting**, the declaration is moved to the top of its scope.
   - However, **only the declaration is hoisted**, not the initialization.
   - So, at this point, `num` is `undefined`.
   - Output:

```javascript
undefined
```

2. `var num;`
   - This declares the variable `num`. Because of hoisting, it is as if this line was moved to the top of the script.
   - At this point, `num` is already declared (hoisted), so this line does nothing visible.
3. `num = 6;`
   - Here, `num` is assigned the value `6`.



3. `num = 6;`
   - Here, `num` is assigned the value `6`.

4. `console.log(num);`
   - At this point, `num` has been assigned the value `6`.
   - Output:

```
6
```

Full Code with Hoisting in Mind:

The JavaScript engine treats the code like this due to hoisting:

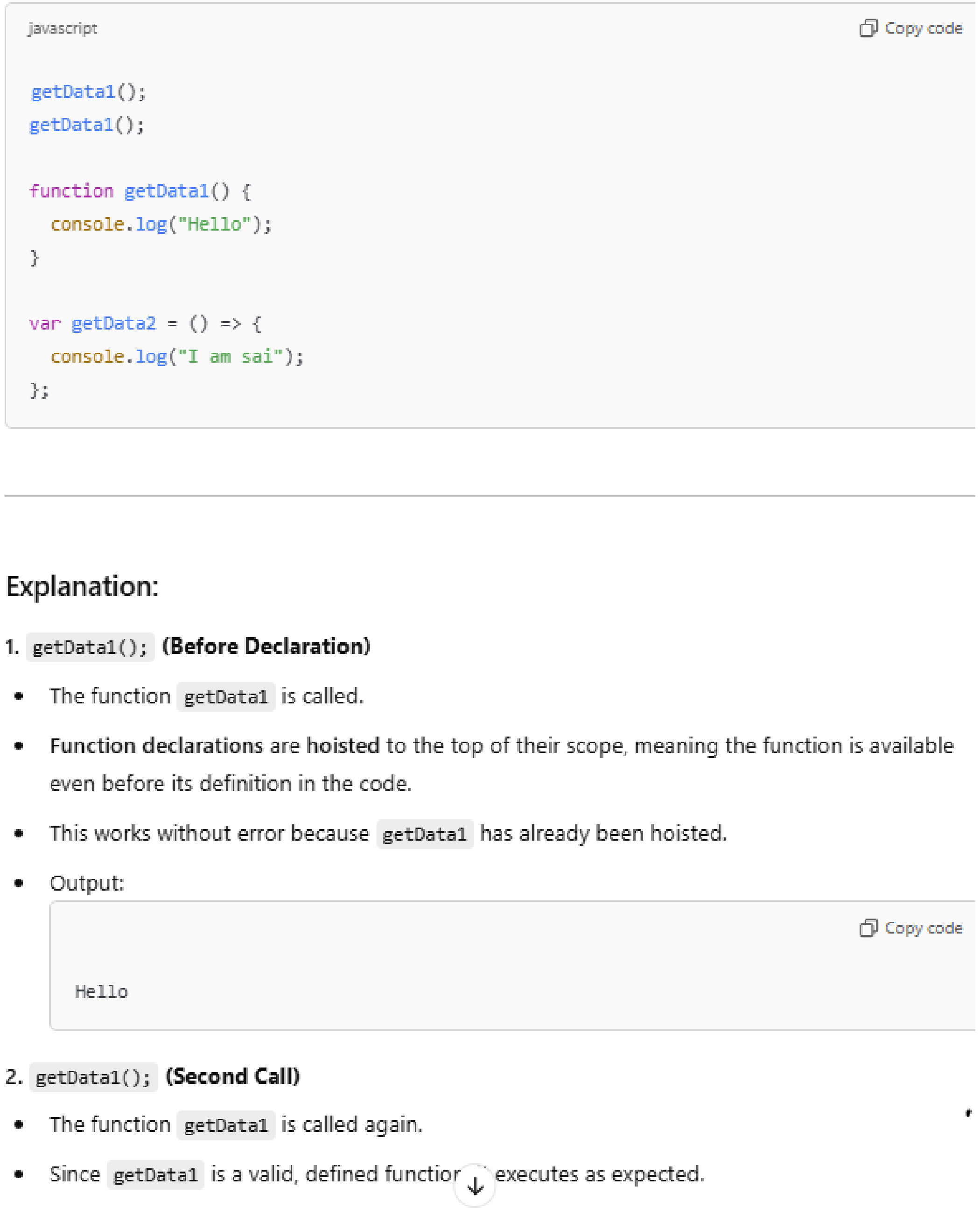
```javascript
var num; // Declaration is hoisted
console.log(num); // `num` is undefined
num = 6; // Assignment happens here
console.log(num); // `num` is 6
```

Final Output:

```javascript
undefined
6
```

Key Points About Hoisting:

- Variable declarations using `var` are hoisted to the top of their scope.
- Only the **declaration** is hoisted, not the **initialization**.
- If you use `let` or `const`, the variable is not accessible before its declaration due to the **temporal dead zone** (TDZ), and accessing it earlier will throw a `ReferenceError`.



```javascript
getData1();
getData1();

function getData1() {
  console.log("Hello");
}

var getData2 = () => {
  console.log("I am sai");
};
```

Explanation:

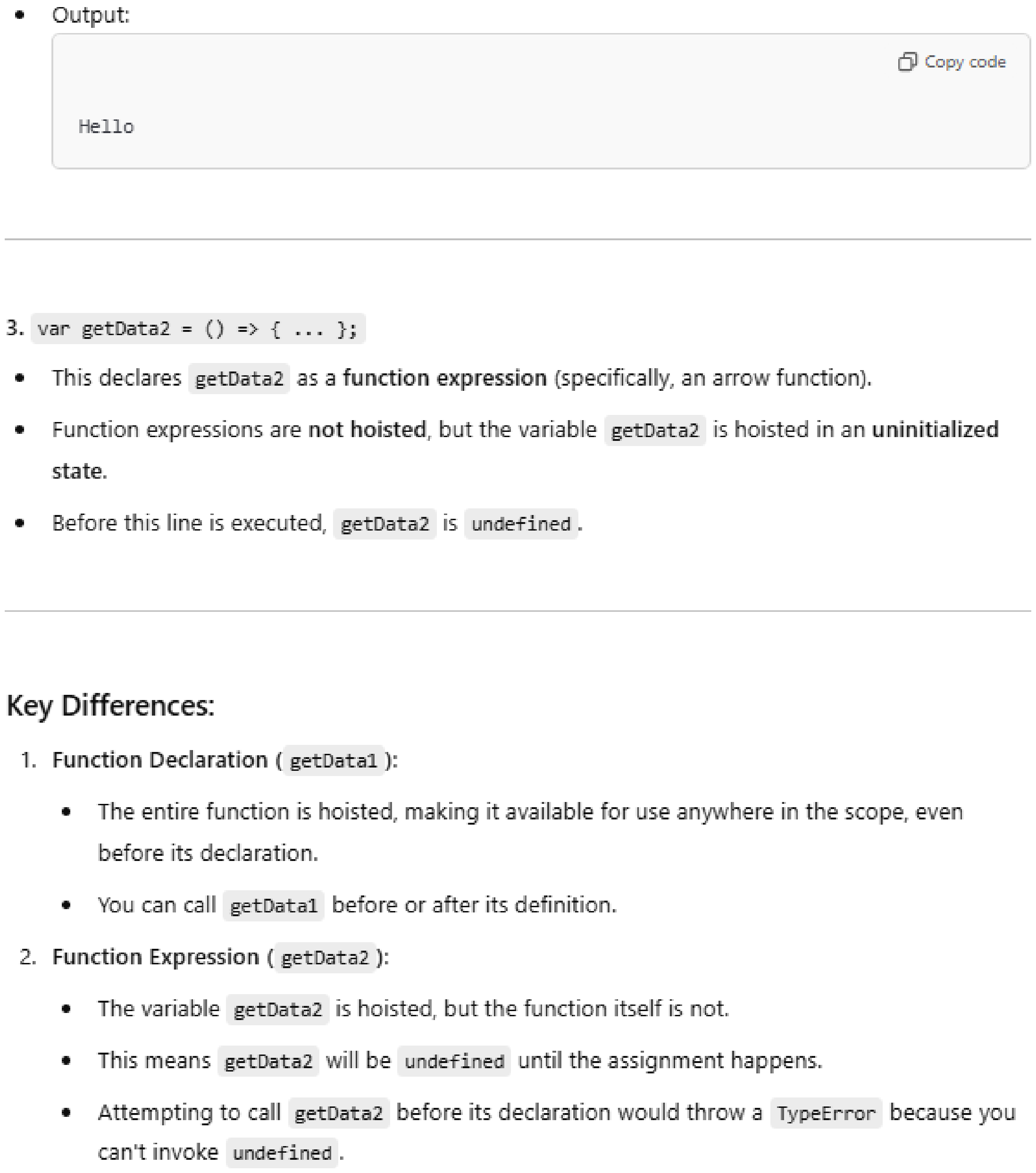
1. `getData1();` **(Before Declaration)**

- The function `getData1` is called.
- **Function declarations** are **hoisted** to the top of their scope, meaning the function is available even before its definition in the code.
- This works without error because `getData1` has already been hoisted.
- Output:

```
Hello
```

2. `getData1();` **(Second Call)**

- The function `getData1` is called again.
- Since `getData1` is a valid, defined function, it executes as expected.



- Output:

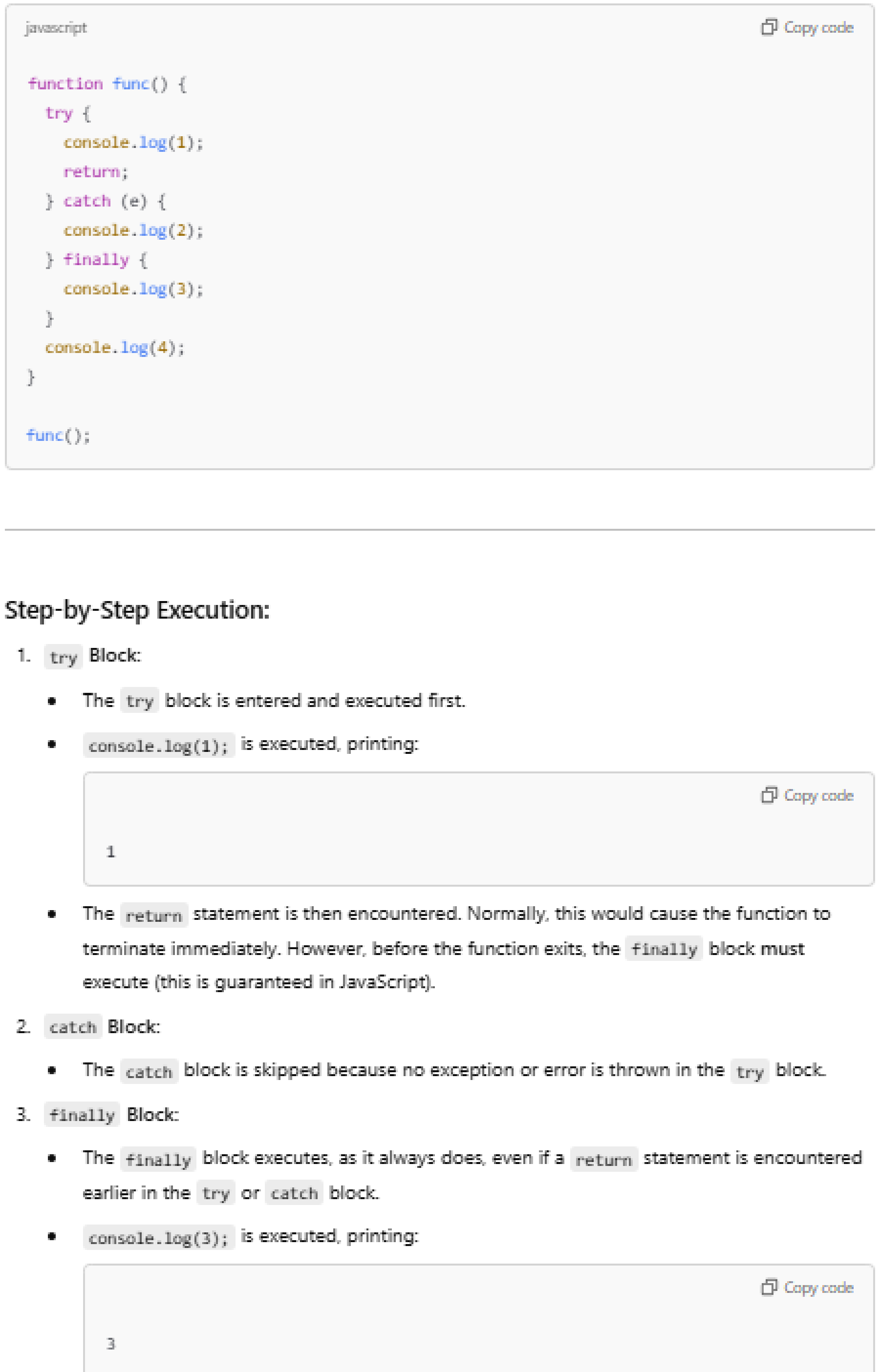
```
Hello
```

3. `var getData2 = () => { ... };`

- This declares `getData2` as a **function expression** (specifically, an arrow function).
- Function expressions are **not hoisted**, but the variable `getData2` is hoisted in an **uninitialized state.**
- Before this line is executed, `getData2` is `undefined`.

Key Differences:

1. **Function Declaration (** `getData1` **)**:
   - The entire function is hoisted, making it available for use anywhere in the scope, even before its declaration.
   - You can call `getData1` before or after its definition.
2. **Function Expression (** `getData2` **)**:
   - The variable `getData2` is hoisted, but the function itself is not.
   - This means `getData2` will be `undefined` until the assignment happens.
   - Attempting to call `getData2` before its declaration would throw a `TypeError` because you can't invoke `undefined`.



```javascript
function func() {
  try {
    console.log(1);
    return;
  } catch (e) {
    console.log(2);
  } finally {
    console.log(3);
  }
  console.log(4);
}

func();
```

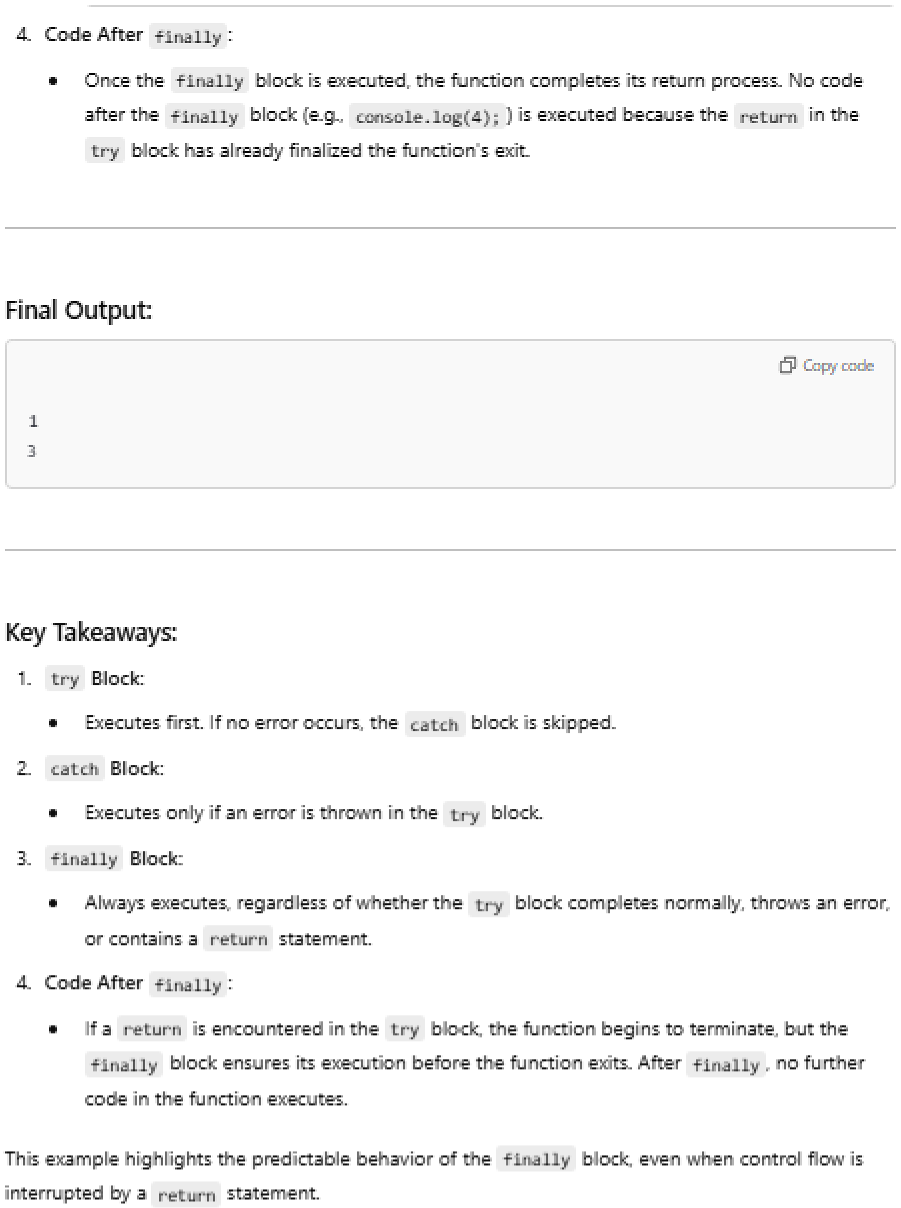
Step-by-Step Execution:

1. `try` Block:
   - The `try` block is entered and executed first.
   - `console.log(1);` is executed, printing:

     ```
     1
     ```

   - The `return` statement is then encountered. Normally, this would cause the function to terminate immediately. However, before the function exits, the `finally` block must execute (this is guaranteed in JavaScript).
2. `catch` Block:
   - The `catch` block is skipped because no exception or error is thrown in the `try` block.
3. `finally` Block:
   - The `finally` block executes, as it always does, even if a `return` statement is encountered earlier in the `try` or `catch` block.
   - `console.log(3);` is executed, printing:

     ```
     3
     ```



4. Code After `finally`:
   - Once the `finally` block is executed, the function completes its return process. No code after the `finally` block (e.g., `console.log(4);`) is executed because the `return` in the `try` block has already finalized the function's exit.

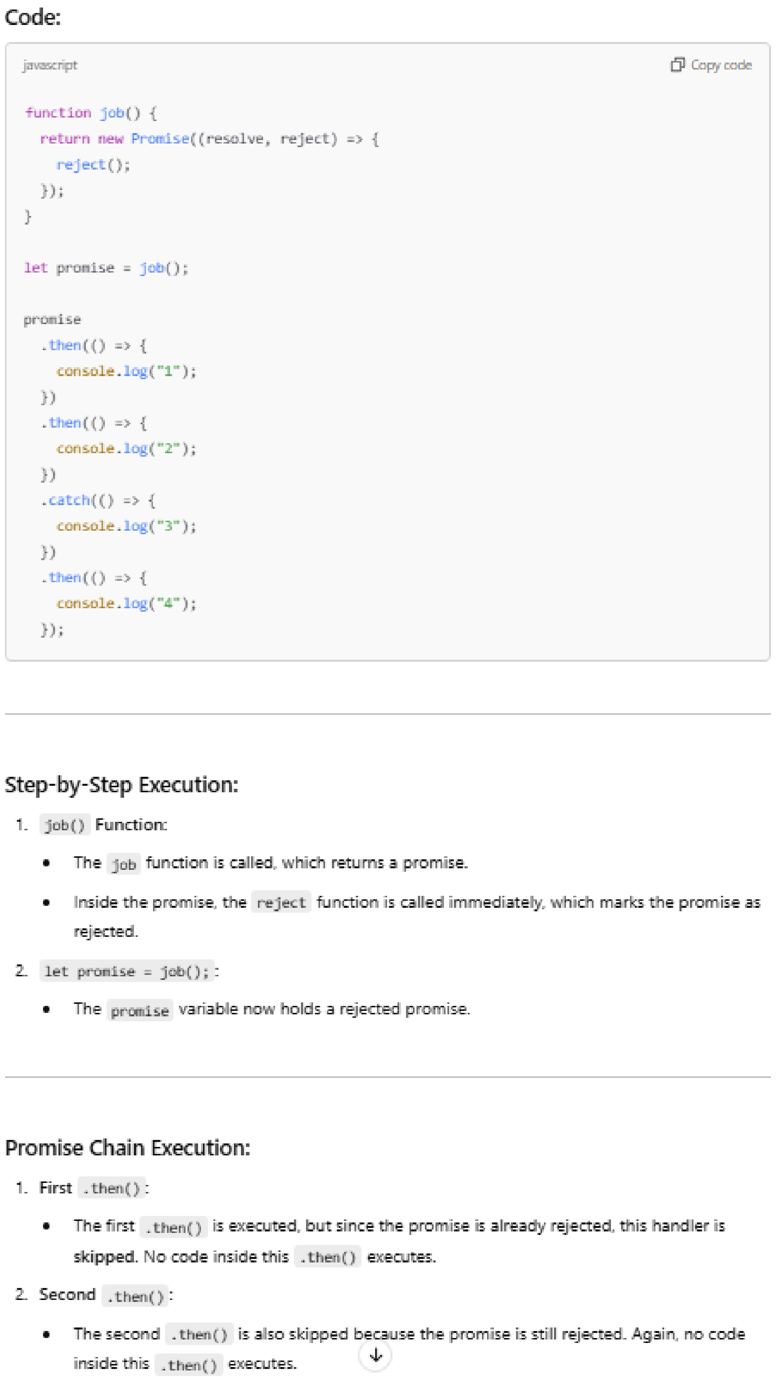
Final Output:

```
1
3
```

Key Takeaways:

1. `try` Block:
   - Executes first. If no error occurs, the `catch` block is skipped.
2. `catch` Block:
   - Executes only if an error is thrown in the `try` block.
3. `finally` Block:
   - Always executes, regardless of whether the `try` block completes normally, throws an error, or contains a `return` statement.
4. Code After `finally`:
   - If a `return` is encountered in the `try` block, the function begins to terminate, but the `finally` block ensures its execution before the function exits. After `finally`, no further code in the function executes.

This example highlights the predictable behavior of the `finally` block, even when control flow is interrupted by a `return` statement.



Code:

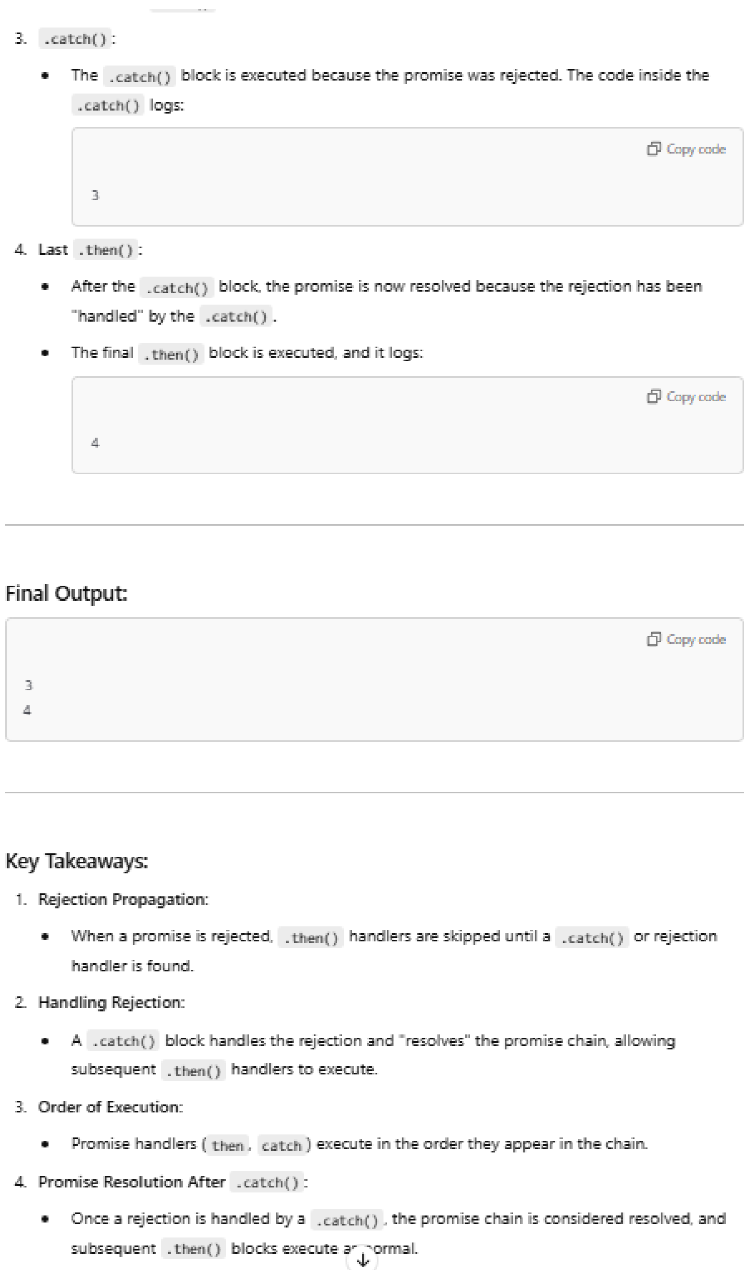
```javascript
function job() {
  return new Promise((resolve, reject) => {
    reject();
  });
}

let promise = job();

promise
  .then(() => {
    console.log("1");
  })
  .then(() => {
    console.log("2");
  })
  .catch(() => {
    console.log("3");
  })
  .then(() => {
    console.log("4");
  });
```

Step-by-Step Execution:

1. `job()` Function:
   - The `job` function is called, which returns a promise.
   - Inside the promise, the `reject` function is called immediately, which marks the promise as rejected.
2. `let promise = job();`:
   - The `promise` variable now holds a rejected promise.

Promise Chain Execution:

1. First `.then()`:
   - The first `.then()` is executed, but since the promise is already rejected, this handler is **skipped**. No code inside this `.then()` executes.
2. Second `.then()`:
   - The second `.then()` is also skipped because the promise is still rejected. Again, no code inside this `.then()` executes.



3. `.catch()`:
   - The `.catch()` block is executed because the promise was rejected. The code inside the `.catch()` logs:

```
3
```

4. Last `.then()`:
   - After the `.catch()` block, the promise is now resolved because the rejection has been "handled" by the `.catch()`.
   - The final `.then()` block is executed, and it logs:

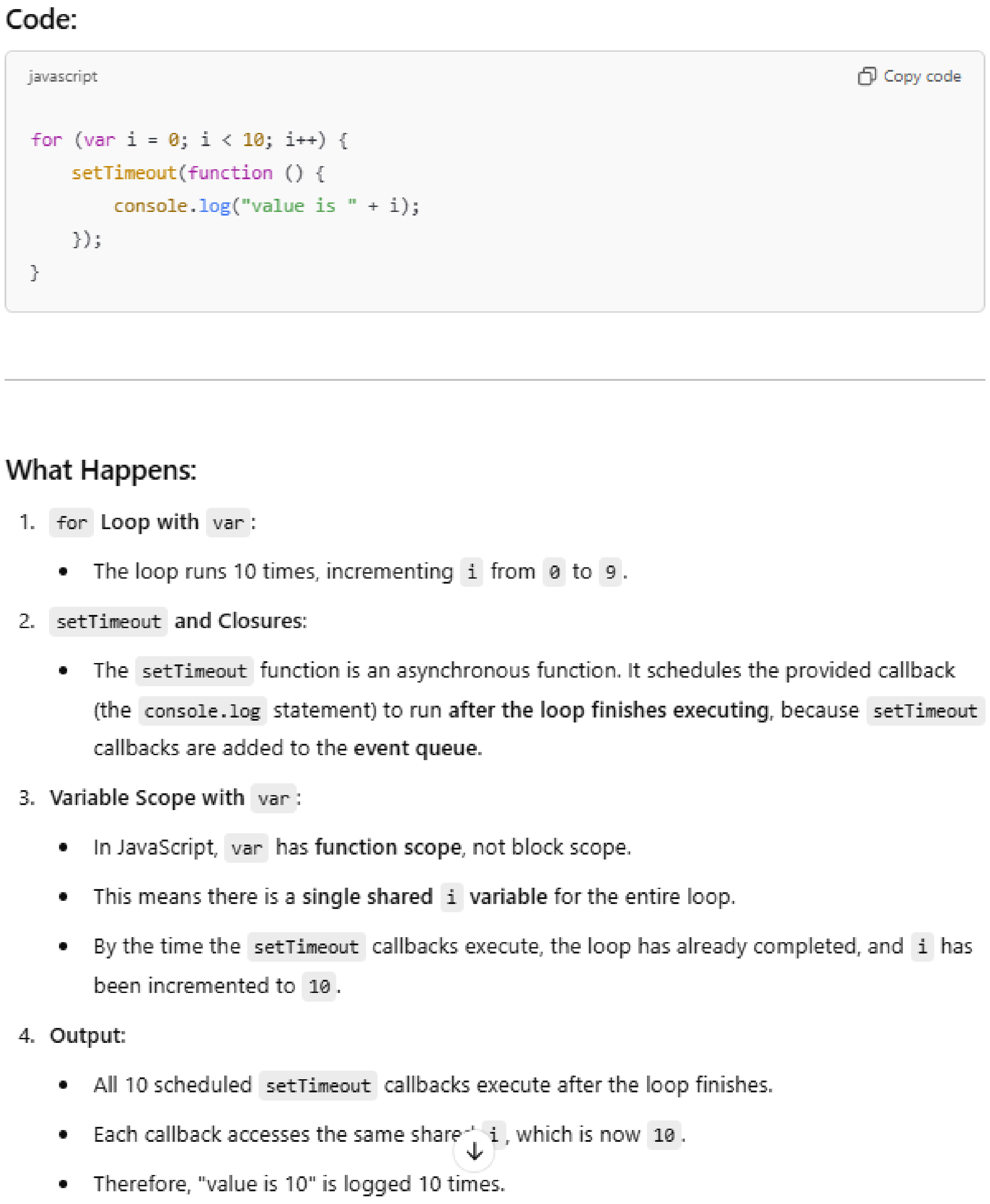
```
4
```

Final Output:

```
3
4
```

Key Takeaways:

1. Rejection Propagation:
   - When a promise is rejected, `.then()` handlers are skipped until a `.catch()` or rejection handler is found.
2. Handling Rejection:
   - A `.catch()` block handles the rejection and "resolves" the promise chain, allowing subsequent `.then()` handlers to execute.
3. Order of Execution:
   - Promise handlers (`then`, `catch`) execute in the order they appear in the chain.
4. Promise Resolution After `.catch()`:
   - Once a rejection is handled by a `.catch()`, the promise chain is considered resolved, and subsequent `.then()` blocks execute as normal.

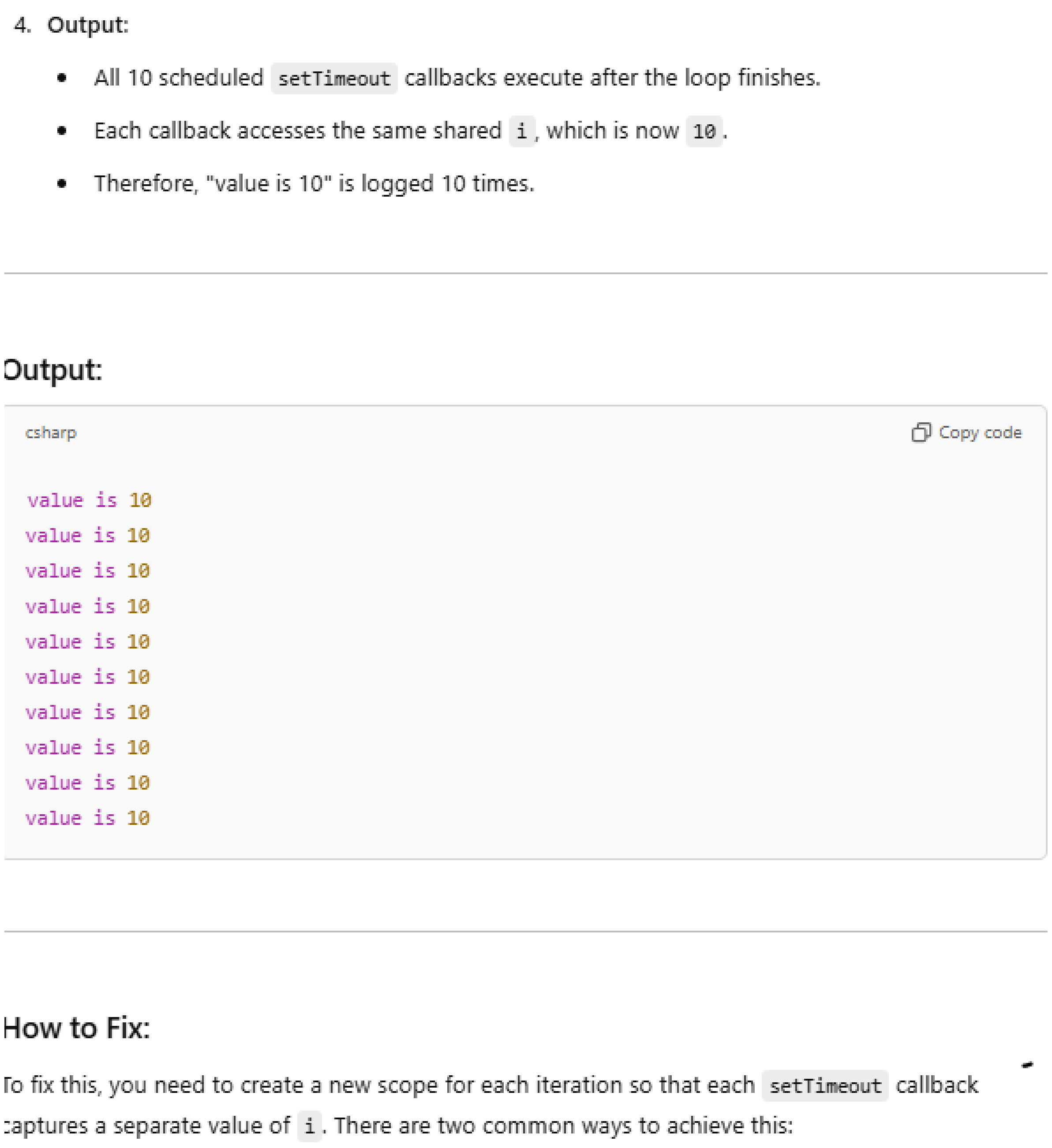


Code:

```javascript
for (var i = 0; i < 10; i++) {
    setTimeout(function () {
        console.log("value is " + i);
    });
}
```

What Happens:

1. `for` **Loop with** `var`:
   - The loop runs 10 times, incrementing `i` from `0` to `9`.
2. `setTimeout` **and Closures:**
   - The `setTimeout` function is an asynchronous function. It schedules the provided callback (the `console.log` statement) to run **after the loop finishes executing**, because `setTimeout` callbacks are added to the **event queue.**
3. **Variable Scope with** `var`:
   - In JavaScript, `var` has **function scope**, not block scope.
   - This means there is a **single shared** `i` **variable** for the entire loop.
   - By the time the `setTimeout` callbacks execute, the loop has already completed, and `i` has been incremented to `10`.
4. **Output:**
   - All 10 scheduled `setTimeout` callbacks execute after the loop finishes.
   - Each callback accesses the same shared `i`, which is now `10`.
   - Therefore, "value is 10" is logged 10 times.



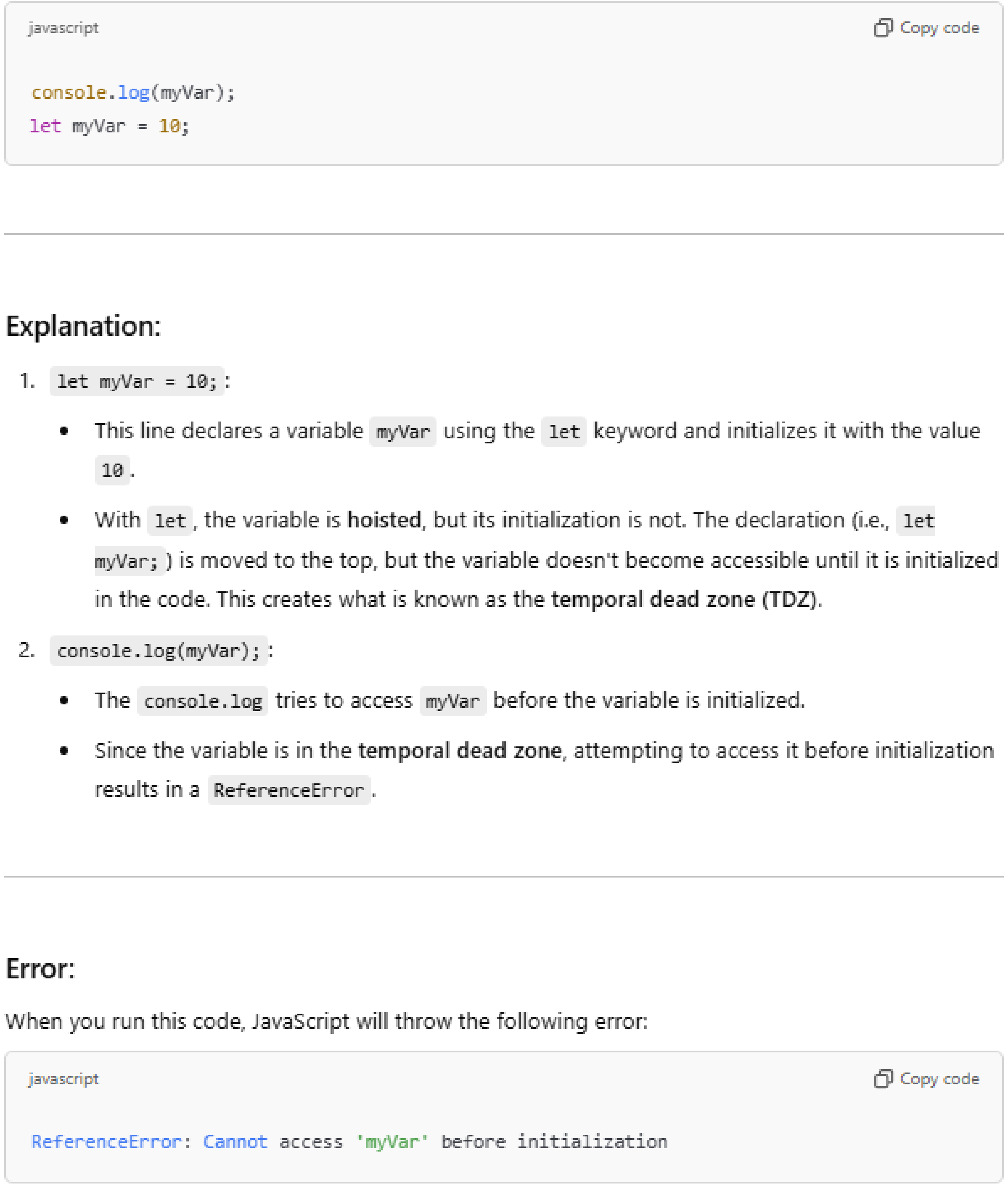
4. **Output:**
   - All 10 scheduled `setTimeout` callbacks execute after the loop finishes.
   - Each callback accesses the same shared `i`, which is now `10`.
   - Therefore, "value is 10" is logged 10 times.

Output:

```csharp
value is 10
value is 10
value is 10
value is 10
value is 10
value is 10
value is 10
value is 10
value is 10
value is 10
```

How to Fix:

To fix this, you need to create a new scope for each iteration so that each `setTimeout` callback captures a separate value of `i`. There are two common ways to achieve this:



javascript

Copy code

```javascript
console.log(myVar);
let myVar = 10;
```

Explanation:

1. `let myVar = 10;`:
   - This line declares a variable `myVar` using the `let` keyword and initializes it with the value `10`.
   - With `let`, the variable is **hoisted**, but its initialization is not. The declaration (i.e., `let myVar;`) is moved to the top, but the variable doesn't become accessible until it is initialized in the code. This creates what is known as the **temporal dead zone** (TDZ).
2. `console.log(myVar);`:
   - The `console.log` tries to access `myVar` before the variable is initialized.
   - Since the variable is in the **temporal dead zone**, attempting to access it before initialization results in a `ReferenceError`.

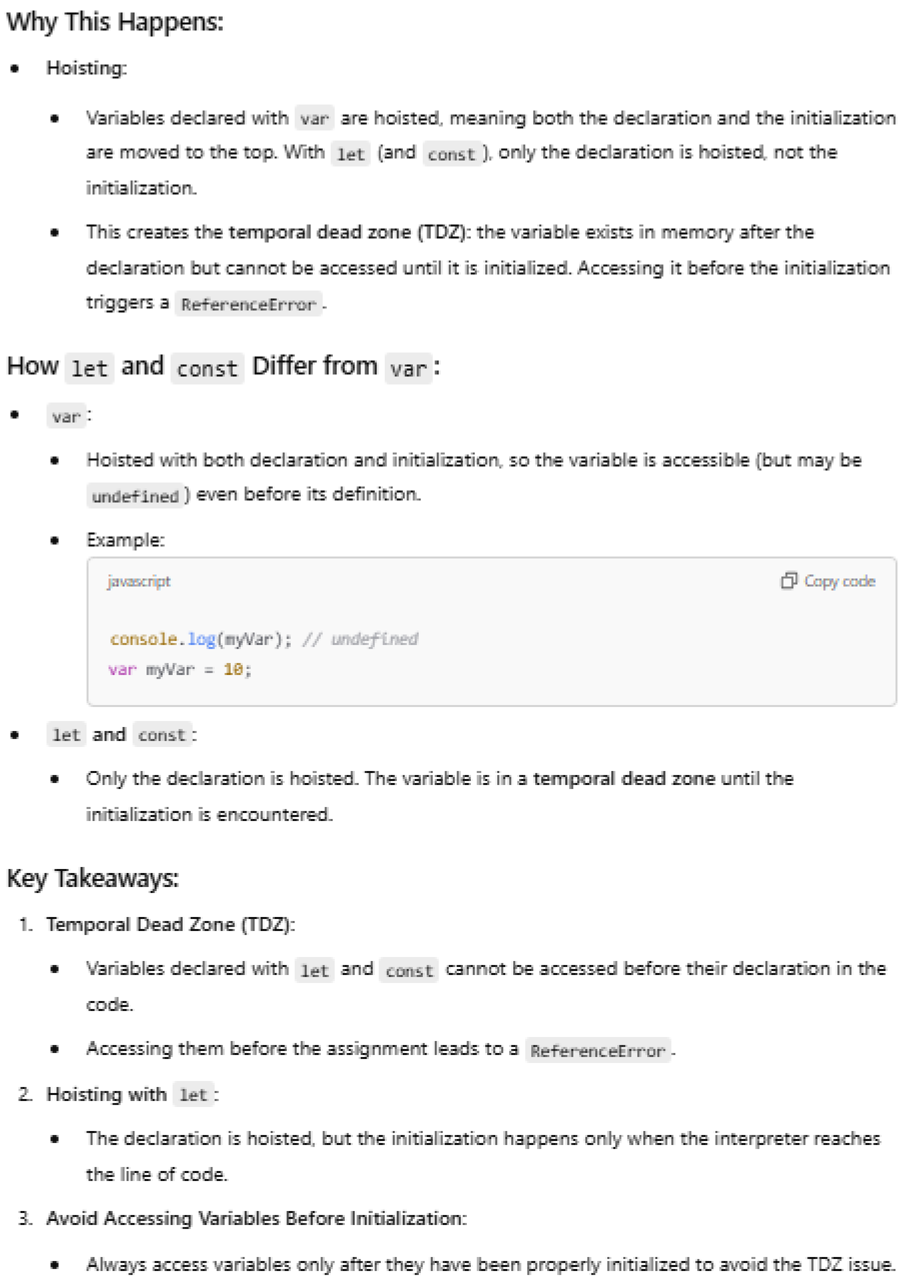
Error:

When you run this code, JavaScript will throw the following error:

javascript

Copy code

```javascript
ReferenceError: Cannot access 'myVar' before initialization
```



Why This Happens:

- **Hoisting:**
  - Variables declared with `var` are hoisted, meaning both the declaration and the initialization are moved to the top. With `let` (and `const`), only the declaration is hoisted, not the initialization.
  - This creates the **temporal dead zone (TDZ)**: the variable exists in memory after the declaration but cannot be accessed until it is initialized. Accessing it before the initialization triggers a `ReferenceError`.

How `let` and `const` Differ from `var`:

- `var`:
  - Hoisted with both declaration and initialization, so the variable is accessible (but may be `undefined`) even before its definition.
  - Example:

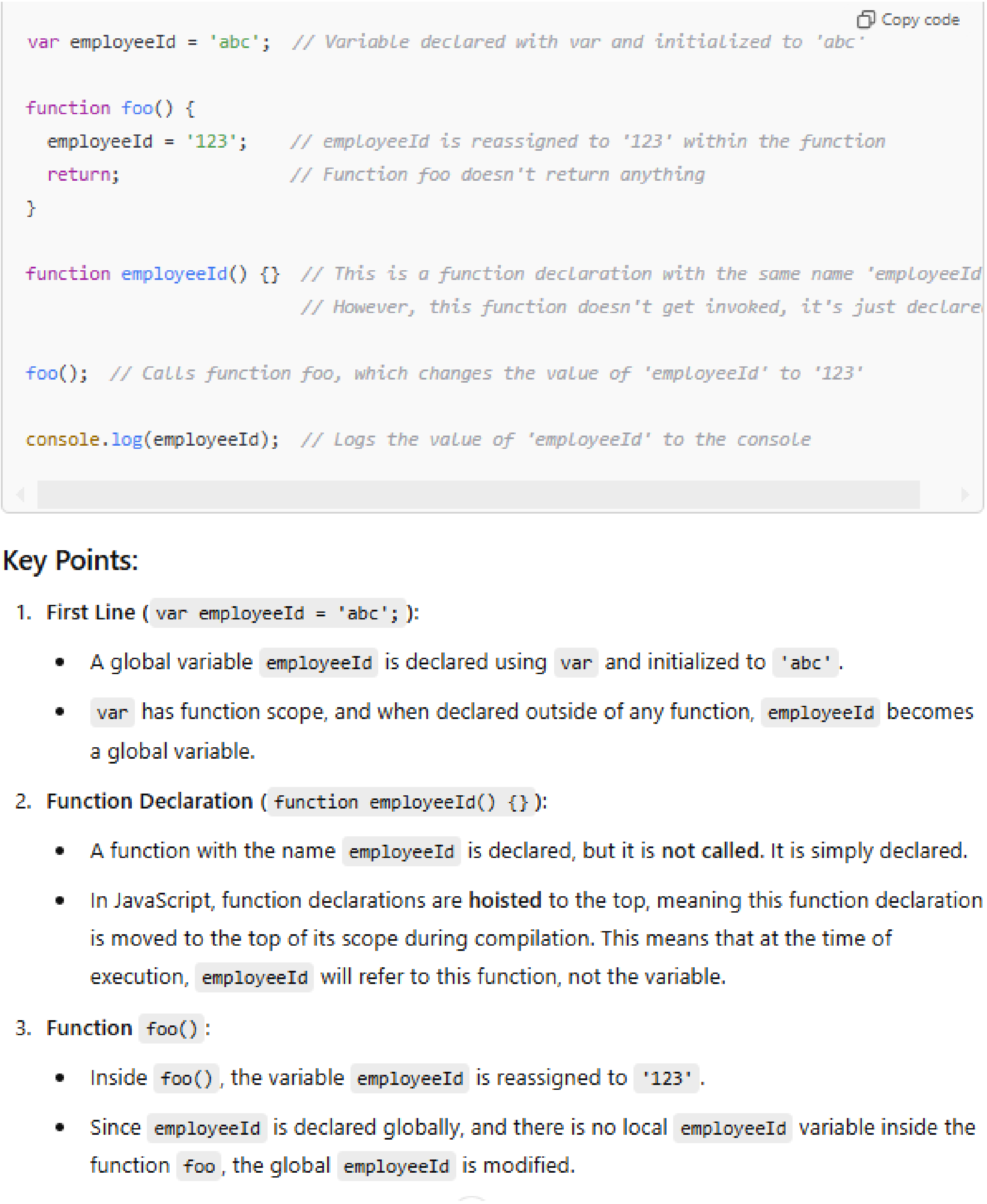
    javascript

    Copy code

    ```javascript
    console.log(myVar); // undefined
    var myVar = 10;
    ```
- `let` and `const`:
  - Only the declaration is hoisted. The variable is in a **temporal dead zone** until the initialization is encountered.

Key Takeaways:

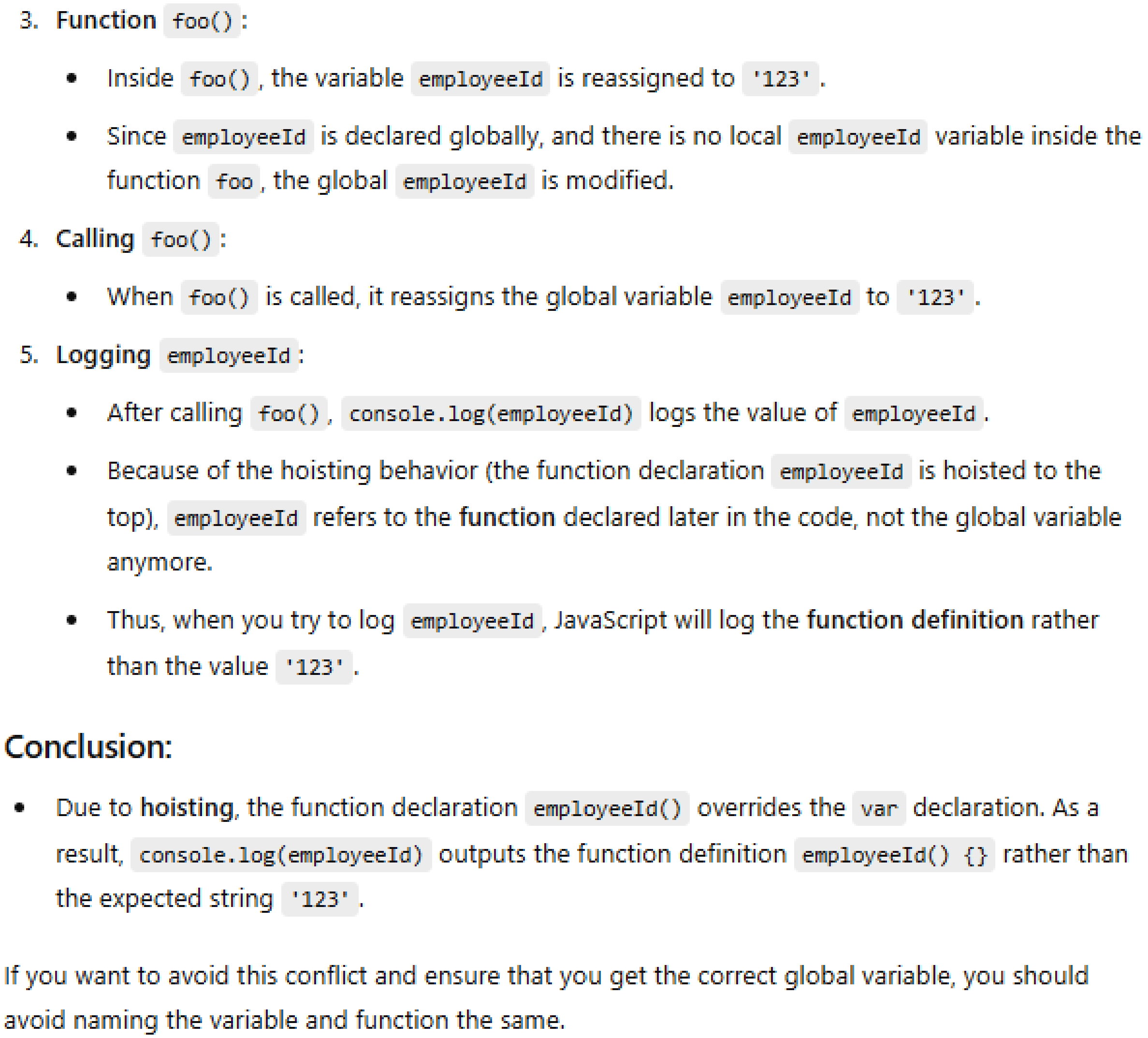
1. **Temporal Dead Zone (TDZ):**
   - Variables declared with `let` and `const` cannot be accessed before their declaration in the code.
   - Accessing them before the assignment leads to a `ReferenceError`.
2. **Hoisting with `let`:**
   - The declaration is hoisted, but the initialization happens only when the interpreter reaches the line of code.
3. **Avoid Accessing Variables Before Initialization:**
   - Always access variables only after they have been properly initialized to avoid the TDZ issue.



```
var employeeId = 'abc';  // Variable declared with var and initialized to 'abc'

function foo() {
  employeeId = '123';    // employeeId is reassigned to '123' within the function
  return;                // Function foo doesn't return anything
}

function employeeId() {}  // This is a function declaration with the same name 'employeeId
                          // However, this function doesn't get invoked, it's just declare

foo();  // Calls function foo, which changes the value of 'employeeId' to '123'

console.log(employeeId);  // Logs the value of 'employeeId' to the console
```

Key Points:

1. **First Line** (`var employeeId = 'abc';`):
   - A global variable `employeeId` is declared using `var` and initialized to `'abc'`.
   - `var` has function scope, and when declared outside of any function, `employeeId` becomes a global variable.
2. **Function Declaration** (`function employeeId() {}`):
   - A function with the name `employeeId` is declared, but it is **not called.** It is simply declared.
   - In JavaScript, function declarations are **hoisted** to the top, meaning this function declaration is moved to the top of its scope during compilation. This means that at the time of execution, `employeeId` will refer to this function, not the variable.
3. **Function** `foo()`:
   - Inside `foo()`, the variable `employeeId` is reassigned to `'123'`.
   - Since `employeeId` is declared globally, and there is no local `employeeId` variable inside the function `foo`, the global `employeeId` is modified.



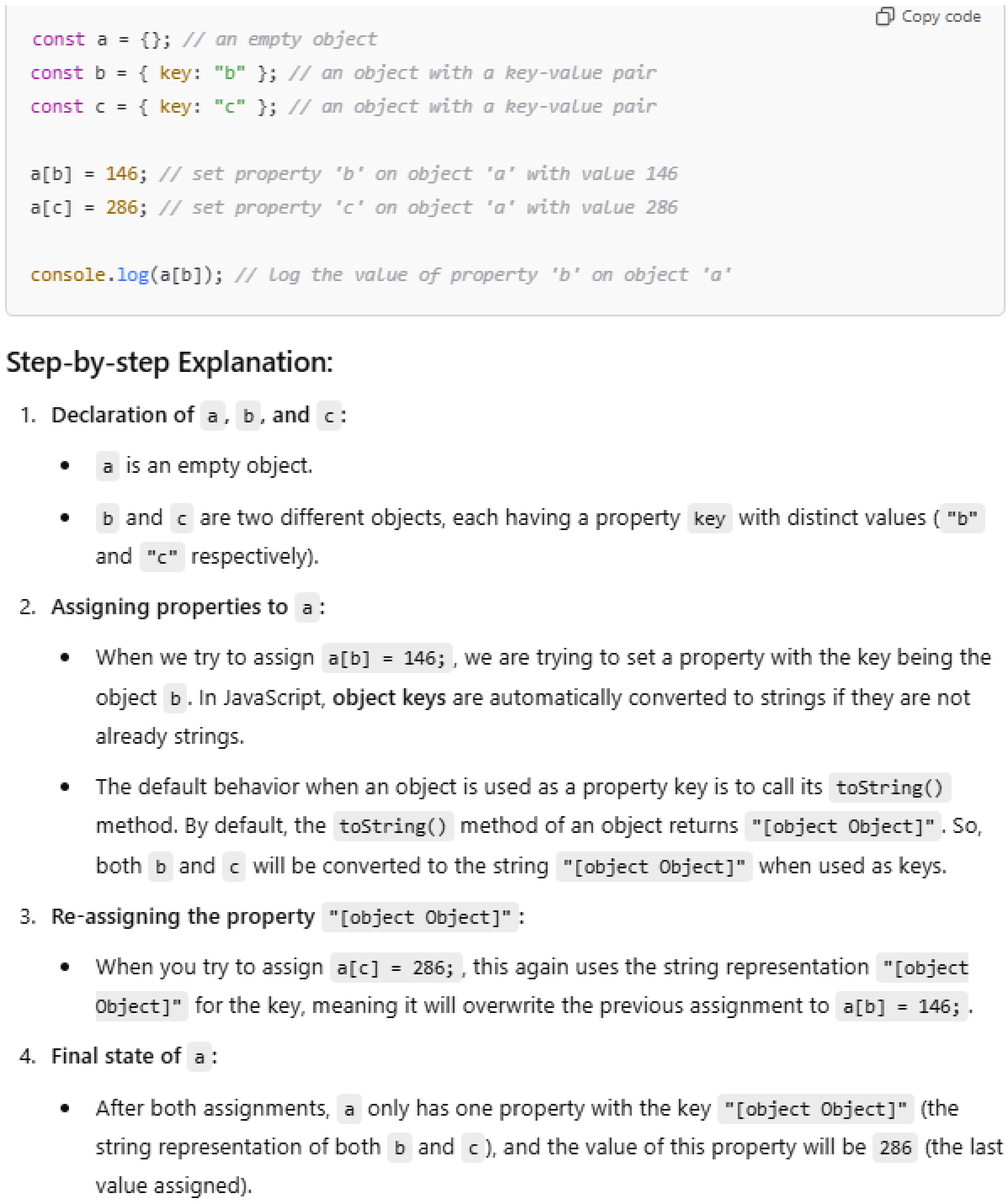
3. **Function** `foo()`:
   - Inside `foo()`, the variable `employeeId` is reassigned to `'123'`.
   - Since `employeeId` is declared globally, and there is no local `employeeId` variable inside the function `foo`, the global `employeeId` is modified.

4. **Calling** `foo()`:
   - When `foo()` is called, it reassigns the global variable `employeeId` to `'123'`.
5. **Logging** `employeeId`:
   - After calling `foo()`, `console.log(employeeId)` logs the value of `employeeId`.
   - Because of the hoisting behavior (the function declaration `employeeId` is hoisted to the top), `employeeId` refers to the **function** declared later in the code, not the global variable anymore.
   - Thus, when you try to log `employeeId`, JavaScript will log the **function definition** rather than the value `'123'`.

Conclusion:

- Due to **hoisting**, the function declaration `employeeId()` overrides the `var` declaration. As a result, `console.log(employeeId)` outputs the function definition `employeeId() {}` rather than the expected string `'123'`.

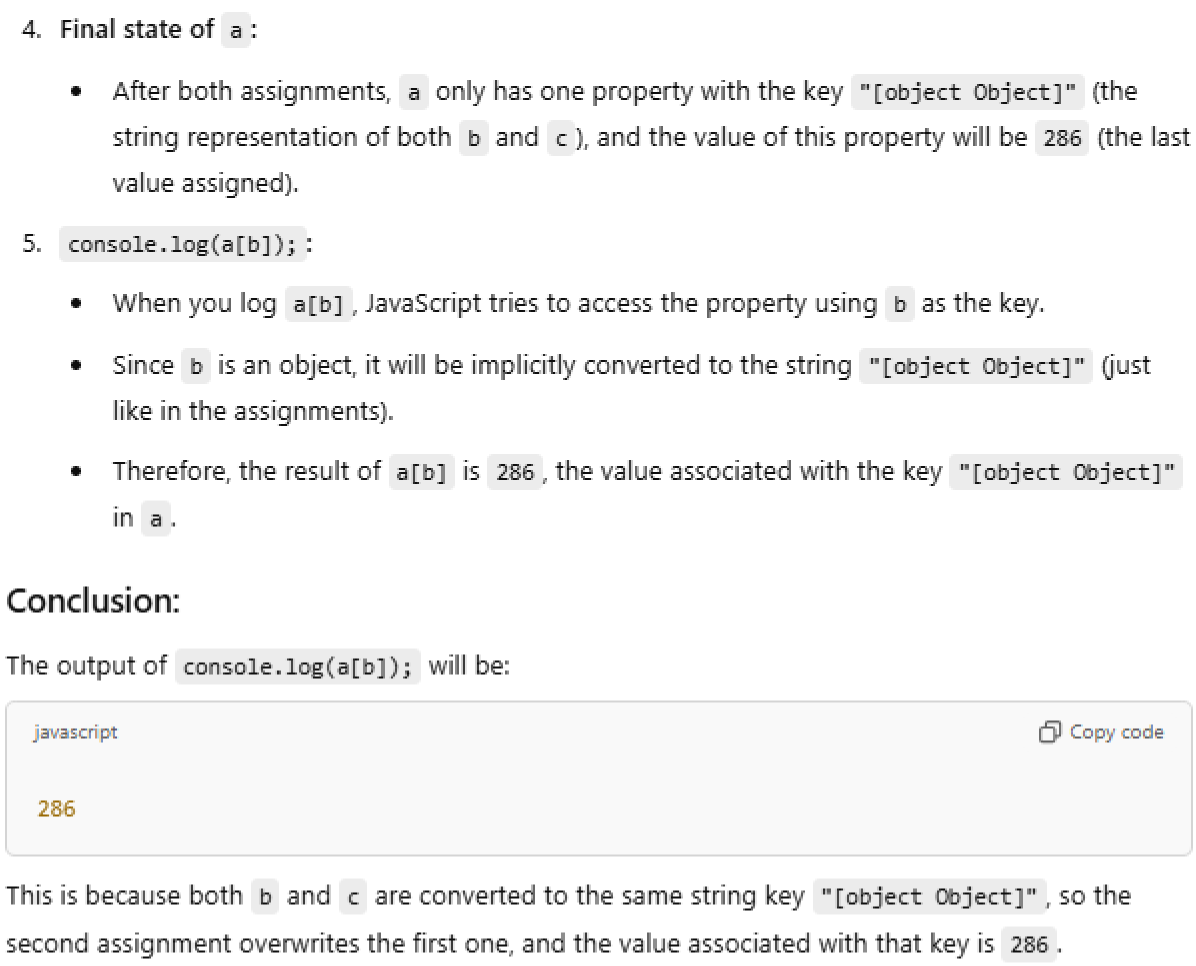
If you want to avoid this conflict and ensure that you get the correct global variable, you should avoid naming the variable and function the same.



```javascript
const a = {}; // an empty object
const b = { key: "b" }; // an object with a key-value pair
const c = { key: "c" }; // an object with a key-value pair

a[b] = 146; // set property 'b' on object 'a' with value 146
a[c] = 286; // set property 'c' on object 'a' with value 286

console.log(a[b]); // log the value of property 'b' on object 'a'
```

Step-by-step Explanation:

1. **Declaration of `a`, `b`, and `c`:**
   - `a` is an empty object.
   - `b` and `c` are two different objects, each having a property `key` with distinct values (`"b"` and `"c"` respectively).
2. **Assigning properties to `a`:**
   - When we try to assign `a[b] = 146;`, we are trying to set a property with the key being the object `b`. In JavaScript, **object keys** are automatically converted to strings if they are not already strings.
   - The default behavior when an object is used as a property key is to call its `toString()` method. By default, the `toString()` method of an object returns `"[object Object]"`. So, both `b` and `c` will be converted to the string `"[object Object]"` when used as keys.
3. **Re-assigning the property `"[object Object]"`:**
   - When you try to assign `a[c] = 286;`, this again uses the string representation `"[object Object]"` for the key, meaning it will overwrite the previous assignment to `a[b] = 146;`.
4. **Final state of `a`:**
   - After both assignments, `a` only has one property with the key `"[object Object]"` (the string representation of both `b` and `c`), and the value of this property will be `286` (the last value assigned).



4. **Final state of `a`:**
   - After both assignments, `a` only has one property with the key `"[object Object]"` (the string representation of both `b` and `c`), and the value of this property will be `286` (the last value assigned).

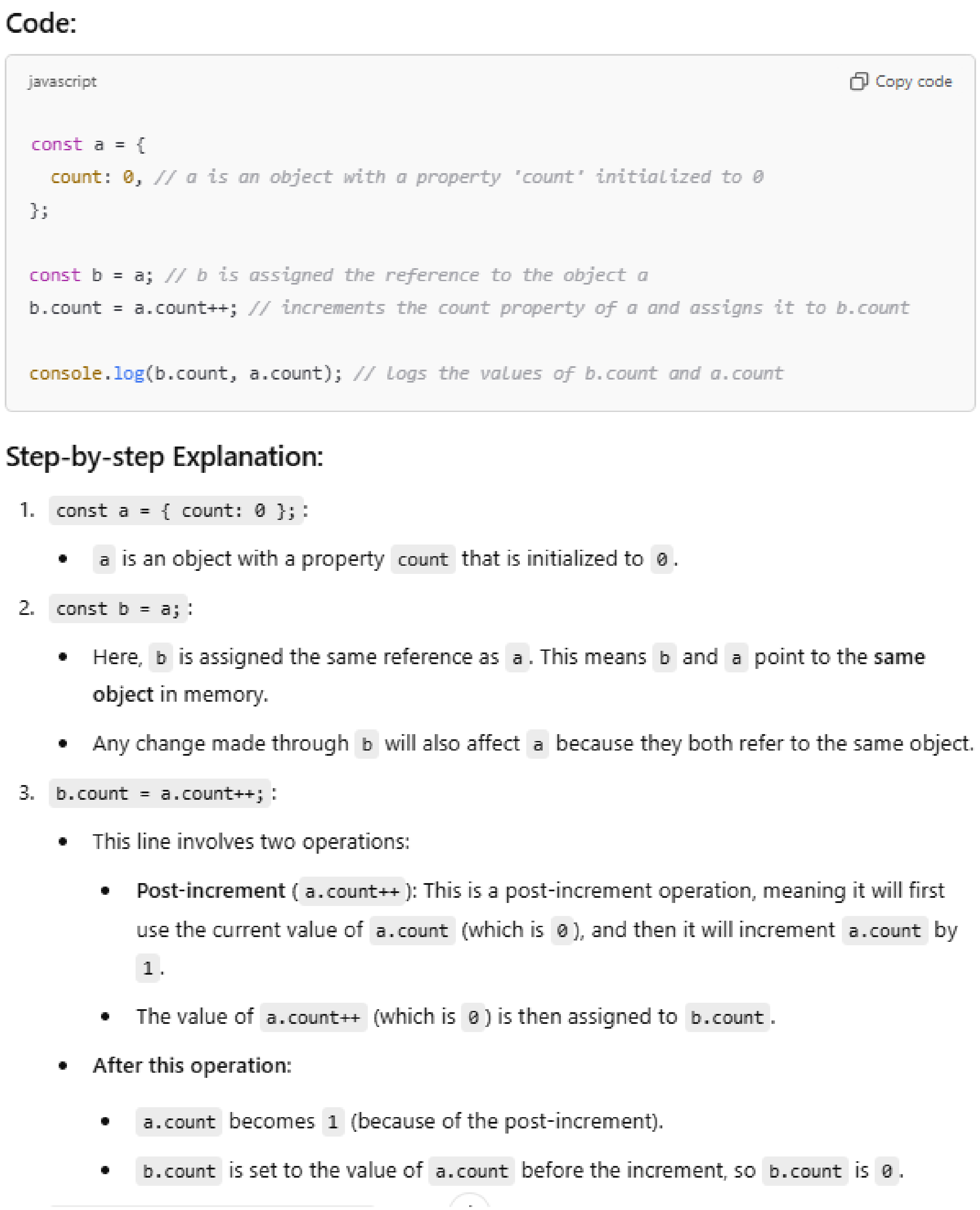
5. **`console.log(a[b]);`:**
   - When you log `a[b]`, JavaScript tries to access the property using `b` as the key.
   - Since `b` is an object, it will be implicitly converted to the string `"[object Object]"` (just like in the assignments).
   - Therefore, the result of `a[b]` is `286`, the value associated with the key `"[object Object]"` in `a`.

Conclusion:

The output of `console.log(a[b]);` will be:

```javascript
286
```

This is because both `b` and `c` are converted to the same string key `"[object Object]"`, so the second assignment overwrites the first one, and the value associated with that key is `286`.

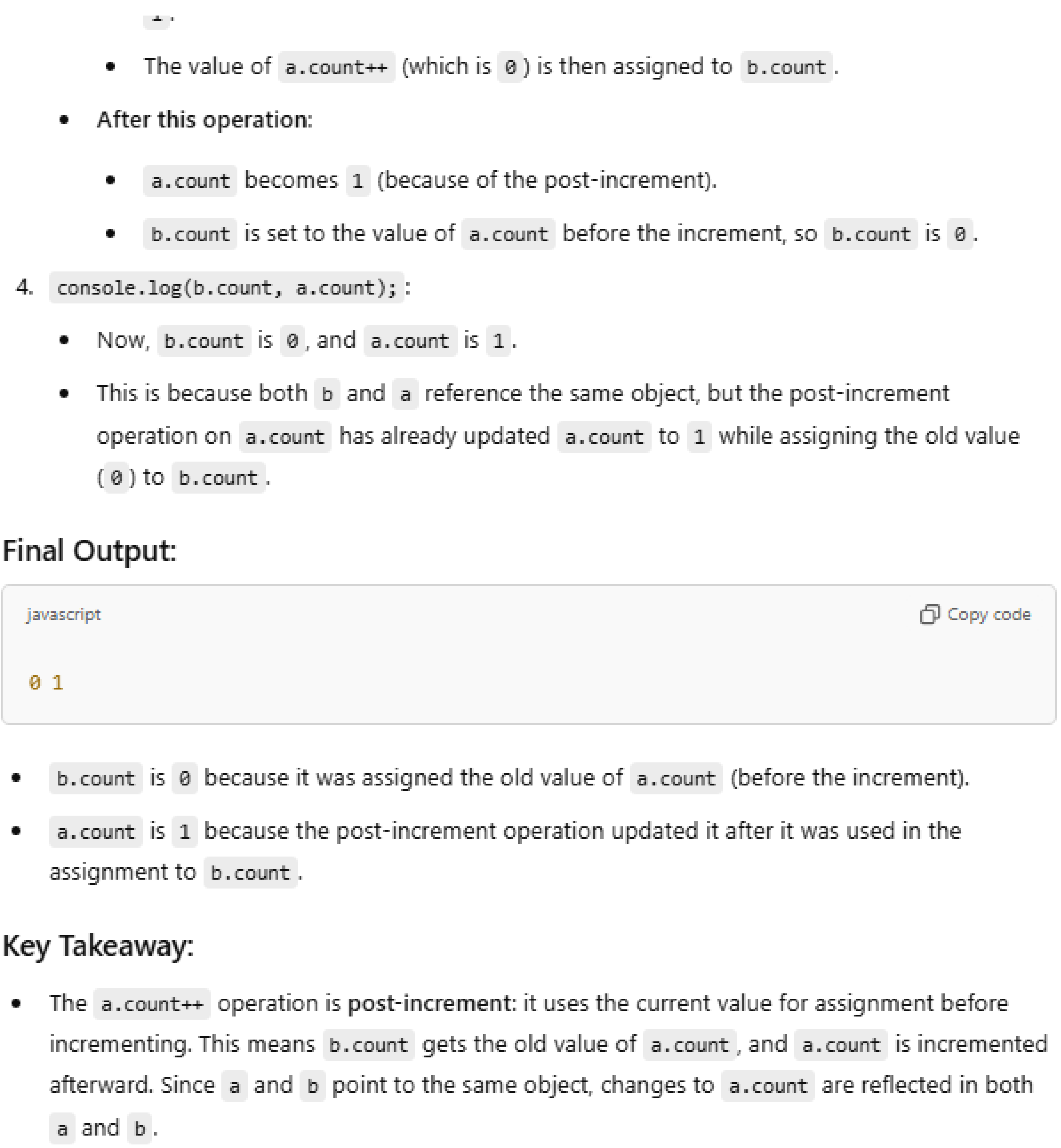


Code:

```javascript
const a = {
  count: 0, // a is an object with a property 'count' initialized to 0
};

const b = a; // b is assigned the reference to the object a
b.count = a.count++; // increments the count property of a and assigns it to b.count

console.log(b.count, a.count); // logs the values of b.count and a.count
```

Step-by-step Explanation:

1. `const a = { count: 0 };`:
   - `a` is an object with a property `count` that is initialized to `0`.
2. `const b = a;`:
   - Here, `b` is assigned the same reference as `a`. This means `b` and `a` point to the **same object** in memory.
   - Any change made through `b` will also affect `a` because they both refer to the same object.
3. `b.count = a.count++;`:
   - This line involves two operations:
     - **Post-increment** (`a.count++`): This is a post-increment operation, meaning it will first use the current value of `a.count` (which is `0`), and then it will increment `a.count` by `1`.
     - The value of `a.count++` (which is `0`) is then assigned to `b.count`.
   - **After this operation:**
     - `a.count` becomes `1` (because of the post-increment).
     - `b.count` is set to the value of `a.count` before the increment, so `b.count` is `0`.



`1`.

- The value of `a.count++` (which is `0`) is then assigned to `b.count`.
- **After this operation:**
  - `a.count` becomes `1` (because of the post-increment).
  - `b.count` is set to the value of `a.count` before the increment, so `b.count` is `0`.

4. `console.log(b.count, a.count);`:
   - Now, `b.count` is `0`, and `a.count` is `1`.
   - This is because both `b` and `a` reference the same object, but the post-increment operation on `a.count` has already updated `a.count` to `1` while assigning the old value (`0`) to `b.count`.

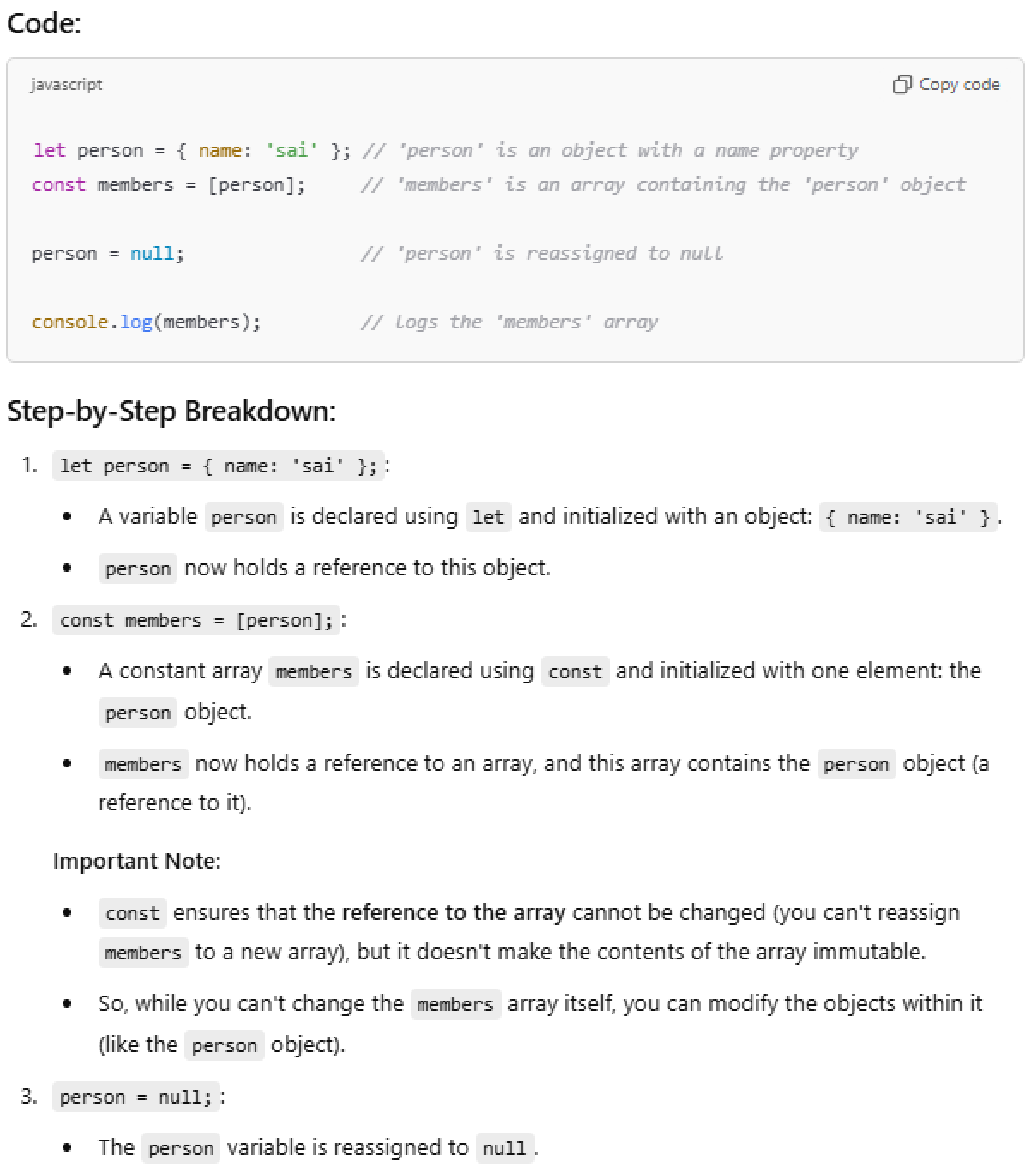
Final Output:

```javascript
0 1
```

- `b.count` is `0` because it was assigned the old value of `a.count` (before the increment).
- `a.count` is `1` because the post-increment operation updated it after it was used in the assignment to `b.count`.

Key Takeaway:

- The `a.count++` operation is **post-increment**: it uses the current value for assignment before incrementing. This means `b.count` gets the old value of `a.count`, and `a.count` is incremented afterward. Since `a` and `b` point to the same object, changes to `a.count` are reflected in both `a` and `b`.



Code:

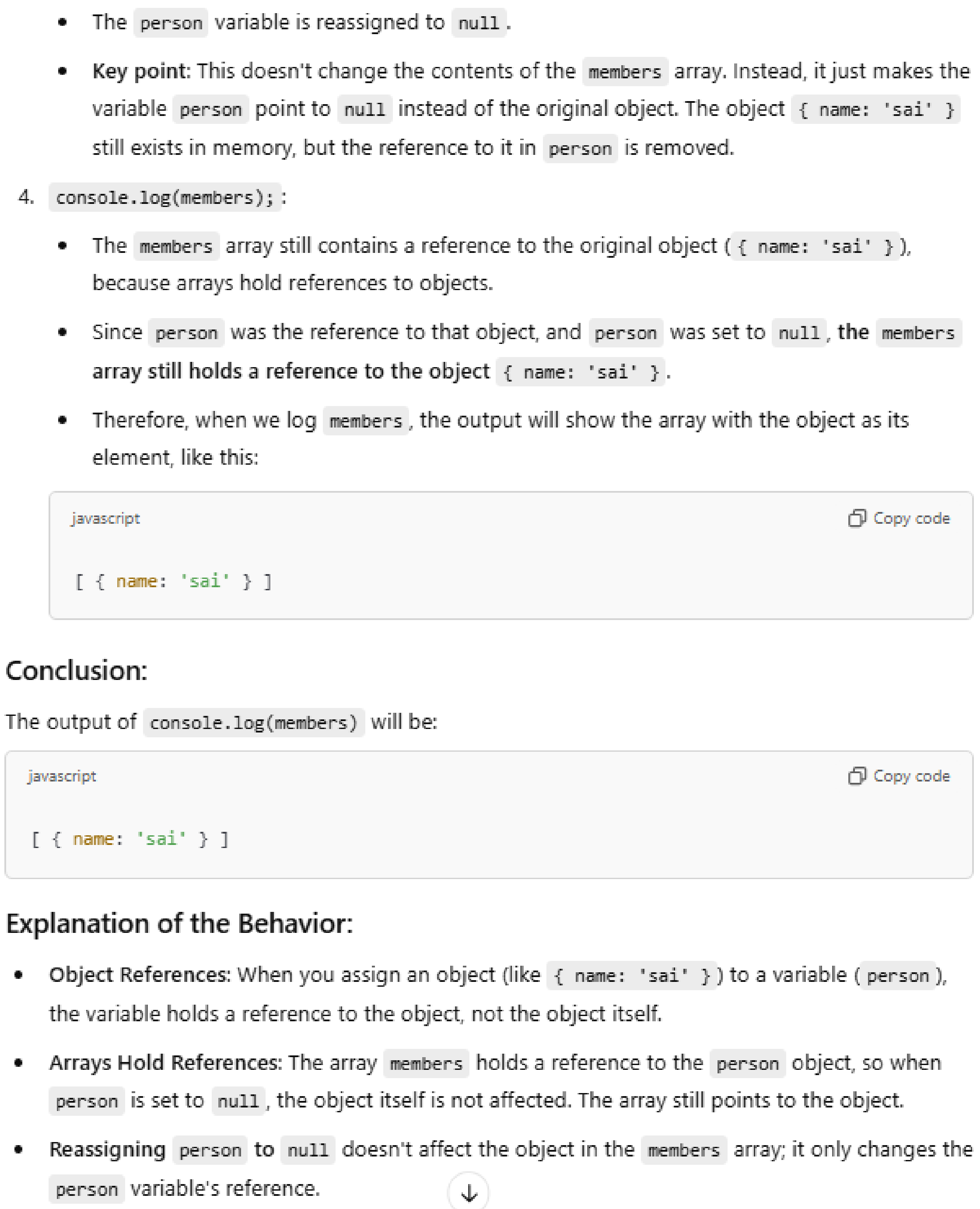
```javascript
let person = { name: 'sai' }; // 'person' is an object with a name property
const members = [person];     // 'members' is an array containing the 'person' object

person = null;                // 'person' is reassigned to null

console.log(members);         // logs the 'members' array
```

Step-by-Step Breakdown:

1. `let person = { name: 'sai' };`:
   - A variable `person` is declared using `let` and initialized with an object: `{ name: 'sai' }`.
   - `person` now holds a reference to this object.
2. `const members = [person];`:
   - A constant array `members` is declared using `const` and initialized with one element: the `person` object.
   - `members` now holds a reference to an array, and this array contains the `person` object (a reference to it).

   Important Note:

   - `const` ensures that the **reference to the array** cannot be changed (you can't reassign `members` to a new array), but it doesn't make the contents of the array immutable.
   - So, while you can't change the `members` array itself, you can modify the objects within it (like the `person` object).
3. `person = null;`:
   - The `person` variable is reassigned to `null`.



- The `person` variable is reassigned to `null`.

   - **Key point:** This doesn't change the contents of the `members` array. Instead, it just makes the variable `person` point to `null` instead of the original object. The object `{ name: 'sai' }` still exists in memory, but the reference to it in `person` is removed.
4. `console.log(members);`:
   - The `members` array still contains a reference to the original object (`{ name: 'sai' }`), because arrays hold references to objects.
   - Since `person` was the reference to that object, and `person` was set to `null`, **the** `members` **array still holds a reference to the object** `{ name: 'sai' }`.
   - Therefore, when we log `members`, the output will show the array with the object as its element, like this:

```javascript
[ { name: 'sai' } ]
```

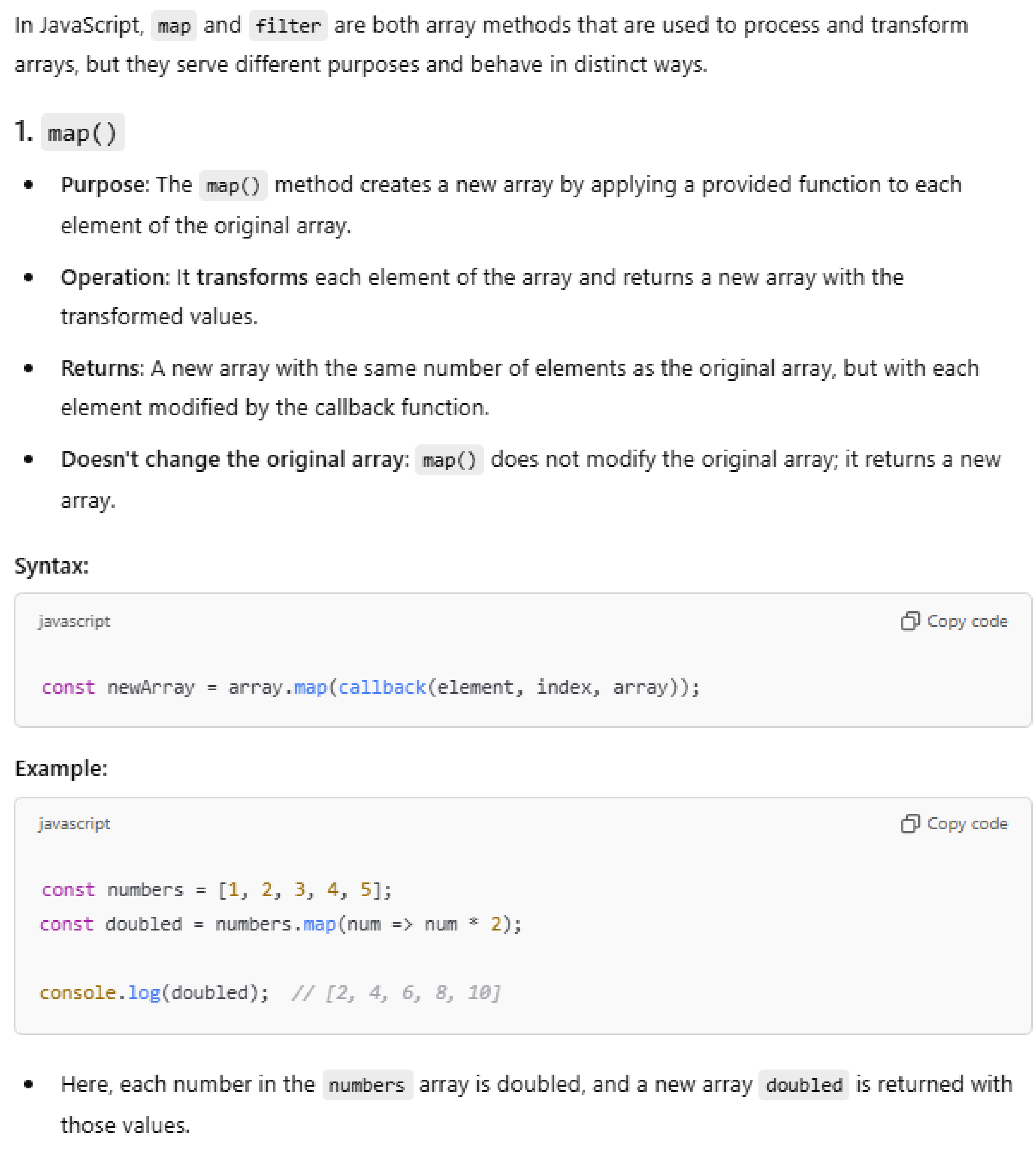
Conclusion:

The output of `console.log(members)` will be:

```javascript
[ { name: 'sai' } ]
```

Explanation of the Behavior:

- **Object References:** When you assign an object (like `{ name: 'sai' }`) to a variable (`person`), the variable holds a reference to the object, not the object itself.
- **Arrays Hold References:** The array `members` holds a reference to the `person` object, so when `person` is set to `null`, the object itself is not affected. The array still points to the object.
- **Reassigning** `person` **to** `null` doesn't affect the object in the `members` array; it only changes the `person` variable's reference.



In JavaScript, `map` and `filter` are both array methods that are used to process and transform arrays, but they serve different purposes and behave in distinct ways.

1. `map()`

- **Purpose:** The `map()` method creates a new array by applying a provided function to each element of the original array.
- **Operation:** It **transforms** each element of the array and returns a new array with the transformed values.
- **Returns:** A new array with the same number of elements as the original array, but with each element modified by the callback function.
- **Doesn't change the original array:** `map()` does not modify the original array; it returns a new array.

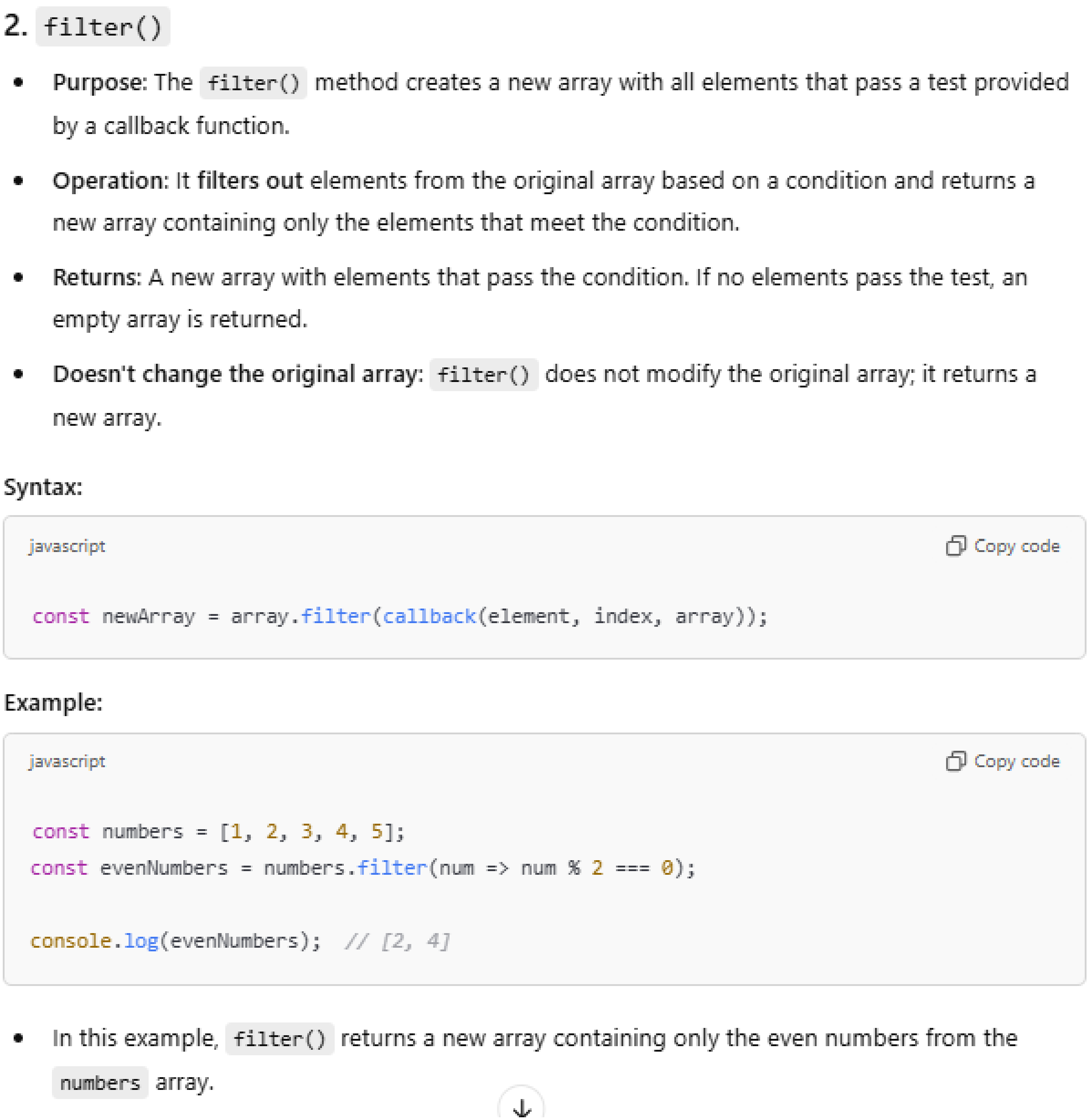
Syntax:

```
const newArray = array.map(callback(element, index, array));
```

Example:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(num => num * 2);

console.log(doubled);  // [2, 4, 6, 8, 10]
```

- Here, each number in the `numbers` array is doubled, and a new array `doubled` is returned with those values.



2. `filter()`

- **Purpose:** The `filter()` method creates a new array with all elements that pass a test provided by a callback function.
- **Operation:** It **filters out** elements from the original array based on a condition and returns a new array containing only the elements that meet the condition.
- **Returns:** A new array with elements that pass the condition. If no elements pass the test, an empty array is returned.
- **Doesn't change the original array:** `filter()` does not modify the original array; it returns a new array.

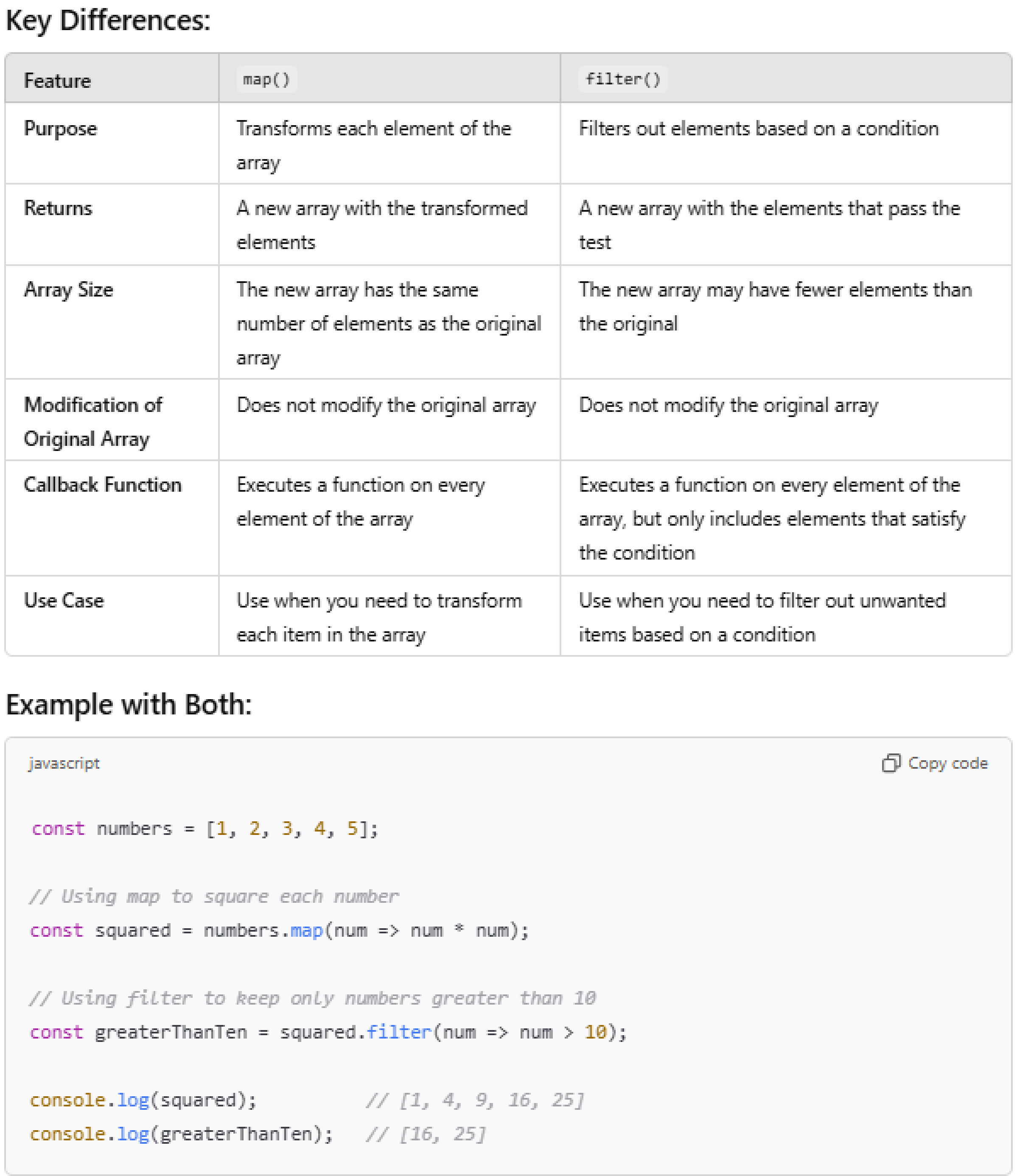
Syntax:

```
const newArray = array.filter(callback(element, index, array));
```

Example:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);

console.log(evenNumbers);  // [2, 4]
```

- In this example, `filter()` returns a new array containing only the even numbers from the `numbers` array.

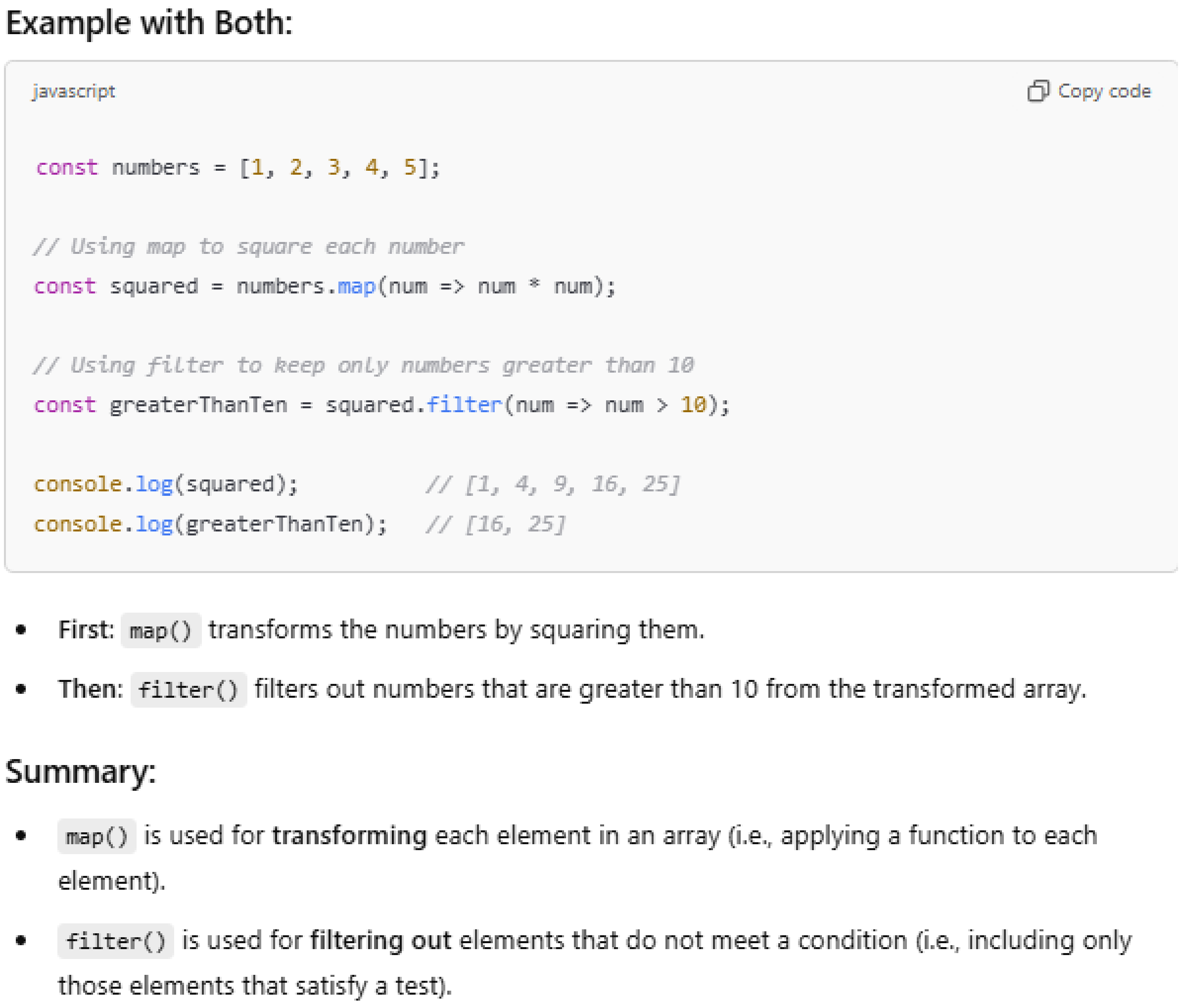


Key Differences:

Feature	`map()`	`filter()`
Purpose	Transforms each element of the array	Filters out elements based on a condition
Returns	A new array with the transformed elements	A new array with the elements that pass the test
Array Size	The new array has the same number of elements as the original array	The new array may have fewer elements than the original
Modification of Original Array	Does not modify the original array	Does not modify the original array
Callback Function	Executes a function on every element of the array	Executes a function on every element of the array, but only includes elements that satisfy the condition
Use Case	Use when you need to transform each item in the array	Use when you need to filter out unwanted items based on a condition

Example with Both:

```javascript
const numbers = [1, 2, 3, 4, 5];

// Using map to square each number
const squared = numbers.map(num => num * num);

// Using filter to keep only numbers greater than 10
const greaterThanTen = squared.filter(num => num > 10);

console.log(squared);         // [1, 4, 9, 16, 25]
console.log(greaterThanTen);  // [16, 25]
```



Example with Both:

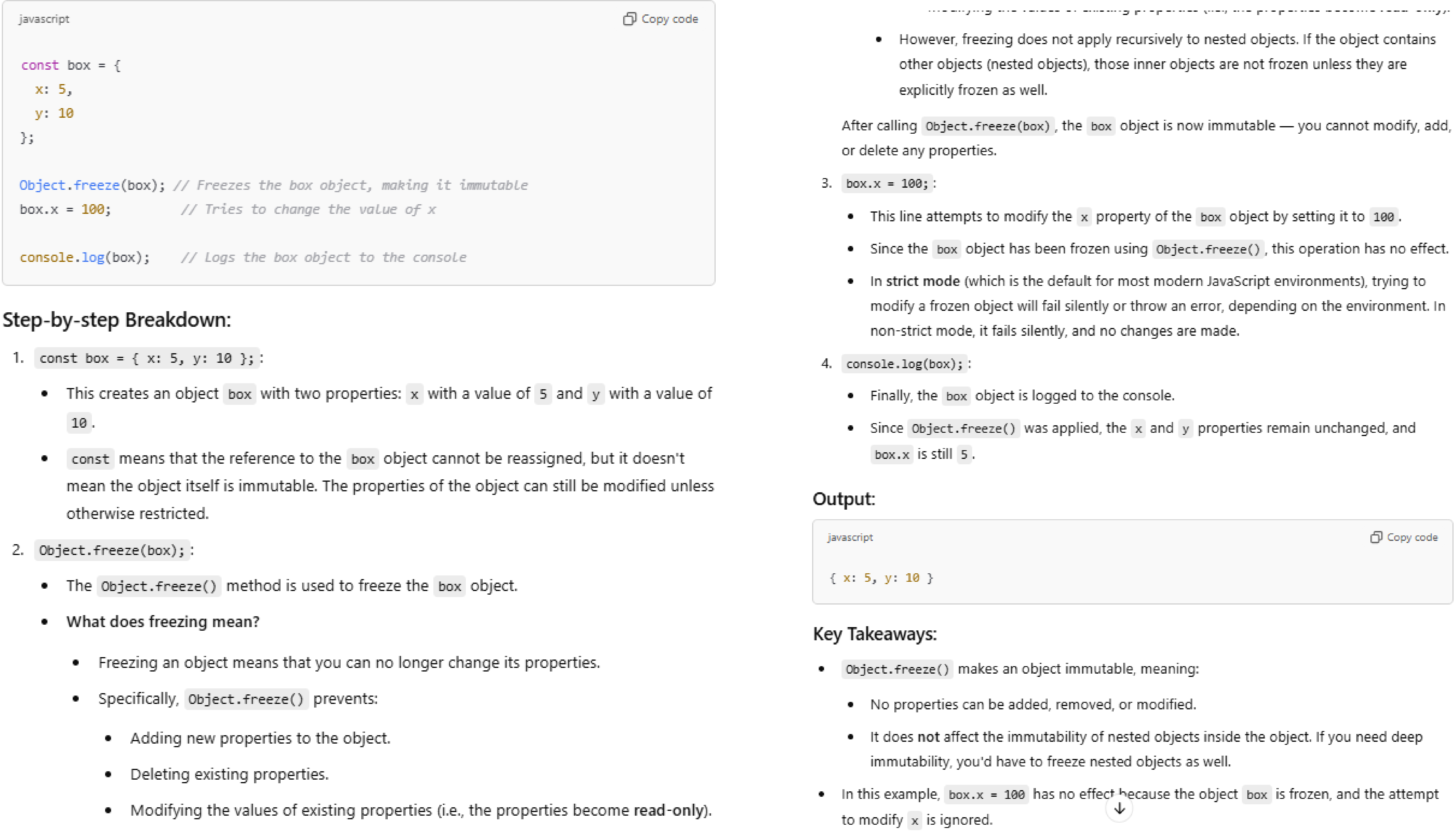
javascript Copy code

```javascript
const numbers = [1, 2, 3, 4, 5];

// Using map to square each number
const squared = numbers.map(num => num * num);

// Using filter to keep only numbers greater than 10
const greaterThanTen = squared.filter(num => num > 10);

console.log(squared);         // [1, 4, 9, 16, 25]
console.log(greaterThanTen);  // [16, 25]
```

- **First:** `map()` transforms the numbers by squaring them.
- **Then:** `filter()` filters out numbers that are greater than 10 from the transformed array.

Summary:

- `map()` is used for **transforming** each element in an array (i.e., applying a function to each element).
- `filter()` is used for **filtering out** elements that do not meet a condition (i.e., including only those elements that satisfy a test).



```javascript
const box = {
  x: 5,
  y: 10
};

Object.freeze(box); // Freezes the box object, making it immutable
box.x = 100;          // Tries to change the value of x

console.log(box);    // Logs the box object to the console
```

Step-by-step Breakdown:

1. `const box = { x: 5, y: 10 };`:
   - This creates an object `box` with two properties: `x` with a value of `5` and `y` with a value of `10`.
   - `const` means that the reference to the `box` object cannot be reassigned, but it doesn't mean the object itself is immutable. The properties of the object can still be modified unless otherwise restricted.
2. `Object.freeze(box);`:
   - The `Object.freeze()` method is used to freeze the `box` object.
   - **What does freezing mean?**
     - Freezing an object means that you can no longer change its properties.
     - Specifically, `Object.freeze()` prevents:
       - Adding new properties to the object.
       - Deleting existing properties.
       - Modifying the values of existing properties (i.e., the properties become **read-only**).

Modifying the values of existing properties (i.e., the properties become read-only).

     - However, freezing does not apply recursively to nested objects. If the object contains other objects (nested objects), those inner objects are not frozen unless they are explicitly frozen as well.

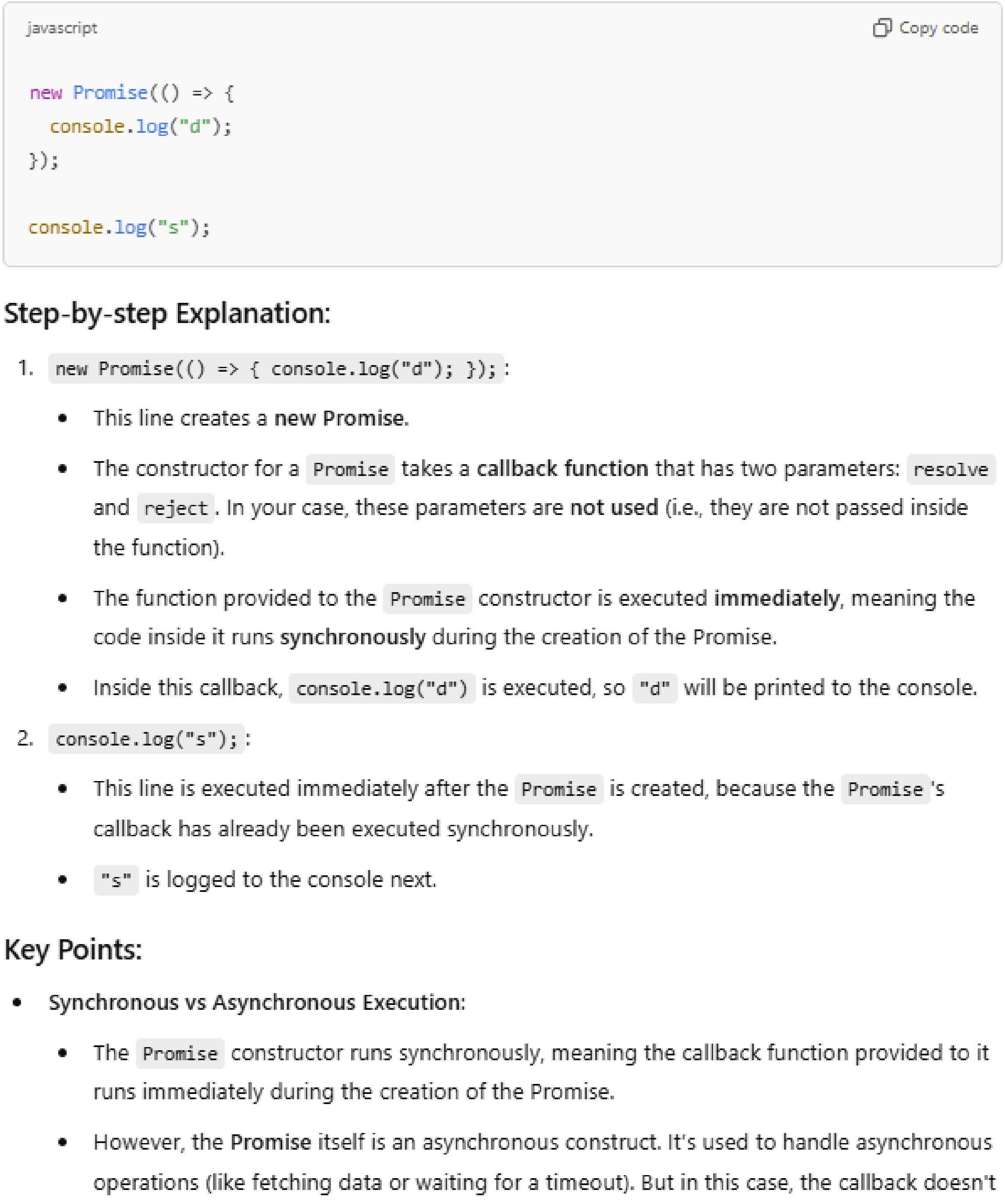
   After calling `Object.freeze(box)`, the `box` object is now immutable — you cannot modify, add, or delete any properties.
3. `box.x = 100;`:
   - This line attempts to modify the `x` property of the `box` object by setting it to `100`.
   - Since the `box` object has been frozen using `Object.freeze()`, this operation has no effect.
   - In **strict mode** (which is the default for most modern JavaScript environments), trying to modify a frozen object will fail silently or throw an error, depending on the environment. In non-strict mode, it fails silently, and no changes are made.
4. `console.log(box);`:
   - Finally, the `box` object is logged to the console.
   - Since `Object.freeze()` was applied, the `x` and `y` properties remain unchanged, and `box.x` is still `5`.

Output:

```javascript
{ x: 5, y: 10 }
```

Key Takeaways:

- `Object.freeze()` makes an object immutable, meaning:
  - No properties can be added, removed, or modified.
  - It does **not** affect the immutability of nested objects inside the object. If you need deep immutability, you'd have to freeze nested objects as well.
- In this example, `box.x = 100` has no effect because the object `box` is frozen, and the attempt to modify `x` is ignored.



```javascript
new Promise(() => {
  console.log("d");
});

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

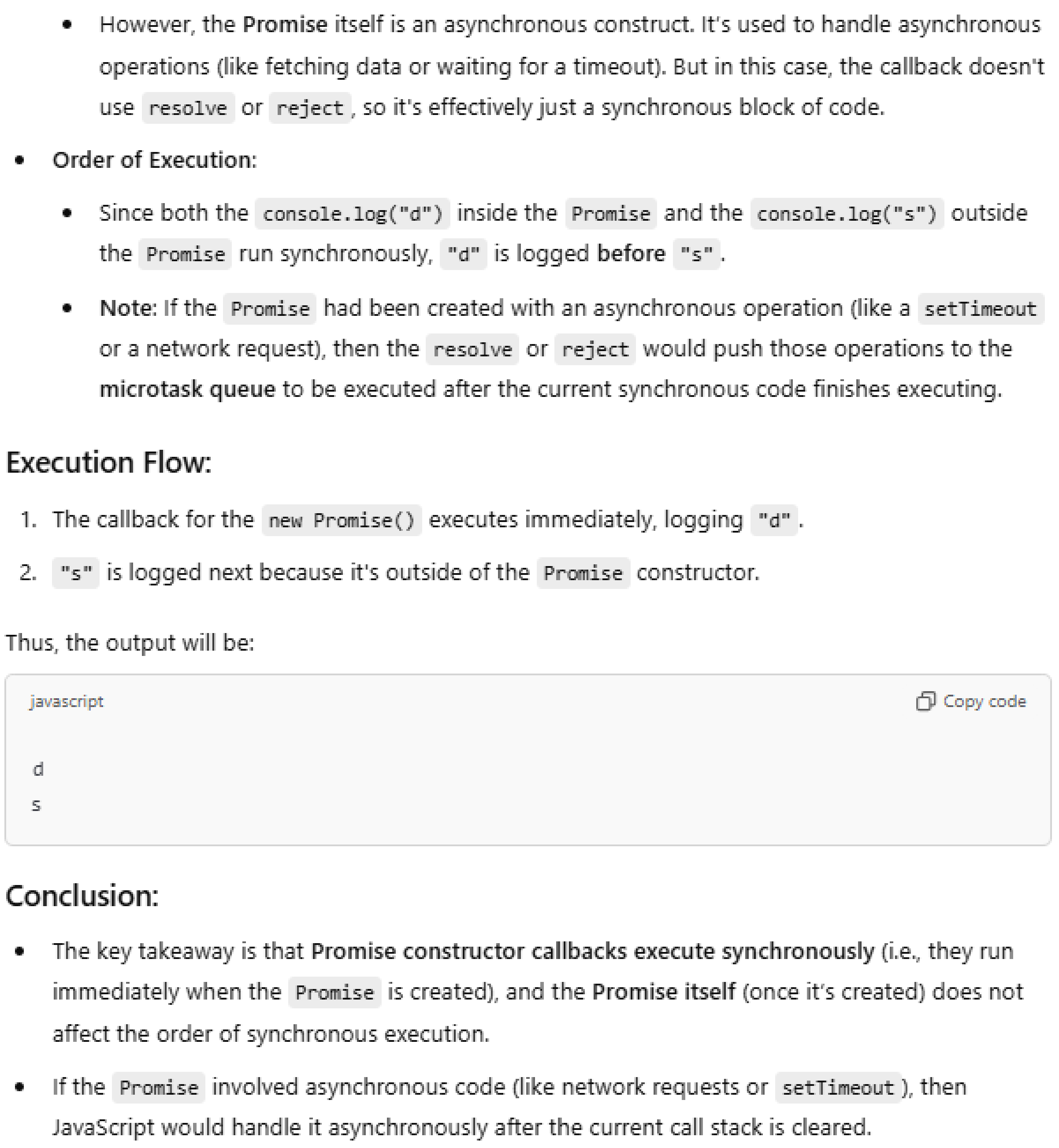
Step-by-step Explanation:

1. `new Promise(() => { console.log("d"); });`:
   - This line creates a **new Promise**.
   - The constructor for a `Promise` takes a **callback function** that has two parameters: `resolve` and `reject`. In your case, these parameters are **not used** (i.e., they are not passed inside the function).
   - The function provided to the `Promise` constructor is executed **immediately**, meaning the code inside it runs **synchronously** during the creation of the Promise.
   - Inside this callback, `console.log("d")` is executed, so `"d"` will be printed to the console.
2. `console.log("s");`:
   - This line is executed immediately after the `Promise` is created, because the `Promise`'s callback has already been executed synchronously.
   - `"s"` is logged to the console next.

Key Points:

- **Synchronous vs Asynchronous Execution**:
  - The `Promise` constructor runs synchronously, meaning the callback function provided to it runs immediately during the creation of the Promise.

However, the **Promise** itself is an asynchronous construct. It's used to handle asynchronous operations (like fetching data or waiting for a timeout). But in this case, the callback doesn't



  - However, the **Promise** itself is an asynchronous construct. It's used to handle asynchronous operations (like fetching data or waiting for a timeout). But in this case, the callback doesn't use `resolve` or `reject`, so it's effectively just a synchronous block of code.
- **Order of Execution**:
  - Since both the `console.log("d")` inside the `Promise` and the `console.log("s")` outside the `Promise` run synchronously, `"d"` is logged **before** `"s"`.
  - **Note**: If the `Promise` had been created with an asynchronous operation (like a `setTimeout` or a network request), then the `resolve` or `reject` would push those operations to the **microtask queue** to be executed after the current synchronous code finishes executing.

Execution Flow:

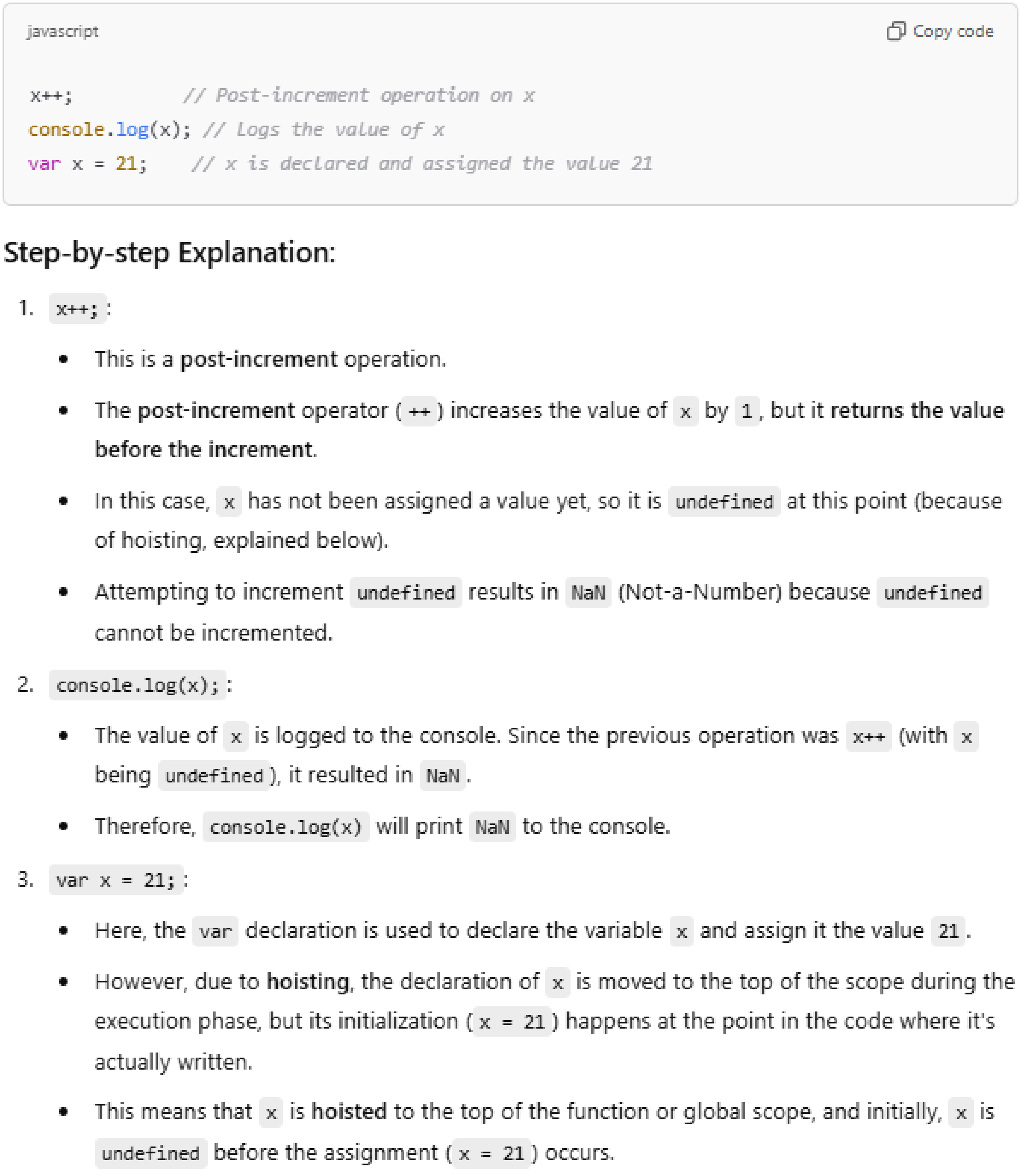
1. The callback for the `new Promise()` executes immediately, logging `"d"`.
2. `"s"` is logged next because it's outside of the `Promise` constructor.

Thus, the output will be:

```
d
s
```

Conclusion:

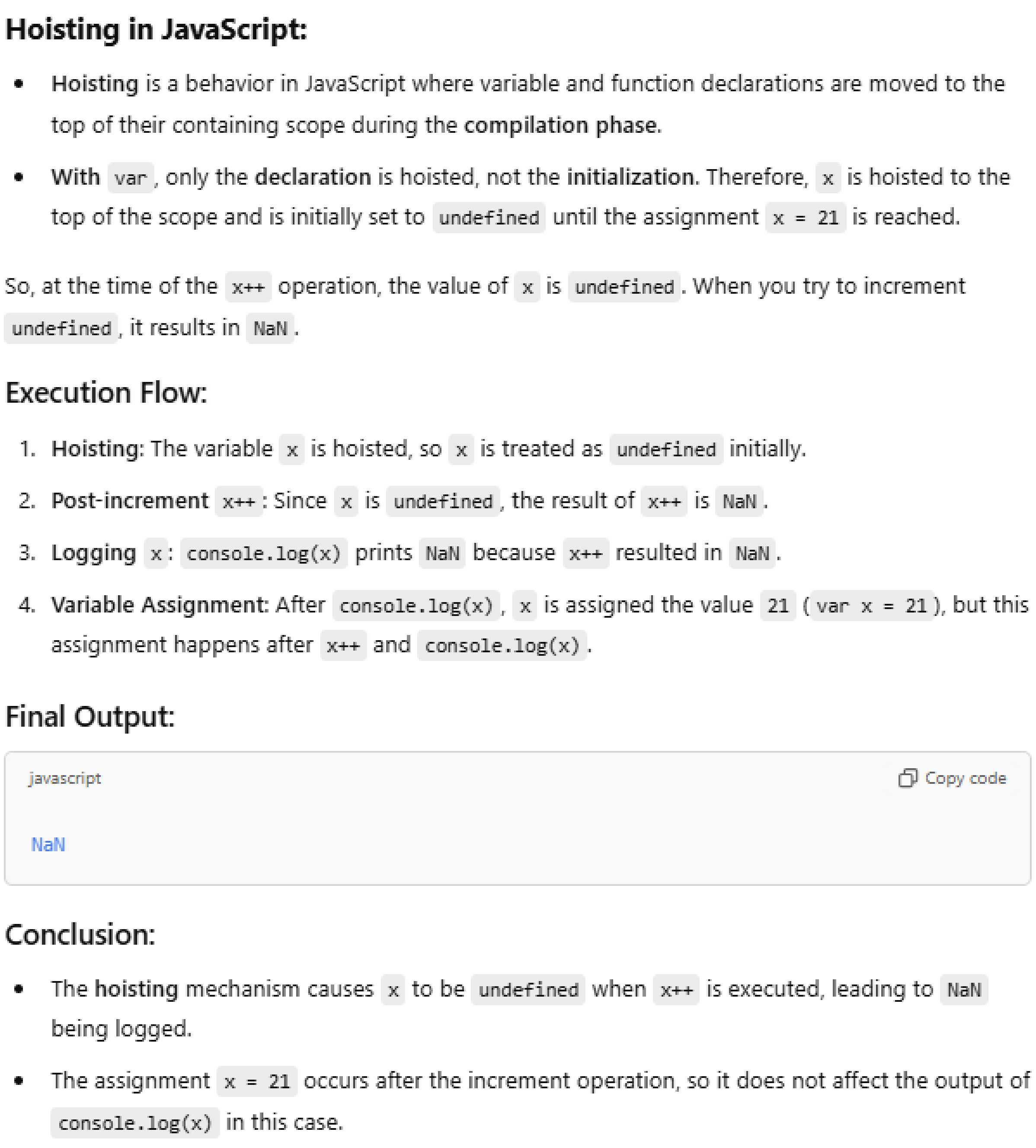
- The key takeaway is that **Promise constructor callbacks execute synchronously** (i.e., they run immediately when the `Promise` is created), and the **Promise itself** (once it's created) does not affect the order of synchronous execution.
- If the `Promise` involved asynchronous code (like network requests or `setTimeout`), then JavaScript would handle it asynchronously after the current call stack is cleared.



```javascript
x++;          // Post-increment operation on x
console.log(x); // Logs the value of x
var x = 21;    // x is declared and assigned the value 21
```

Step-by-step Explanation:

1. `x++;` :
   - This is a **post-increment** operation.
   - The **post-increment** operator (`++`) increases the value of `x` by `1`, but it **returns the value before the increment.**
   - In this case, `x` has not been assigned a value yet, so it is `undefined` at this point (because of hoisting, explained below).
   - Attempting to increment `undefined` results in `NaN` (Not-a-Number) because `undefined` cannot be incremented.
2. `console.log(x);` :
   - The value of `x` is logged to the console. Since the previous operation was `x++` (with `x` being `undefined`), it resulted in `NaN` .
   - Therefore, `console.log(x)` will print `NaN` to the console.
3. `var x = 21;` :
   - Here, the `var` declaration is used to declare the variable `x` and assign it the value `21` .
   - However, due to **hoisting**, the declaration of `x` is moved to the top of the scope during the execution phase, but its initialization (`x = 21`) happens at the point in the code where it's actually written.
   - This means that `x` is **hoisted** to the top of the function or global scope, and initially, `x` is `undefined` before the assignment (`x = 21`) occurs.



Hoisting in JavaScript:

- **Hoisting** is a behavior in JavaScript where variable and function declarations are moved to the top of their containing scope during the **compilation phase**.
- **With** `var` , only the **declaration** is hoisted, not the **initialization**. Therefore, `x` is hoisted to the top of the scope and is initially set to `undefined` until the assignment `x = 21` is reached.

So, at the time of the `x++` operation, the value of `x` is `undefined` . When you try to increment `undefined` , it results in `NaN` .

Execution Flow:

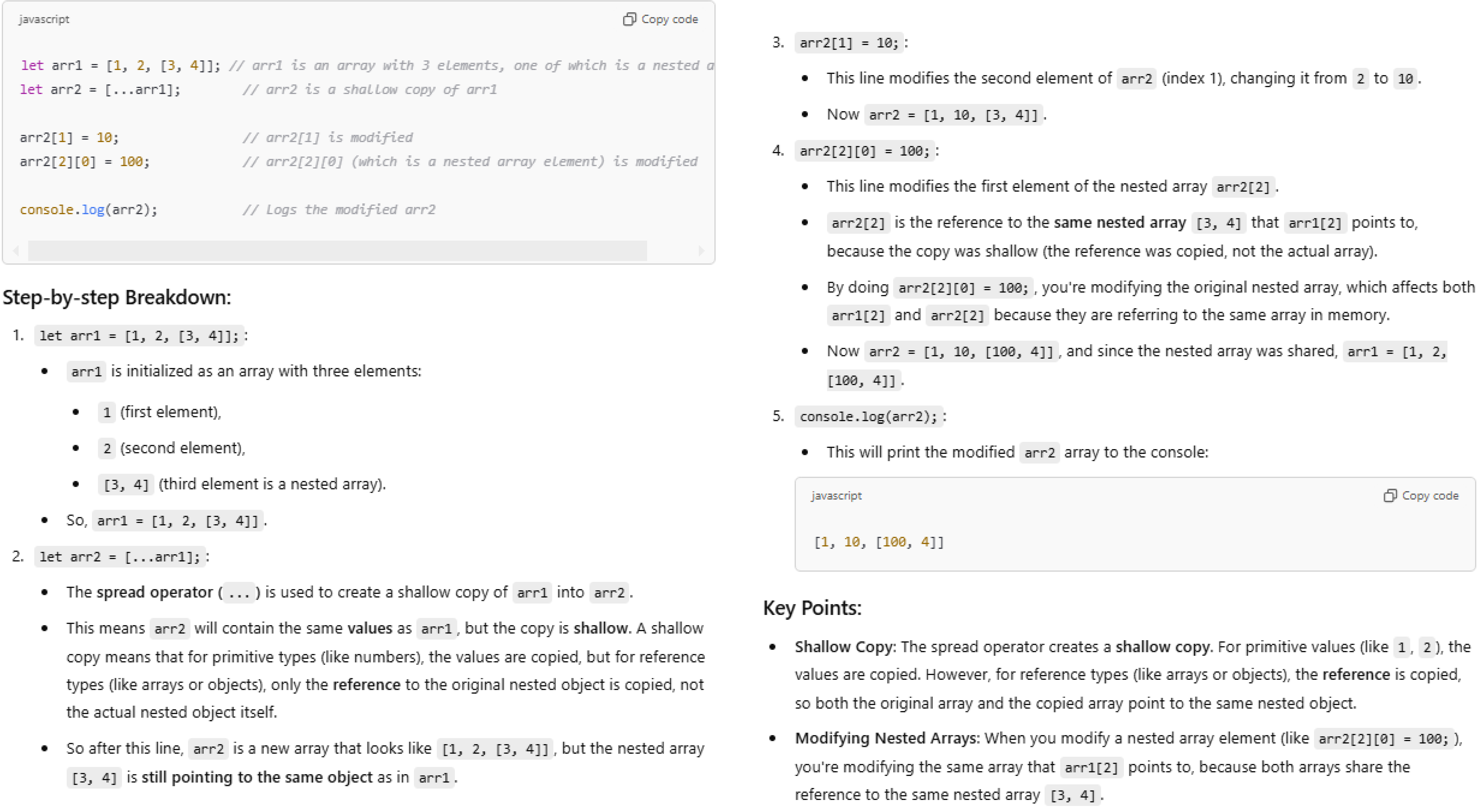
1. **Hoisting:** The variable `x` is hoisted, so `x` is treated as `undefined` initially.
2. **Post-increment** `x++` : Since `x` is `undefined` , the result of `x++` is `NaN` .
3. **Logging** `x` : `console.log(x)` prints `NaN` because `x++` resulted in `NaN` .
4. **Variable Assignment:** After `console.log(x)` , `x` is assigned the value `21` (`var x = 21`), but this assignment happens after `x++` and `console.log(x)` .

Final Output:

```javascript
NaN
```

Conclusion:

- The **hoisting** mechanism causes `x` to be `undefined` when `x++` is executed, leading to `NaN` being logged.
- The assignment `x = 21` occurs after the increment operation, so it does not affect the output of `console.log(x)` in this case.



```javascript
let arr1 = [1, 2, [3, 4]]; // arr1 is an array with 3 elements, one of which is a nested a
let arr2 = [...arr1];        // arr2 is a shallow copy of arr1

arr2[1] = 10;                // arr2[1] is modified
arr2[2][0] = 100;            // arr2[2][0] (which is a nested array element) is modified

console.log(arr2);           // Logs the modified arr2
```

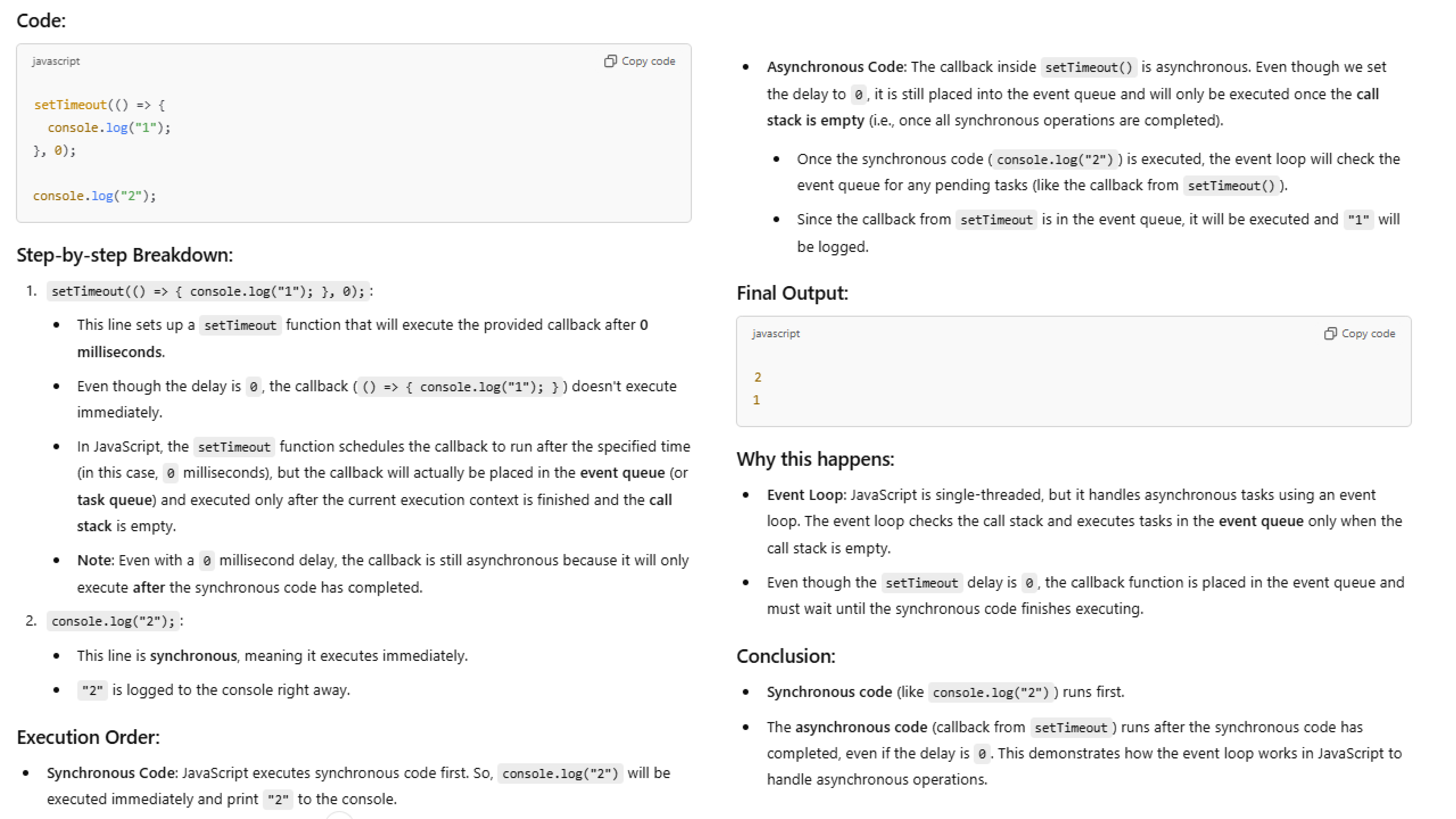
Step-by-step Breakdown:

1. `let arr1 = [1, 2, [3, 4]];`:
   - `arr1` is initialized as an array with three elements:
     - `1` (first element),
     - `2` (second element),
     - `[3, 4]` (third element is a nested array).
   - So, `arr1 = [1, 2, [3, 4]]`.
2. `let arr2 = [...arr1];`:
   - The **spread operator** (`...`) is used to create a shallow copy of `arr1` into `arr2`.
   - This means `arr2` will contain the same **values** as `arr1`, but the copy is **shallow.** A shallow copy means that for primitive types (like numbers), the values are copied, but for reference types (like arrays or objects), only the **reference** to the original nested object is copied, not the actual nested object itself.
   - So after this line, `arr2` is a new array that looks like `[1, 2, [3, 4]]`, but the nested array `[3, 4]` is **still pointing to the same object** as in `arr1`.
3. `arr2[1] = 10;`:
   - This line modifies the second element of `arr2` (index 1), changing it from `2` to `10`.
   - Now `arr2 = [1, 10, [3, 4]]`.
4. `arr2[2][0] = 100;`:
   - This line modifies the first element of the nested array `arr2[2]`.
   - `arr2[2]` is the reference to the **same nested array** `[3, 4]` that `arr1[2]` points to, because the copy was shallow (the reference was copied, not the actual array).
   - By doing `arr2[2][0] = 100;`, you're modifying the original nested array, which affects both `arr1[2]` and `arr2[2]` because they are referring to the same array in memory.
   - Now `arr2 = [1, 10, [100, 4]]`, and since the nested array was shared, `arr1 = [1, 2, [100, 4]]`.
5. `console.log(arr2);`:
   - This will print the modified `arr2` array to the console:

```javascript
[1, 10, [100, 4]]
```

Key Points:

- **Shallow Copy:** The spread operator creates a **shallow copy.** For primitive values (like `1`, `2`), the values are copied. However, for reference types (like arrays or objects), the **reference** is copied, so both the original array and the copied array point to the same nested object.
- **Modifying Nested Arrays:** When you modify a nested array element (like `arr2[2][0] = 100;`), you're modifying the same array that `arr1[2]` points to, because both arrays share the reference to the same nested array `[3, 4]`.



Code:

javascript

```javascript
setTimeout(() => {
  console.log("1");
}, 0);

console.log("2");
```

Step-by-step Breakdown:

1. `setTimeout(() => { console.log("1"); }, 0);`:
   - This line sets up a `setTimeout` function that will execute the provided callback after **0 milliseconds**.
   - Even though the delay is `0`, the callback (`() => { console.log("1"); }`) doesn't execute immediately.
   - In JavaScript, the `setTimeout` function schedules the callback to run after the specified time (in this case, `0` milliseconds), but the callback will actually be placed in the **event queue** (or **task queue**) and executed only after the current execution context is finished and the **call stack** is empty.
   - **Note**: Even with a `0` millisecond delay, the callback is still asynchronous because it will only execute **after** the synchronous code has completed.
2. `console.log("2");`:
   - This line is **synchronous**, meaning it executes immediately.
   - `"2"` is logged to the console right away.

Execution Order:

- **Synchronous Code**: JavaScript executes synchronous code first. So, `console.log("2")` will be executed immediately and print `"2"` to the console.
- **Asynchronous Code**: The callback inside `setTimeout()` is asynchronous. Even though we set the delay to `0`, it is still placed into the event queue and will only be executed once the **call stack is empty** (i.e., once all synchronous operations are completed).
  - Once the synchronous code (`console.log("2")`) is executed, the event loop will check the event queue for any pending tasks (like the callback from `setTimeout()`).
  - Since the callback from `setTimeout` is in the event queue, it will be executed and `"1"` will be logged.

Final Output:

javascript

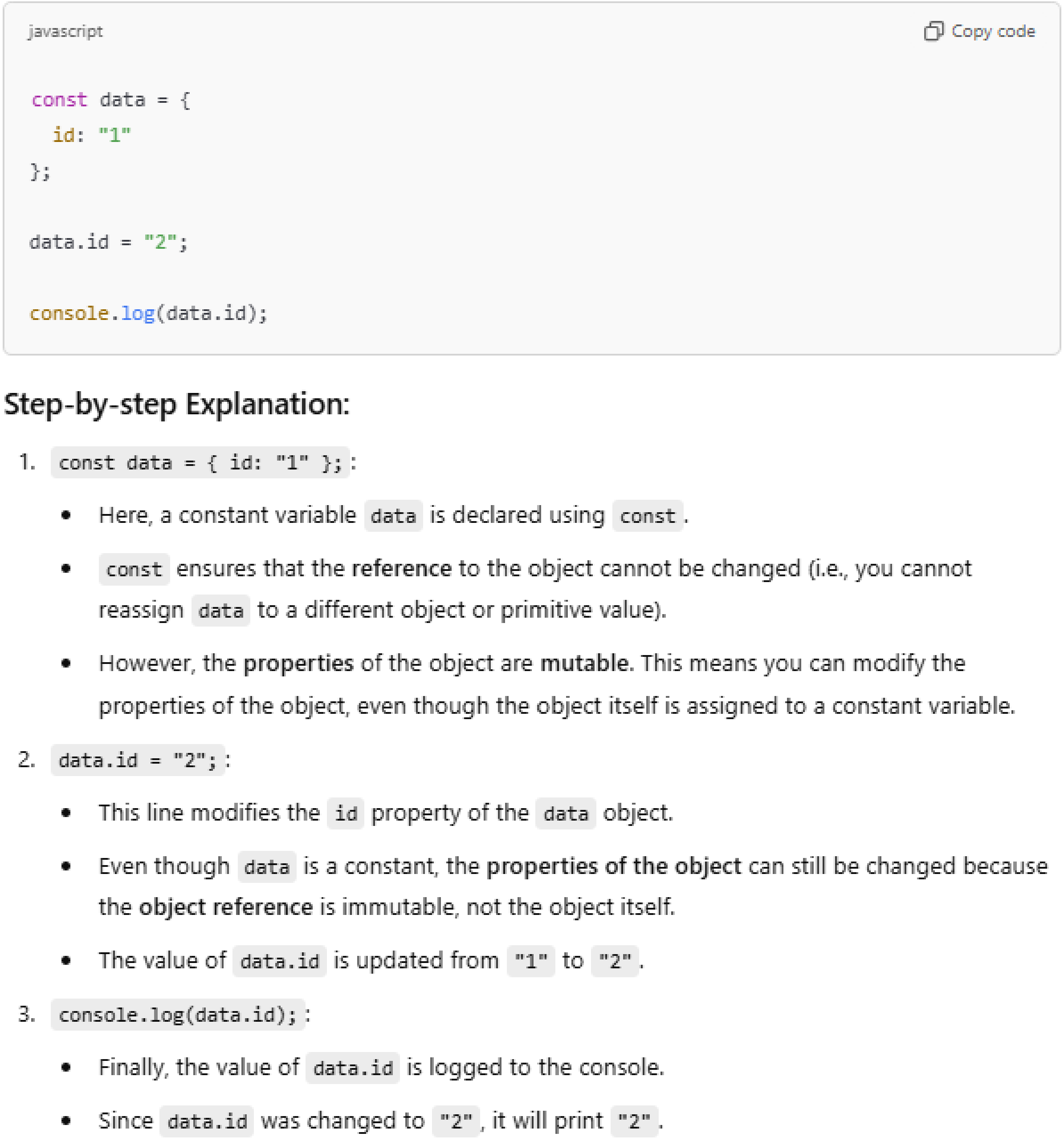
```javascript
2
1
```

Why this happens:

- **Event Loop**: JavaScript is single-threaded, but it handles asynchronous tasks using an event loop. The event loop checks the call stack and executes tasks in the **event queue** only when the call stack is empty.
- Even though the `setTimeout` delay is `0`, the callback function is placed in the event queue and must wait until the synchronous code finishes executing.

Conclusion:

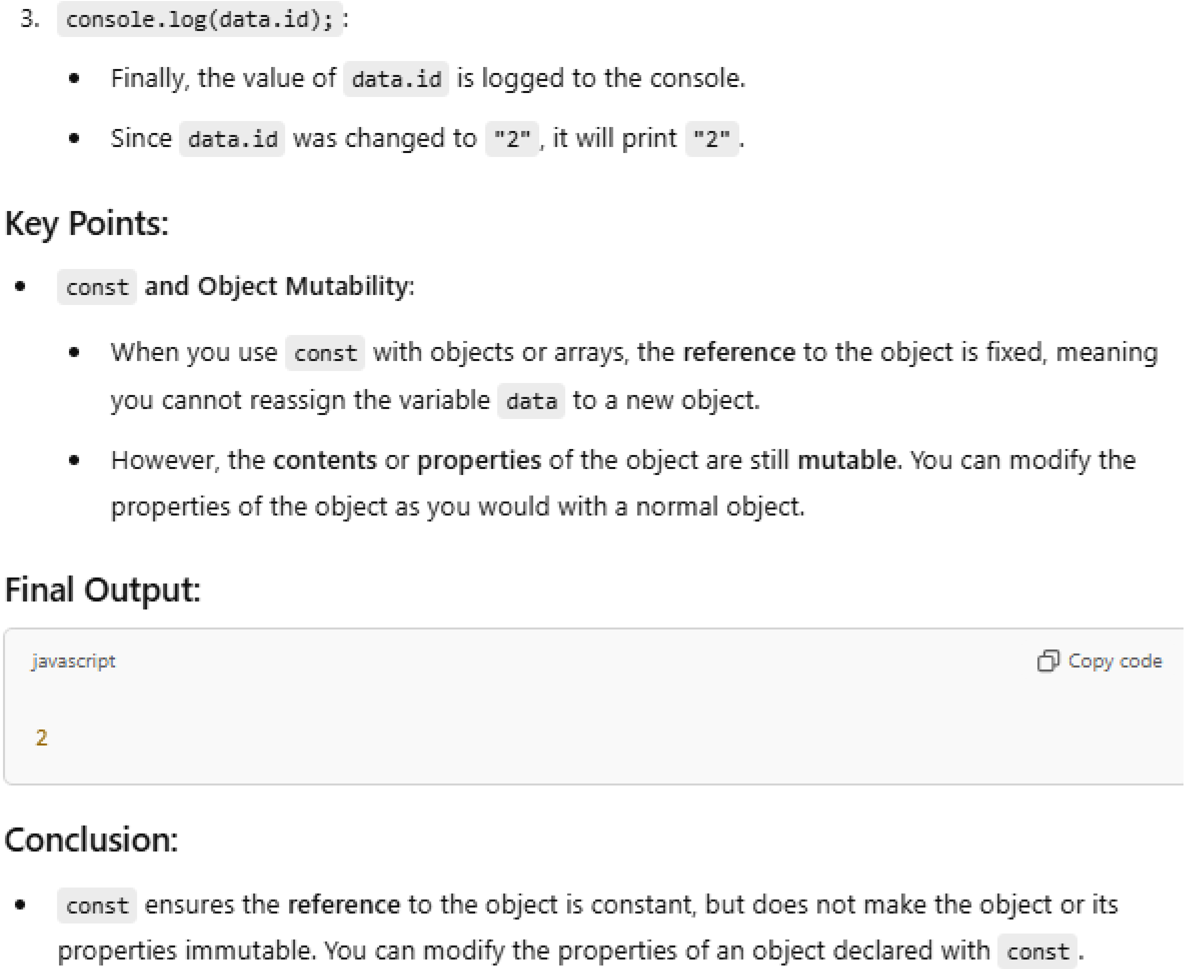
- **Synchronous code** (like `console.log("2")`) runs first.
- The **asynchronous code** (callback from `setTimeout`) runs after the synchronous code has completed, even if the delay is `0`. This demonstrates how the event loop works in JavaScript to handle asynchronous operations.



```javascript
const data = {
  id: "1"
};

data.id = "2";

console.log(data.id);
```

Step-by-step Explanation:

1. `const data = { id: "1" };`:
   - Here, a constant variable `data` is declared using `const`.
   - `const` ensures that the **reference** to the object cannot be changed (i.e., you cannot reassign `data` to a different object or primitive value).
   - However, the **properties** of the object are **mutable**. This means you can modify the properties of the object, even though the object itself is assigned to a constant variable.
2. `data.id = "2";`:
   - This line modifies the `id` property of the `data` object.
   - Even though `data` is a constant, the **properties of the object** can still be changed because the **object reference** is immutable, not the object itself.
   - The value of `data.id` is updated from `"1"` to `"2"`.
3. `console.log(data.id);`:
   - Finally, the value of `data.id` is logged to the console.
   - Since `data.id` was changed to `"2"`, it will print `"2"`.



3. `console.log(data.id);`:
   - Finally, the value of `data.id` is logged to the console.
   - Since `data.id` was changed to `"2"`, it will print `"2"`.

Key Points:

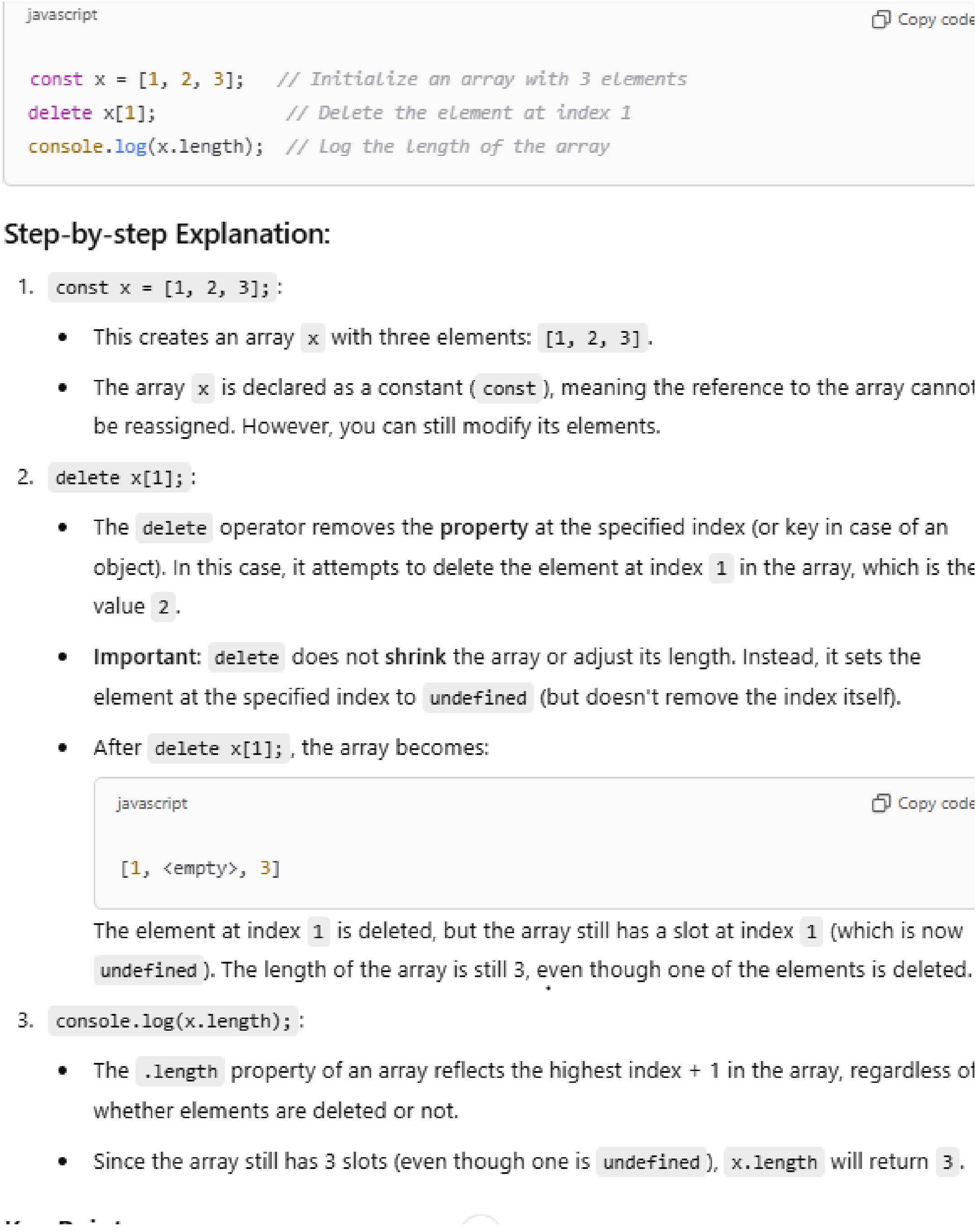
- `const` **and Object Mutability**:
  - When you use `const` with objects or arrays, the **reference** to the object is fixed, meaning you cannot reassign the variable `data` to a new object.
  - However, the **contents** or **properties** of the object are still **mutable**. You can modify the properties of the object as you would with a normal object.

Final Output:

```javascript
2
```

Conclusion:

- `const` ensures the **reference** to the object is constant, but does not make the object or its properties immutable. You can modify the properties of an object declared with `const`.



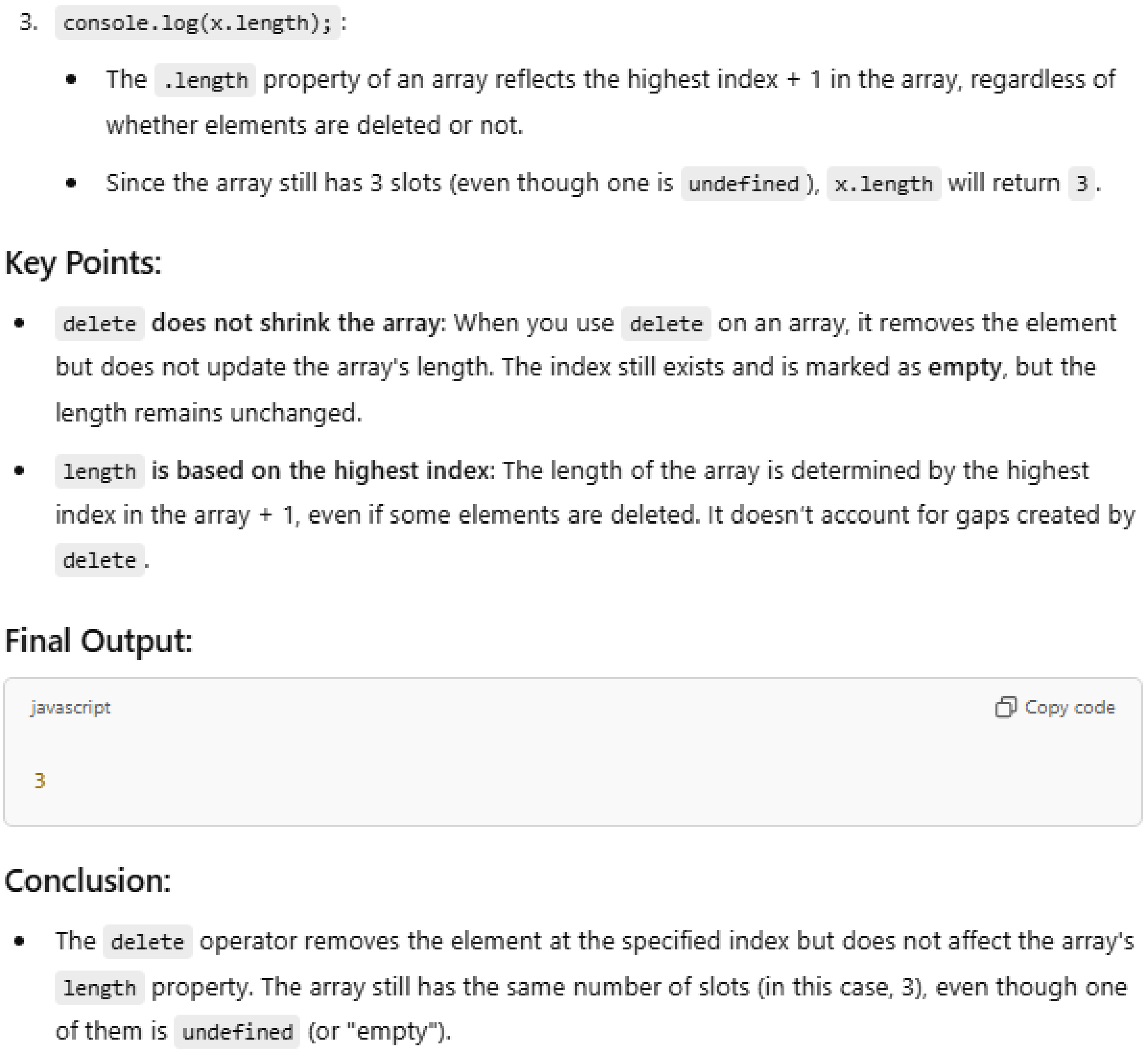
```javascript
const x = [1, 2, 3];   // Initialize an array with 3 elements
delete x[1];            // Delete the element at index 1
console.log(x.length);  // Log the length of the array
```

Step-by-step Explanation:

1. `const x = [1, 2, 3];`:
   - This creates an array `x` with three elements: `[1, 2, 3]`.
   - The array `x` is declared as a constant (`const`), meaning the reference to the array cannot be reassigned. However, you can still modify its elements.
2. `delete x[1];`:
   - The `delete` operator removes the **property** at the specified index (or key in case of an object). In this case, it attempts to delete the element at index `1` in the array, which is the value `2`.
   - **Important:** `delete` does not **shrink** the array or adjust its length. Instead, it sets the element at the specified index to `undefined` (but doesn't remove the index itself).
   - After `delete x[1];`, the array becomes:

     ```javascript
     [1, <empty>, 3]
     ```

     The element at index `1` is deleted, but the array still has a slot at index `1` (which is now `undefined`). The length of the array is still 3, even though one of the elements is deleted.
3. `console.log(x.length);`:
   - The `.length` property of an array reflects the highest index + 1 in the array, regardless of whether elements are deleted or not.
   - Since the array still has 3 slots (even though one is `undefined`), `x.length` will return `3`.



3. `console.log(x.length);`:
   - The `.length` property of an array reflects the highest index + 1 in the array, regardless of whether elements are deleted or not.
   - Since the array still has 3 slots (even though one is `undefined`), `x.length` will return `3`.

Key Points:

- `delete` **does not shrink the array**: When you use `delete` on an array, it removes the element but does not update the array's length. The index still exists and is marked as **empty**, but the length remains unchanged.
- `length` **is based on the highest index**: The length of the array is determined by the highest index in the array + 1, even if some elements are deleted. It doesn't account for gaps created by `delete`.

Final Output:

```javascript
3
```

Conclusion:

- The `delete` operator removes the element at the specified index but does not affect the array's `length` property. The array still has the same number of slots (in this case, 3), even though one of them is `undefined` (or "empty").