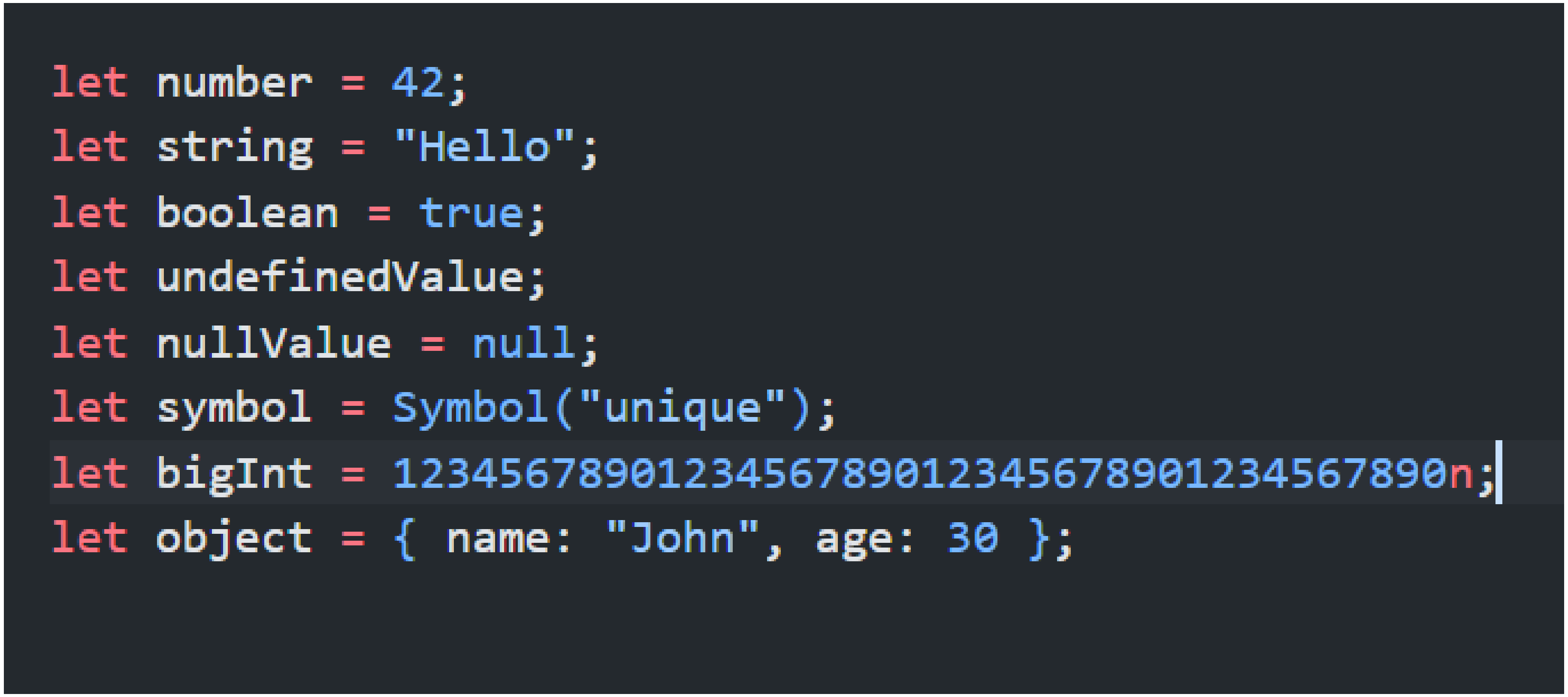
1. What are the data types in JavaScript?

JavaScript has eight data types, divided into two categories: primitive and non-primitive (object).

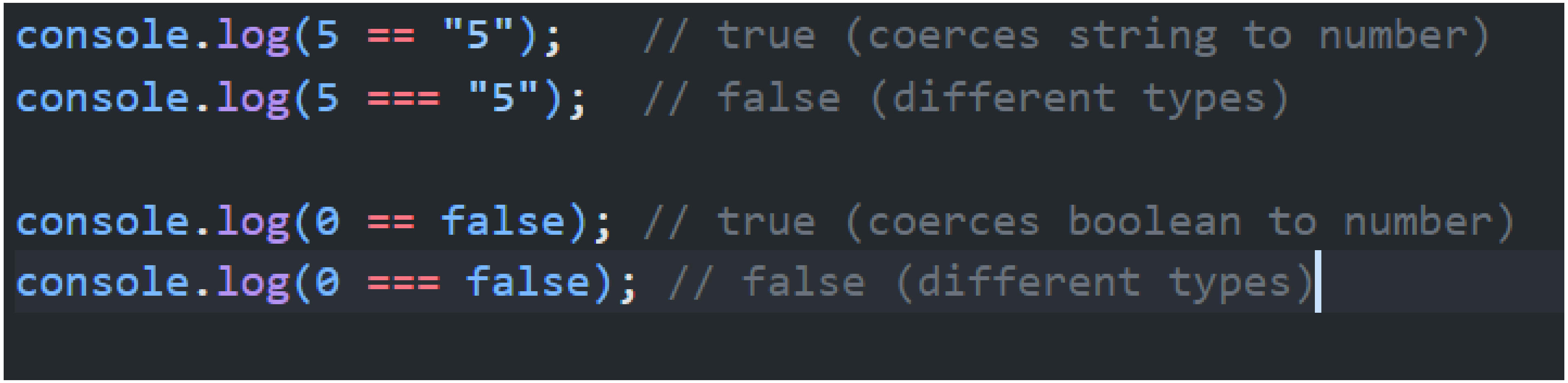


```
let number = 42;
let string = "Hello";
let boolean = true;
let undefinedValue;
let nullValue = null;
let symbol = Symbol("unique");
let bigInt = 1234567890123456789012345678901234567890n;
let object = { name: "John", age: 30 };
```

2. What is the difference between == and ===?

The == (equality) and === (strict equality) operators are used for comparison in JavaScript.
== (Equality): Performs type coercion before comparison and then it compares values after converting them to a common type.

=== (Strict Equality): Does not perform type coercion and it compares both value and type

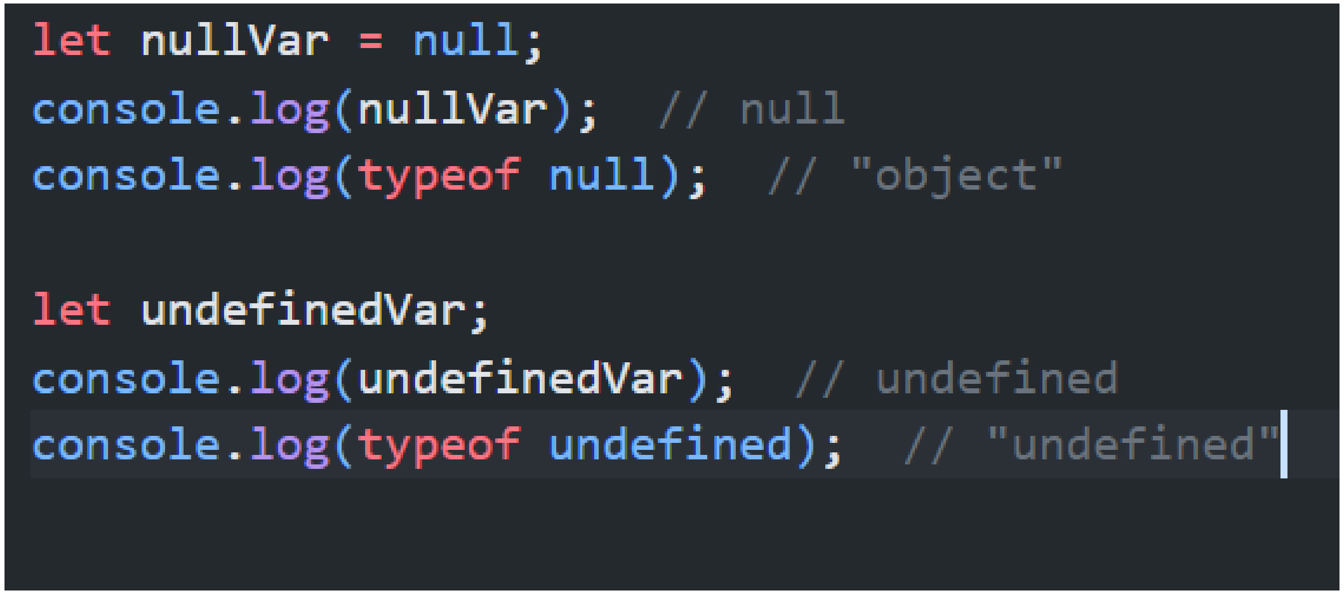


```
console.log(5 == "5");   // true (coerces string to number)
console.log(5 === "5");  // false (different types)

console.log(0 == false); // true (coerces boolean to number)
console.log(0 === false); // false (different types)
```

3. What is the difference between null and undefined?

Null and undefined are both used to represent the absence of a value.
Null: Must be explicitly assigned. Typeof null returns "object" (this is a known JavaScript bug)

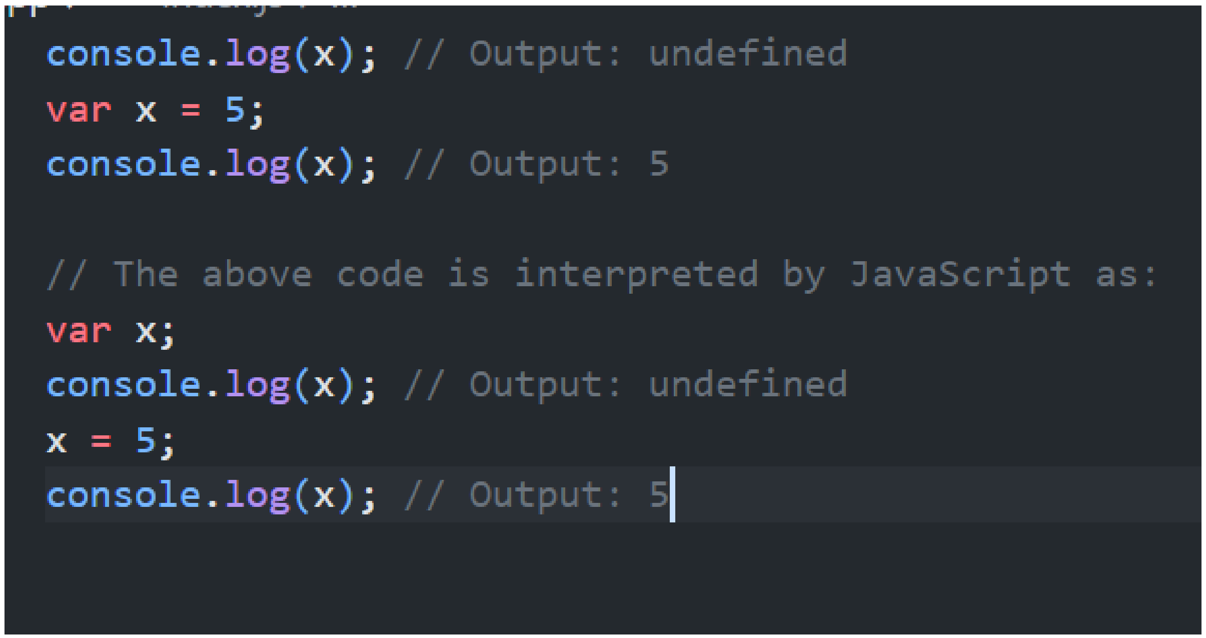
Undefined: Represents a variable that has been declared but not assigned a value .Typeof undefined returns "undefined"



```
let nullVar = null;
console.log(nullVar);  // null
console.log(typeof null);  // "object"

let undefinedVar;
console.log(undefinedVar);  // undefined
console.log(typeof undefined);  // "undefined"
```

4. Explain the concept of hoisting in JavaScript.

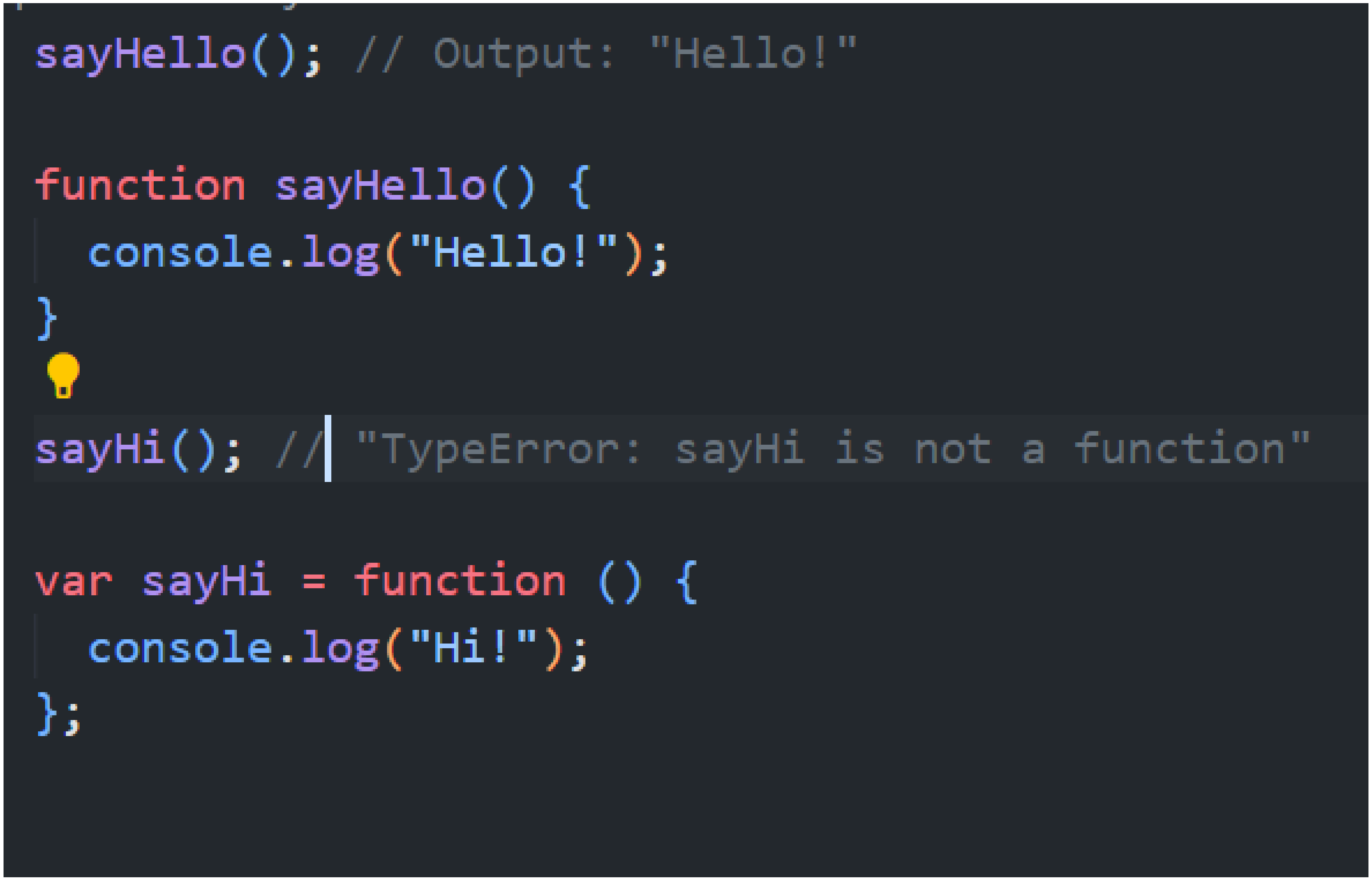
Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their respective scopes during the compilation phase, before the code is executed. This means that regardless of where variables and functions are declared in the code, they are treated as if they are declared at the beginning of their scope.



```
console.log(x); // Output: undefined
var x = 5;
console.log(x); // Output: 5

// The above code is interpreted by JavaScript as:
var x;
console.log(x); // Output: undefined
x = 5;
console.log(x); // Output: 5
```

the declaration of x is hoisted to the top, but not its initialization. That's why the first `console.log outputs undefined

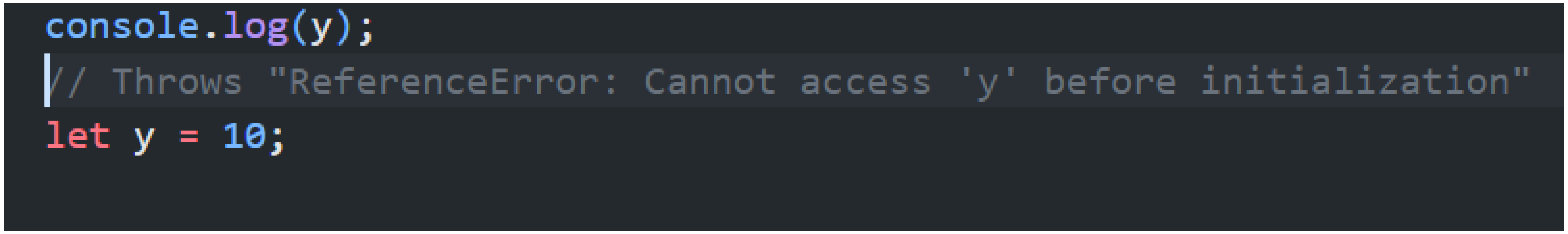
4. Explain the concept of hoisting in JavaScript.



```javascript
sayHello(); // Output: "Hello!"

function sayHello() {
  console.log("Hello!");
}

sayHi(); // "TypeError: sayHi is not a function"

var sayHi = function () {
  console.log("Hi!");
};
```

4. Explain the concept of hoisting in JavaScript.

let and const declarations are hoisted but not initialized. This leads to a "temporal dead zone" where accessing the variable before its declaration results in a ReferenceError



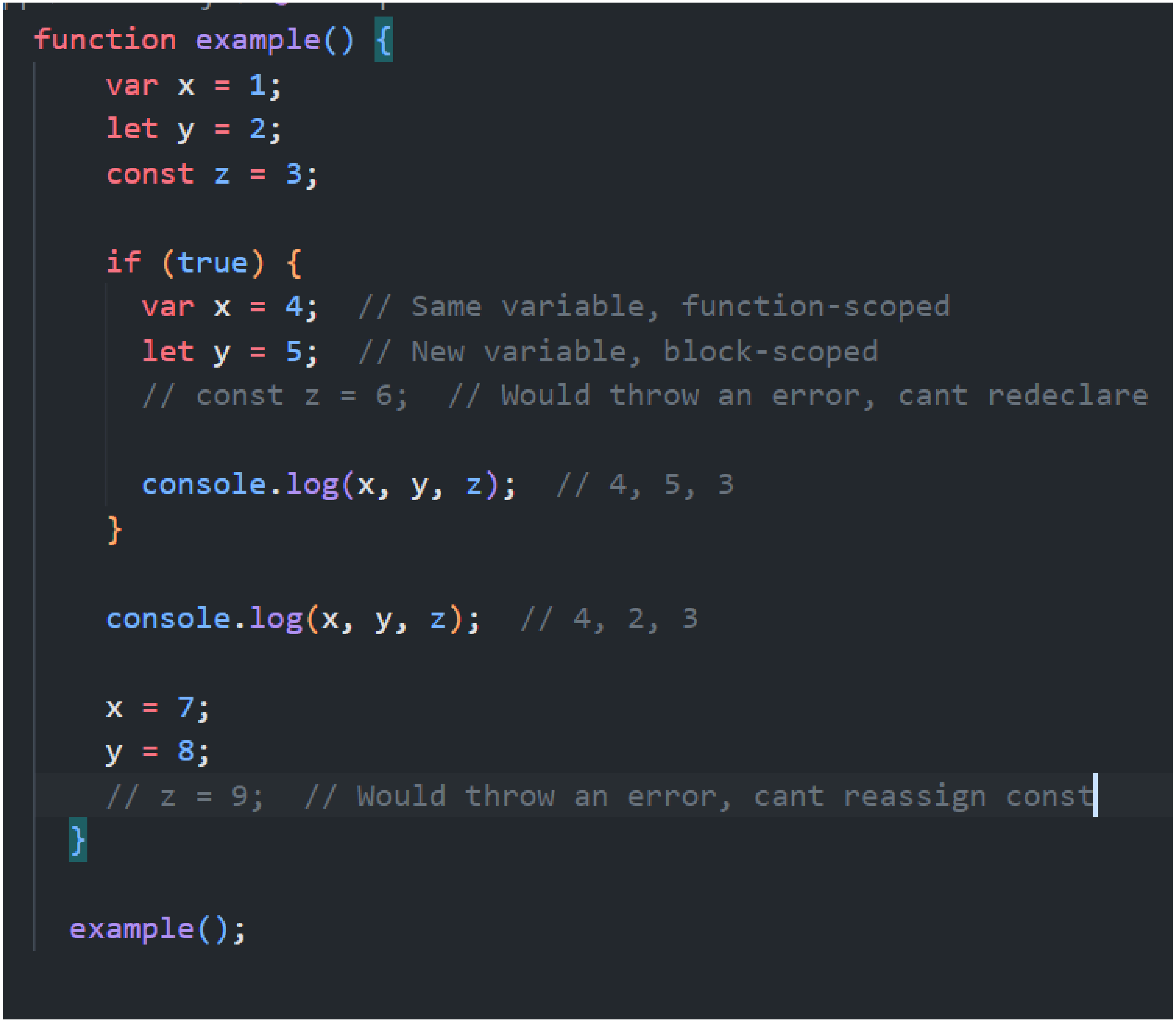
```javascript
console.log(y);
// Throws "ReferenceError: Cannot access 'y' before initialization"
let y = 10;
```

5. What is the difference between let, const, and var?

var: Function-scoped or globally-scoped. Can be redeclared and updated and Hoisted and initialized with undefined.

let: Block-scoped. Hoisted but not initialized (Temporal Dead Zone).

const: Block-scoped. Cannot be updated or redeclared. Hoisted but not initialized (Temporal Dead Zone)

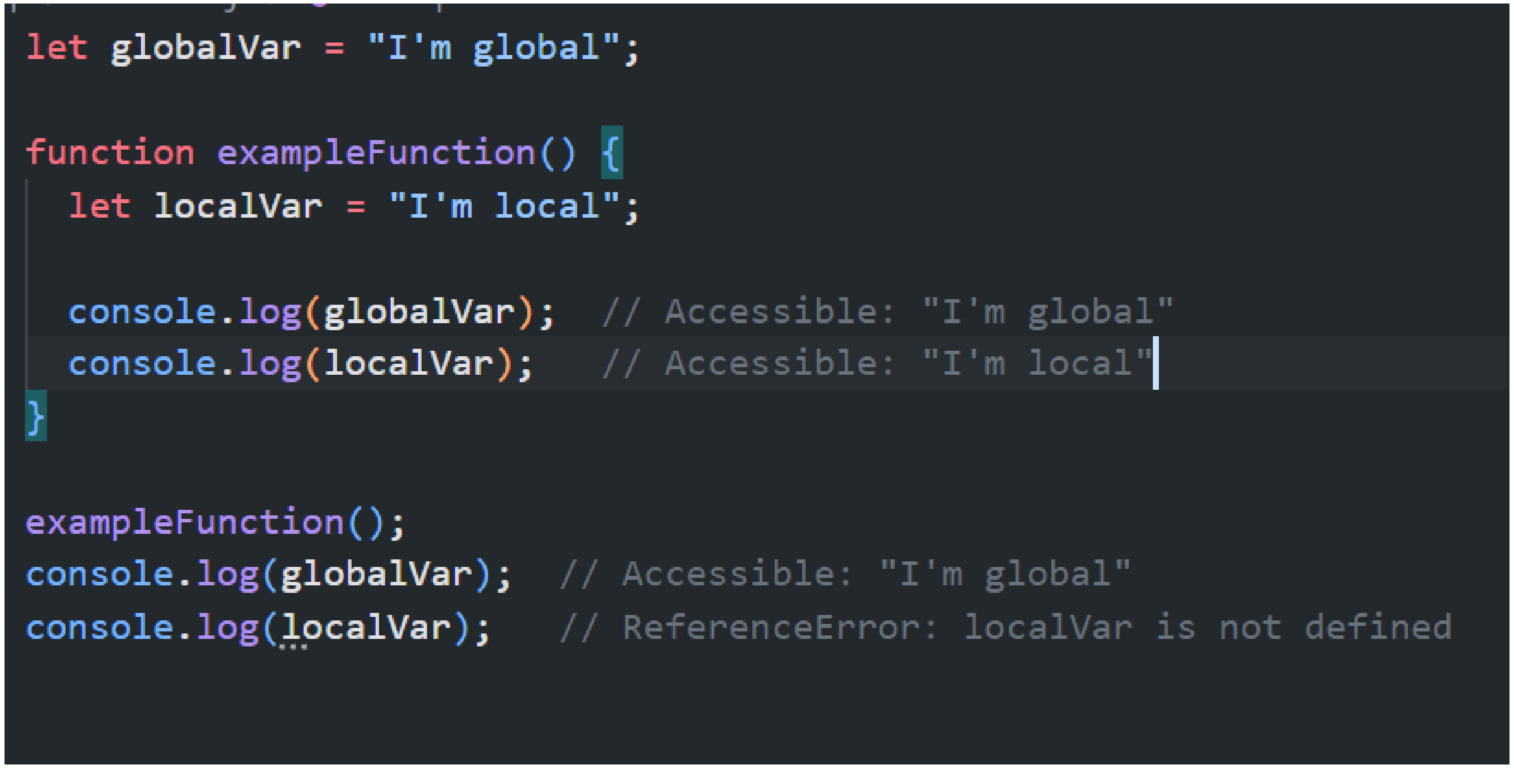


```
function example() {
    var x = 1;
    let y = 2;
    const z = 3;

    if (true) {
      var x = 4;  // Same variable, function-scoped
      let y = 5;  // New variable, block-scoped
      // const z = 6;  // Would throw an error, cant redeclare

      console.log(x, y, z);  // 4, 5, 3
    }

    console.log(x, y, z);  // 4, 2, 3

    x = 7;
    y = 8;
    // z = 9;  // Would throw an error, cant reassign const
  }

  example();
```

6. What is variable scope in JavaScript?

Variable scope refers to the context in which a variable is declared and can be accessed. In JavaScript, there are two main types of scope:
1. Global Scope: Variables declared outside any function or block
2. Local Scope: Variables declared inside a function or block

JavaScript uses lexical scoping, which means that inner functions have access to variables in their outer scope.

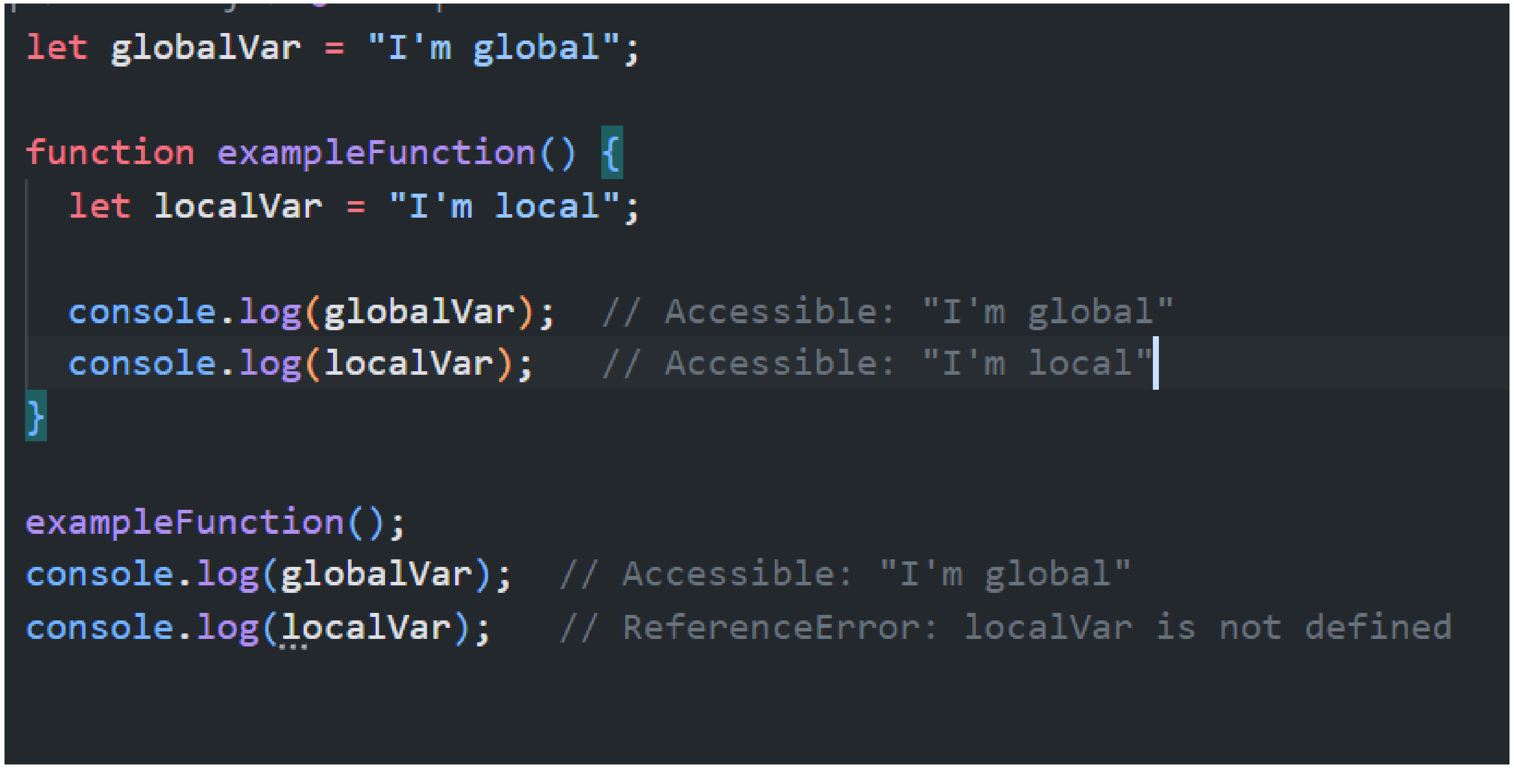


```
let globalVar = "I'm global";

function exampleFunction() {
  let localVar = "I'm local";

  console.log(globalVar);  // Accessible: "I'm global"
  console.log(localVar);   // Accessible: "I'm local"
}

exampleFunction();
console.log(globalVar);  // Accessible: "I'm global"
console.log(localVar);   // ReferenceError: localVar is not defined
```

7. Explain the difference between global and local variables.

Global Variables: Declared outside any function or block. Accessible from anywhere in the code, including inside functions. Have global scope

Local Variables: Declared inside a function or block. Only accessible within that function or block. Have local scope

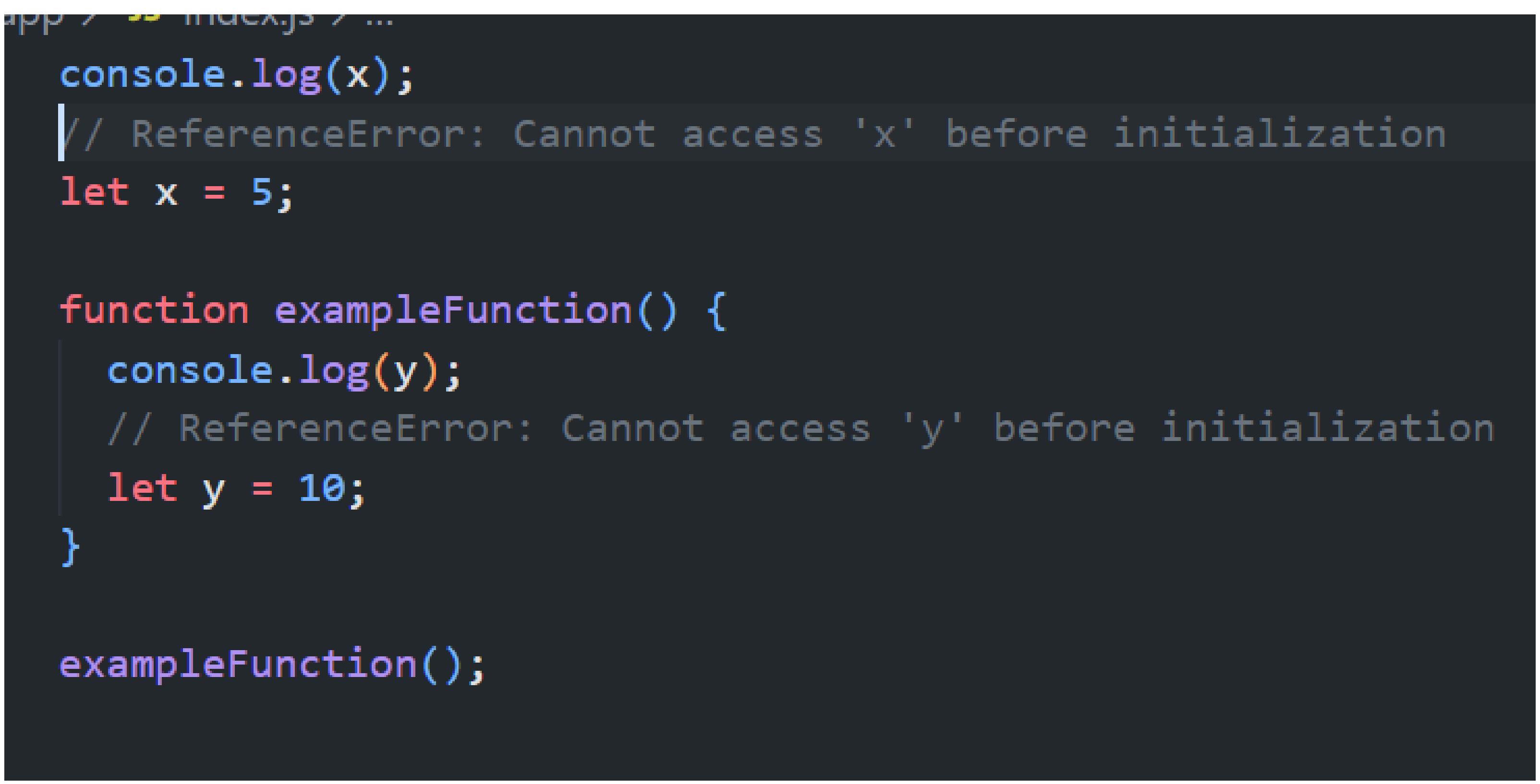


```javascript
let globalVar = "I'm global";

function exampleFunction() {
  let localVar = "I'm local";

  console.log(globalVar);  // Accessible: "I'm global"
  console.log(localVar);   // Accessible: "I'm local"
}

exampleFunction();
console.log(globalVar);  // Accessible: "I'm global"
console.log(localVar);   // ReferenceError: localVar is not defined
```

8. What is the temporal dead zone?

The Temporal Dead Zone (TDZ) is the period between entering a scope and the point where a variable is declared and initialized.

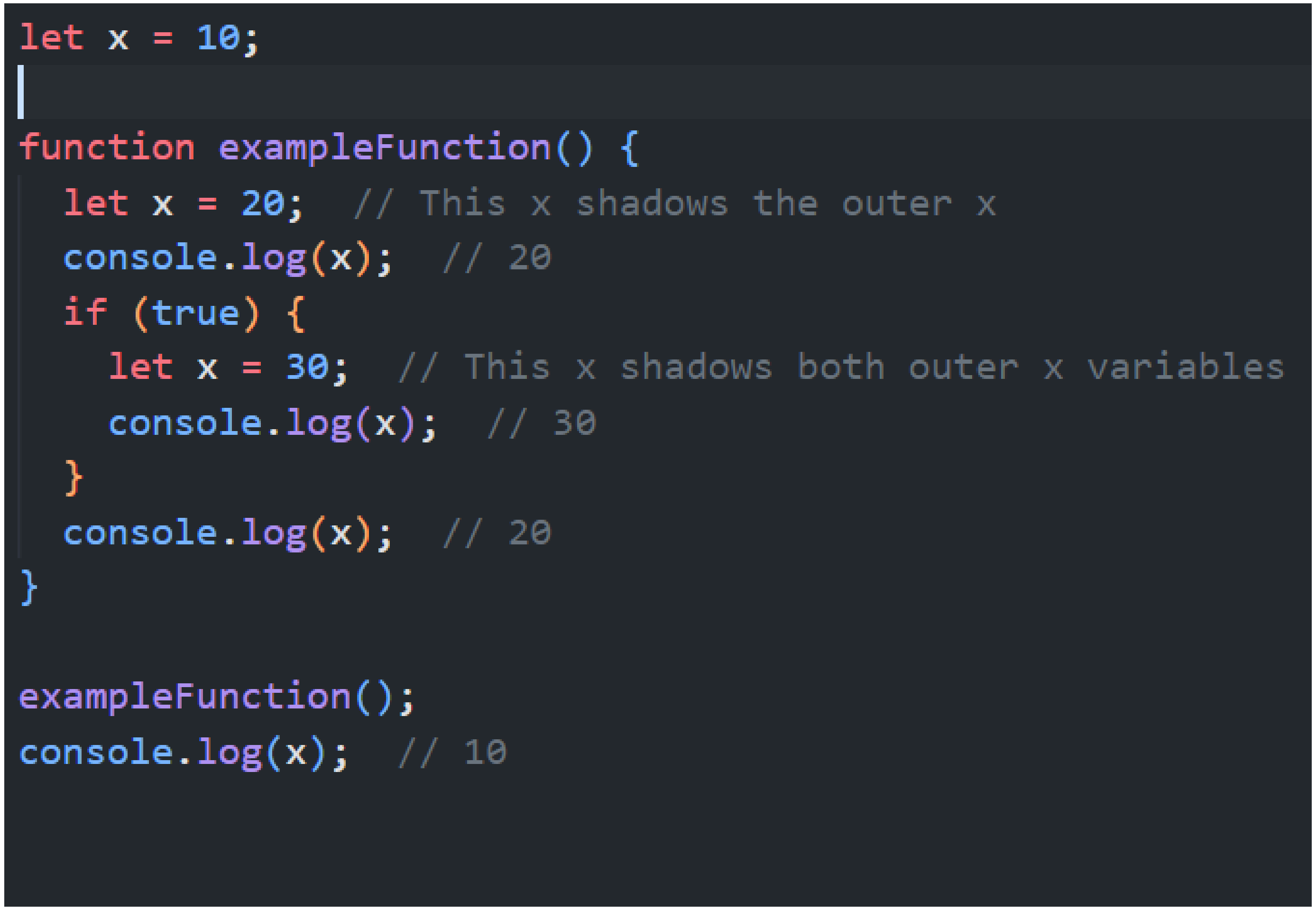
Applies to variables declared with let and const.
Helps catch errors by preventing access to variables before they're declared



```
console.log(x);
// ReferenceError: Cannot access 'x' before initialization
let x = 5;

function exampleFunction() {
  console.log(y);
  // ReferenceError: Cannot access 'y' before initialization
  let y = 10;
}

exampleFunction();
```

9. What is variable shadowing?

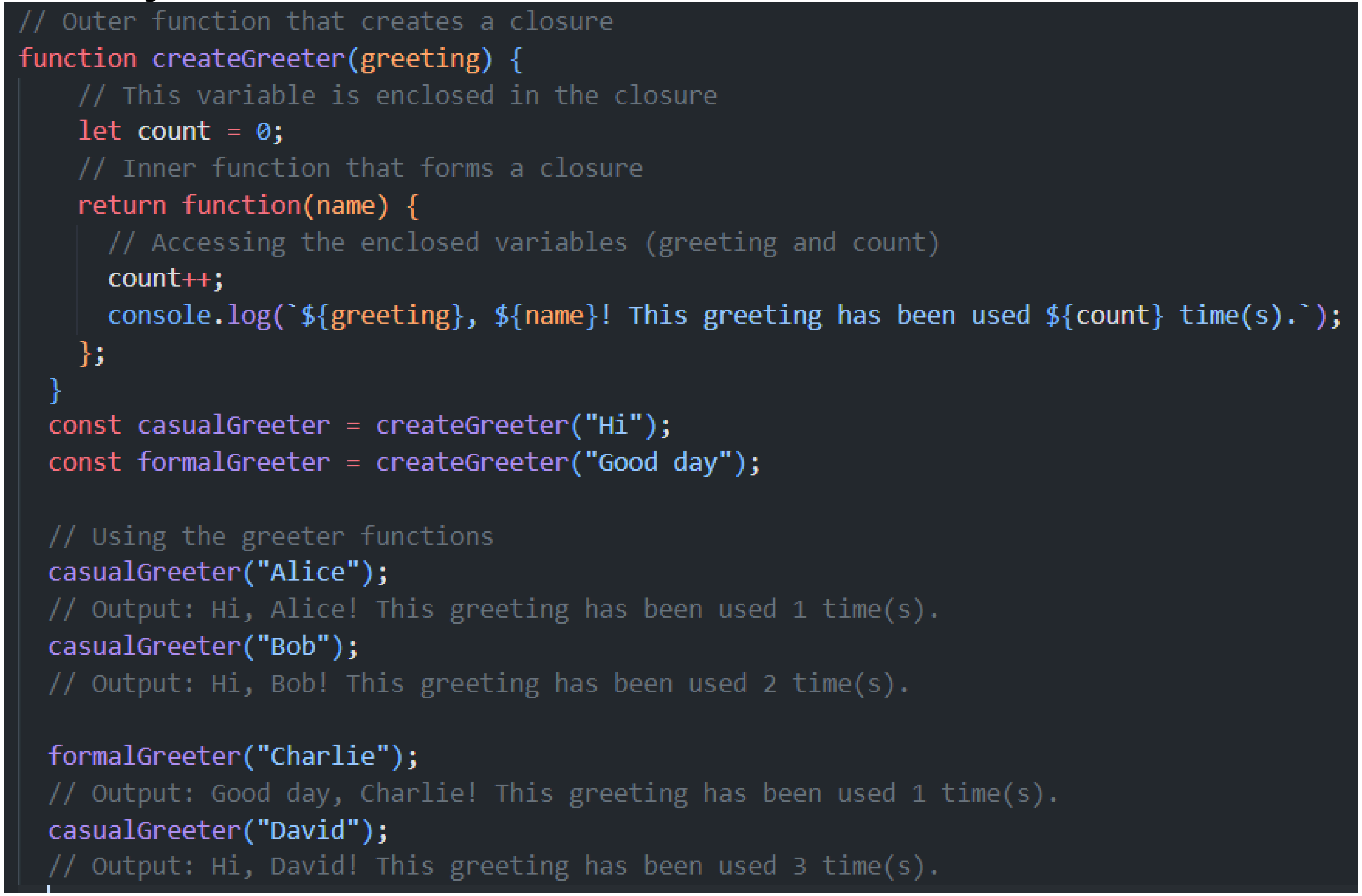
Variable shadowing occurs when a variable declared in a certain scope has the same name as a variable in an outer scope. The inner variable "shadows" the outer one, effectively hiding it.



```javascript
let x = 10;

function exampleFunction() {
  let x = 20;  // This x shadows the outer x
  console.log(x);  // 20
  if (true) {
    let x = 30;  // This x shadows both outer x variables
    console.log(x);  // 30
  }
  console.log(x);  // 20
}

exampleFunction();
console.log(x);  // 10
```

10. What is a closure in JavaScript?

A closure in JavaScript is a function that has access to variables in its outer (enclosing) lexical scope, even after the outer function has returned. Closures are created every time a function is created

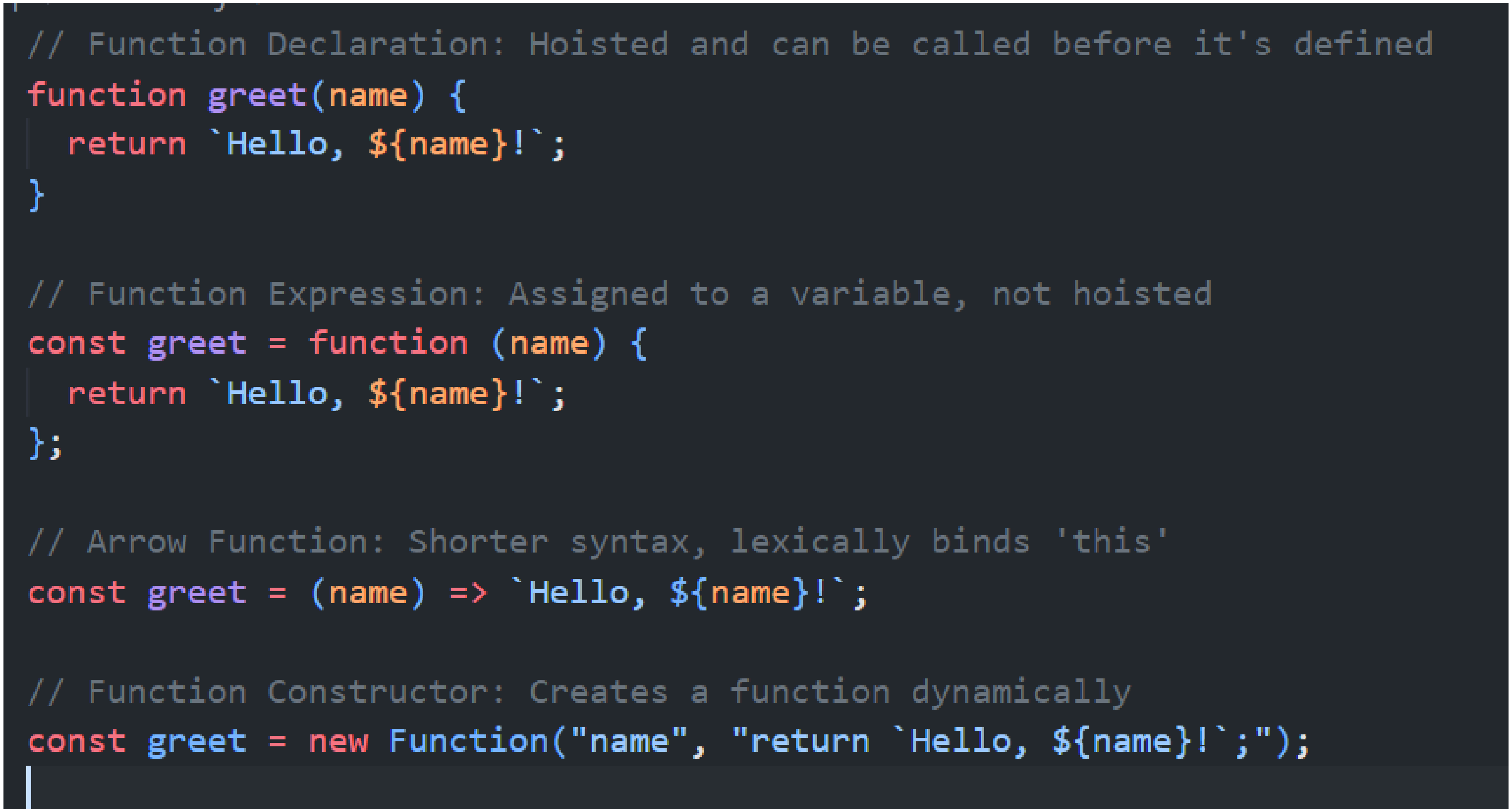


```javascript
// Outer function that creates a closure
function createGreeter(greeting) {
    // This variable is enclosed in the closure
    let count = 0;
    // Inner function that forms a closure
    return function(name) {
      // Accessing the enclosed variables (greeting and count)
      count++;
      console.log(`${greeting}, ${name}! This greeting has been used ${count} time(s).`);
    };
  }
  const casualGreeter = createGreeter("Hi");
  const formalGreeter = createGreeter("Good day");

  // Using the greeter functions
  casualGreeter("Alice");
  // Output: Hi, Alice! This greeting has been used 1 time(s).
  casualGreeter("Bob");
  // Output: Hi, Bob! This greeting has been used 2 time(s).

  formalGreeter("Charlie");
  // Output: Good day, Charlie! This greeting has been used 1 time(s).
  casualGreeter("David");
  // Output: Hi, David! This greeting has been used 3 time(s).
```

10. What is a closure in JavaScript?

1. **The createGreeter function returns an inner function that forms a closure.**
2. **The inner function has access to the \`greeting\` parameter and the \`count\` variable from its outer scope.**
3. **Each time we call createGreeter, it creates a new closure with its own enclosed count variable.**
4. **The returned functions (casualGreeter and formalGreeter) maintain their own separate counts, demonstrating how closures preserve state.**

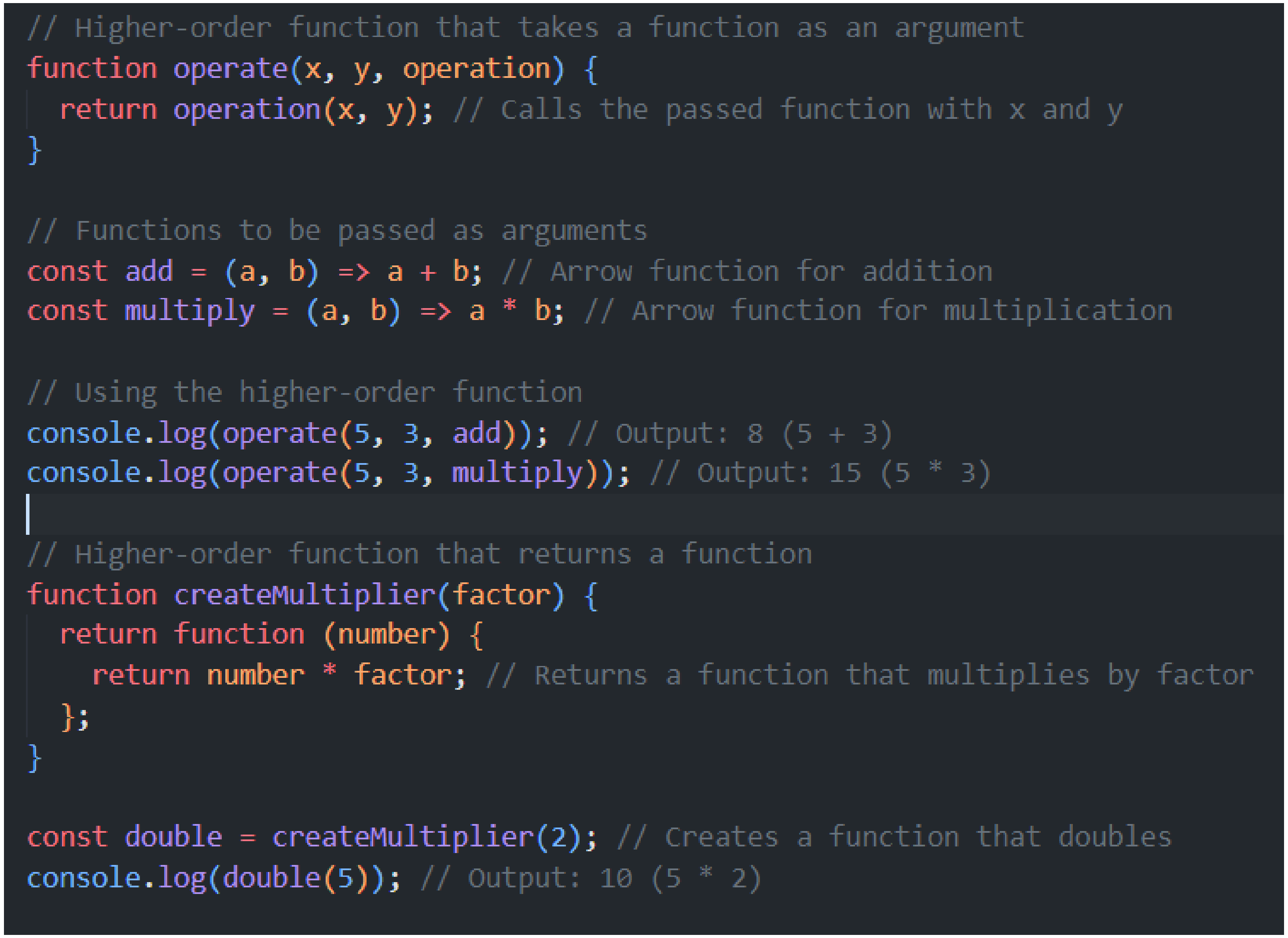
11. What are the different ways to define a function in JavaScript?



```javascript
// Function Declaration: Hoisted and can be called before it's defined
function greet(name) {
  return `Hello, ${name}!`;
}

// Function Expression: Assigned to a variable, not hoisted
const greet = function (name) {
  return `Hello, ${name}!`;
};

// Arrow Function: Shorter syntax, lexically binds 'this'
const greet = (name) => `Hello, ${name}!`;

// Function Constructor: Creates a function dynamically
const greet = new Function("name", "return `Hello, ${name}!`;");
```

12. What is a higher-order function?

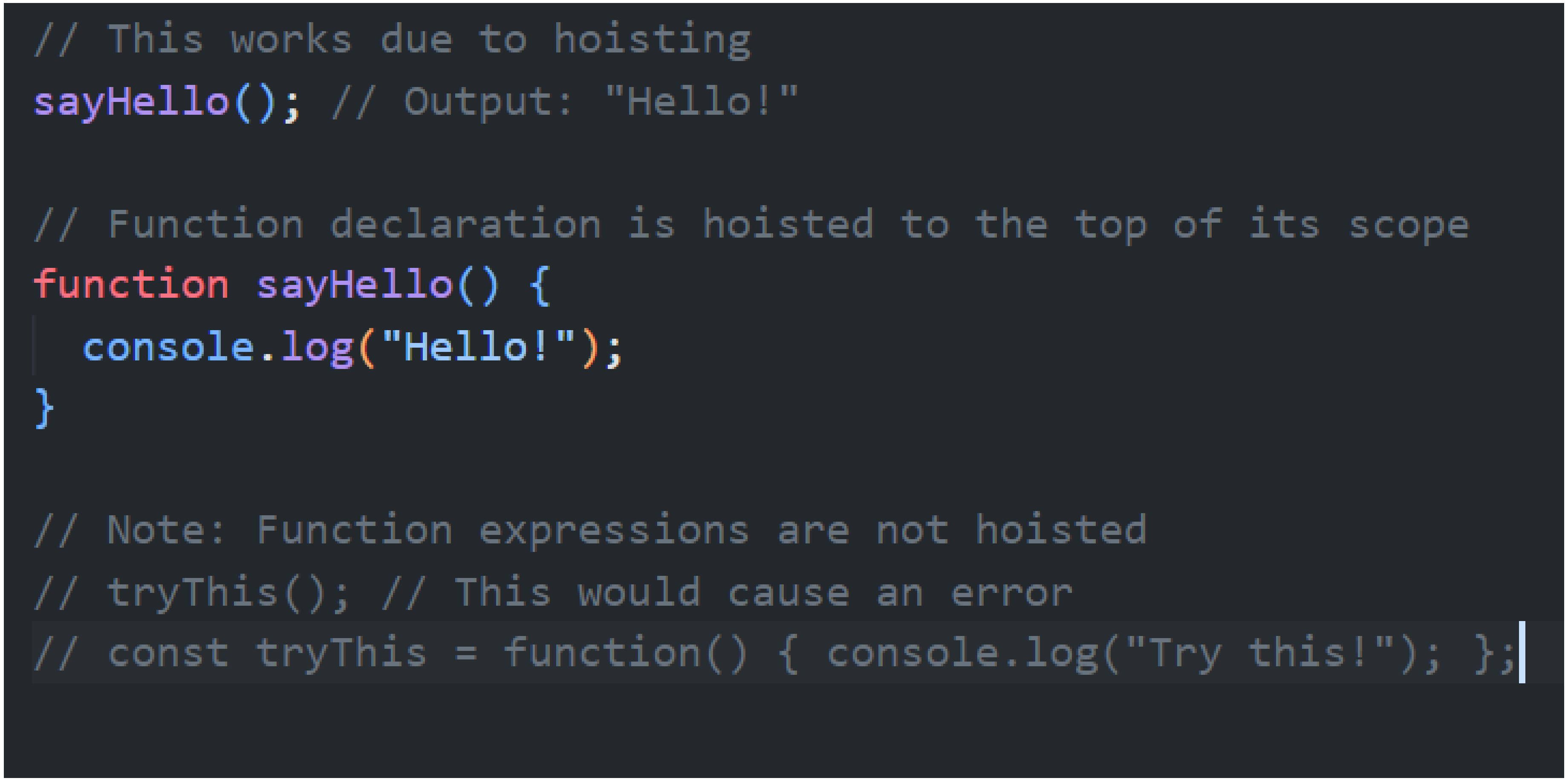
A higher-order function is a function that treats other functions as data, either by taking them as arguments or returning them.



```javascript
// Higher-order function that takes a function as an argument
function operate(x, y, operation) {
  return operation(x, y); // Calls the passed function with x and y
}

// Functions to be passed as arguments
const add = (a, b) => a + b; // Arrow function for addition
const multiply = (a, b) => a * b; // Arrow function for multiplication

// Using the higher-order function
console.log(operate(5, 3, add)); // Output: 8 (5 + 3)
console.log(operate(5, 3, multiply)); // Output: 15 (5 * 3)

// Higher-order function that returns a function
function createMultiplier(factor) {
  return function (number) {
    return number * factor; // Returns a function that multiplies by factor
  };
}

const double = createMultiplier(2); // Creates a function that doubles
console.log(double(5)); // Output: 10 (5 * 2)
```

13. Explain the concept of function hoisting.

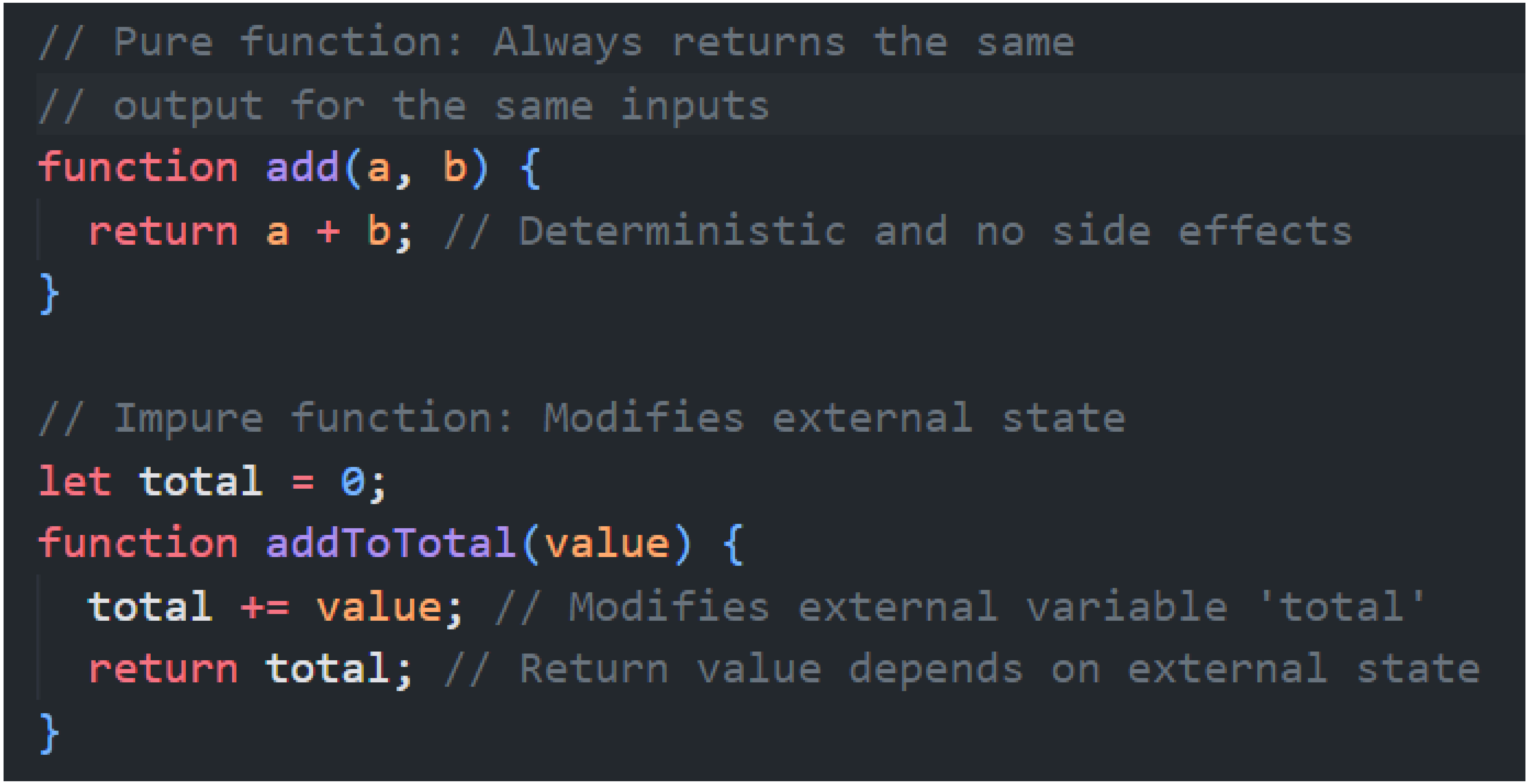
Function hoisting is JavaScript's behavior of moving function declarations to the top of their scope during the compilation phase. This allows you to call a function before it appears to be defined in the code.



```javascript
// This works due to hoisting
sayHello(); // Output: "Hello!"

// Function declaration is hoisted to the top of its scope
function sayHello() {
  console.log("Hello!");
}

// Note: Function expressions are not hoisted
// tryThis(); // This would cause an error
// const tryThis = function() { console.log("Try this!"); };
```

14. What is a pure function?

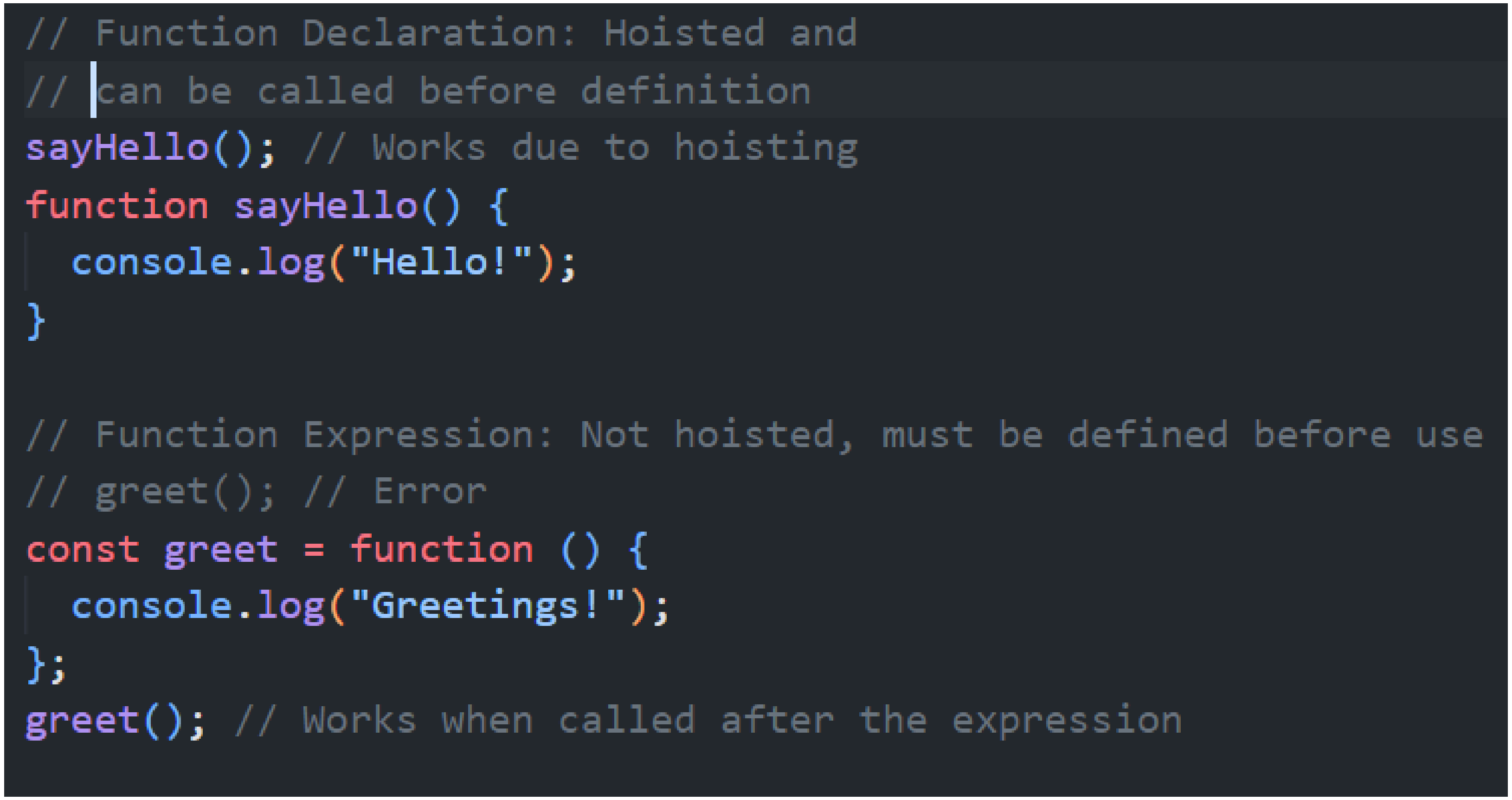
A pure function is a function that: Always returns the same output for the same inputs. Has no side effects (doesn't modify external state). Doesn't rely on external state. Make code more predictable and easier to test.



```javascript
// Pure function: Always returns the same
// output for the same inputs
function add(a, b) {
  return a + b; // Deterministic and no side effects
}

// Impure function: Modifies external state
let total = 0;
function addToTotal(value) {
  total += value; // Modifies external variable 'total'
  return total; // Return value depends on external state
}
```

15. Difference between function declaration and function expression?

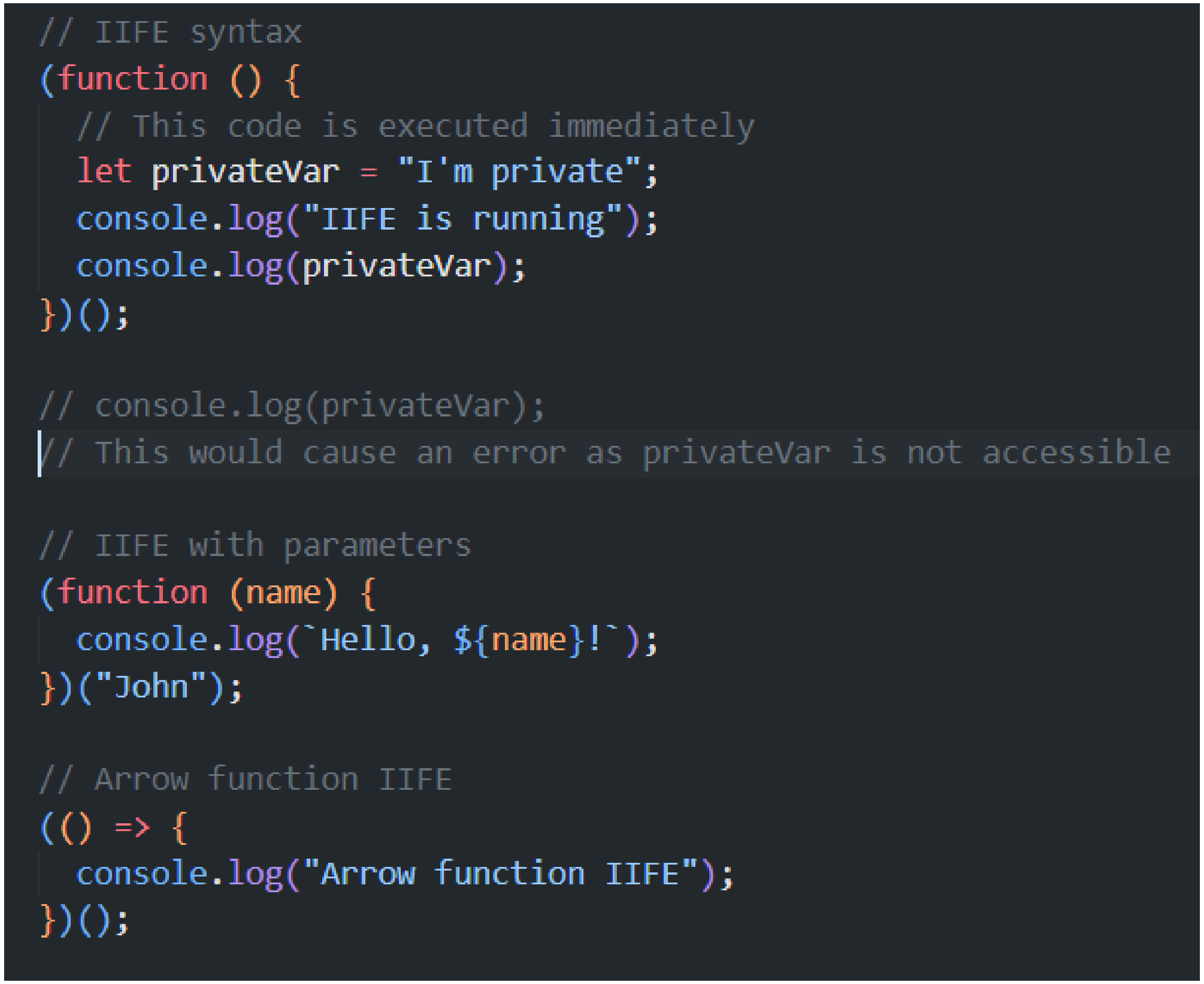
Hoisting: Function declarations are hoisted, function expressions are not.
Usage: Function declarations can be called before they appear in the code, function expressions cannot.
Naming: Function declarations require a name, function expressions can be anonymous.



```javascript
// Function Declaration: Hoisted and
// can be called before definition
sayHello(); // Works due to hoisting
function sayHello() {
  console.log("Hello!");
}

// Function Expression: Not hoisted, must be defined before use
// greet(); // Error
const greet = function () {
  console.log("Greetings!");
};
greet(); // Works when called after the expression
```

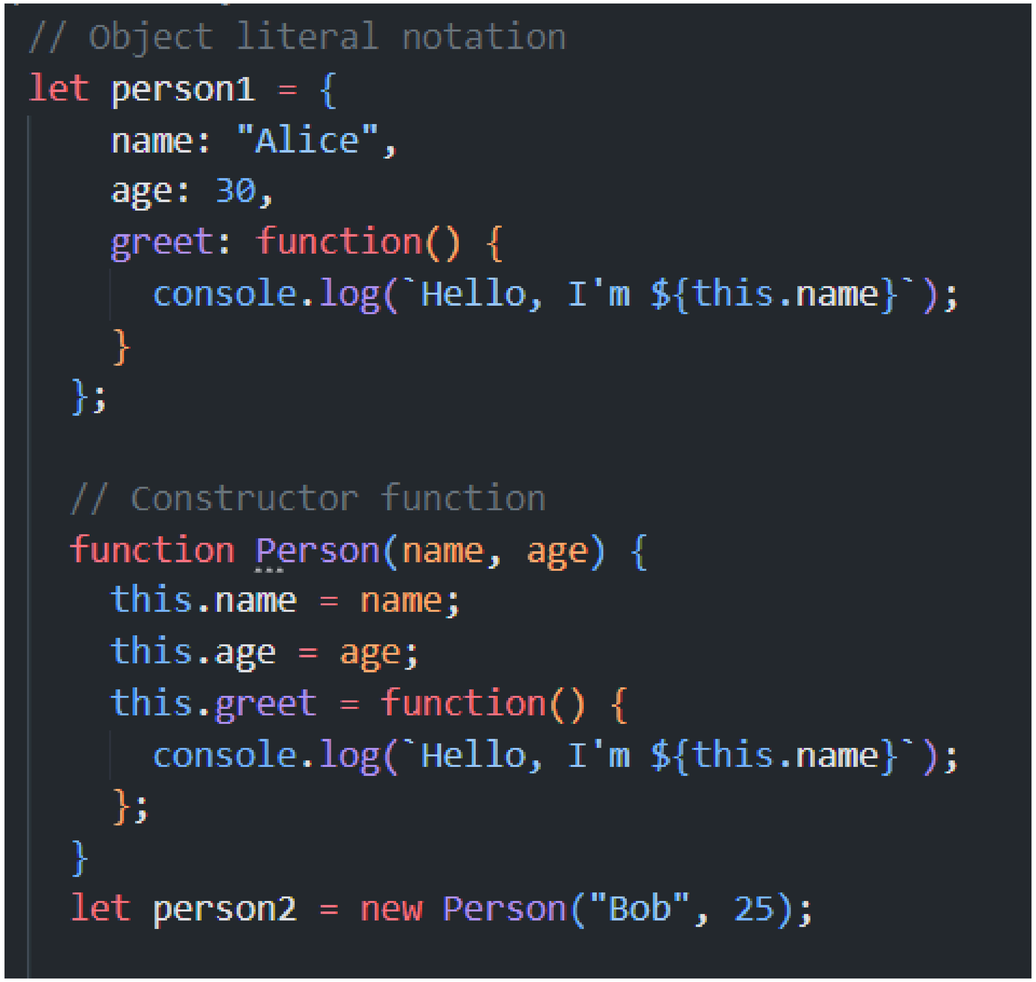
16. What is an Immediately Invoked Function Expression (IIFE)?

An IIFE is a JavaScript function that runs as soon as it is defined.

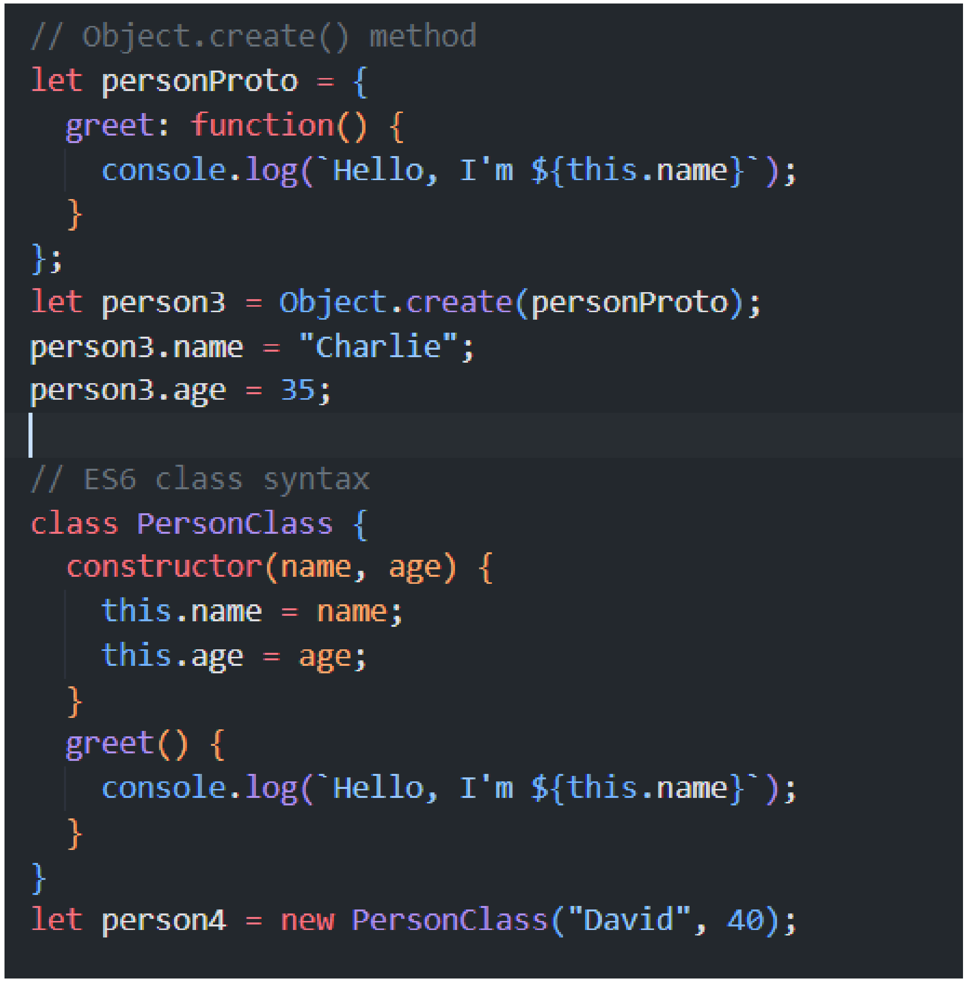


```javascript
// IIFE syntax
(function () {
  // This code is executed immediately
  let privateVar = "I'm private";
  console.log("IIFE is running");
  console.log(privateVar);
})();

// console.log(privateVar);
// This would cause an error as privateVar is not accessible

// IIFE with parameters
(function (name) {
  console.log(`Hello, ${name}!`);
})("John");

// Arrow function IIFE
(() => {
  console.log("Arrow function IIFE");
})();
```

17. How do you create an object in JavaScript?

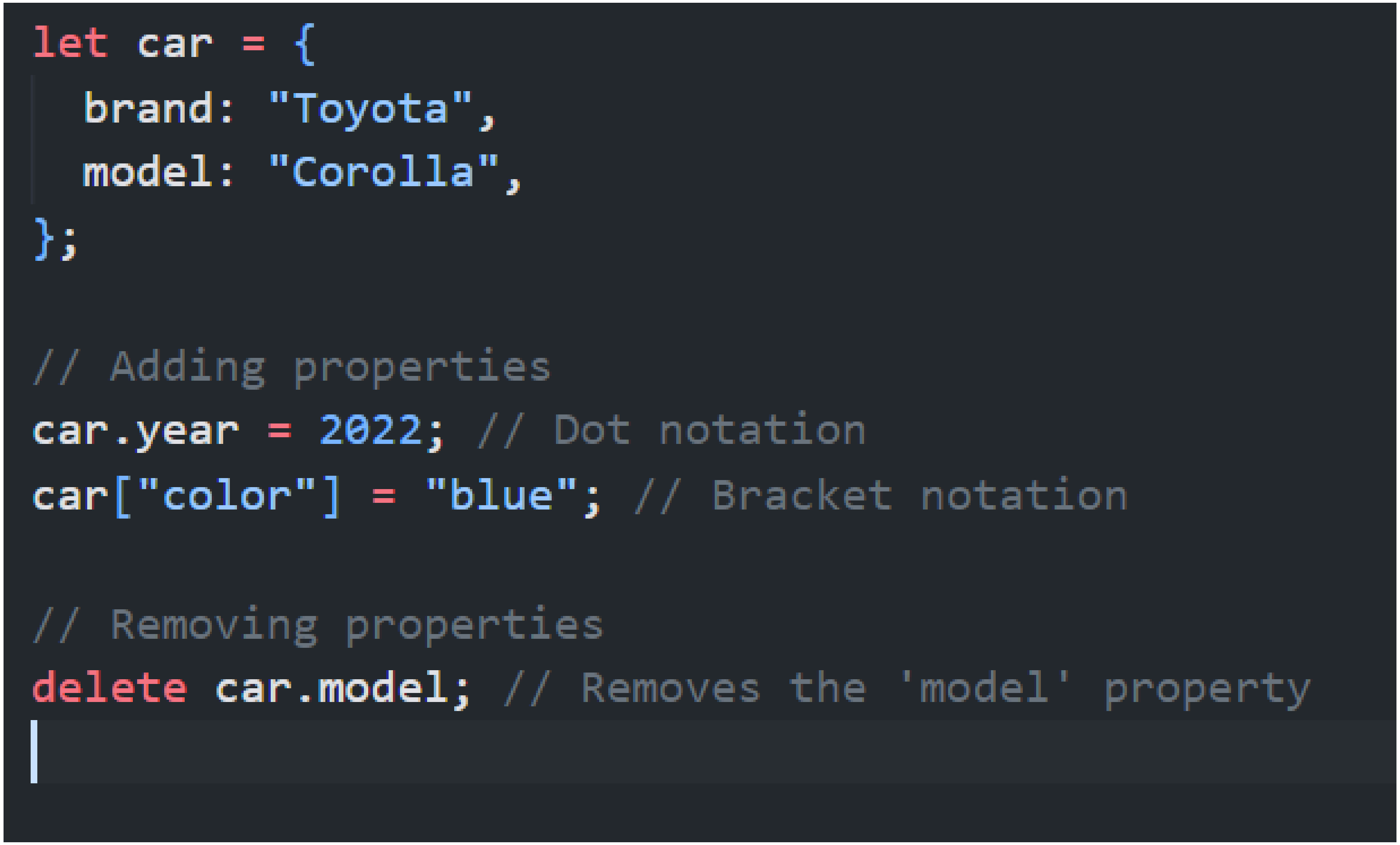


```javascript
// Object literal notation
let person1 = {
    name: "Alice",
    age: 30,
    greet: function() {
      console.log(`Hello, I'm ${this.name}`);
    }
  };

  // Constructor function
  function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
      console.log(`Hello, I'm ${this.name}`);
    };
  }
  let person2 = new Person("Bob", 25);
```



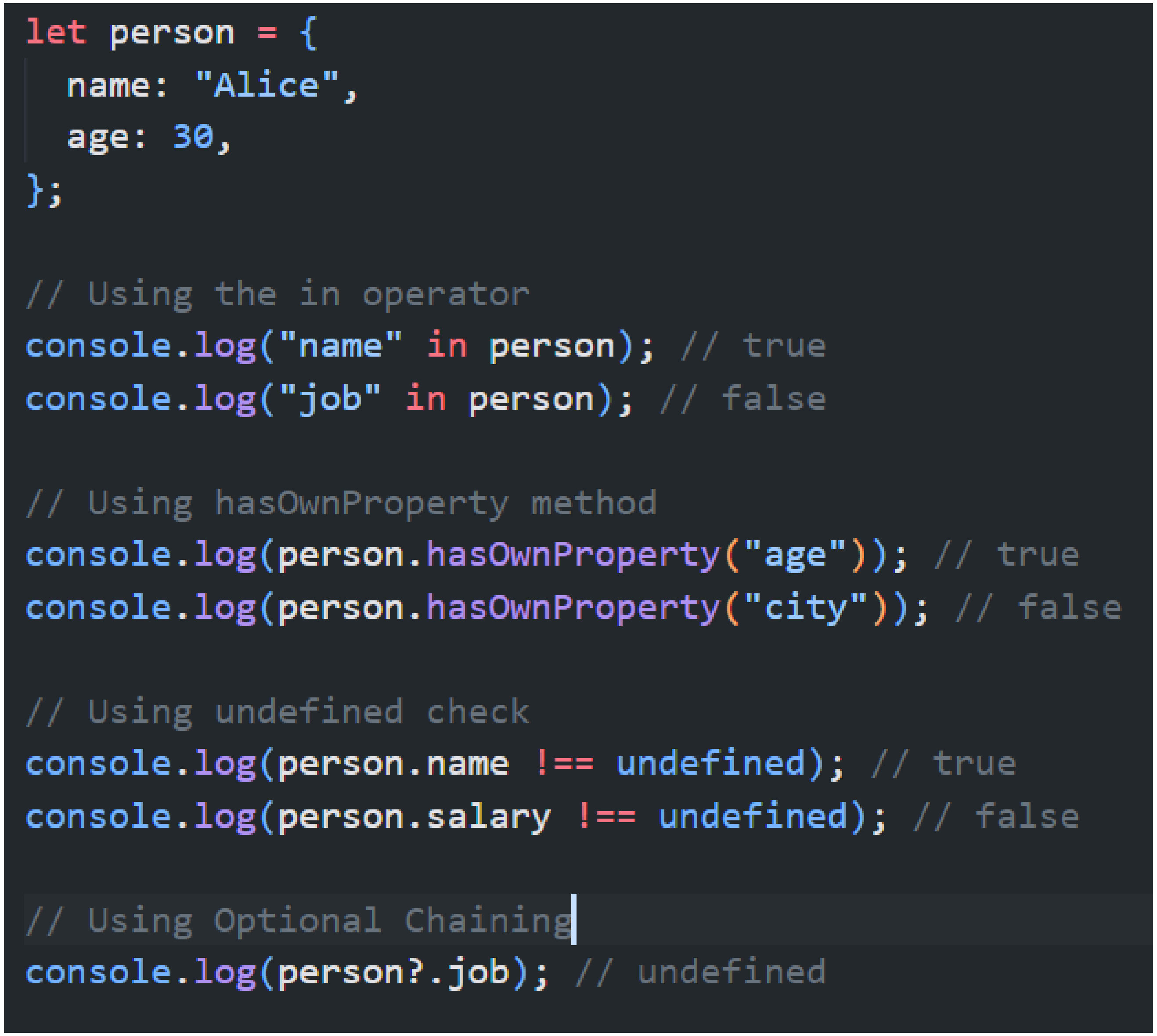
```javascript
// Object.create() method
let personProto = {
  greet: function() {
    console.log(`Hello, I'm ${this.name}`);
  }
};
let person3 = Object.create(personProto);
person3.name = "Charlie";
person3.age = 35;

// ES6 class syntax
class PersonClass {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hello, I'm ${this.name}`);
  }
}
let person4 = new PersonClass("David", 40);
```

18. How do you add/remove properties to an object dynamically?



```javascript
let car = {
  brand: "Toyota",
  model: "Corolla",
};

// Adding properties
car.year = 2022; // Dot notation
car["color"] = "blue"; // Bracket notation

// Removing properties
delete car.model; // Removes the 'model' property
```

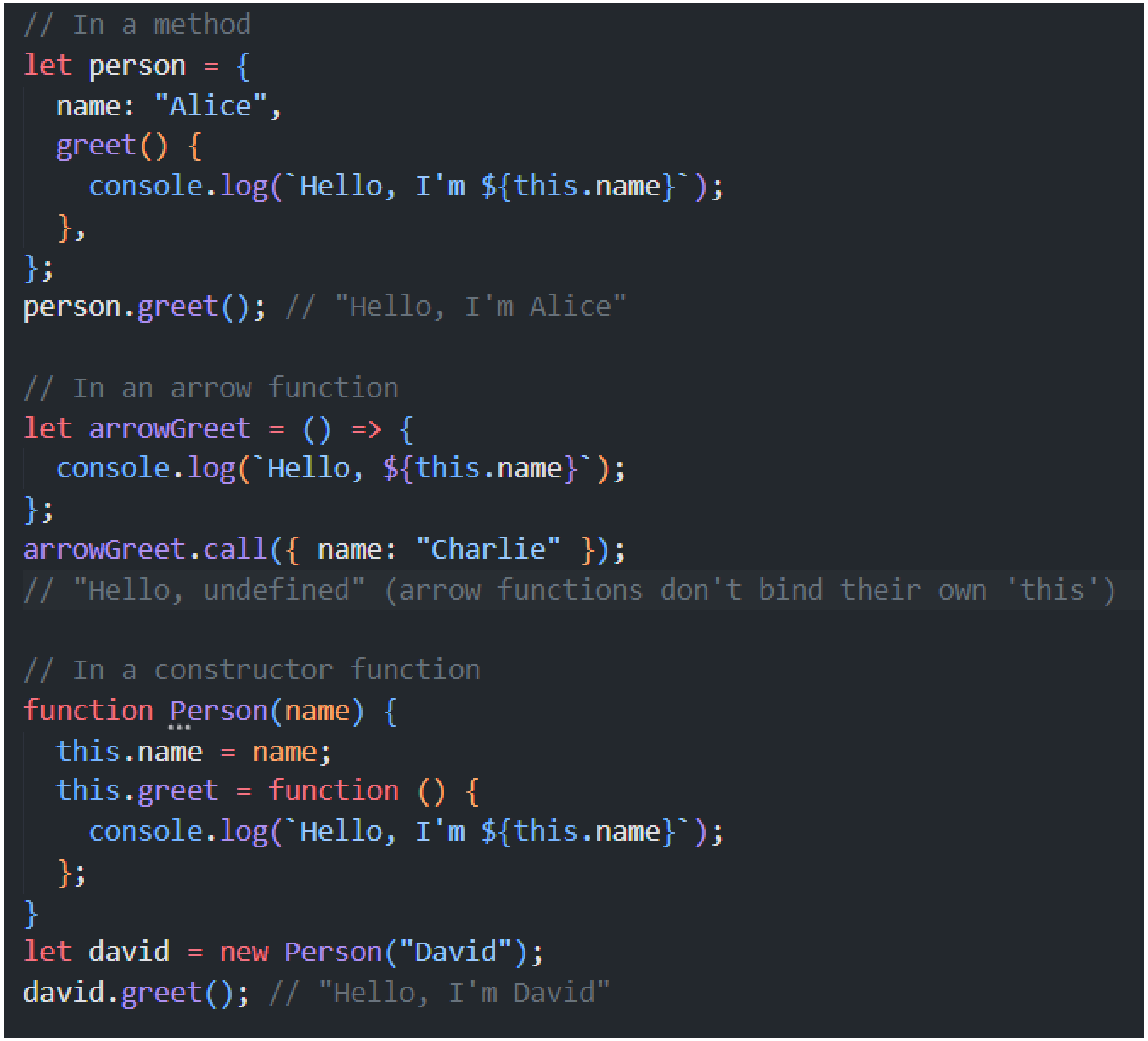
19. How do you check if a property exists in an object?



```javascript
let person = {
  name: "Alice",
  age: 30,
};

// Using the in operator
console.log("name" in person); // true
console.log("job" in person); // false

// Using hasOwnProperty method
console.log(person.hasOwnProperty("age")); // true
console.log(person.hasOwnProperty("city")); // false

// Using undefined check
console.log(person.name !== undefined); // true
console.log(person.salary !== undefined); // false

// Using Optional Chaining
console.log(person?.job); // undefined
```

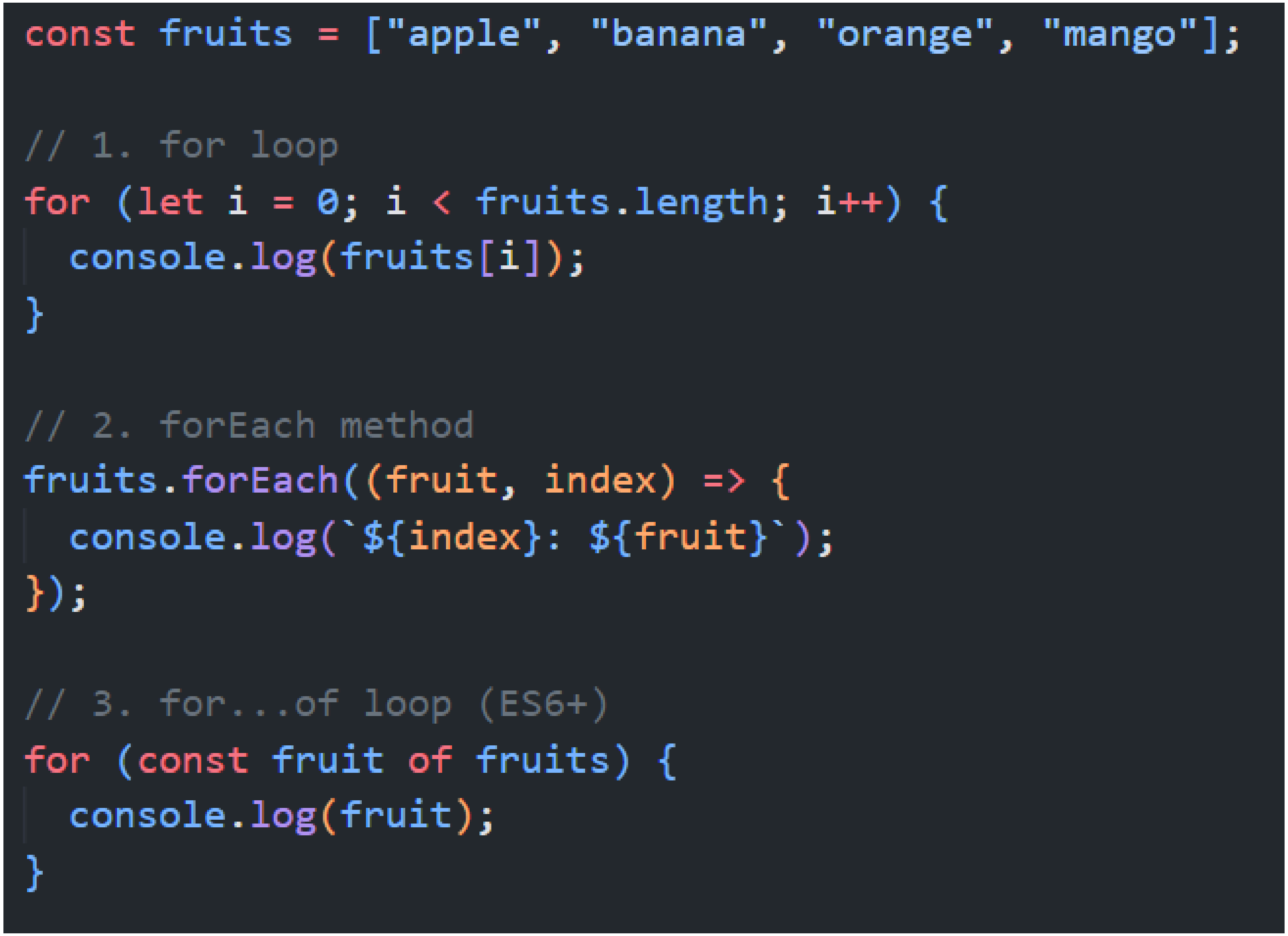
20. What is the “this” keyword in JavaScript?

The `this` keyword refers to the object that is executing the current function. Its value is determined by how a function is called.



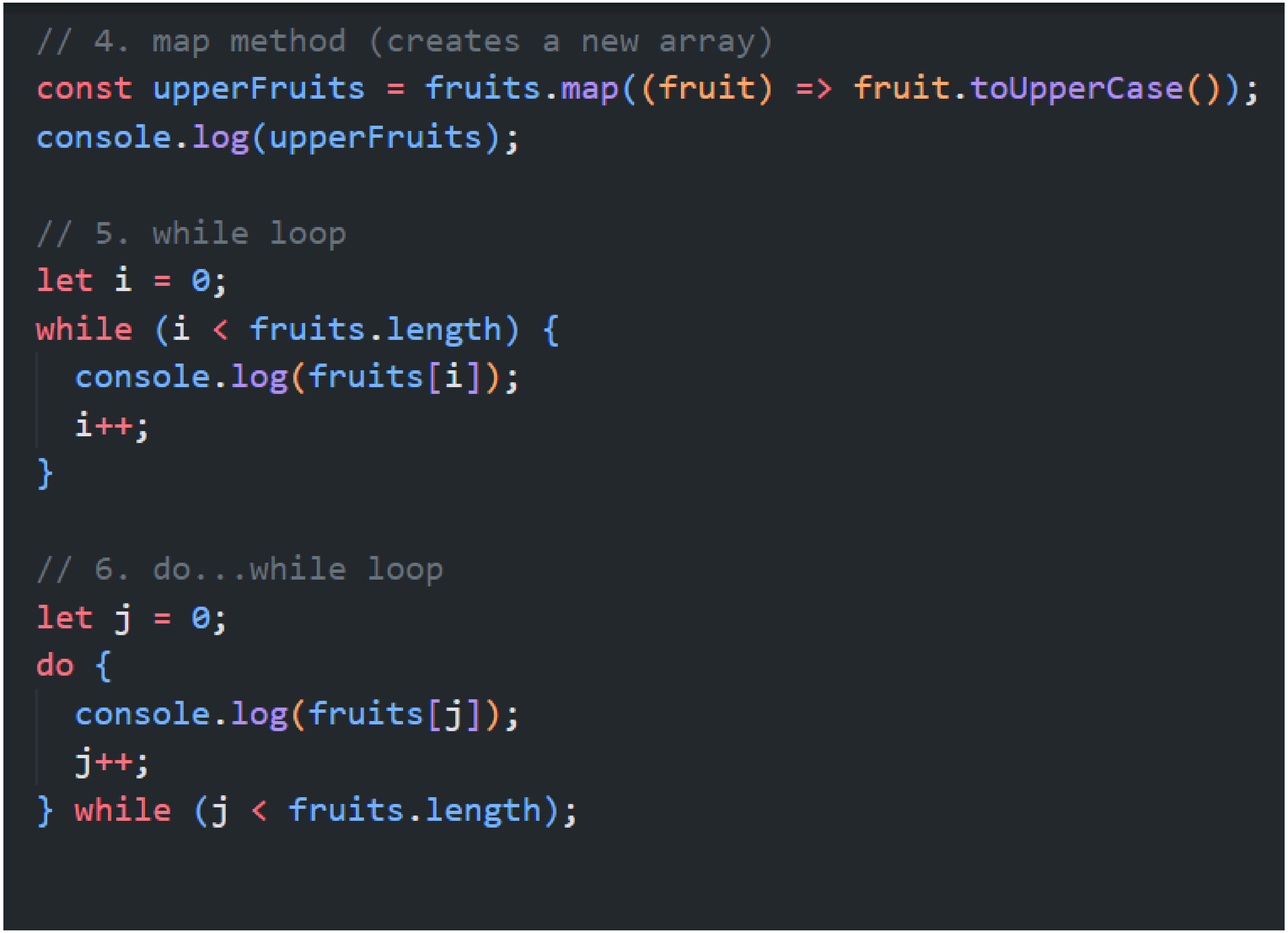
```javascript
// In a method
let person = {
  name: "Alice",
  greet() {
    console.log(`Hello, I'm ${this.name}`);
  },
};
person.greet(); // "Hello, I'm Alice"

// In an arrow function
let arrowGreet = () => {
  console.log(`Hello, ${this.name}`);
};
arrowGreet.call({ name: "Charlie" });
// "Hello, undefined" (arrow functions don't bind their own 'this')

// In a constructor function
function Person(name) {
  this.name = name;
  this.greet = function () {
    console.log(`Hello, I'm ${this.name}`);
  };
}
let david = new Person("David");
david.greet(); // "Hello, I'm David"
```

21. What are the different ways to loop through an array in JavaScript?



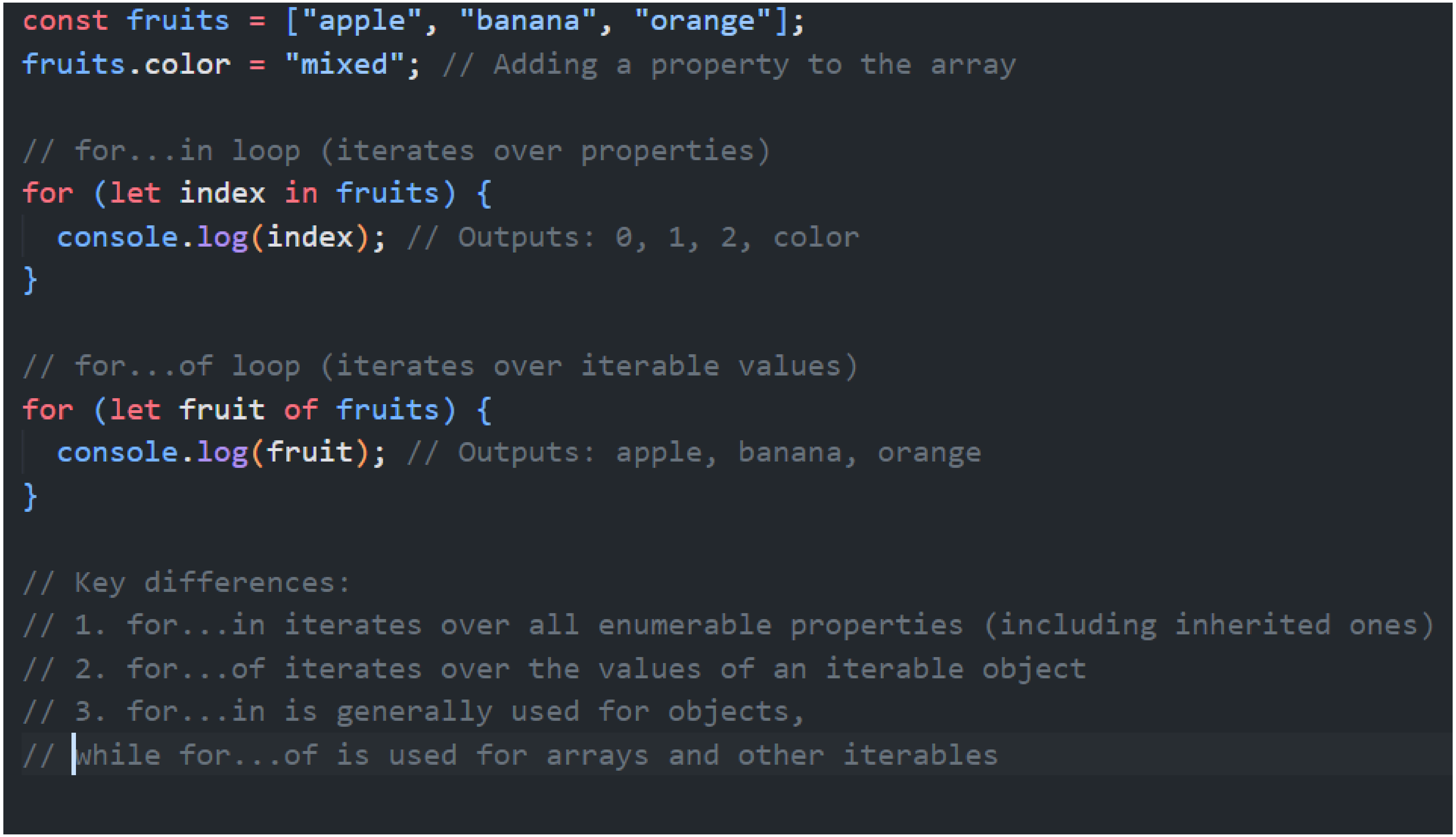
```javascript
const fruits = ["apple", "banana", "orange", "mango"];

// 1. for loop
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}

// 2. forEach method
fruits.forEach((fruit, index) => {
  console.log(`${index}: ${fruit}`);
});

// 3. for...of loop (ES6+)
for (const fruit of fruits) {
  console.log(fruit);
}
```

21. What are the different ways to loop through an array in JavaScript?



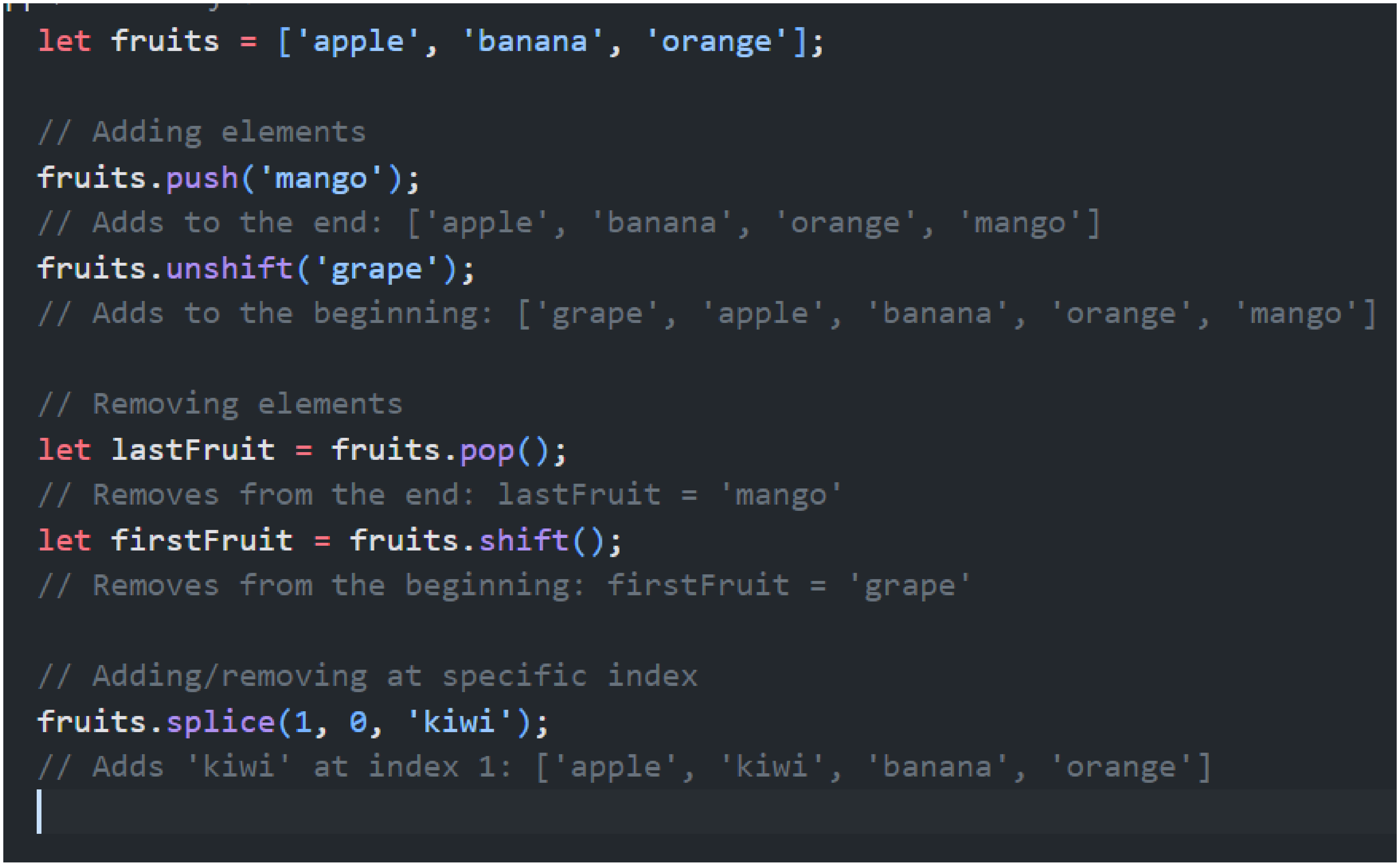
```javascript
// 4. map method (creates a new array)
const upperFruits = fruits.map((fruit) => fruit.toUpperCase());
console.log(upperFruits);

// 5. while loop
let i = 0;
while (i < fruits.length) {
  console.log(fruits[i]);
  i++;
}

// 6. do...while loop
let j = 0;
do {
  console.log(fruits[j]);
  j++;
} while (j < fruits.length);
```

22. Explain the difference between for...in and for...of loops.



```javascript
const fruits = ["apple", "banana", "orange"];
fruits.color = "mixed"; // Adding a property to the array

// for...in loop (iterates over properties)
for (let index in fruits) {
  console.log(index); // Outputs: 0, 1, 2, color
}

// for...of loop (iterates over iterable values)
for (let fruit of fruits) {
  console.log(fruit); // Outputs: apple, banana, orange
}

// Key differences:
// 1. for...in iterates over all enumerable properties (including inherited ones)
// 2. for...of iterates over the values of an iterable object
// 3. for...in is generally used for objects,
// while for...of is used for arrays and other iterables
```

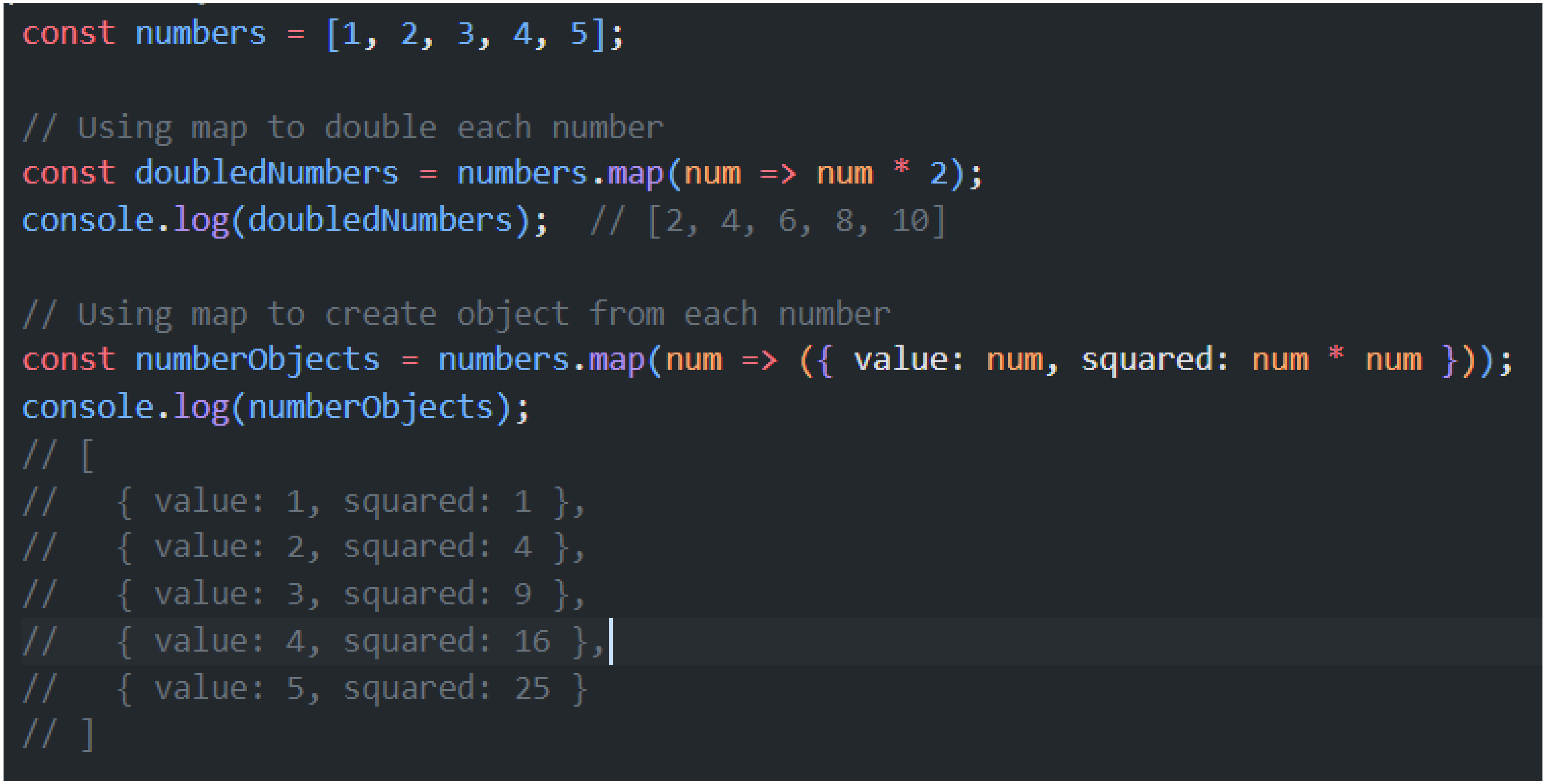
23. How do you add/remove elements from an array?



```
let fruits = ['apple', 'banana', 'orange'];

// Adding elements
fruits.push('mango');
// Adds to the end: ['apple', 'banana', 'orange', 'mango']
fruits.unshift('grape');
// Adds to the beginning: ['grape', 'apple', 'banana', 'orange', 'mango']

// Removing elements
let lastFruit = fruits.pop();
// Removes from the end: lastFruit = 'mango'
let firstFruit = fruits.shift();
// Removes from the beginning: firstFruit = 'grape'

// Adding/removing at specific index
fruits.splice(1, 0, 'kiwi');
// Adds 'kiwi' at index 1: ['apple', 'kiwi', 'banana', 'orange']
```

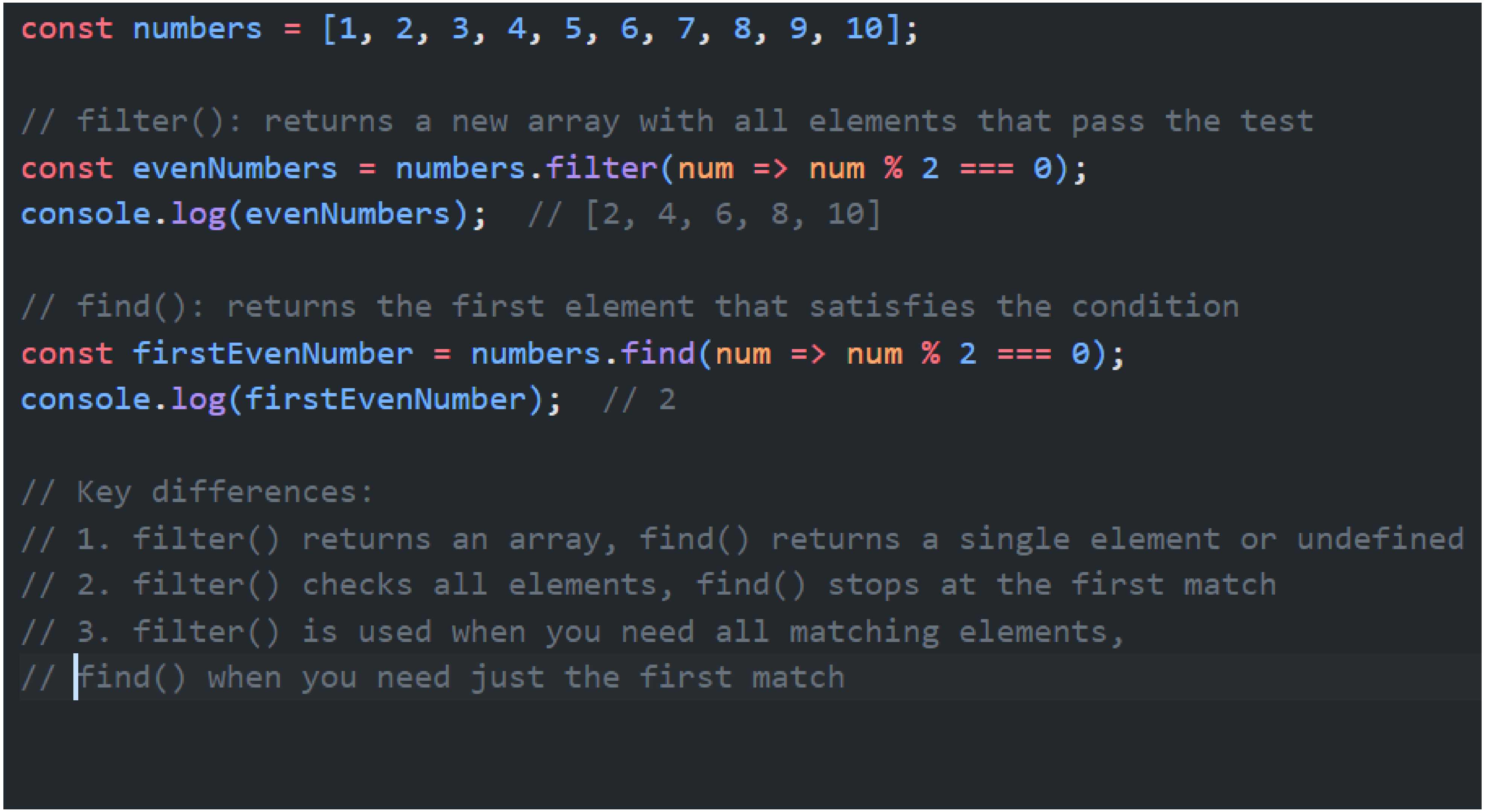
24. What is the purpose of the map() function?

The map() method creates a new array with the results of calling a provided function on every element in the array.



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

// Using map to double each number
const doubledNumbers = numbers.map(num => num * 2);
console.log(doubledNumbers);  // [2, 4, 6, 8, 10]

// Using map to create object from each number
const numberObjects = numbers.map(num => ({ value: num, squared: num * num }));
console.log(numberObjects);
// [
//   { value: 1, squared: 1 },
//   { value: 2, squared: 4 },
//   { value: 3, squared: 9 },
//   { value: 4, squared: 16 },
//   { value: 5, squared: 25 }
// ]
```

25. Explain the difference between filter() and find() methods.

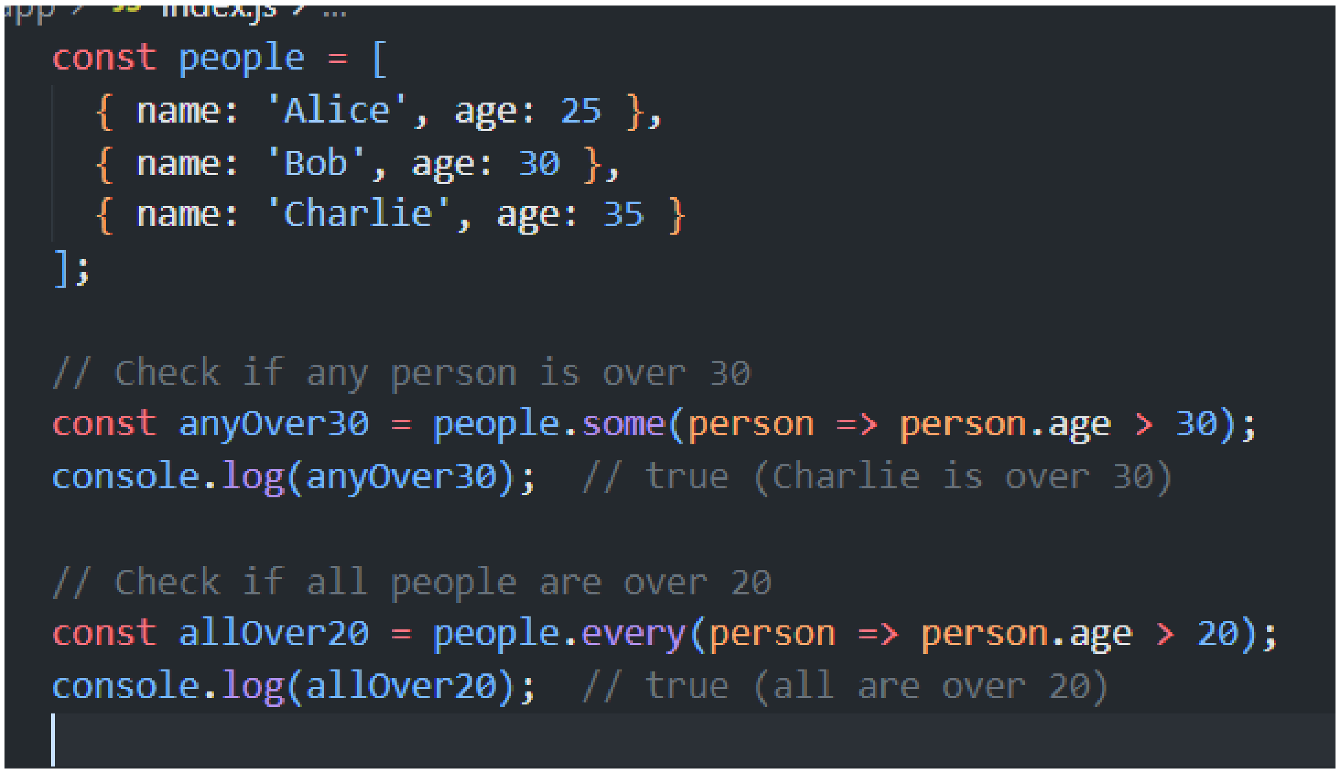


```javascript
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// filter(): returns a new array with all elements that pass the test
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers);  // [2, 4, 6, 8, 10]

// find(): returns the first element that satisfies the condition
const firstEvenNumber = numbers.find(num => num % 2 === 0);
console.log(firstEvenNumber);  // 2

// Key differences:
// 1. filter() returns an array, find() returns a single element or undefined
// 2. filter() checks all elements, find() stops at the first match
// 3. filter() is used when you need all matching elements,
// find() when you need just the first match
```

26. Explain the difference between some() and every() methods.

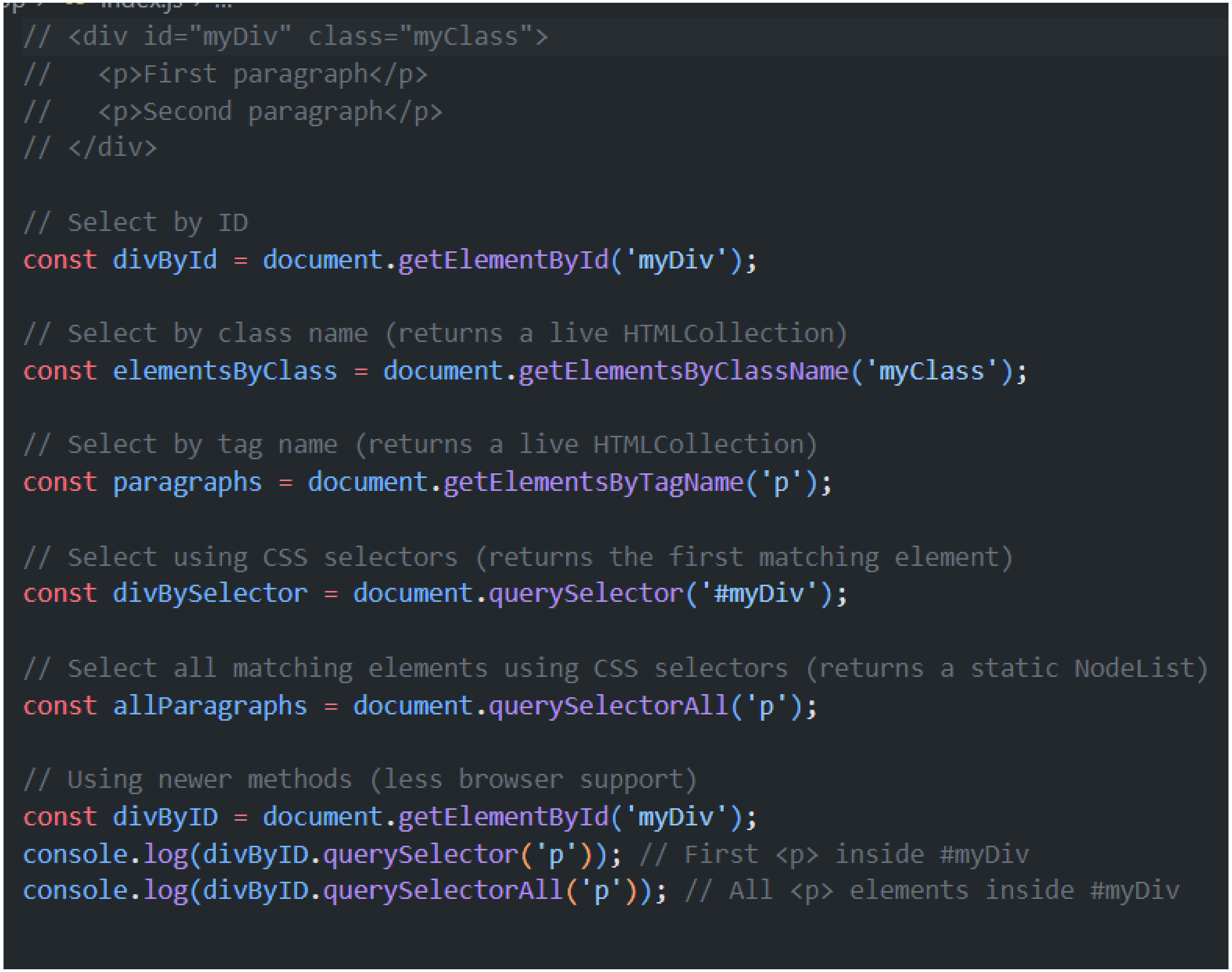
some() method: Returns true if at least one element in the array satisfies the provided testing function. Stops iterating as soon as it finds an element that satisfies the condition. Returns false if no elements satisfy the condition.

every() method: Returns true if all elements in the array satisfy the provided testing function. Stops iterating as soon as it finds an element that doesn't satisfy the condition.



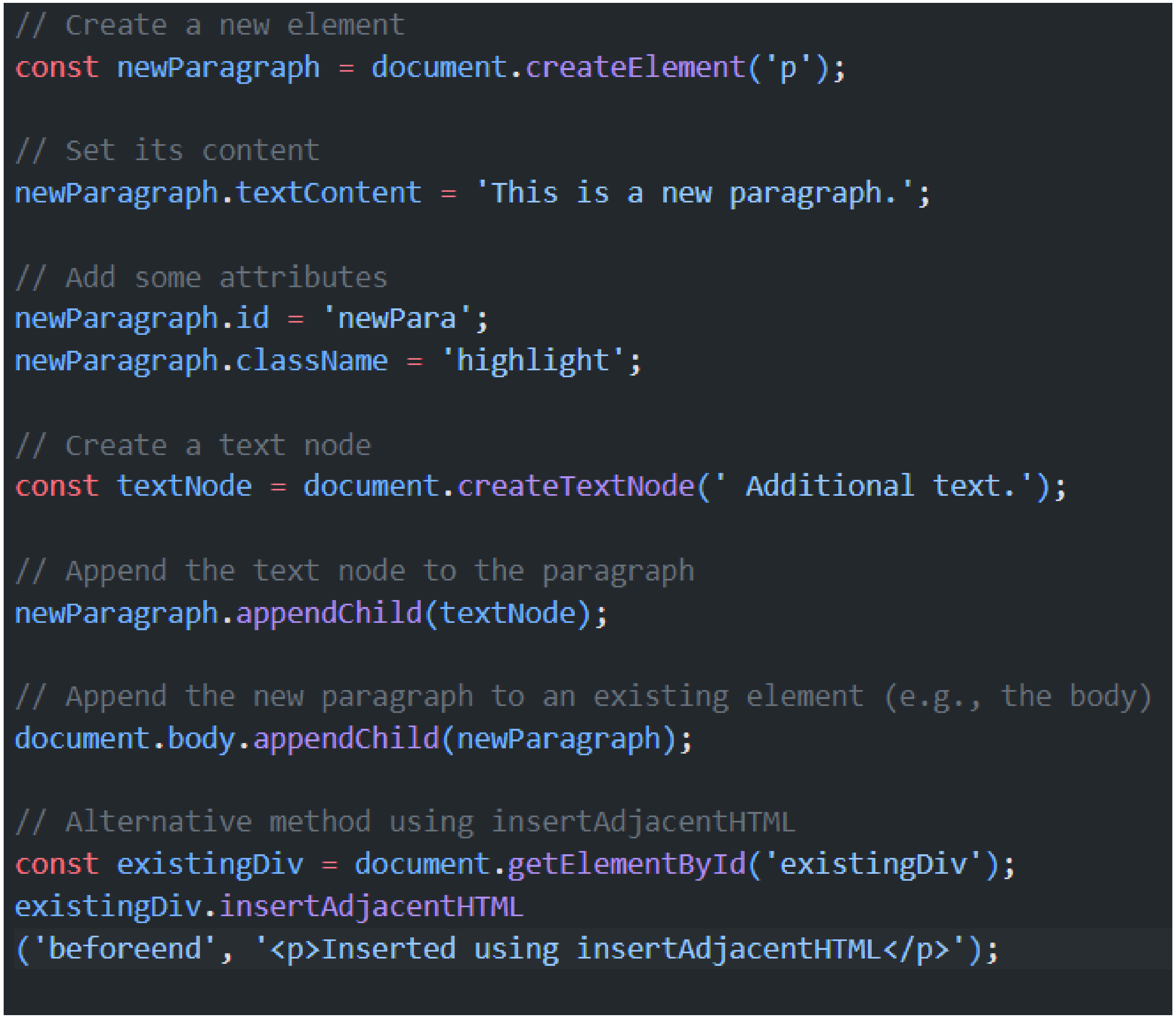
```
const people = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 }
];

// Check if any person is over 30
const anyOver30 = people.some(person => person.age > 30);
console.log(anyOver30);  // true (Charlie is over 30)

// Check if all people are over 20
const allOver20 = people.every(person => person.age > 20);
console.log(allOver20);  // true (all are over 20)
```

27. How do you select elements in the DOM using JavaScript?



```javascript
// <div id="myDiv" class="myClass">
//   <p>First paragraph</p>
//   <p>Second paragraph</p>
// </div>

// Select by ID
const divById = document.getElementById('myDiv');

// Select by class name (returns a live HTMLCollection)
const elementsByClass = document.getElementsByClassName('myClass');

// Select by tag name (returns a live HTMLCollection)
const paragraphs = document.getElementsByTagName('p');

// Select using CSS selectors (returns the first matching element)
const divBySelector = document.querySelector('#myDiv');

// Select all matching elements using CSS selectors (returns a static NodeList)
const allParagraphs = document.querySelectorAll('p');

// Using newer methods (less browser support)
const divByID = document.getElementById('myDiv');
console.log(divByID.querySelector('p')); // First <p> inside #myDiv
console.log(divByID.querySelectorAll('p')); // All <p> elements inside #myDiv
```

28. How do you create and append elements to the DOM?

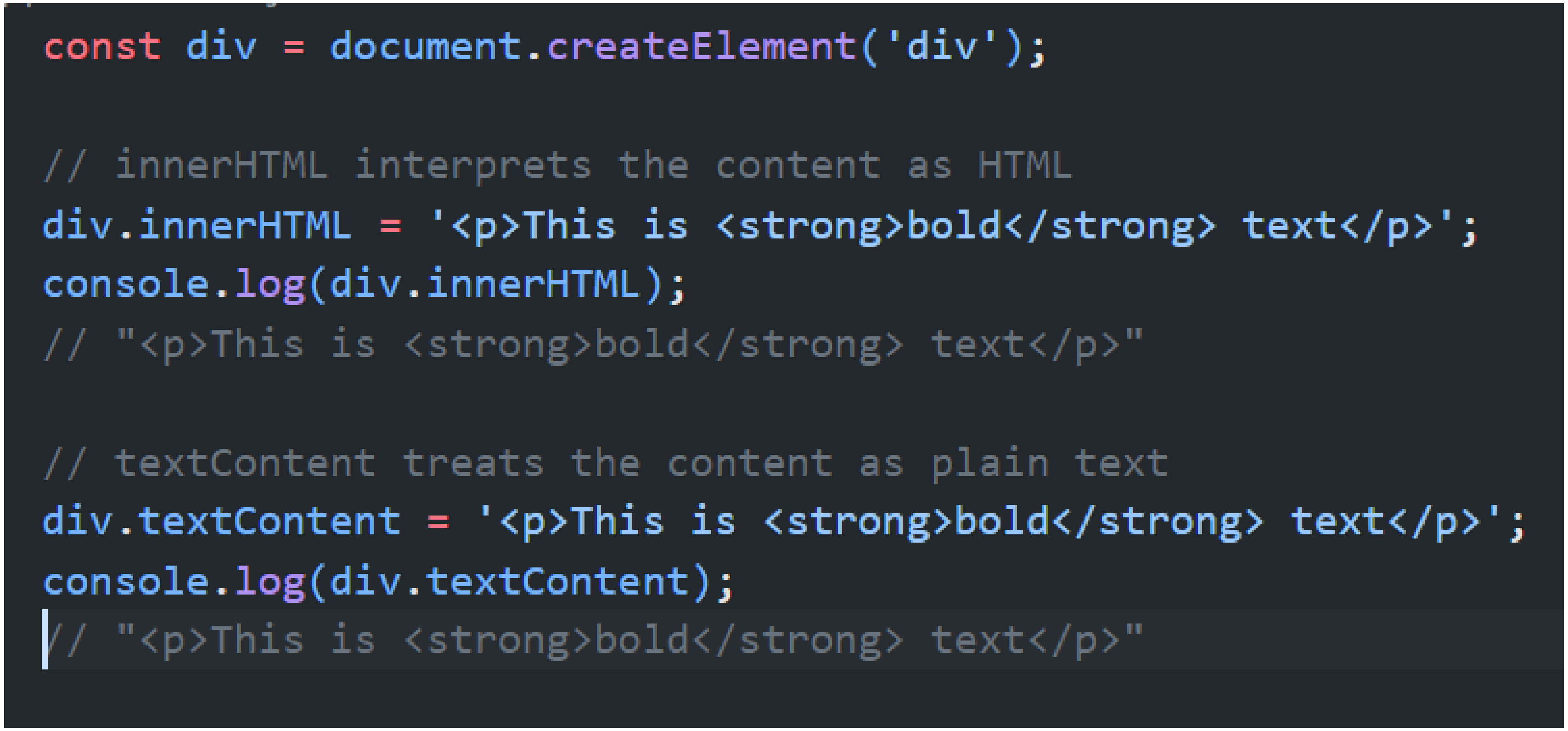


```javascript
// Create a new element
const newParagraph = document.createElement('p');

// Set its content
newParagraph.textContent = 'This is a new paragraph.';

// Add some attributes
newParagraph.id = 'newPara';
newParagraph.className = 'highlight';

// Create a text node
const textNode = document.createTextNode(' Additional text.');

// Append the text node to the paragraph
newParagraph.appendChild(textNode);

// Append the new paragraph to an existing element (e.g., the body)
document.body.appendChild(newParagraph);

// Alternative method using insertAdjacentHTML
const existingDiv = document.getElementById('existingDiv');
existingDiv.insertAdjacentHTML
('beforeend', '<p>Inserted using insertAdjacentHTML</p>');
```

29. Explain the difference between innerHTML and textContent.

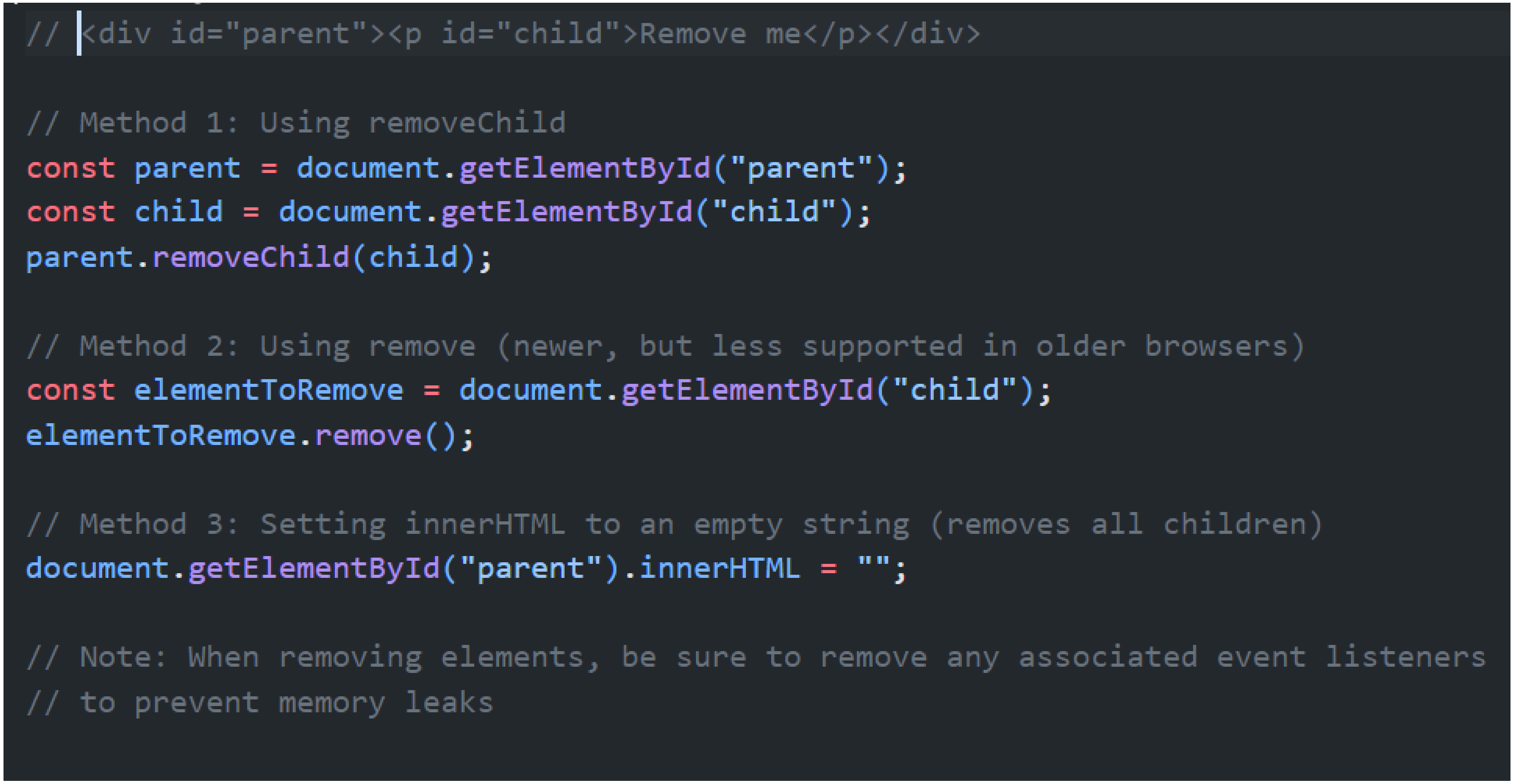
Key differences:

1. innerHTML parses content as HTML, textContent does not
2. innerHTML can be slower and less secure (risk of XSS attacks)
3. textContent is generally faster and safer
4. innerHTML returns all content, including script and style element content
5. textContent returns the text content of all elements, ignoring tags



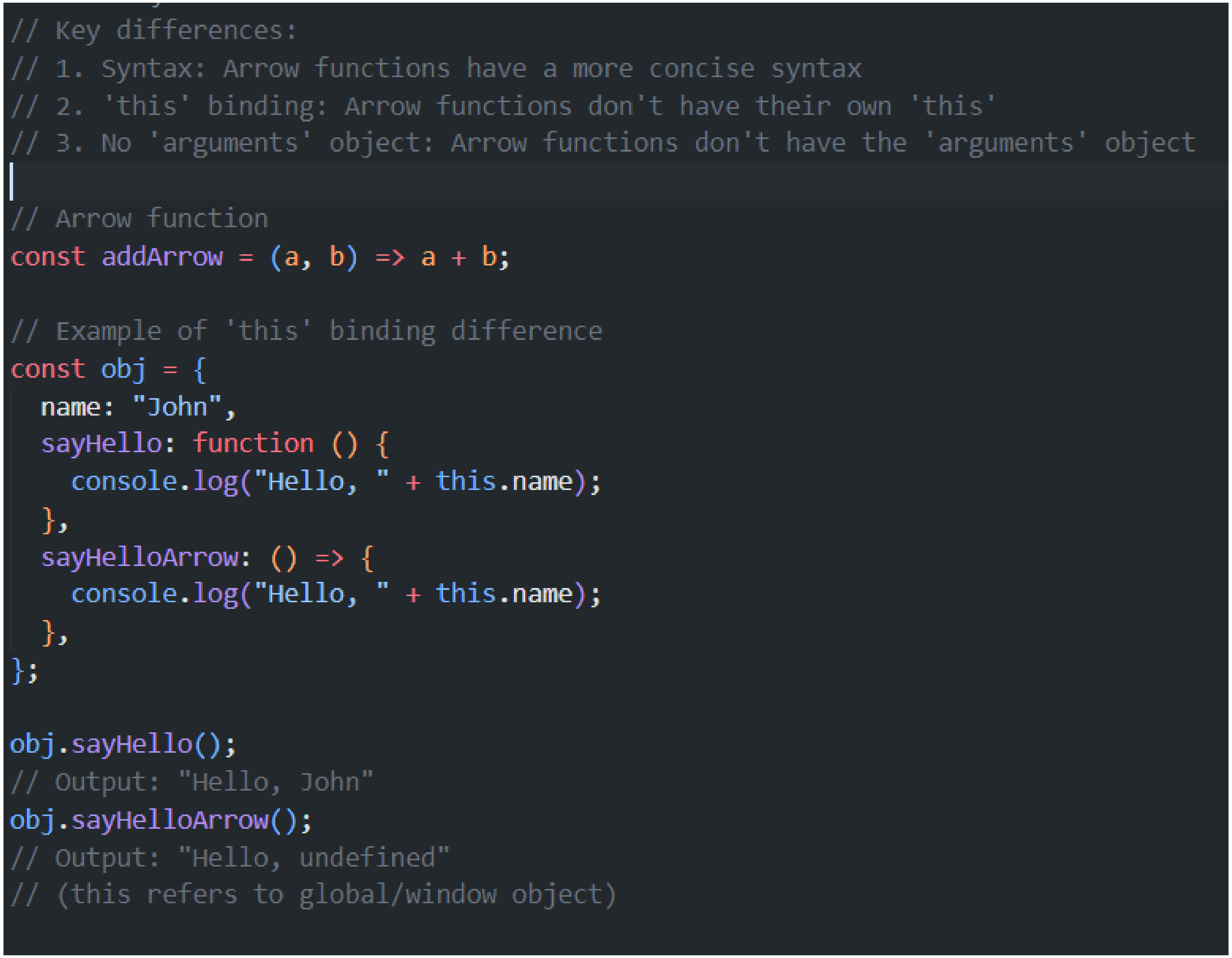
```javascript
const div = document.createElement('div');

// innerHTML interprets the content as HTML
div.innerHTML = '<p>This is <strong>bold</strong> text</p>';
console.log(div.innerHTML);
// "<p>This is <strong>bold</strong> text</p>"

// textContent treats the content as plain text
div.textContent = '<p>This is <strong>bold</strong> text</p>';
console.log(div.textContent);
// "<p>This is <strong>bold</strong> text</p>"
```

30. How do you remove an element from the DOM?



```javascript
// <div id="parent"><p id="child">Remove me</p></div>

// Method 1: Using removeChild
const parent = document.getElementById("parent");
const child = document.getElementById("child");
parent.removeChild(child);

// Method 2: Using remove (newer, but less supported in older browsers)
const elementToRemove = document.getElementById("child");
elementToRemove.remove();

// Method 3: Setting innerHTML to an empty string (removes all children)
document.getElementById("parent").innerHTML = "";

// Note: When removing elements, be sure to remove any associated event listeners
// to prevent memory leaks
```

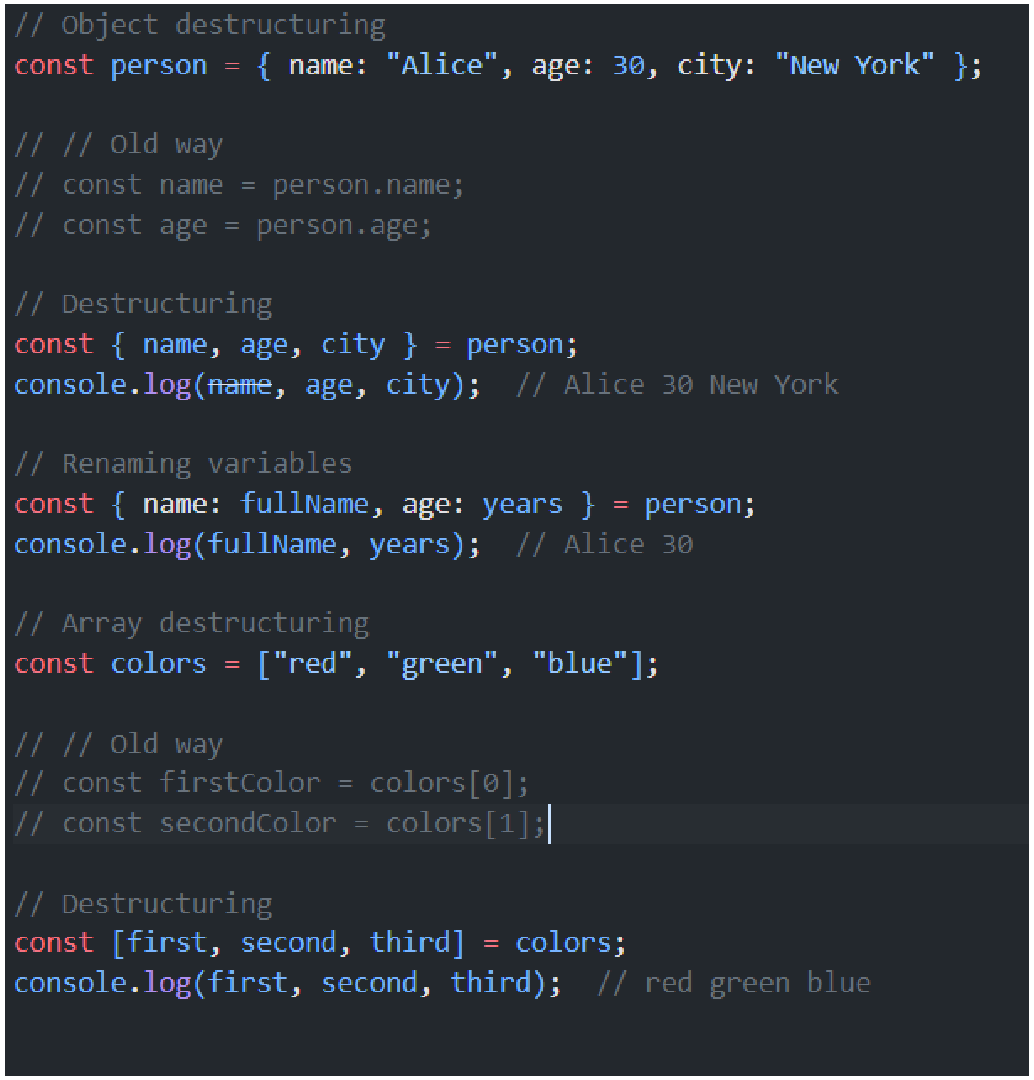
31. What are arrow functions and how do they differ from regular functions?



```
// Key differences:
// 1. Syntax: Arrow functions have a more concise syntax
// 2. 'this' binding: Arrow functions don't have their own 'this'
// 3. No 'arguments' object: Arrow functions don't have the 'arguments' object

// Arrow function
const addArrow = (a, b) => a + b;

// Example of 'this' binding difference
const obj = {
  name: "John",
  sayHello: function () {
    console.log("Hello, " + this.name);
  },
  sayHelloArrow: () => {
    console.log("Hello, " + this.name);
  },
};

obj.sayHello();
// Output: "Hello, John"
obj.sayHelloArrow();
// Output: "Hello, undefined"
// (this refers to global/window object)
```

32. Explain the concept of destructuring in JavaScript.

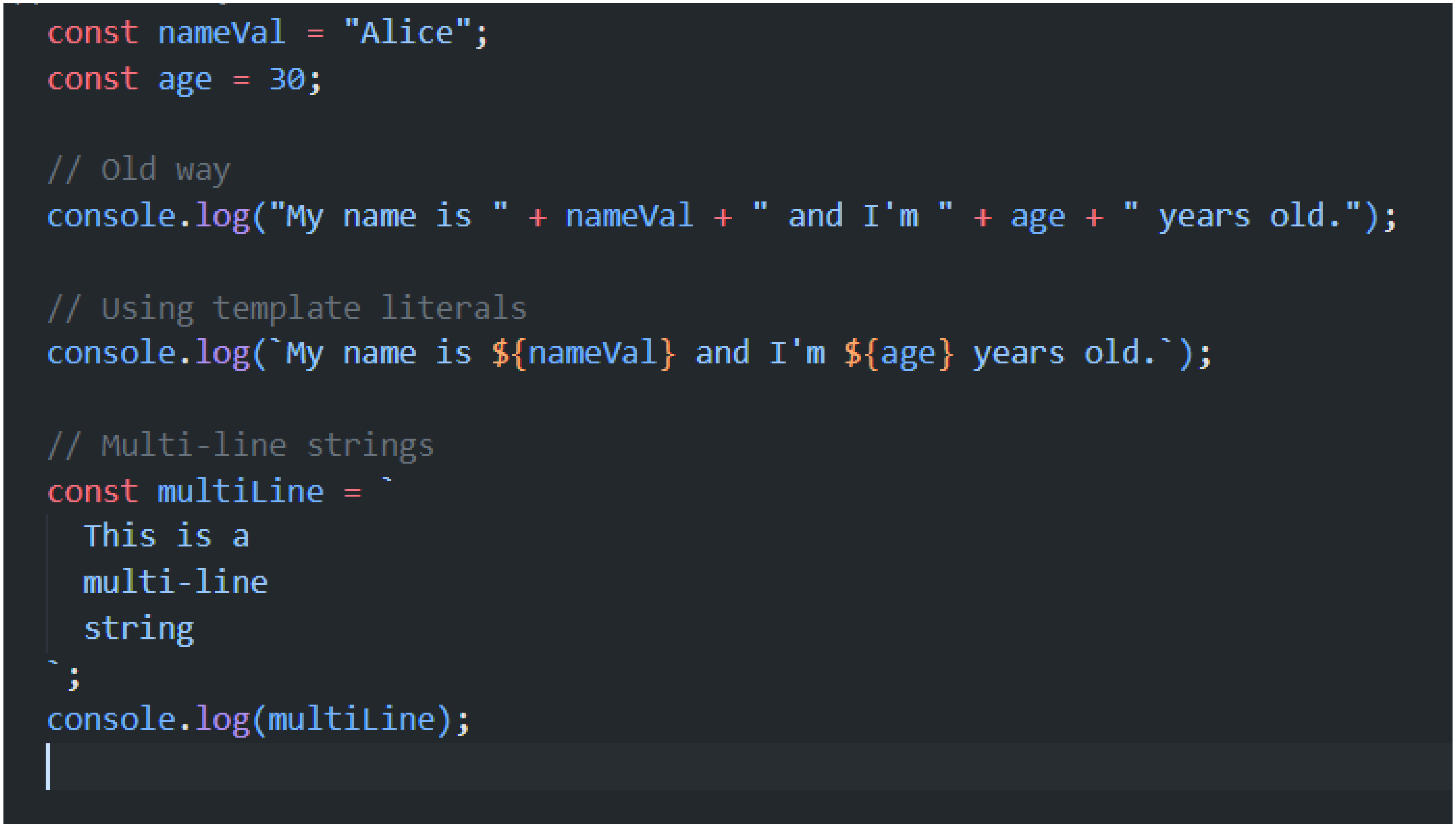
Destructuring is a way to extract multiple values from data stored in objects and arrays.



```
// Object destructuring
const person = { name: "Alice", age: 30, city: "New York" };

// // Old way
// const name = person.name;
// const age = person.age;

// Destructuring
const { name, age, city } = person;
console.log(~~name~~, age, city);  // Alice 30 New York

// Renaming variables
const { name: fullName, age: years } = person;
console.log(fullName, years);  // Alice 30

// Array destructuring
const colors = ["red", "green", "blue"];

// // Old way
// const firstColor = colors[0];
// const secondColor = colors[1];

// Destructuring
const [first, second, third] = colors;
console.log(first, second, third);  // red green blue
```

33. What are template literals?

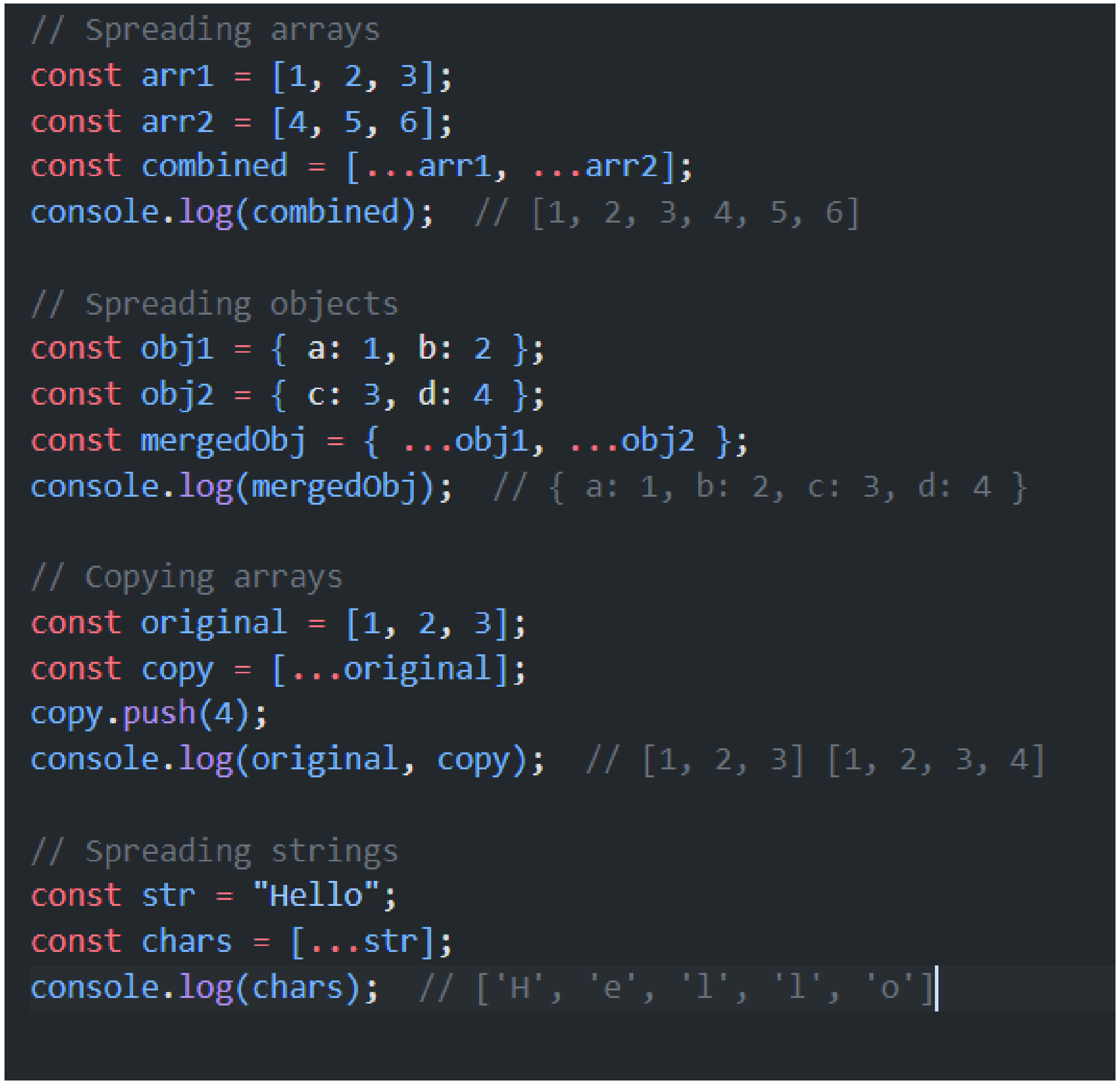
Template literals are string literals that allow embedded expressions and multi-line strings.



```javascript
const nameVal = "Alice";
const age = 30;

// Old way
console.log("My name is " + nameVal + " and I'm " + age + " years old.");

// Using template literals
console.log(`My name is ${nameVal} and I'm ${age} years old.`);

// Multi-line strings
const multiLine = `
  This is a
  multi-line
  string
`;
console.log(multiLine);
```

34. What is spread operator, Explain with example?

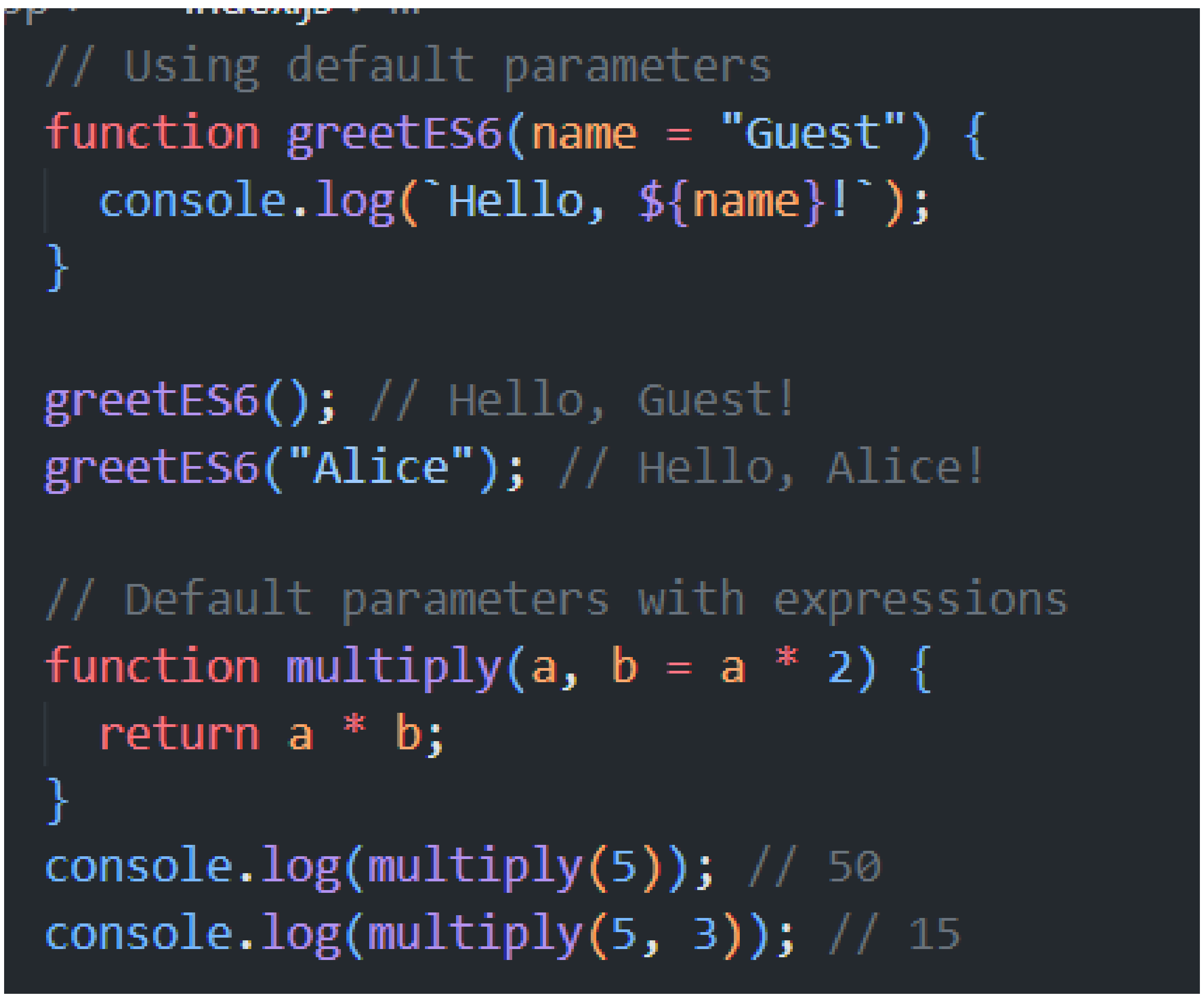
The spread operator (...) allows an iterable to be expanded in places where zero or more arguments or elements are expected.



```javascript
// Spreading arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2];
console.log(combined);  // [1, 2, 3, 4, 5, 6]

// Spreading objects
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const mergedObj = { ...obj1, ...obj2 };
console.log(mergedObj);  // { a: 1, b: 2, c: 3, d: 4 }

// Copying arrays
const original = [1, 2, 3];
const copy = [...original];
copy.push(4);
console.log(original, copy);  // [1, 2, 3] [1, 2, 3, 4]

// Spreading strings
const str = "Hello";
const chars = [...str];
console.log(chars);  // ['H', 'e', 'l', 'l', 'o']
```

35. What are default parameters in ES6?

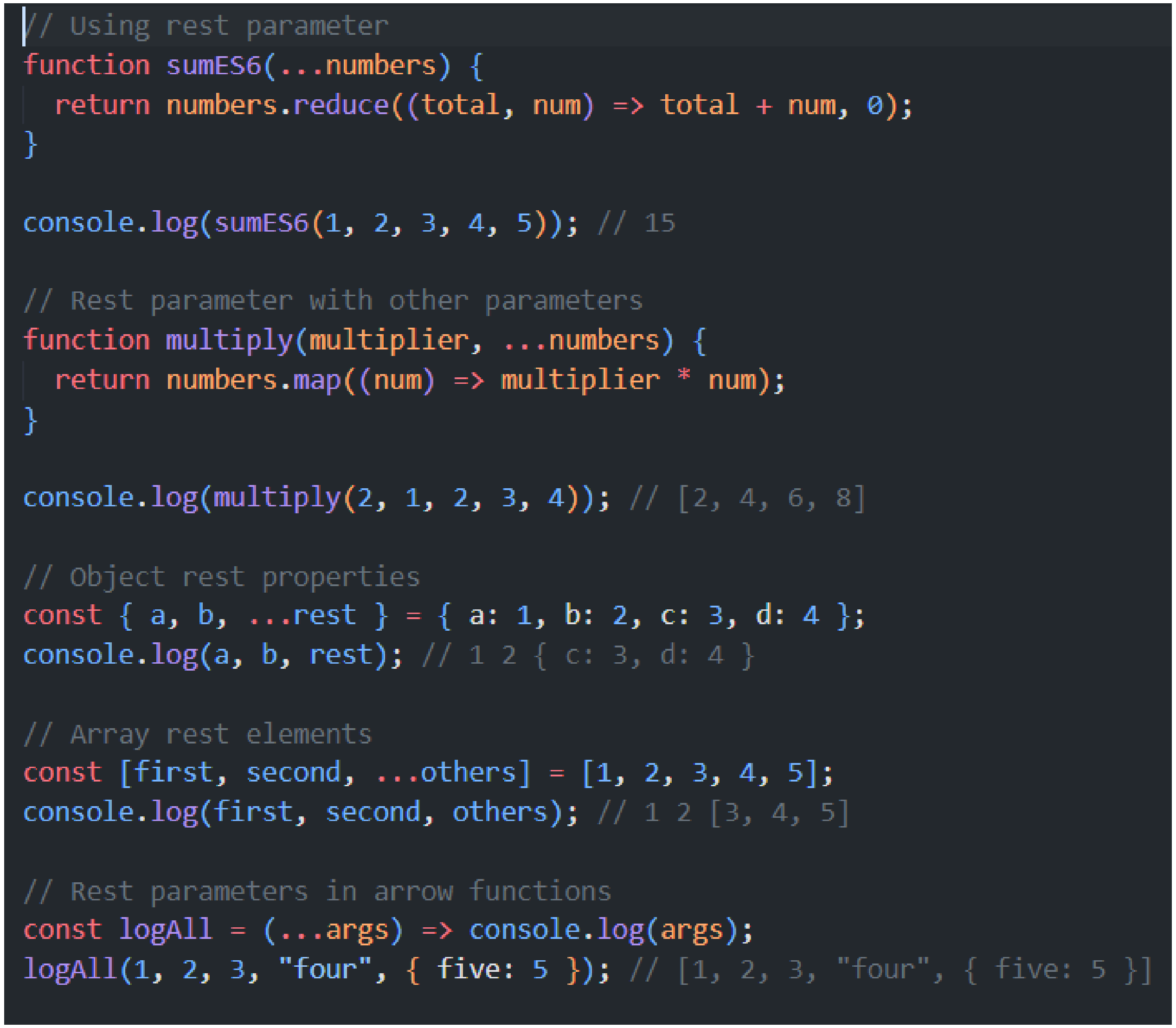
Default parameters allow you to set default values for function parameters if no value or undefined is passed.



```javascript
// Using default parameters
function greetES6(name = "Guest") {
  console.log(`Hello, ${name}!`);
}

greetES6(); // Hello, Guest!
greetES6("Alice"); // Hello, Alice!

// Default parameters with expressions
function multiply(a, b = a * 2) {
  return a * b;
}
console.log(multiply(5)); // 50
console.log(multiply(5, 3)); // 15
```

36. How do you use the rest parameter in functions?

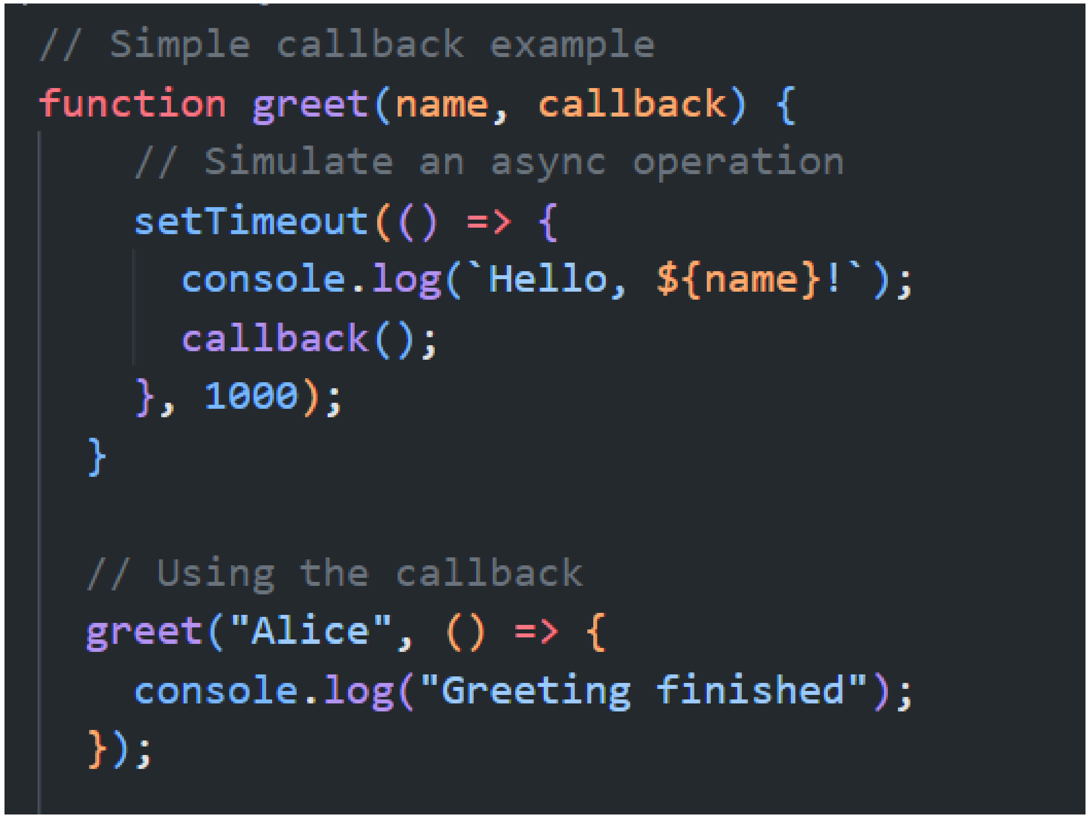
The rest parameter syntax allows a function to accept an indefinite number of arguments as an array.



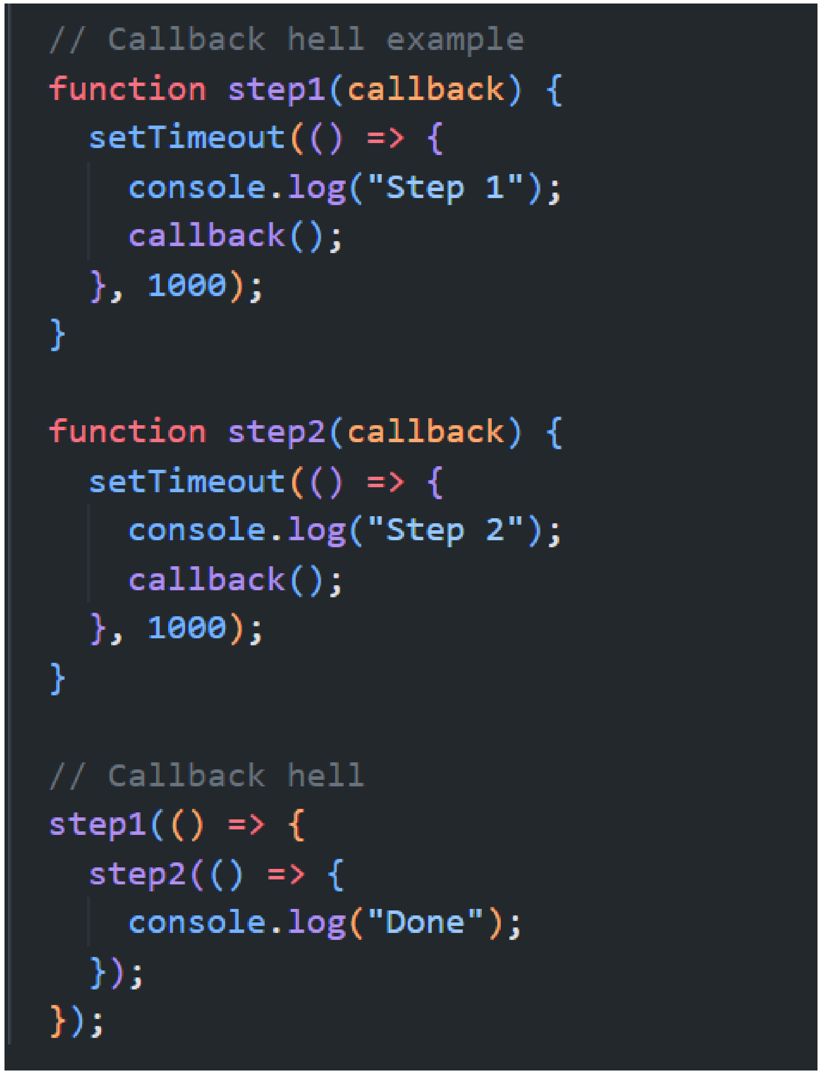
```javascript
// Using rest parameter
function sumES6(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sumES6(1, 2, 3, 4, 5)); // 15

// Rest parameter with other parameters
function multiply(multiplier, ...numbers) {
  return numbers.map((num) => multiplier * num);
}

console.log(multiply(2, 1, 2, 3, 4)); // [2, 4, 6, 8]

// Object rest properties
const { a, b, ...rest } = { a: 1, b: 2, c: 3, d: 4 };
console.log(a, b, rest); // 1 2 { c: 3, d: 4 }

// Array rest elements
const [first, second, ...others] = [1, 2, 3, 4, 5];
console.log(first, second, others); // 1 2 [3, 4, 5]

// Rest parameters in arrow functions
const logAll = (...args) => console.log(args);
logAll(1, 2, 3, "four", { five: 5 }); // [1, 2, 3, "four", { five: 5 }]
```

37. What is callback & callback hell explain with example

Callbacks are functions passed as arguments to other functions, often used for asynchronous operations. Callback hell occurs when multiple nested callbacks make code hard to read and maintain.



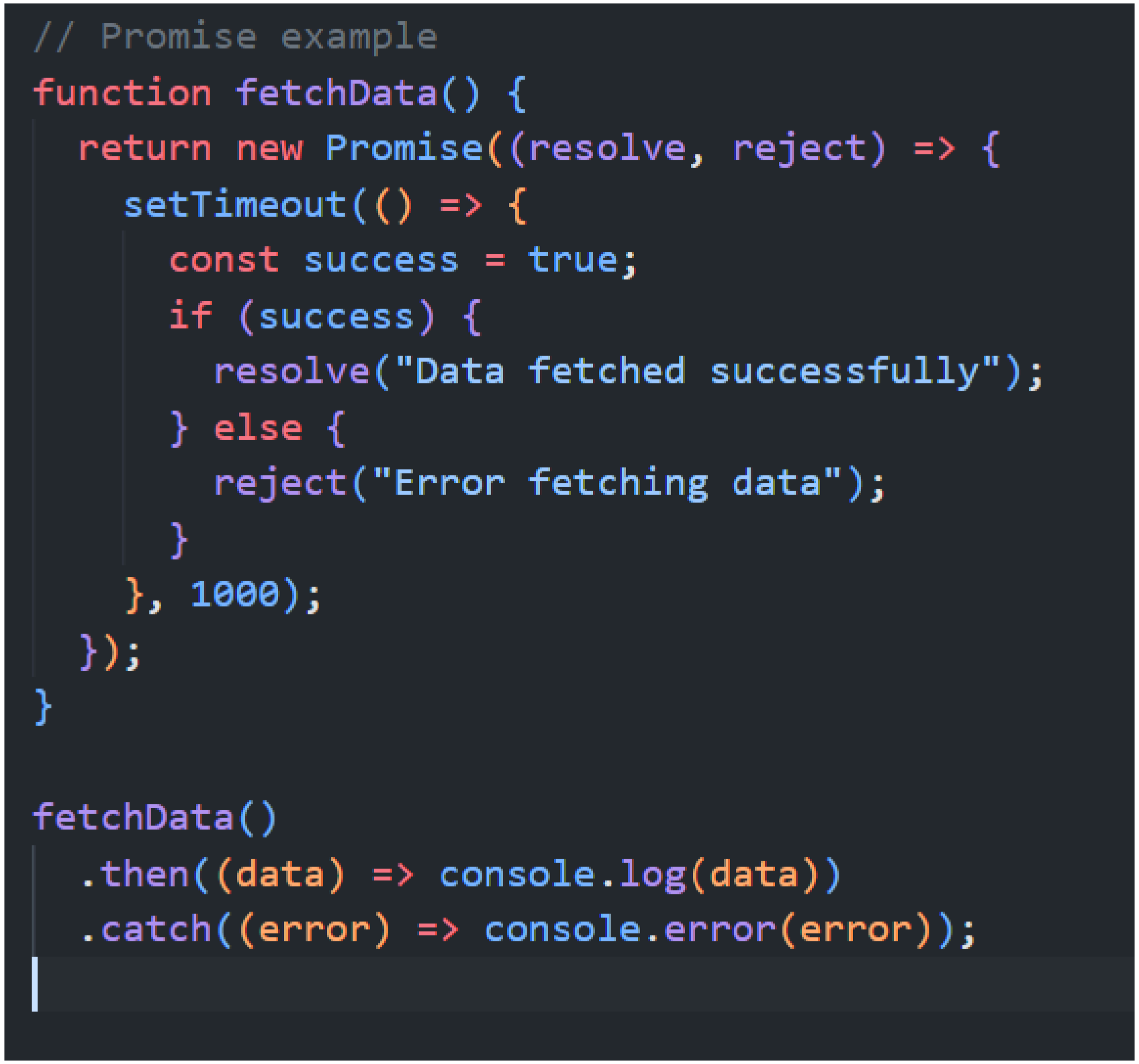
```javascript
// Simple callback example
function greet(name, callback) {
    // Simulate an async operation
    setTimeout(() => {
      console.log(`Hello, ${name}!`);
      callback();
    }, 1000);
  }

  // Using the callback
  greet("Alice", () => {
    console.log("Greeting finished");
  });
```



```javascript
// Callback hell example
function step1(callback) {
  setTimeout(() => {
    console.log("Step 1");
    callback();
  }, 1000);
}

function step2(callback) {
  setTimeout(() => {
    console.log("Step 2");
    callback();
  }, 1000);
}

// Callback hell
step1(() => {
  step2(() => {
    console.log("Done");
  });
});
```

38. What is a Promise in JavaScript with example?

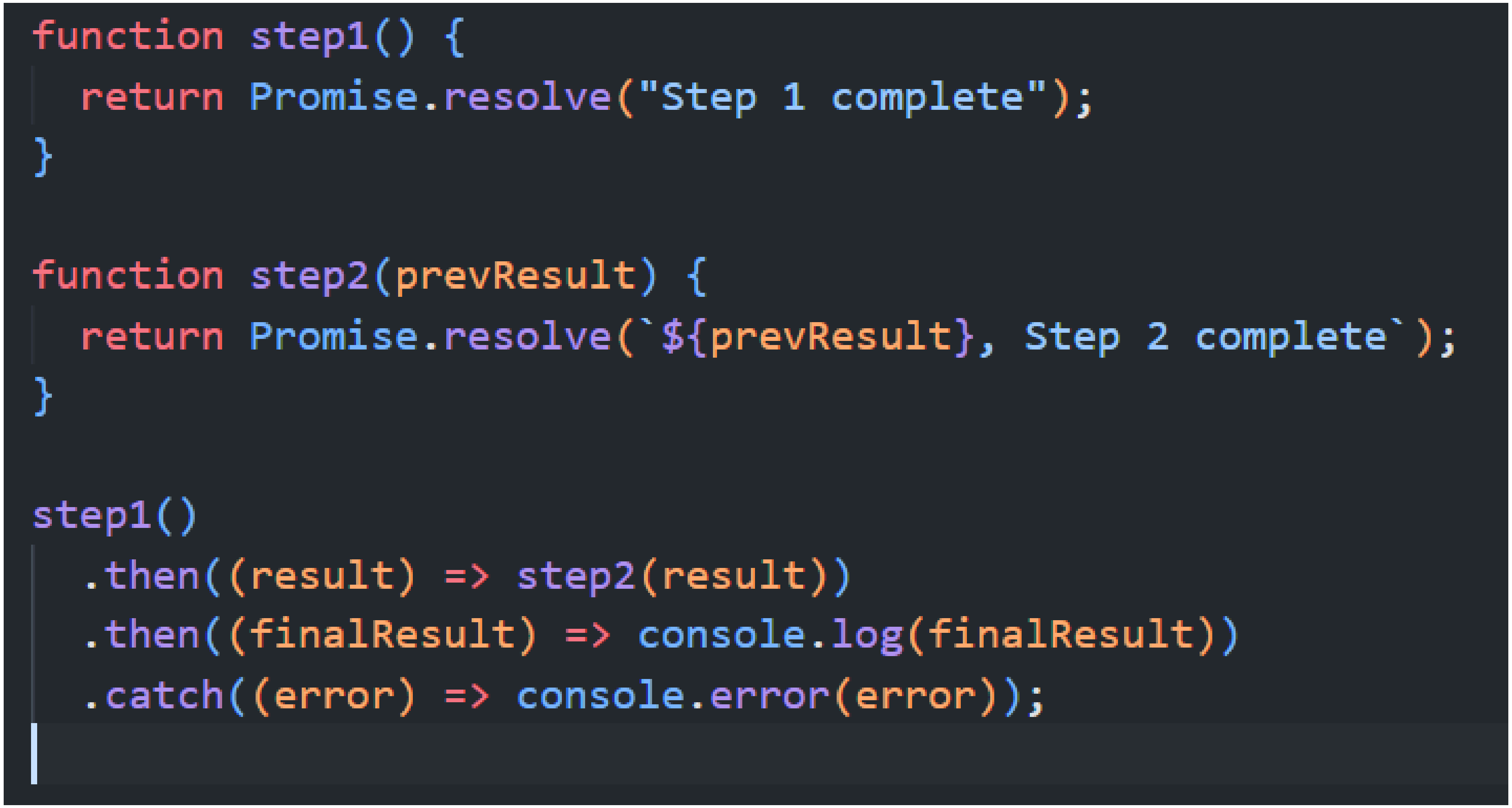
A Promise is an object representing the eventual completion or failure of an asynchronous operation.



```javascript
// Promise example
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Data fetched successfully");
      } else {
        reject("Error fetching data");
      }
    }, 1000);
  });
}

fetchData()
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

39. How do you chain Promises?

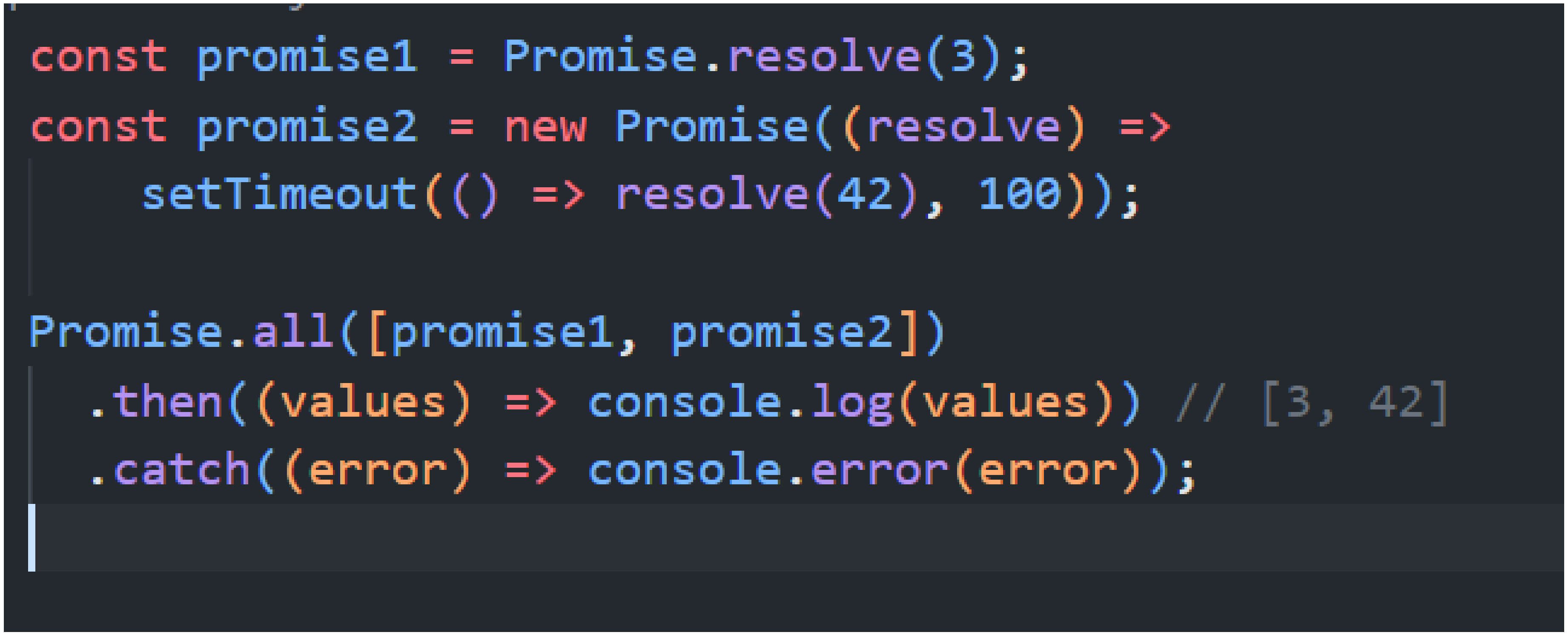
Promise chaining allows you to perform sequential asynchronous operations.



```javascript
function step1() {
  return Promise.resolve("Step 1 complete");
}

function step2(prevResult) {
  return Promise.resolve(`${prevResult}, Step 2 complete`);
}

step1()
  .then((result) => step2(result))
  .then((finalResult) => console.log(finalResult))
  .catch((error) => console.error(error));
```

40. What is the purpose of the Promise.all() method?

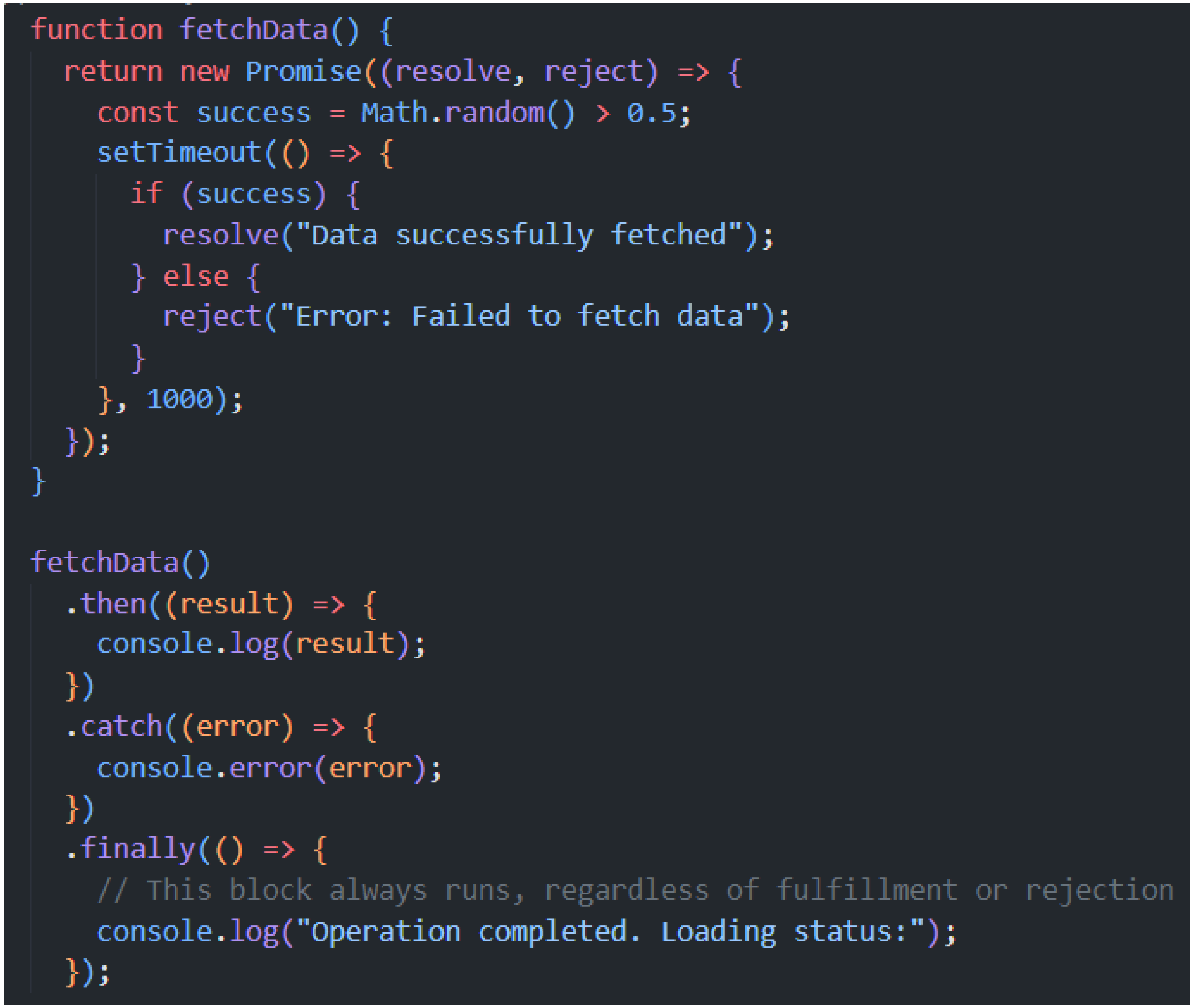
Promise.all() takes an iterable of promises and returns a single Promise that resolves when all input promises have resolved, or rejects if any input promise rejects.



```javascript
const promise1 = Promise.resolve(3);
const promise2 = new Promise((resolve) =>
    setTimeout(() => resolve(42), 100));

Promise.all([promise1, promise2])
  .then((values) => console.log(values)) // [3, 42]
  .catch((error) => console.error(error));
```

41. What is the purpose of the finally() method in Promises?

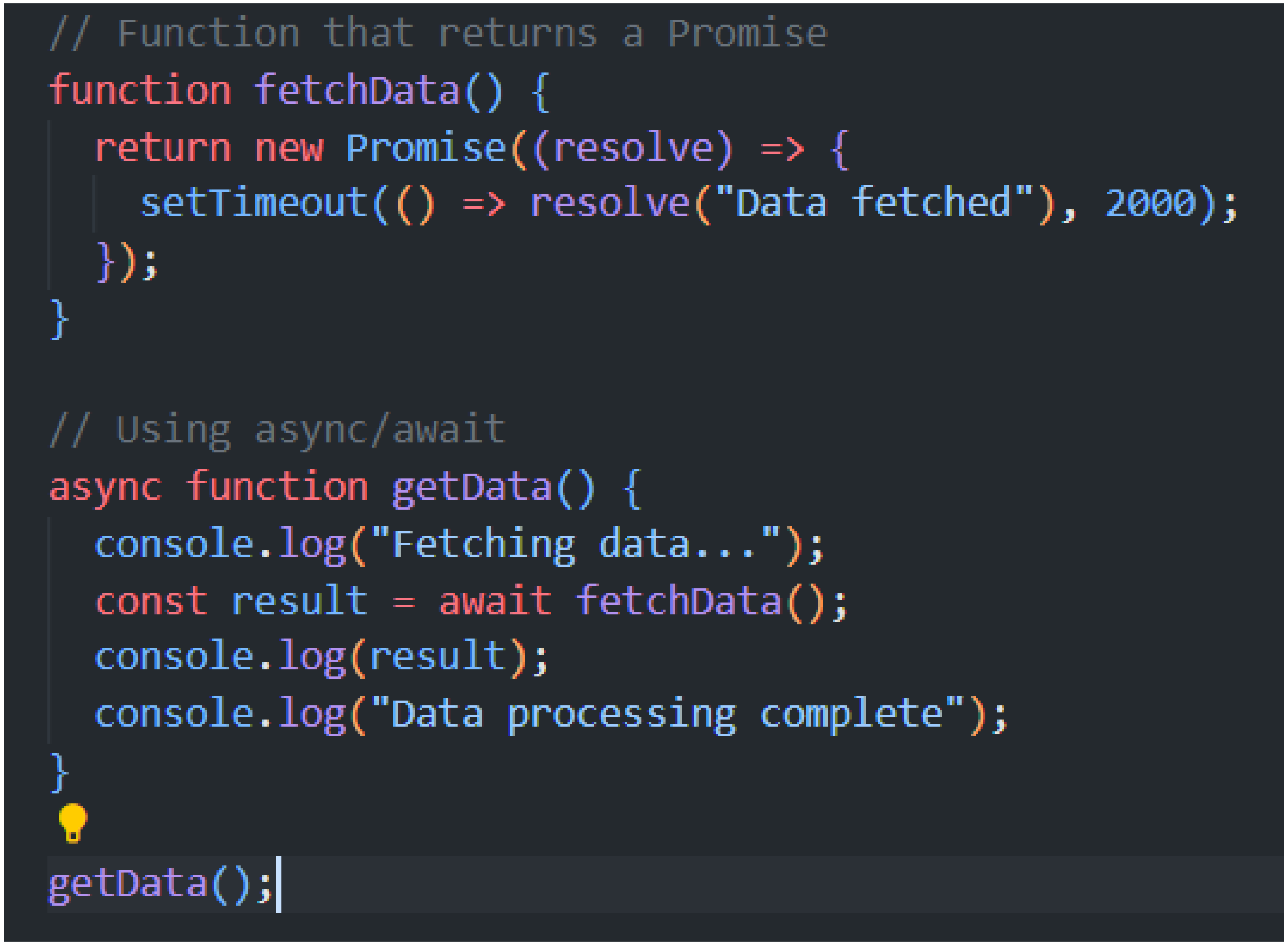
The finally() method is used to specify code that should be executed regardless of whether the promise is fulfilled or rejected.



```javascript
function fetchData() {
  return new Promise((resolve, reject) => {
    const success = Math.random() > 0.5;
    setTimeout(() => {
      if (success) {
        resolve("Data successfully fetched");
      } else {
        reject("Error: Failed to fetch data");
      }
    }, 1000);
  });
}

fetchData()
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  })
  .finally(() => {
    // This block always runs, regardless of fulfillment or rejection
    console.log("Operation completed. Loading status:");
  });
```

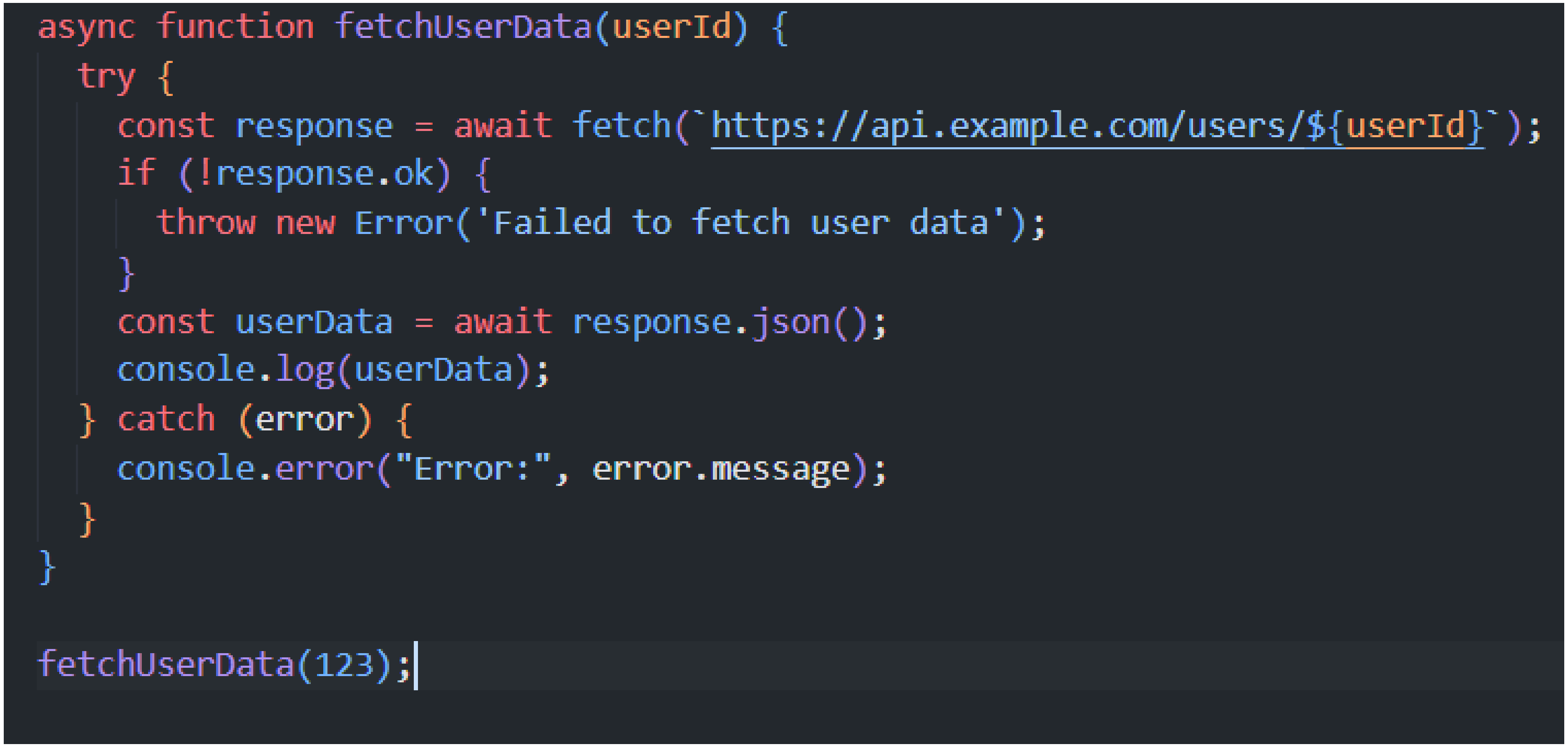
42. What is the purpose of the async await ?

The purpose of async/await is to simplify the syntax for working with Promises, making asynchronous code easier to write and read. It allows you to write asynchronous code that looks and behaves more like synchronous code.



```javascript
// Function that returns a Promise
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data fetched"), 2000);
  });
}

// Using async/await
async function getData() {
  console.log("Fetching data...");
  const result = await fetchData();
  console.log(result);
  console.log("Data processing complete");
}

getData();
```

43. How do you handle errors in async/await?



```javascript
async function fetchUserData(userId) {
  try {
    const response = await fetch(`https://api.example.com/users/${userId}`);
    if (!response.ok) {
      throw new Error('Failed to fetch user data');
    }
    const userData = await response.json();
    console.log(userData);
  } catch (error) {
    console.error("Error:", error.message);
  }
}

fetchUserData(123);
```

44. What is the difference between async/await and Promises?

Syntax: Async/await provides a more linear, synchronous-looking code structure.

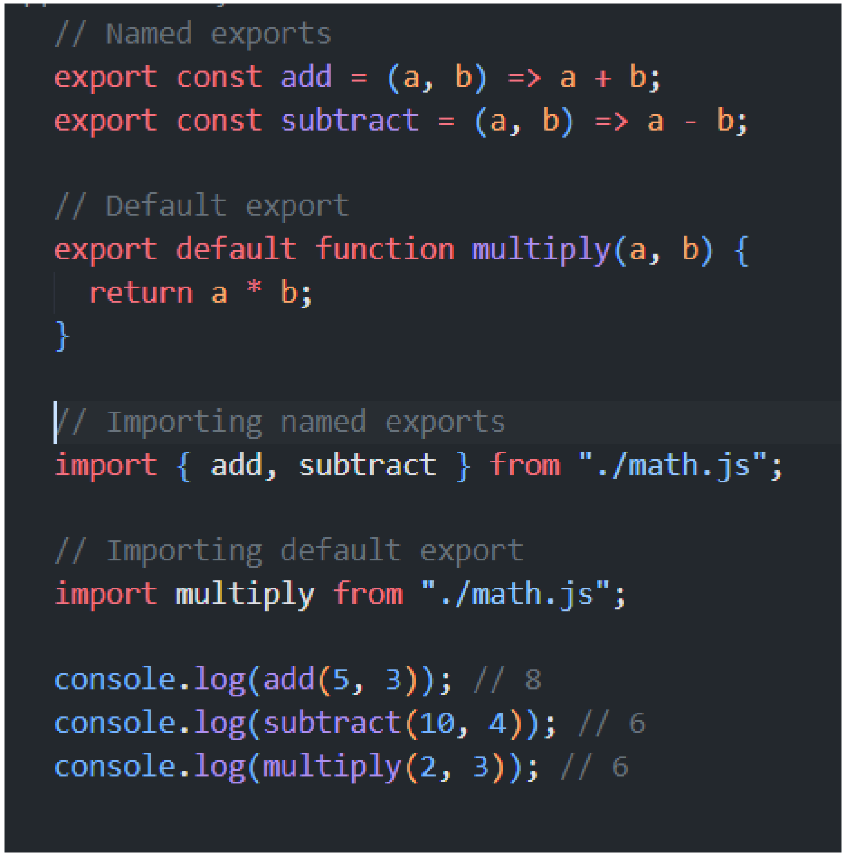
Error handling: Async/await uses try/catch, which is more familiar for synchronous code.

Chaining: Promises use .then() for chaining, while async/await uses regular JavaScript control flow.

Debugging: Async/await can be easier to debug as it behaves more like synchronous code

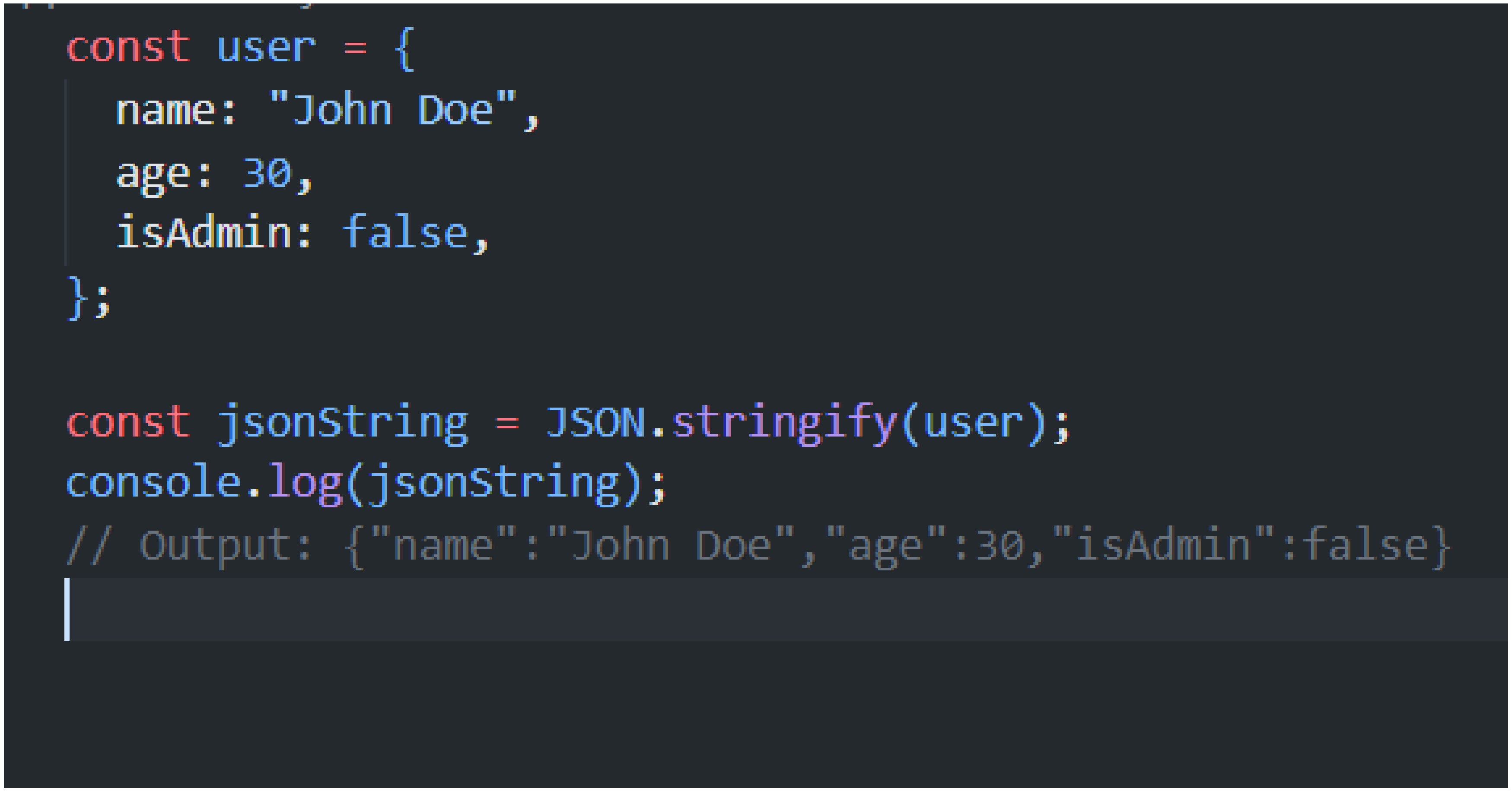
45. What is the difference between default and named exports?

A module can have multiple named exports but only one default export.
Named exports are imported using curly braces, while default exports are imported without them.
Default exports can be imported with any name, while named exports must be imported with their exact names (unless renamed using `as`).



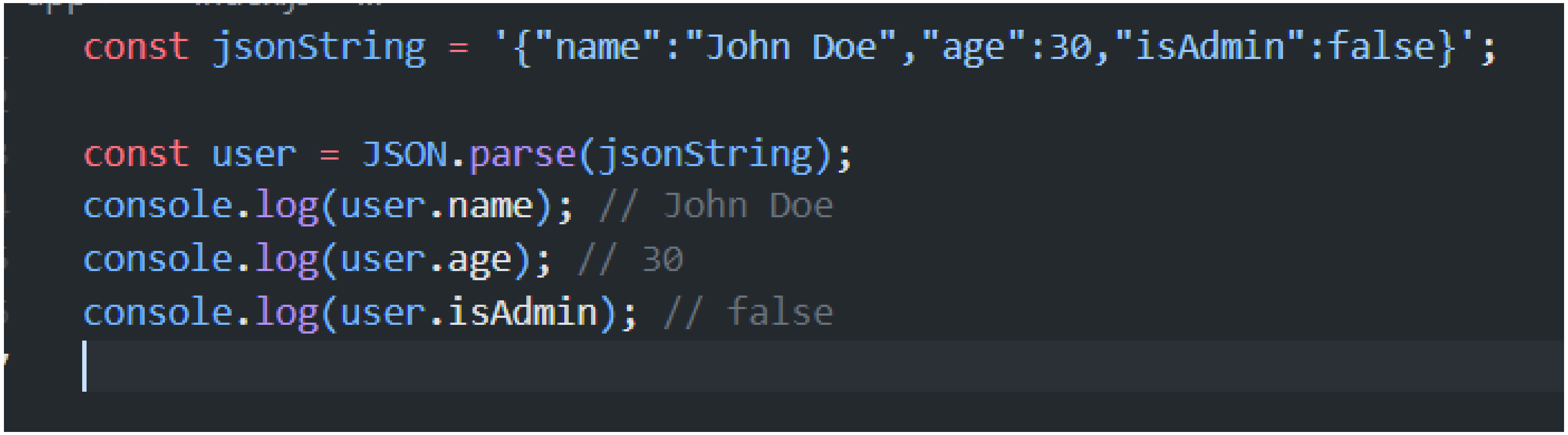
```javascript
// Named exports
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// Default export
export default function multiply(a, b) {
  return a * b;
}

// Importing named exports
import { add, subtract } from "./math.js";

// Importing default export
import multiply from "./math.js";

console.log(add(5, 3)); // 8
console.log(subtract(10, 4)); // 6
console.log(multiply(2, 3)); // 6
```

46. How do you convert a JavaScript object to a JSON string?



```javascript
const user = {
  name: "John Doe",
  age: 30,
  isAdmin: false,
};

const jsonString = JSON.stringify(user);
console.log(jsonString);
// Output: {"name":"John Doe","age":30,"isAdmin":false}
```

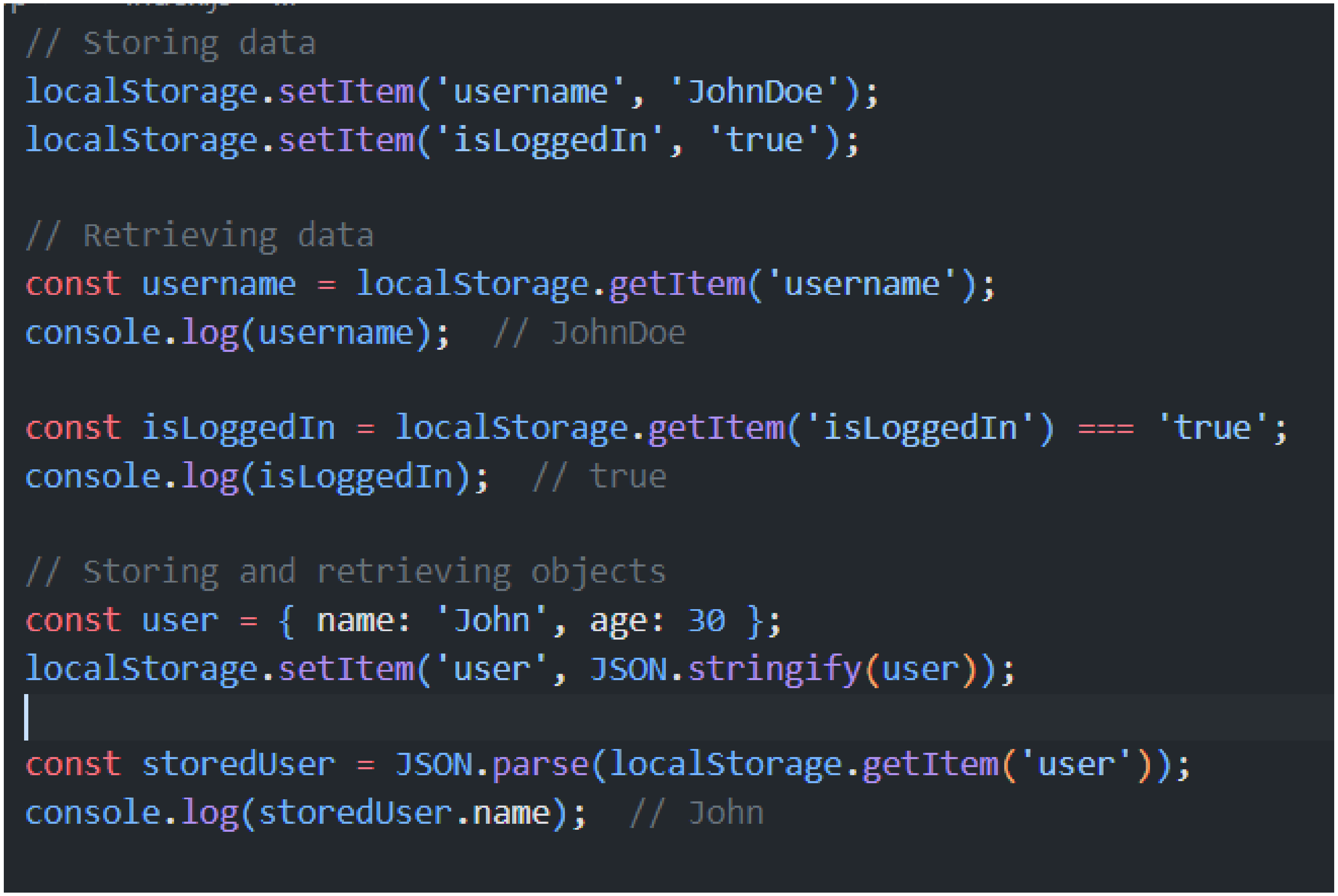
47. How do you parse a JSON string back into a JavaScript object?



```
const jsonString = '{"name":"John Doe","age":30,"isAdmin":false}';

const user = JSON.parse(jsonString);
console.log(user.name); // John Doe
console.log(user.age); // 30
console.log(user.isAdmin); // false
```

48. What is localStorage in JavaScript, and how do you store and retrieve data from it?

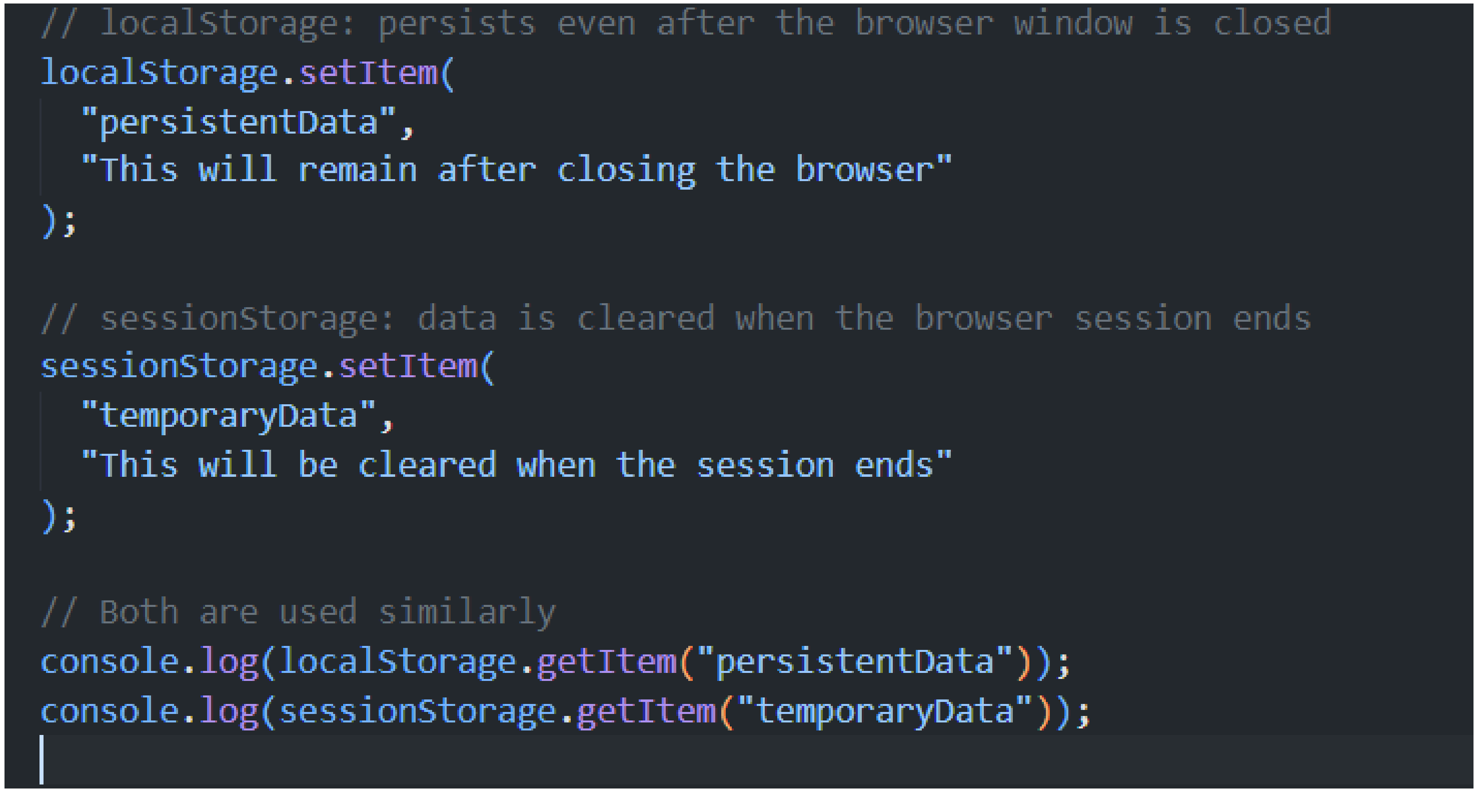
localStorage is a web storage object that allows you to store key-value pairs in the browser with no expiration time



```javascript
// Storing data
localStorage.setItem('username', 'JohnDoe');
localStorage.setItem('isLoggedIn', 'true');

// Retrieving data
const username = localStorage.getItem('username');
console.log(username);  // JohnDoe

const isLoggedIn = localStorage.getItem('isLoggedIn') === 'true';
console.log(isLoggedIn);  // true

// Storing and retrieving objects
const user = { name: 'John', age: 30 };
localStorage.setItem('user', JSON.stringify(user));

const storedUser = JSON.parse(localStorage.getItem('user'));
console.log(storedUser.name);  // John
```

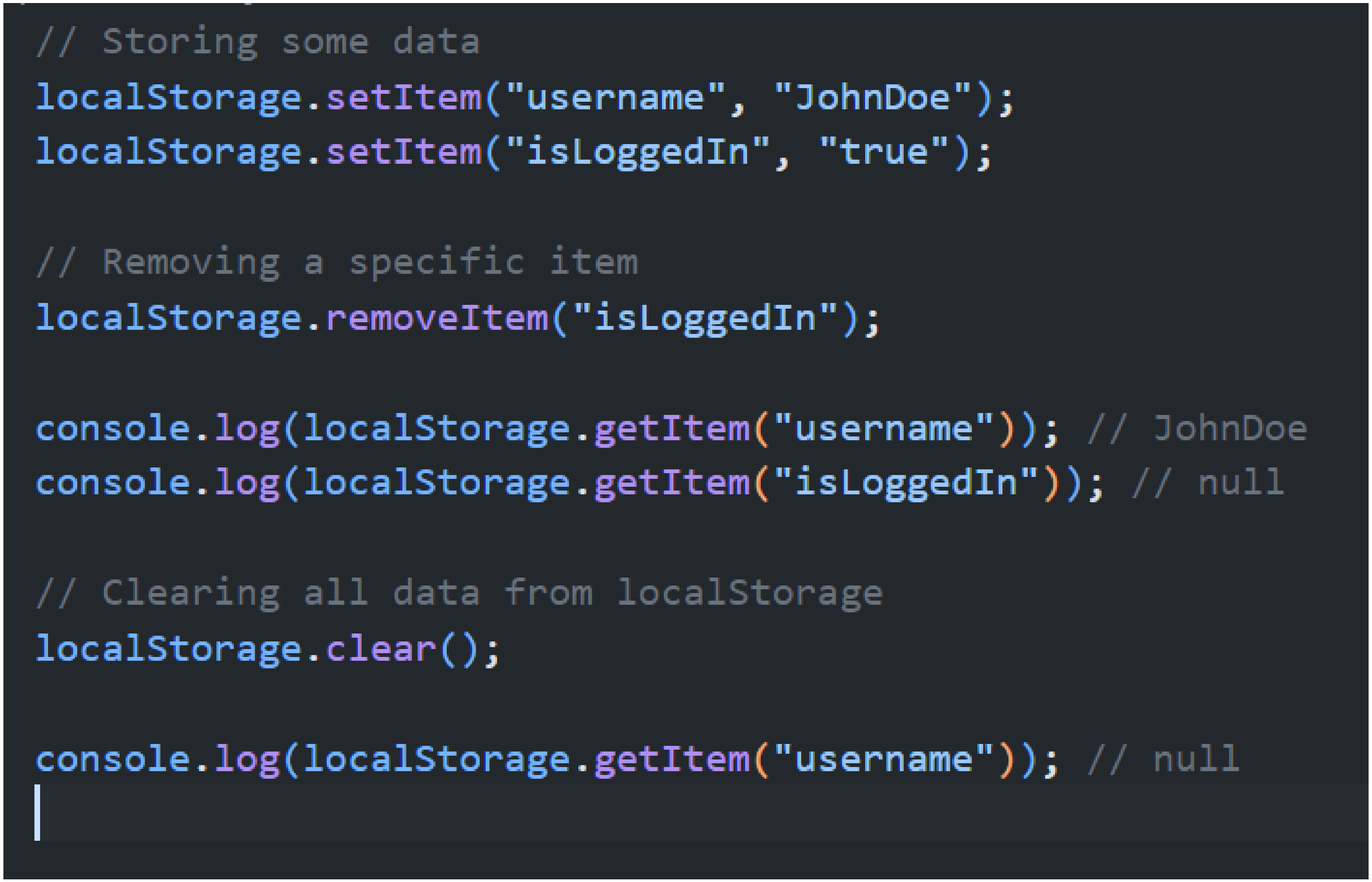
49. What is the difference between localStorage and sessionStorage?

Both localStorage and sessionStorage are web storage objects, but they differ in data persistence.
localStorage data persists even after the browser is closed and reopened.
sessionStorage data is cleared when the page session ends (i.e., when the tab is closed).
Both have the same API and are limited to storing string data.



```
// localStorage: persists even after the browser window is closed
localStorage.setItem(
  "persistentData",
  "This will remain after closing the browser"
);

// sessionStorage: data is cleared when the browser session ends
sessionStorage.setItem(
  "temporaryData",
  "This will be cleared when the session ends"
);

// Both are used similarly
console.log(localStorage.getItem("persistentData"));
console.log(sessionStorage.getItem("temporaryData"));
```

50. How do you delete a specific item from localStorage or clear all data from it?



```javascript
// Storing some data
localStorage.setItem("username", "JohnDoe");
localStorage.setItem("isLoggedIn", "true");

// Removing a specific item
localStorage.removeItem("isLoggedIn");

console.log(localStorage.getItem("username")); // JohnDoe
console.log(localStorage.getItem("isLoggedIn")); // null

// Clearing all data from localStorage
localStorage.clear();

console.log(localStorage.getItem("username")); // null
```