Crisp Notes On JavaScript

JavaScript Essentials: Quick Reference Guide! 🚀💡

Jimmy Ramani
@jimmyramani

Index

Jimmy Ramani
@jimmyramani

Index

Jimmy Ramani

@jimmyramani

Index

Jimmy Ramani
@jimmyramani

Why you should learn JavaScript ?

✓ It Works in The Browser 🌐

Like most languages, you don’t need to setup anything. You can run your code without any Environment.

✓ Easy to Learn

A very beginner friendly languages in which you don’t need to learn deal with complexities.

✓ Versatile Programming Language

From Front-End to Back-End, JavaScript can be used for almost anything. There’s nothing you can’t do with JavaScript.

✓ Big Community Support

Doesn't matter what error you face while learning or else. Just google it, and you’ll see tons of Solutions.

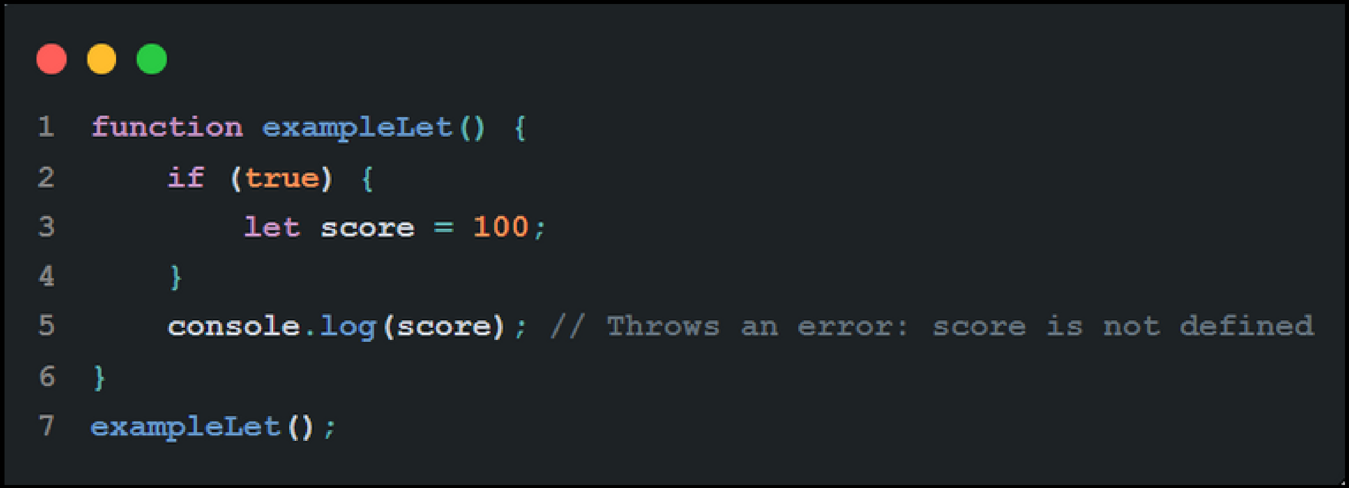
Jimmy Ramani
@jimmyramani

Let , Const & Var

✓ Let :

- Variables declared with let are block-scoped. They are confined to the block (curly braces) in which they are defined.
- They are also hoisted, but not initialized, so you cannot use them before they are declared.
- let variables can be updated but not re-declared within their scope.

Example :



```javascript
function exampleLet() {
    if (true) {
        let score = 100;
    }
    console.log(score); // Throws an error: score is not defined
}
exampleLet();
```

Jimmy Ramani

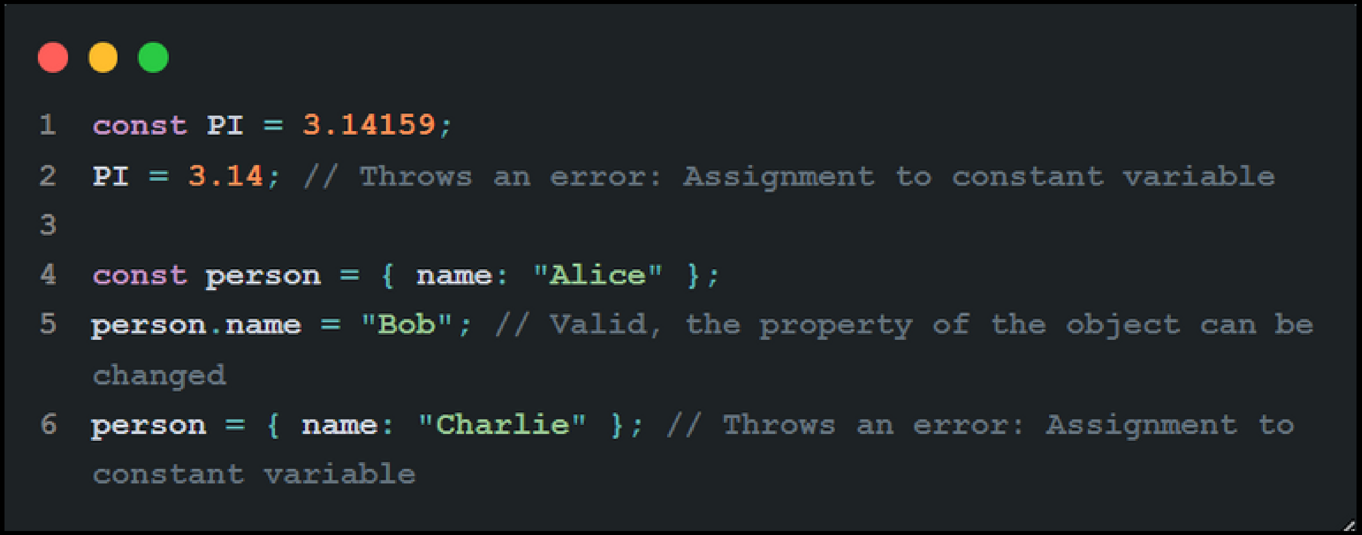
@jimmyramani

Let , Const & Var

✓ Const :

- Variables declared with const are block-scoped and cannot be reassigned after they are initialized.
- Like let, const variables are hoisted but not initialized.
- const is commonly used for values that should remain constant throughout the program.
- If the value being assigned is an object or an array, the variable itself cannot be reassigned to a new object or array, but the properties or elements of the object or array can be modified.

Example :



```
const PI = 3.14159;
PI = 3.14; // Throws an error: Assignment to constant variable

const person = { name: "Alice" };
person.name = "Bob"; // Valid, the property of the object can be changed
person = { name: "Charlie" }; // Throws an error: Assignment to constant variable
```

Jimmy Ramani

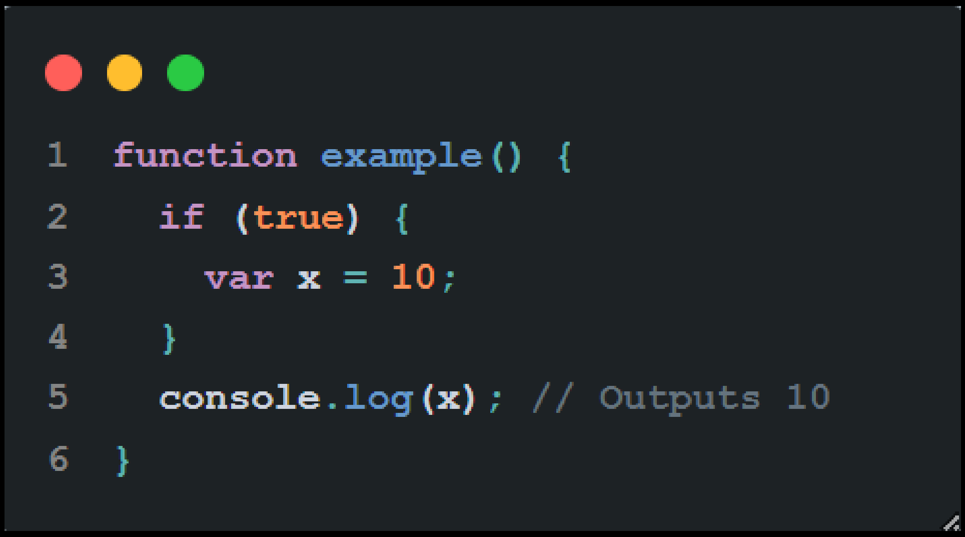
@jimmyramani

Let , Const & Var

✓ Var :

- Variables declared with var are function-scoped or globally scoped, but they are not block-scoped.
- They are hoisted to the top of their scope during execution, which means you can use a variable before it's declared (though its value will be undefined).
- var variables can be re-declared and updated within their scope.
- Since var doesn't have block scope, it can lead to unintended issues when used in loops and conditionals.

Example :



```
function example() {
  if (true) {
    var x = 10;
  }
  console.log(x); // Outputs 10
}
```

Jimmy Ramani

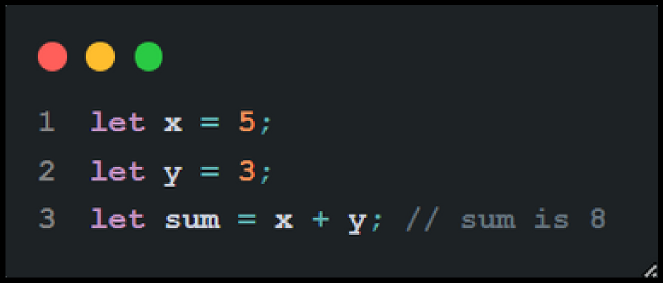
@jimmyramani

Operators

✓ Arithmetic Operators :

- Perform mathematical calculations on numbers.
- Examples: + (addition), - (subtraction), * (multiplication), / (division), % (modulus), ** (exponentiation).

Example :

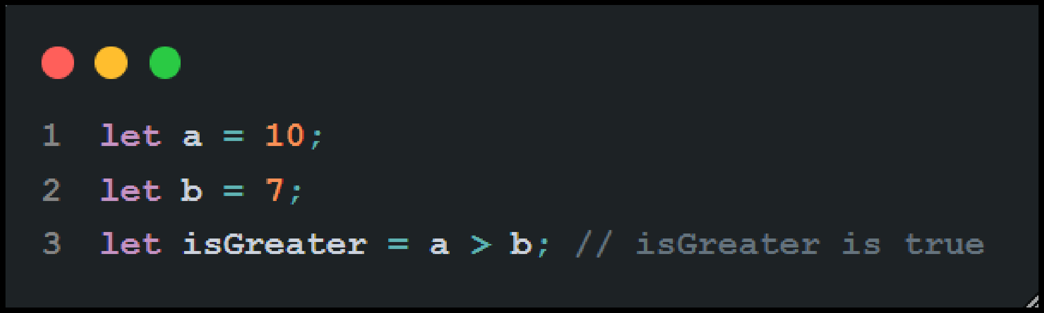


```
let x = 5;
let y = 3;
let sum = x + y; // sum is 8
```

✓ Comparison Operators :

- Compare two values and return a boolean result.
- Examples: == (equal to), != (not equal to), === (strictly equal to), !== (strictly not equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to).

Example :



```
let a = 10;
let b = 7;
let isGreater = a > b; // isGreater is true
```

Jimmy Ramani

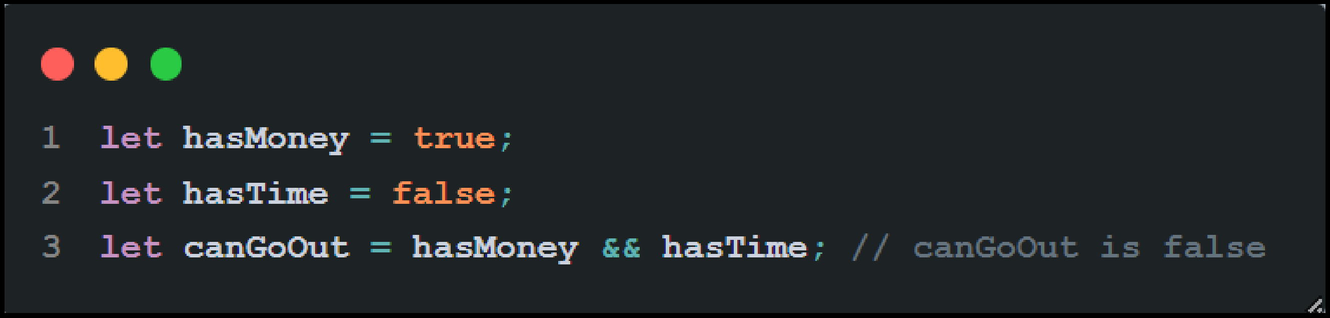
@jimmyramani

Operators

✓ Logical Operators :

- Perform logical operations on boolean values.
- Examples: && (logical AND), || (logical OR), ! (logical NOT).

Example :

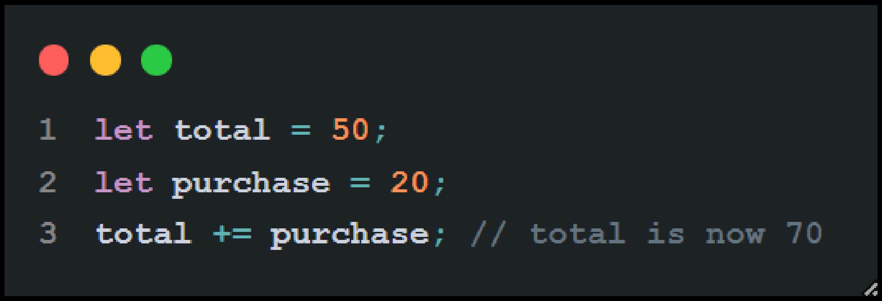


```
let hasMoney = true;
let hasTime = false;
let canGoOut = hasMoney && hasTime; // canGoOut is false
```

✓ Assignment Operators :

- Assign values to variables.
- Examples: = (assignment), += (addition assignment), -= (subtraction assignment), *= (multiplication assignment), /= (division assignment), %= (modulus assignment).

Example :



```
let total = 50;
let purchase = 20;
total += purchase; // total is now 70
```

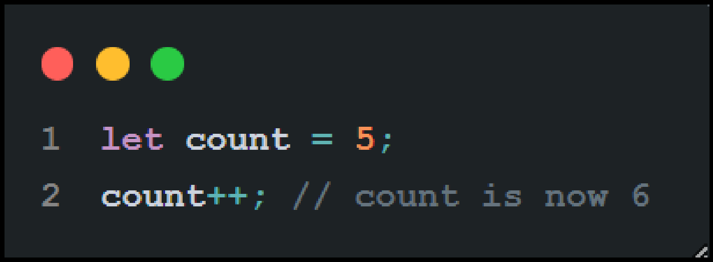
Jimmy Ramani
@jimmyramani

Operators

✓ Unary Operators :

- Operate on a single value.
- Examples: ++ (increment), -- (decrement), + (unary plus), - (unary minus), ! (logical NOT).

Example :

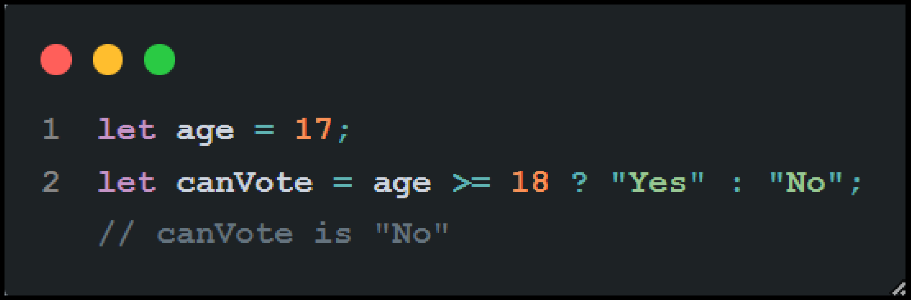


```javascript
let count = 5;
count++; // count is now 6
```

✓ Ternary Operators :

- Determine the type of a value.
- Example: typeof (returns a string representing the type of a value), instanceof (checks if an object is an instance of a particular class or constructor).

Example :



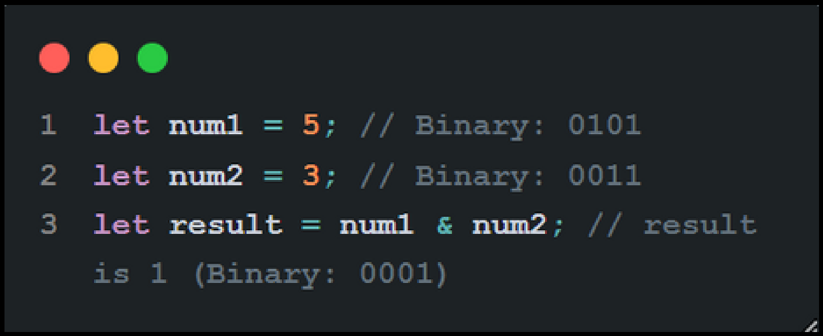
```javascript
let age = 17;
let canVote = age >= 18 ? "Yes" : "No";
// canVote is "No"
```

Operators

✓ Bitwise Operators :

- Perform operations on binary representations of numbers.
- Examples: & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (left shift), >> (right shift), >>> (unsigned right shift).

Example :

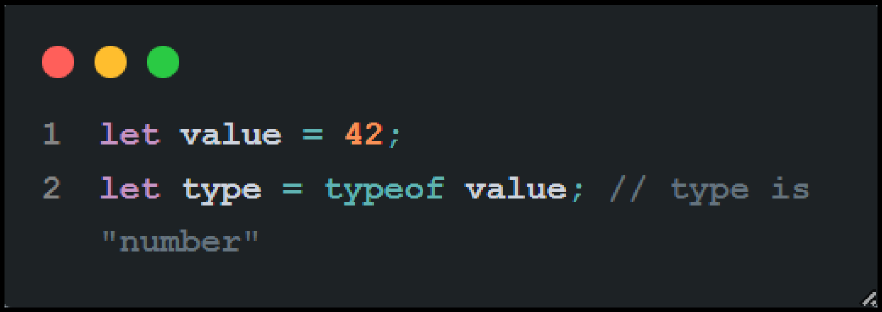


```
let num1 = 5; // Binary: 0101
let num2 = 3; // Binary: 0011
let result = num1 & num2; // result is 1 (Binary: 0001)
```

✓ Type Operators :

- Determine the type of a value.
- Example: typeof (returns a string representing the type of a value), instanceof (checks if an object is an instance of a particular class or constructor).

Example :



```
let value = 42;
let type = typeof value; // type is "number"
```

Jimmy Ramani

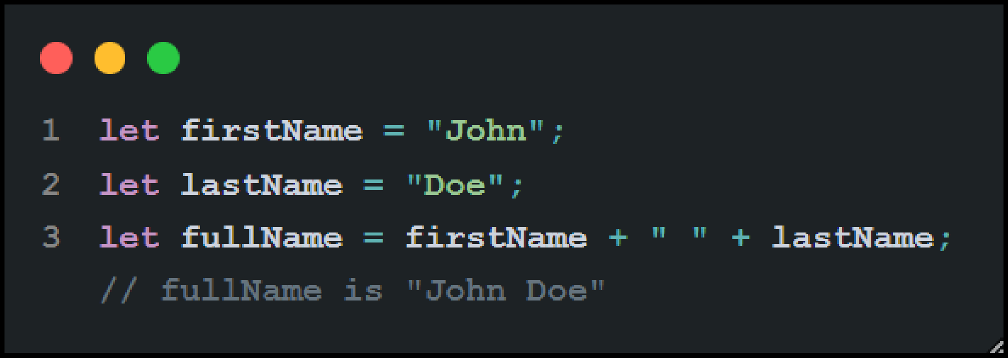
@jimmyramani

Operators

✓ String Operators :

- Concatenate strings.
- Example: + (concatenation).

Example :



```
let firstName = "John";
let lastName = "Doe";
let fullName = firstName + " " + lastName;
// fullName is "John Doe"
```

Jimmy Ramani

@jimmyramani

Data Types

✓ Number Represents both integer and floating-point numbers.

✓ String Represents a sequence of characters (text).

✓ Boolean Represents a logical value, either true or false.

✓ Undefined Represents an uninitialized variable or a function that doesn't return a value.

✓ Null Represents the intentional absence of any value or object.

✓ Object Represents a collection of key-value pairs or properties.

✓ Array Represents an ordered collection of values.

✓ Function Represents a block of code that can be invoked or called.

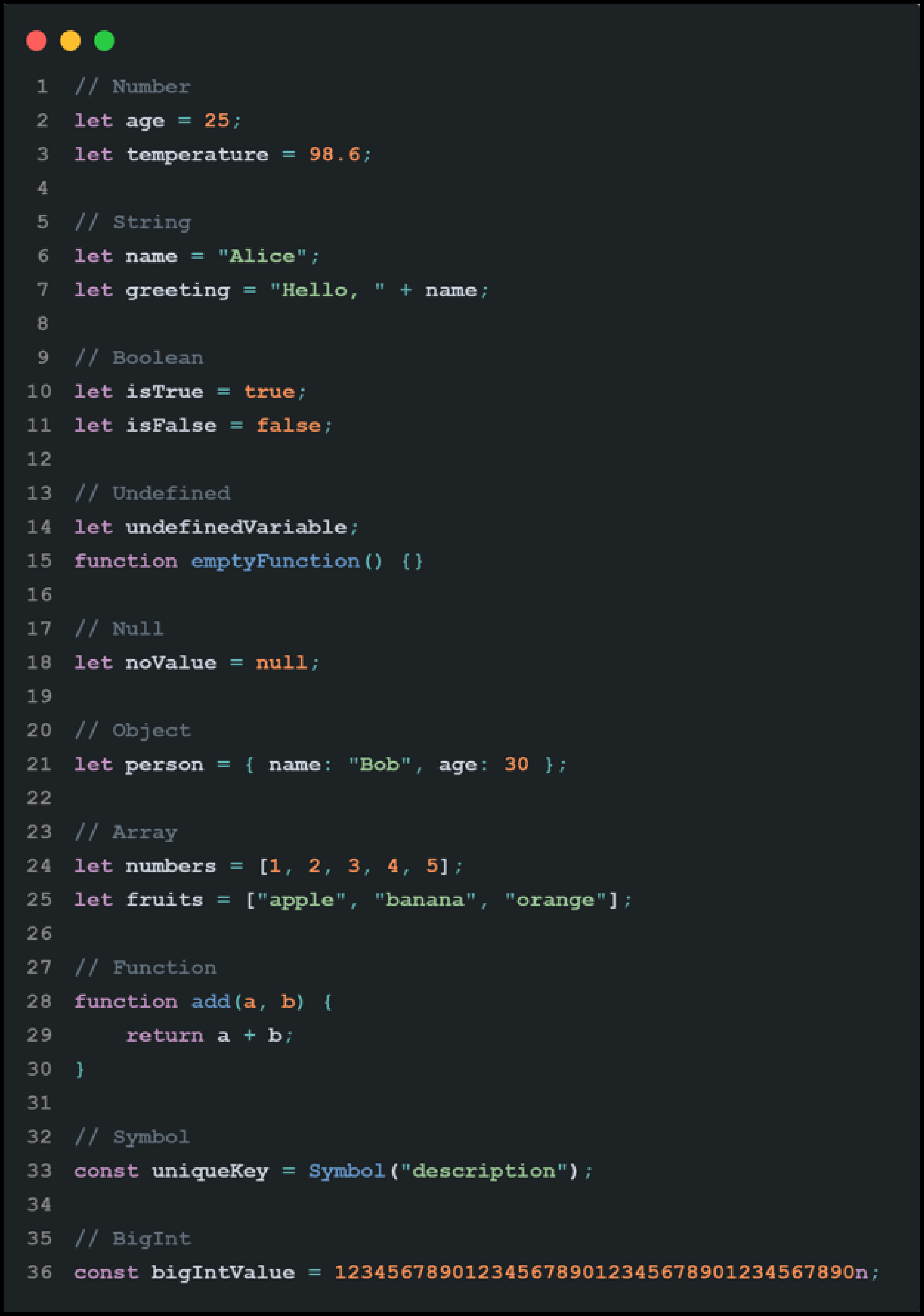
✓ Symbol Represents a unique and immutable value, often used as object property keys.

✓ BigInt Represents arbitrarily large integers.

Jimmy Ramani

@jimmyramani

Data Types

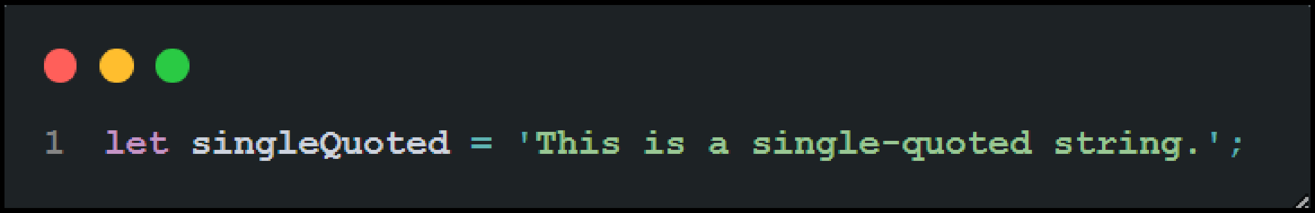


```javascript
// Number
let age = 25;
let temperature = 98.6;

// String
let name = "Alice";
let greeting = "Hello, " + name;

// Boolean
let isTrue = true;
let isFalse = false;

// Undefined
let undefinedVariable;
function emptyFunction() {}

// Null
let noValue = null;

// Object
let person = { name: "Bob", age: 30 };

// Array
let numbers = [1, 2, 3, 4, 5];
let fruits = ["apple", "banana", "orange"];

// Function
function add(a, b) {
    return a + b;
}

// Symbol
const uniqueKey = Symbol("description");

// BigInt
const bigIntValue = 1234567890123456789012345678901234567890n;
```

Strings

✓ Single Quotes (' ')

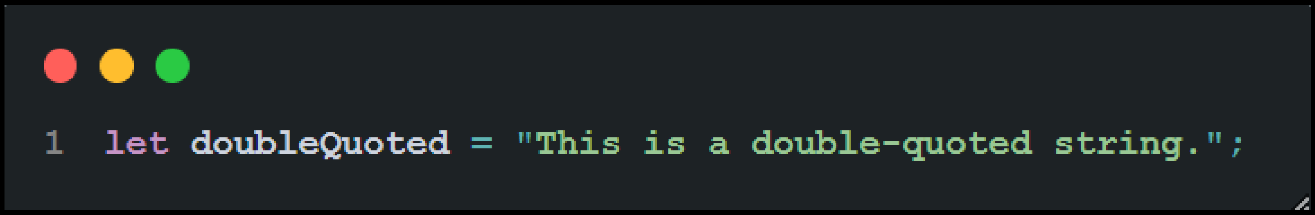
Strings created with single quotes are enclosed within single quotes.



```
let singleQuoted = 'This is a single-quoted string.';
```

✓ Double Quotes (" ")

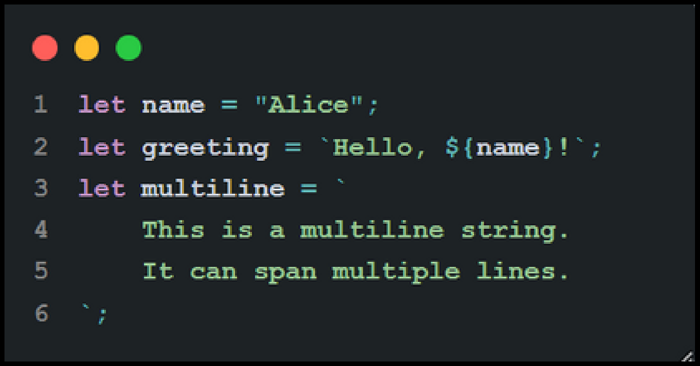
Strings created with double quotes are enclosed within double quotes.



```
let doubleQuoted = "This is a double-quoted string.";
```

✓ Backticks (` `)

Backticks are used to create template literals, which offer more advanced string features, such as string interpolation and multiline strings.



```
let name = "Alice";
let greeting = `Hello, ${name}!`;
let multiline = `
    This is a multiline string.
    It can span multiple lines.
`;
```

Jimmy Ramani

@jimmyramani

Events

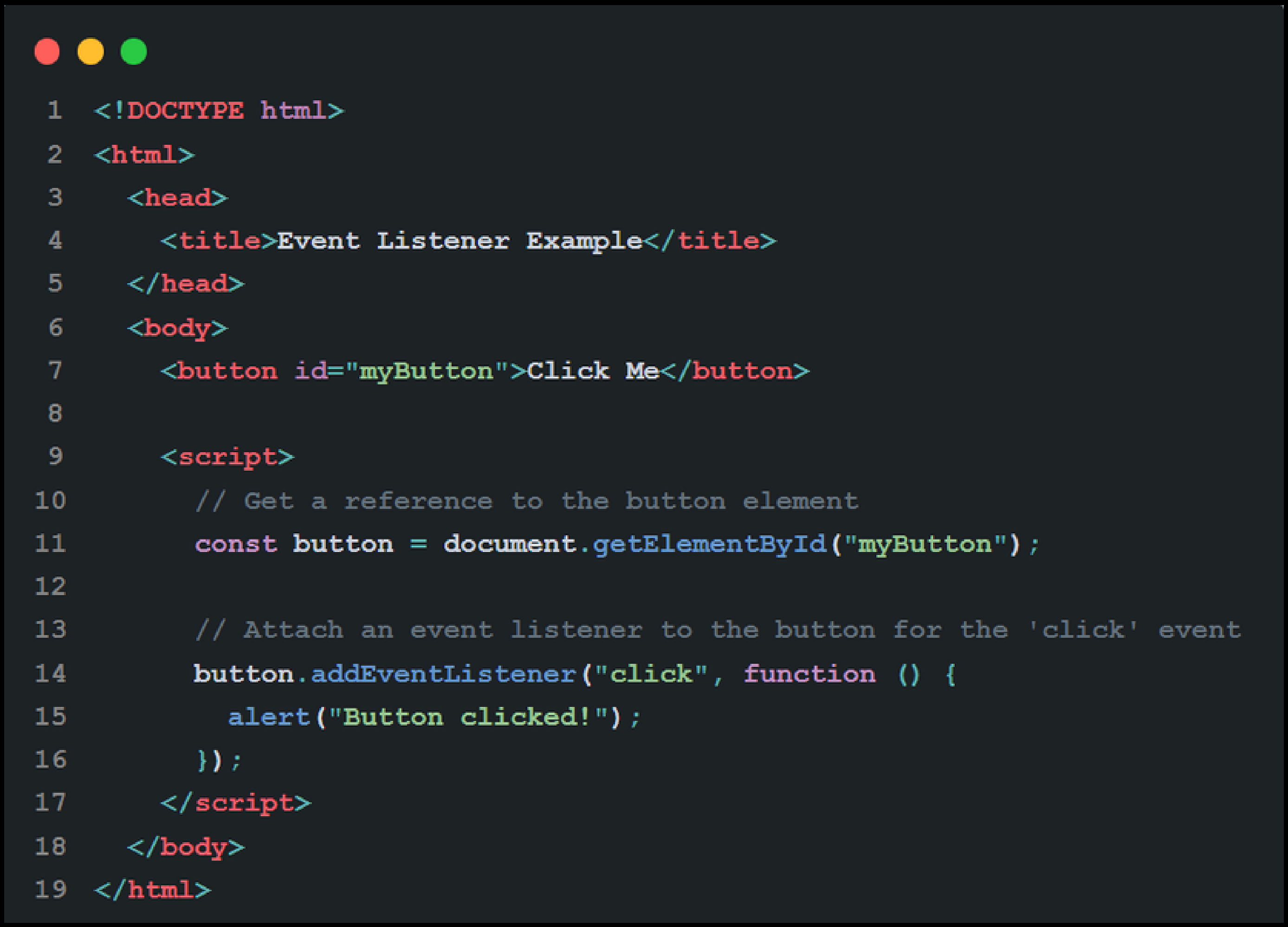
Events are actions or occurrences that happen in the browser, such as a user clicking a button, resizing the window, or pressing a key. JavaScript allows you to capture and handle these events to create interactive and responsive web applications. Events enable you to execute specific code when something happens on a web page.

✓ Types

- User interactions: click, double-click, mouseover, mouseout, keydown, keyup, etc.
- HTML form events: submit, change, input, focus, blur, etc.
- Window events: load, unload, resize, scroll, etc.
- Network events: XMLHttpRequest events (readystatechange, load, error), WebSocket events, etc.

Event Listener

An Event Listener in JavaScript is a function that "listens" for a specific event to occur on an HTML element and then executes a specified block of code in response to that event. Event listeners are used to make web pages interactive and respond to user actions or other events.



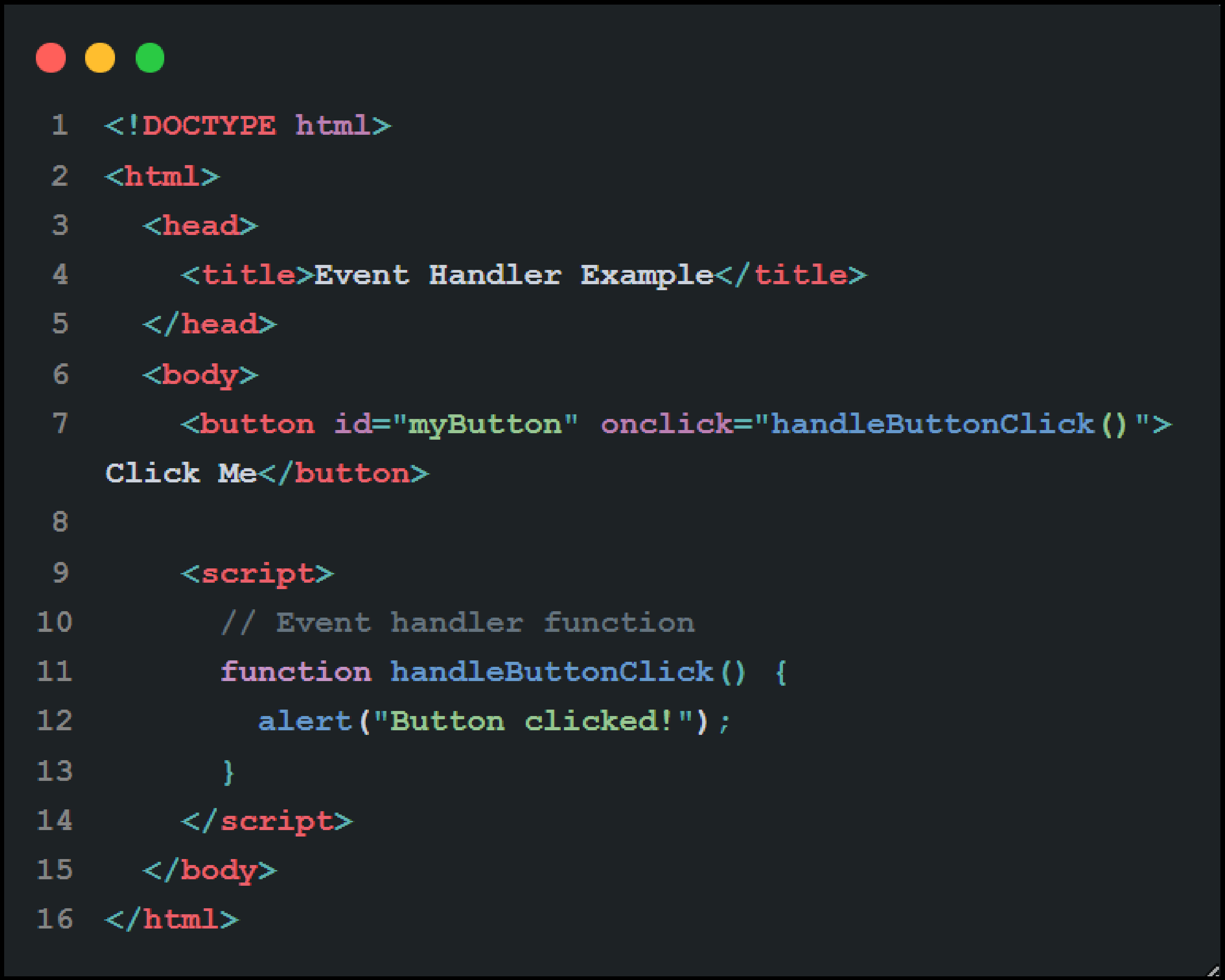
```html
<!DOCTYPE html>
<html>
  <head>
    <title>Event Listener Example</title>
  </head>
  <body>
    <button id="myButton">Click Me</button>

    <script>
      // Get a reference to the button element
      const button = document.getElementById("myButton");

      // Attach an event listener to the button for the 'click' event
      button.addEventListener("click", function () {
        alert("Button clicked!");
      });
    </script>
  </body>
</html>
```

the code sets up an event listener for the 'click' event on the button element with the ID 'myButton'. When the button is clicked, the function provided to the addEventListener method is executed, which shows an alert with the message "Button clicked!".

Jimmy Ramani
@jimmyramani

Event Handler

An Event Handler in JavaScript is a function that gets executed in response to a specific event occurring on an HTML element. Event handlers are assigned directly as attributes of HTML elements and are invoked when the associated event takes place.



```html
<!DOCTYPE html>
<html>
  <head>
    <title>Event Handler Example</title>
  </head>
  <body>
    <button id="myButton" onclick="handleButtonClick()">
Click Me</button>

    <script>
      // Event handler function
      function handleButtonClick() {
        alert("Button clicked!");
      }
    </script>
  </body>
</html>
```

The HTML file includes a button element with the ID 'myButton'. The onclick attribute is added to the button element, which specifies the name of the function (handleButtonClick) that should be executed when the button is clicked. The JavaScript code within the <script> section defines the handleButtonClick function. This function displays an alert when the button is clicked.

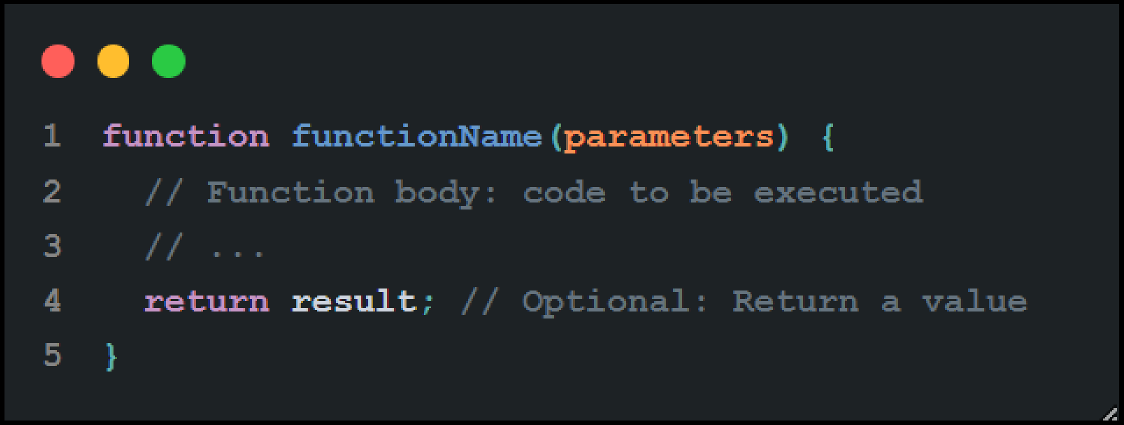
Jimmy Ramani

@jimmyramani

Function

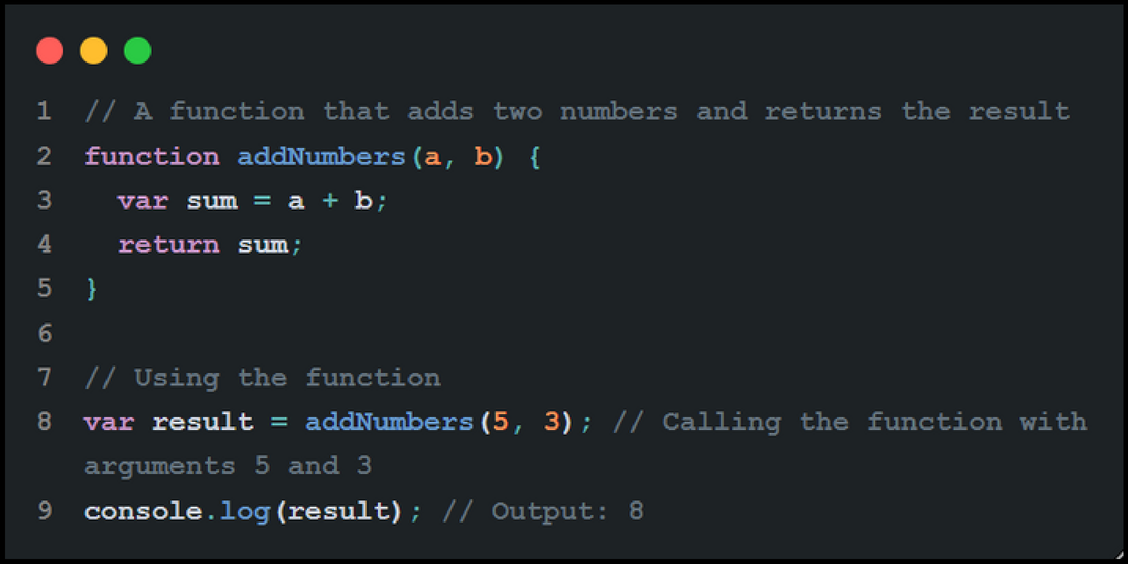
A function is a block of code that performs a specific task or a set of tasks. Functions allow you to encapsulate code into reusable units, making your code more organized, modular, and easier to maintain.

Basic Syntax



```
function functionName(parameters) {
  // Function body: code to be executed
  // ...
  return result; // Optional: Return a value
}
```

Example :

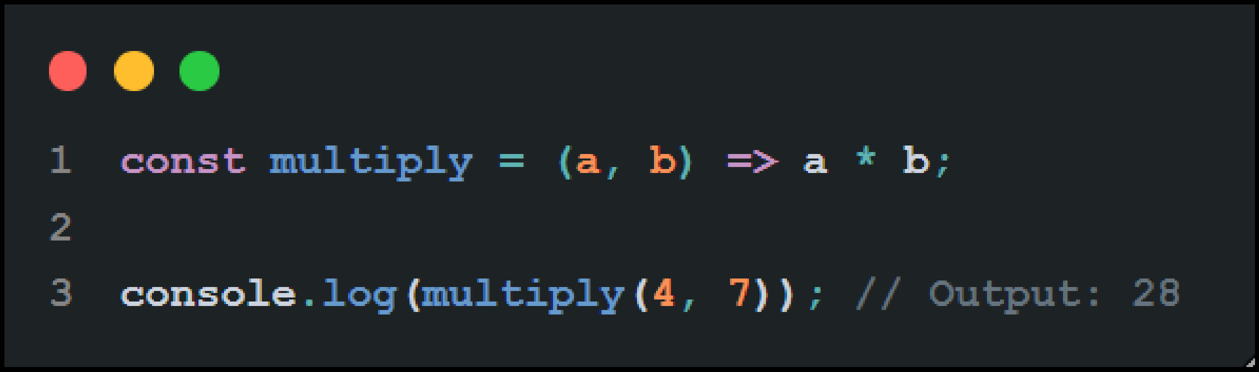


```
// A function that adds two numbers and returns the result
function addNumbers(a, b) {
  var sum = a + b;
  return sum;
}

// Using the function
var result = addNumbers(5, 3); // Calling the function with
arguments 5 and 3
console.log(result); // Output: 8
```

Jimmy Ramani

@jimmyramani

You can also use **arrow functions,** which are a more concise way to define functions:



```
const multiply = (a, b) => a * b;

console.log(multiply(4, 7)); // Output: 28
```

Arrow functions are particularly useful for short and simple functions.

Functions are a foundational concept in JavaScript and are crucial for building modular and maintainable code. They allow you to encapsulate logic, promote code reuse, and improve overall code readability.

More about Function Methods :

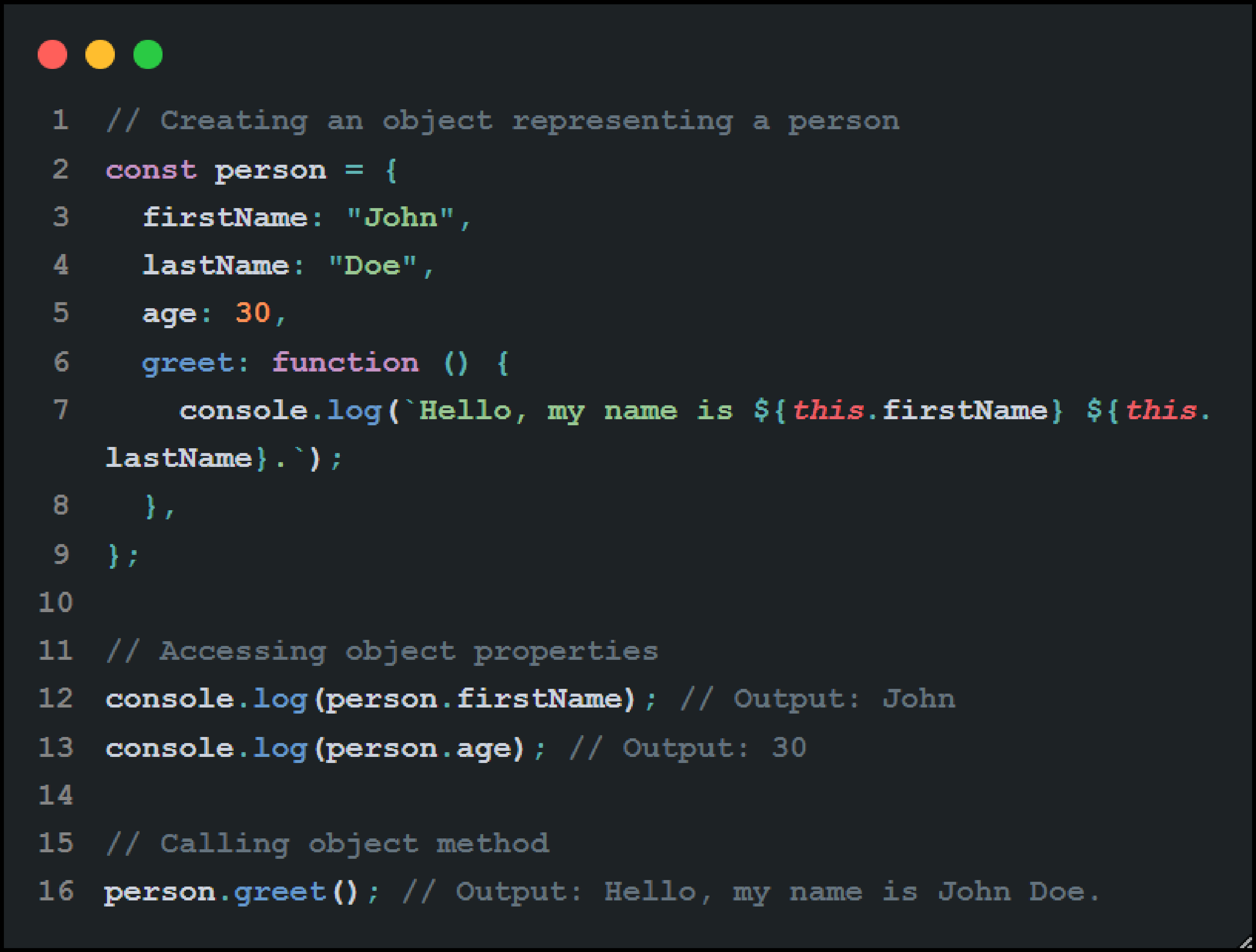
https://www.linkedin.com/feed/update/urn:li:activity:7093226017141506048?utm_source=share&utm_medium=member_desktop

Go Through
QR

Jimmy Ramani
@jimmyramani

Objects

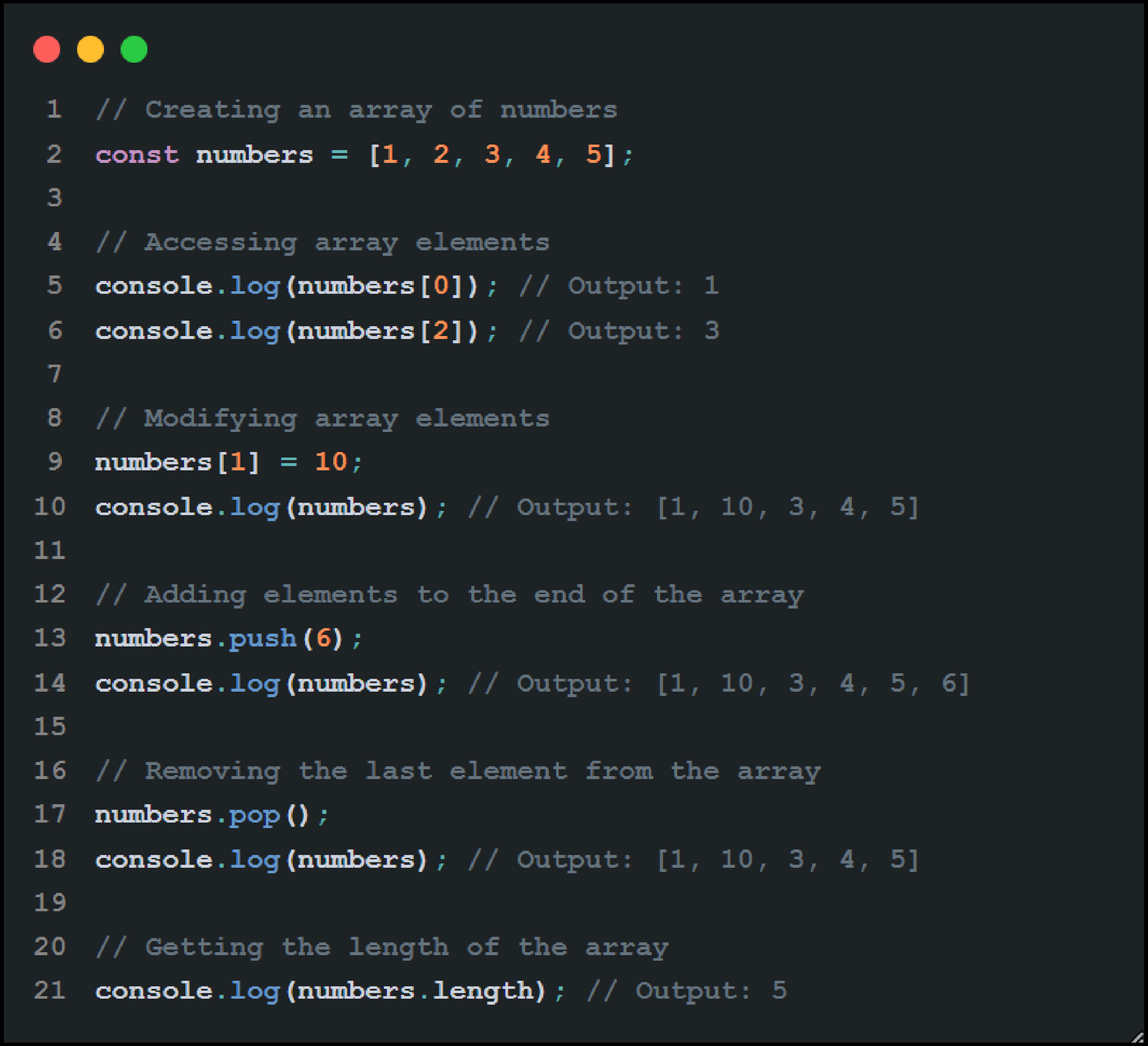
An Object is a complex data type that allows you to store and organize related data and functions as properties and methods. Objects provide a way to model real-world entities and their behaviors in your code. Objects consist of key-value pairs, where each key (property) is associated with a value (data) or a function (method).



```javascript
// Creating an object representing a person
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
  },
};

// Accessing object properties
console.log(person.firstName); // Output: John
console.log(person.age); // Output: 30

// Calling object method
person.greet(); // Output: Hello, my name is John Doe.
```

Array

An **Array** is a data structure that allows you to store and manage a collection of values. Arrays can contain elements of various data types, such as numbers, strings, objects, and even other arrays. Arrays are widely used to organize and manipulate data in a structured manner.



```javascript
// Creating an array of numbers
const numbers = [1, 2, 3, 4, 5];

// Accessing array elements
console.log(numbers[0]); // Output: 1
console.log(numbers[2]); // Output: 3

// Modifying array elements
numbers[1] = 10;
console.log(numbers); // Output: [1, 10, 3, 4, 5]

// Adding elements to the end of the array
numbers.push(6);
console.log(numbers); // Output: [1, 10, 3, 4, 5, 6]

// Removing the last element from the array
numbers.pop();
console.log(numbers); // Output: [1, 10, 3, 4, 5]

// Getting the length of the array
console.log(numbers.length); // Output: 5
```

More about Arrays Methods :

https://www.linkedin.com/feed/update/urn:li:activity:7094369342129139712?utm_source=share&utm_medium=member_desktop

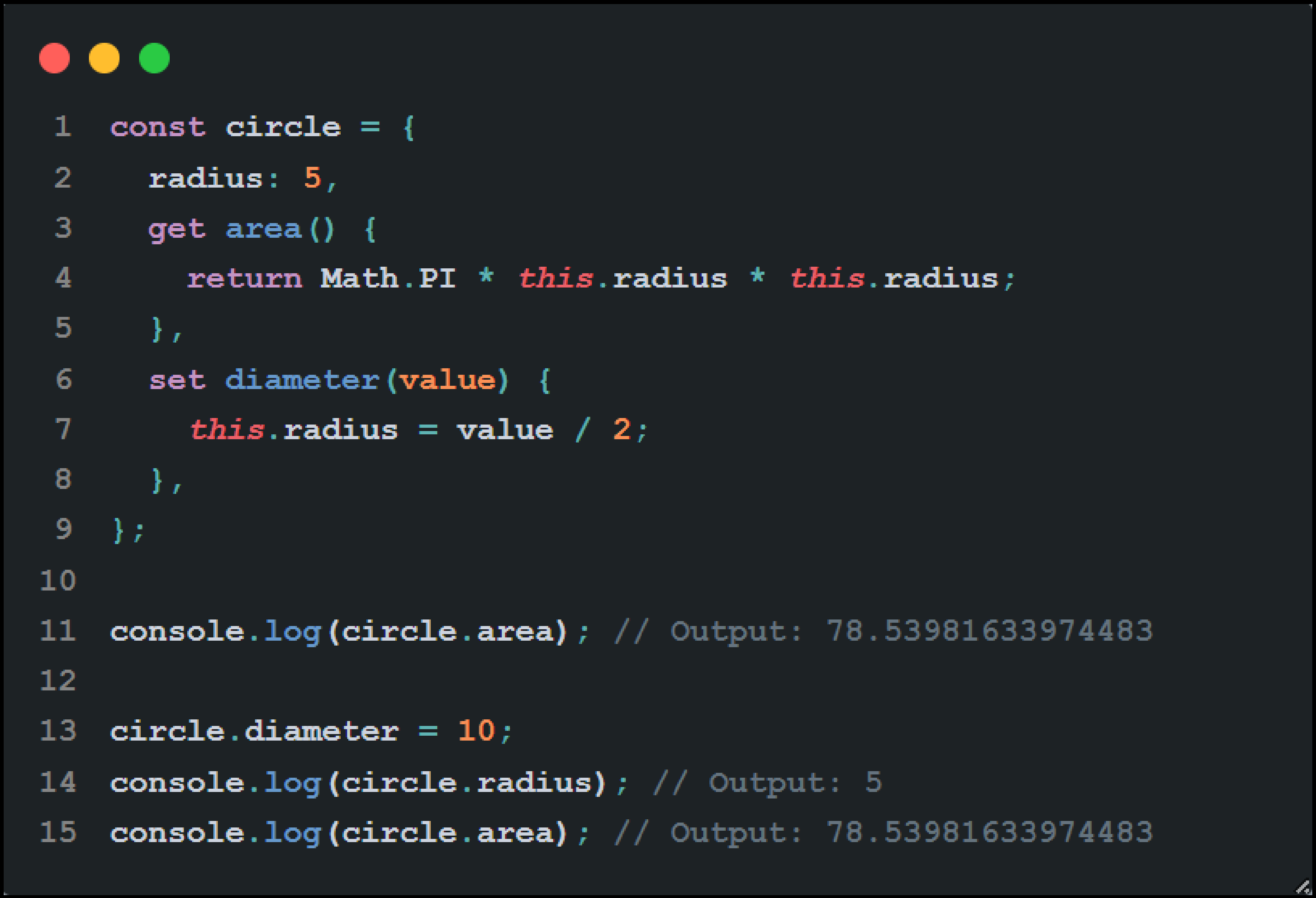
Go Through QR

Jimmy Ramani

@jimmyramani

Getter & Setter

Getters and **Setters** are special methods that allow you to define how the properties of an object are accessed and assigned. Getters are used to retrieve the value of a property, while setters are used to set or modify the value of a property. They provide a way to control the behavior of accessing and modifying object properties.



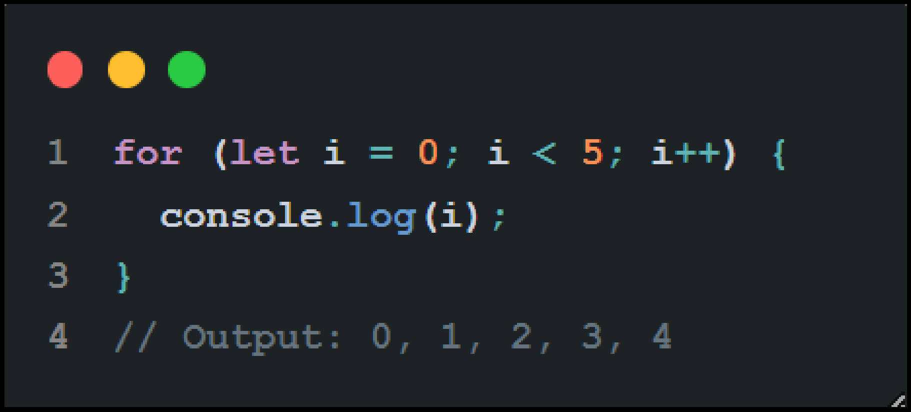
```
const circle = {
  radius: 5,
  get area() {
    return Math.PI * this.radius * this.radius;
  },
  set diameter(value) {
    this.radius = value / 2;
  },
};

console.log(circle.area); // Output: 78.53981633974483

circle.diameter = 10;
console.log(circle.radius); // Output: 5
console.log(circle.area); // Output: 78.53981633974483
```

Jimmy Ramani

@jimmyramani

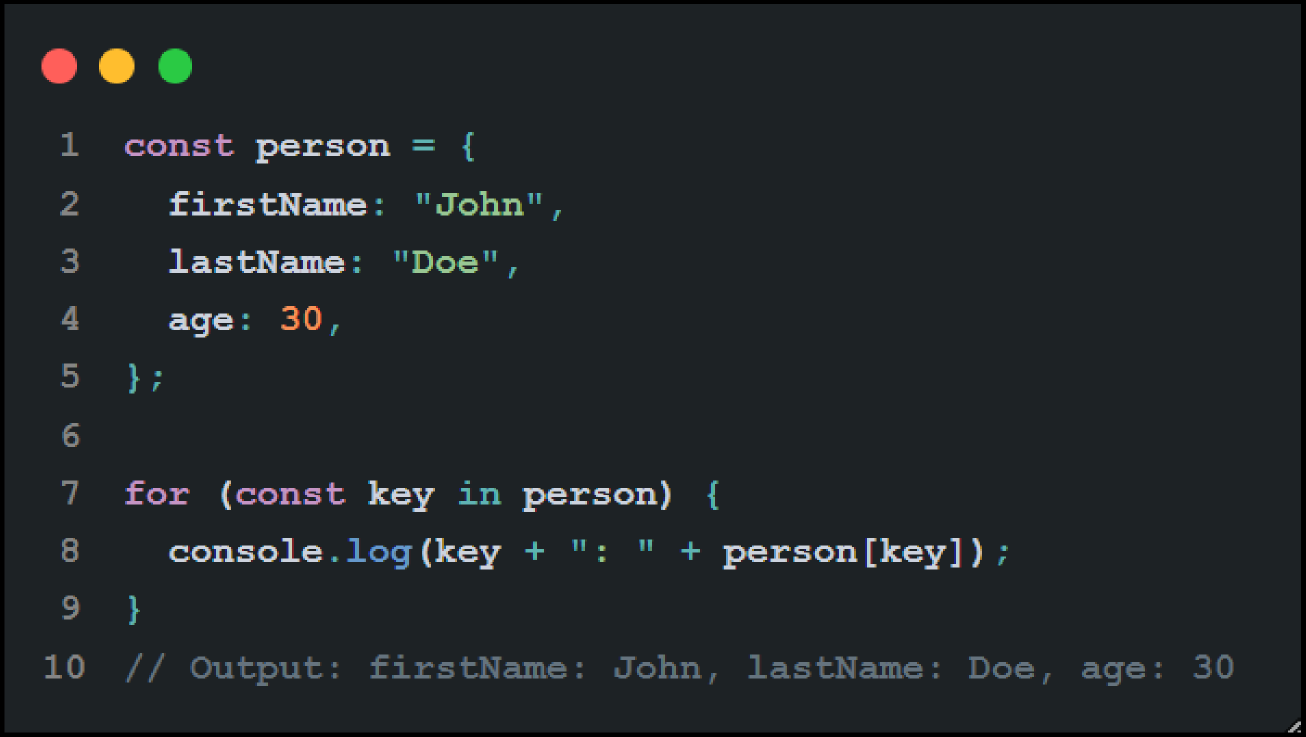
Loops

✓ For Loop



```javascript
for (let i = 0; i < 5; i++) {
  console.log(i);
}
// Output: 0, 1, 2, 3, 4
```

✓ For-in Loop



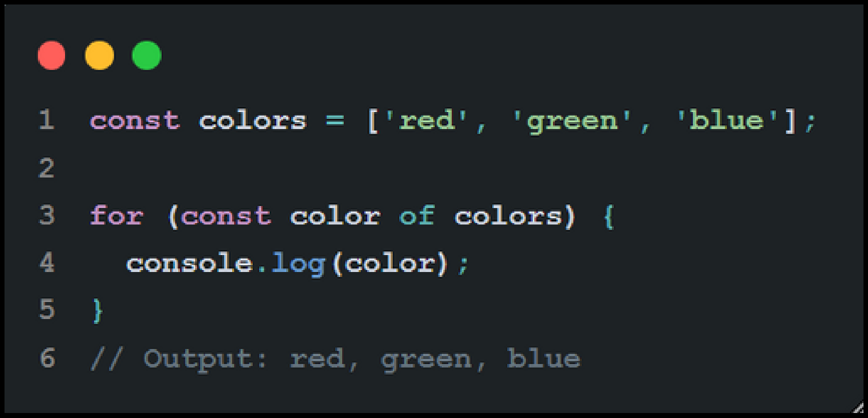
```javascript
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
};

for (const key in person) {
  console.log(key + ": " + person[key]);
}
// Output: firstName: John, lastName: Doe, age: 30
```

Jimmy Ramani

@jimmyramani

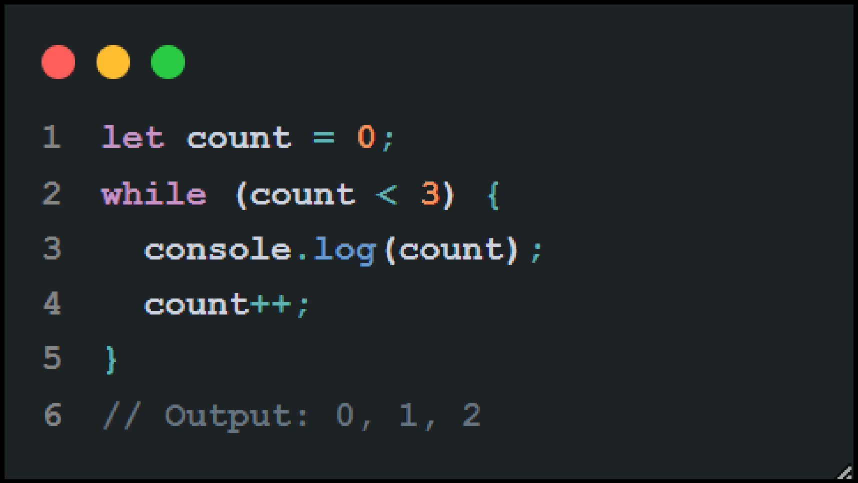
Loops

✓ For-of Loop



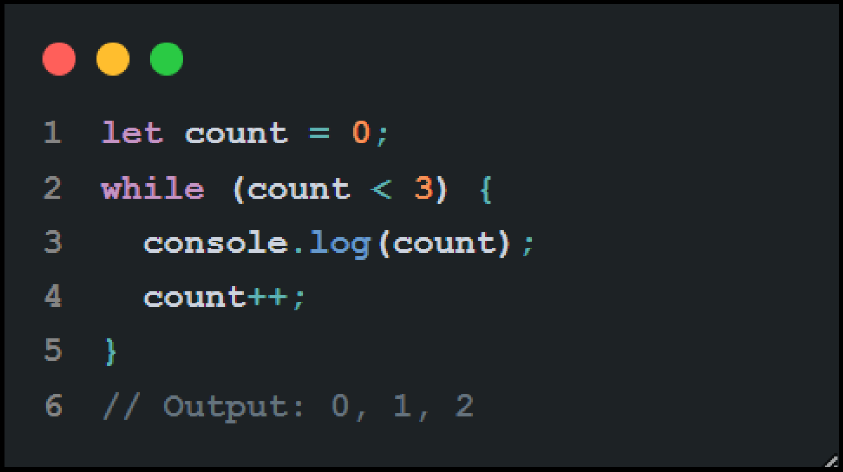
```javascript
const colors = ['red', 'green', 'blue'];

for (const color of colors) {
  console.log(color);
}
// Output: red, green, blue
```

✓ While Loop



```javascript
let count = 0;
while (count < 3) {
  console.log(count);
  count++;
}
// Output: 0, 1, 2
```

✓ Do While Loop



```javascript
let count = 0;
while (count < 3) {
  console.log(count);
  count++;
}
// Output: 0, 1, 2
```

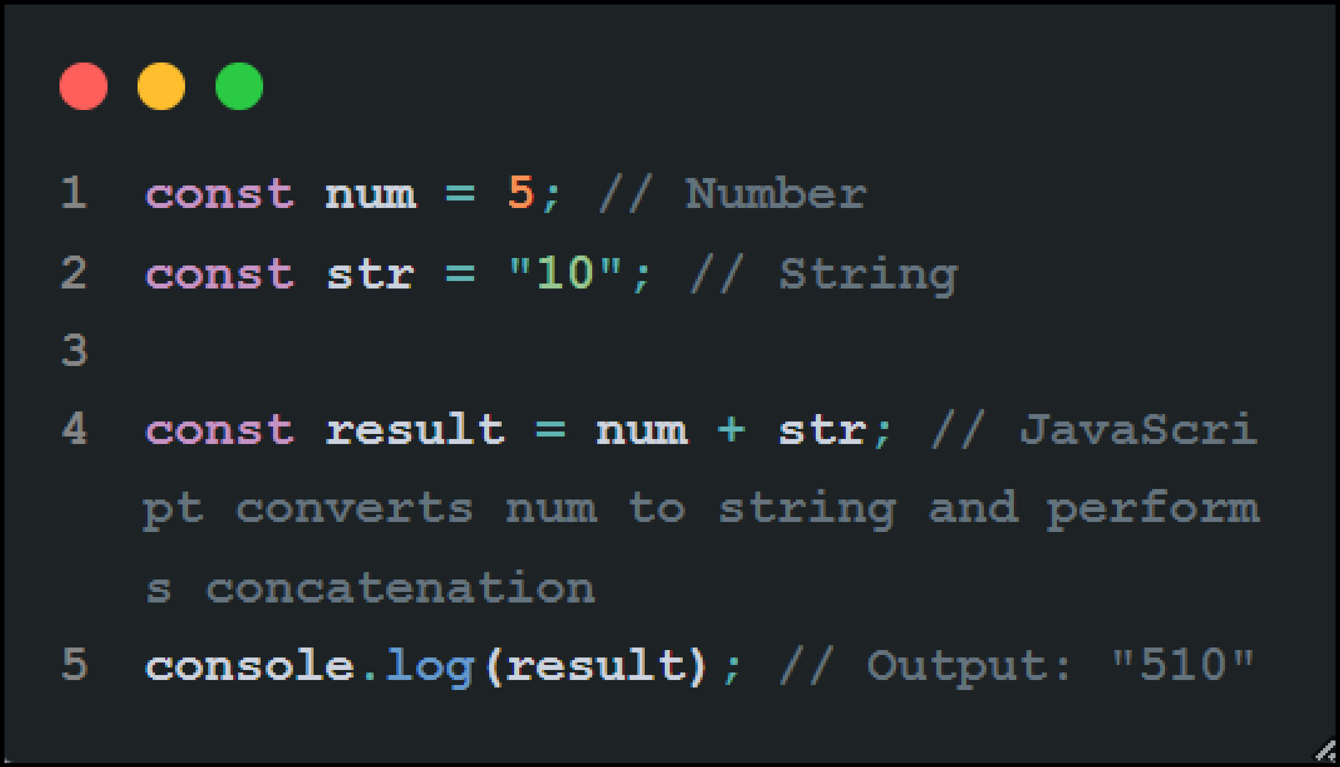
Jimmy Ramani

@jimmyramani

Type Conversion

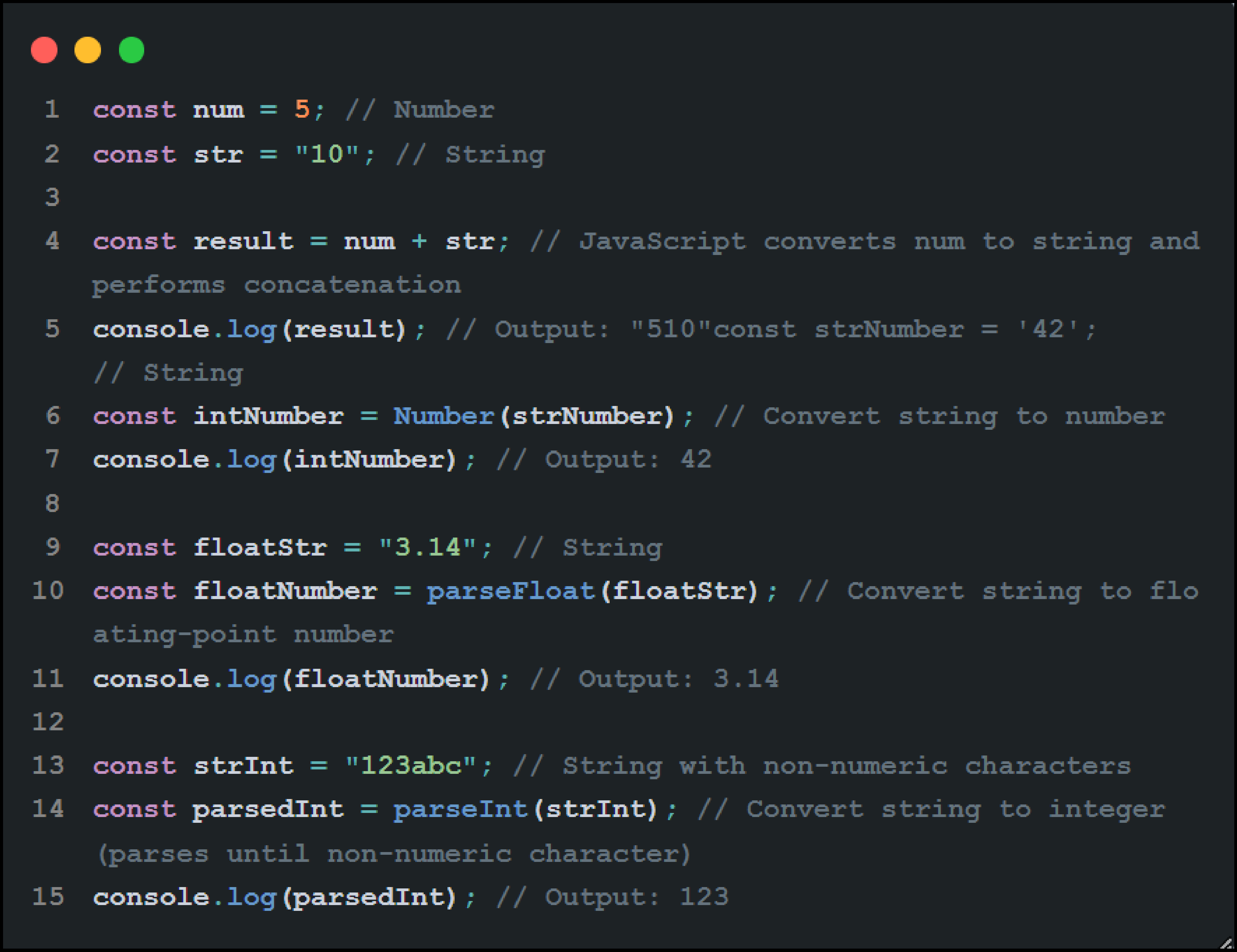
Type conversion, also known as type coercion, refers to the process of converting a value from one data type to another in JavaScript. JavaScript performs automatic type conversion in many situations, but you can also explicitly convert values using various methods and functions

✓ Implicit Type Conversion :



```javascript
const num = 5; // Number
const str = "10"; // String

const result = num + str; // JavaScript converts num to string and performs concatenation
console.log(result); // Output: "510"
```

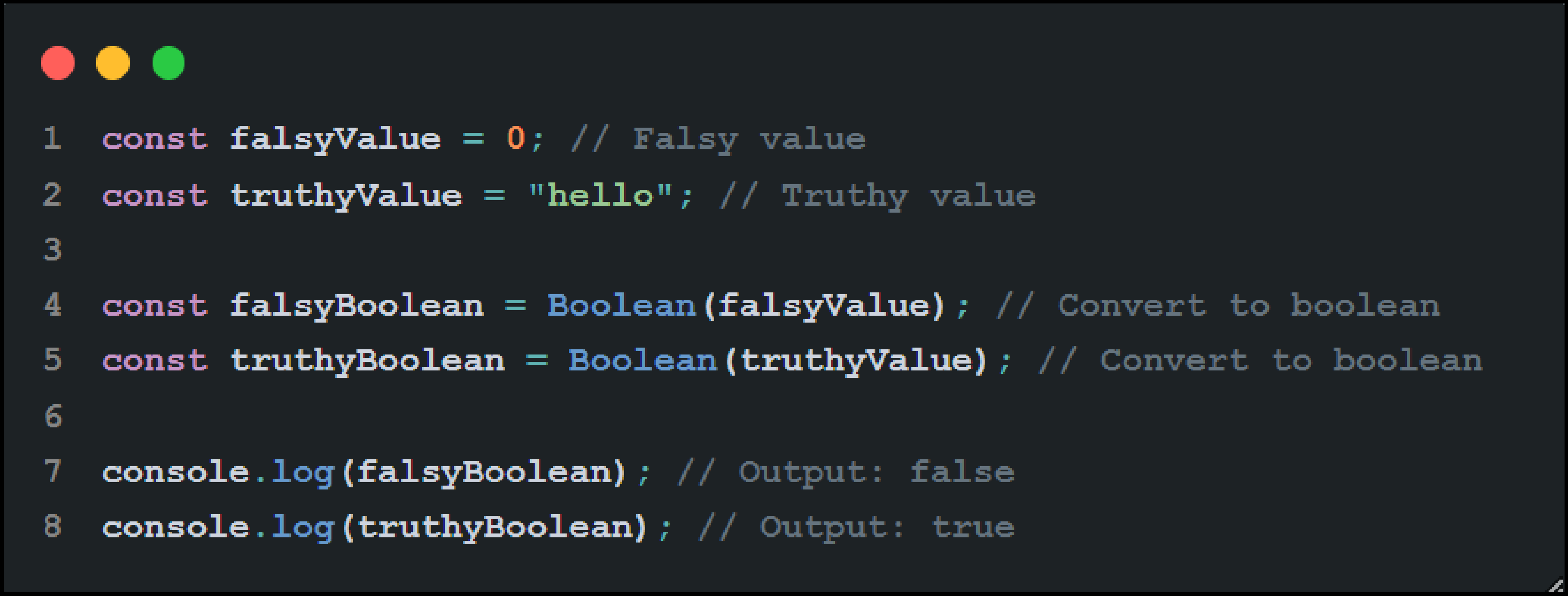
✓ Explicit Type Conversion :



```javascript
const num = 5; // Number
const str = "10"; // String

const result = num + str; // JavaScript converts num to string and performs concatenation
console.log(result); // Output: "510"const strNumber = '42'; // String
const intNumber = Number(strNumber); // Convert string to number
console.log(intNumber); // Output: 42

const floatStr = "3.14"; // String
const floatNumber = parseFloat(floatStr); // Convert string to floating-point number
console.log(floatNumber); // Output: 3.14

const strInt = "123abc"; // String with non-numeric characters
const parsedInt = parseInt(strInt); // Convert string to integer (parses until non-numeric character)
console.log(parsedInt); // Output: 123
```

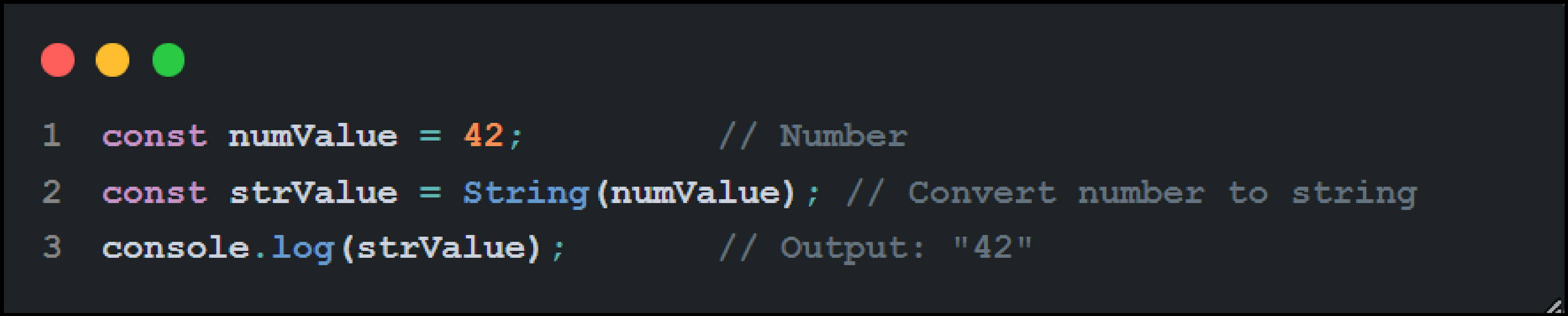
Jimmy Ramani
@jimmyramani

✓ Using Boolean() for Truthy/Falsy Conversion :



```javascript
const falsyValue = 0; // Falsy value
const truthyValue = "hello"; // Truthy value

const falsyBoolean = Boolean(falsyValue); // Convert to boolean
const truthyBoolean = Boolean(truthyValue); // Convert to boolean

console.log(falsyBoolean); // Output: false
console.log(truthyBoolean); // Output: true
```

✓ Using String() for Explicit String Conversion :



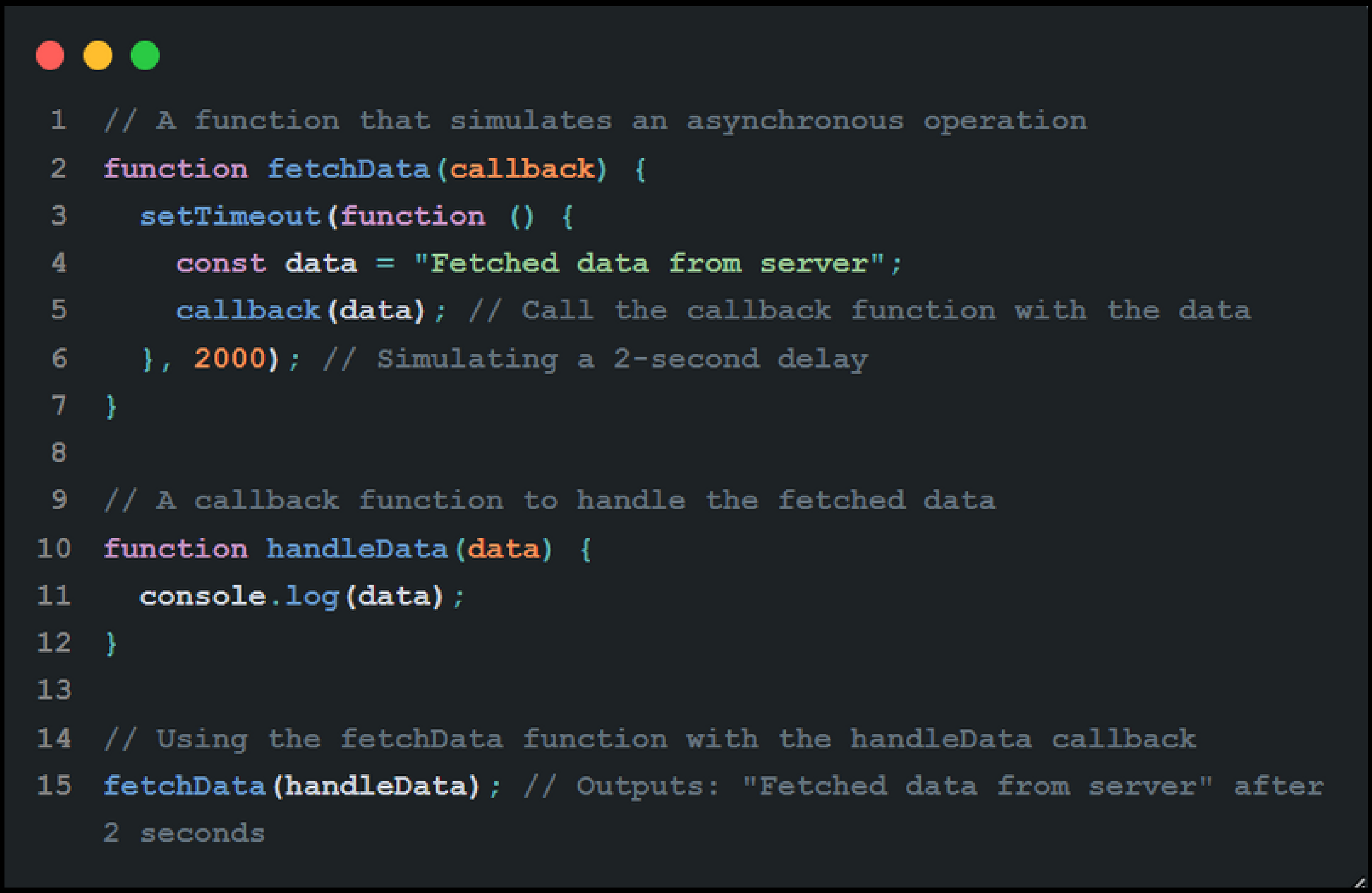
```javascript
const numValue = 42;         // Number
const strValue = String(numValue); // Convert number to string
console.log(strValue);       // Output: "42"
```

Jimmy Ramani

@jimmyramani

CallBack

a **callback** is a function that is passed as an argument to another function and is intended to be executed later, often after an asynchronous operation or a certain event occurs. Callbacks are a fundamental concept in JavaScript and are commonly used to handle asynchronous operations, such as AJAX requests, timers, and event handling.



```javascript
// A function that simulates an asynchronous operation
function fetchData(callback) {
  setTimeout(function () {
    const data = "Fetched data from server";
    callback(data); // Call the callback function with the data
  }, 2000); // Simulating a 2-second delay
}

// A callback function to handle the fetched data
function handleData(data) {
  console.log(data);
}

// Using the fetchData function with the handleData callback
fetchData(handleData); // Outputs: "Fetched data from server" after 2 seconds
```

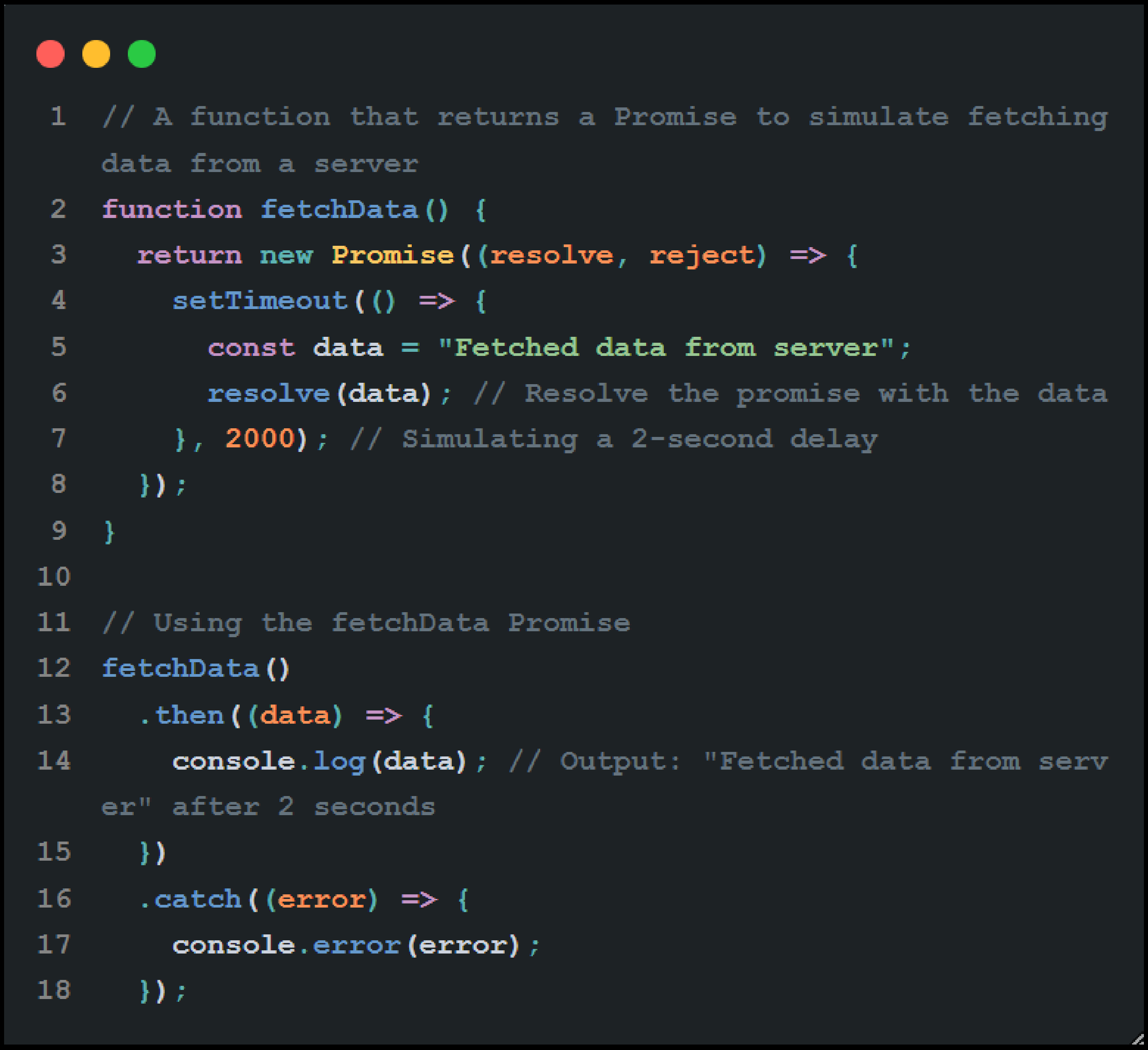
Callbacks are used extensively in JavaScript for scenarios involving asynchronous operations, event handling, and more. However, as JavaScript codebases grow, using callbacks alone can lead to callback hell (also known as the "pyramid of doom"), where code becomes hard to read and maintain due to deeply nested callbacks.

Jimmy Ramani

@jimmyramani

Promises

Promises are a more structured way of handling asynchronous operations in JavaScript. They provide a cleaner and more maintainable alternative to using callbacks for managing asynchronous code. Promises represent a value that might be available now, or in the future, or never.



```javascript
// A function that returns a Promise to simulate fetching data from a server
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = "Fetched data from server";
      resolve(data); // Resolve the promise with the data
    }, 2000); // Simulating a 2-second delay
  });
}

// Using the fetchData Promise
fetchData()
  .then((data) => {
    console.log(data); // Output: "Fetched data from server" after 2 seconds
  })
  .catch((error) => {
    console.error(error);
  });
```

More about Promises :

https://www.linkedin.com/feed/update/urn:li:activity:7102310801448853505?utm_source=share&utm_medium=member_desktop

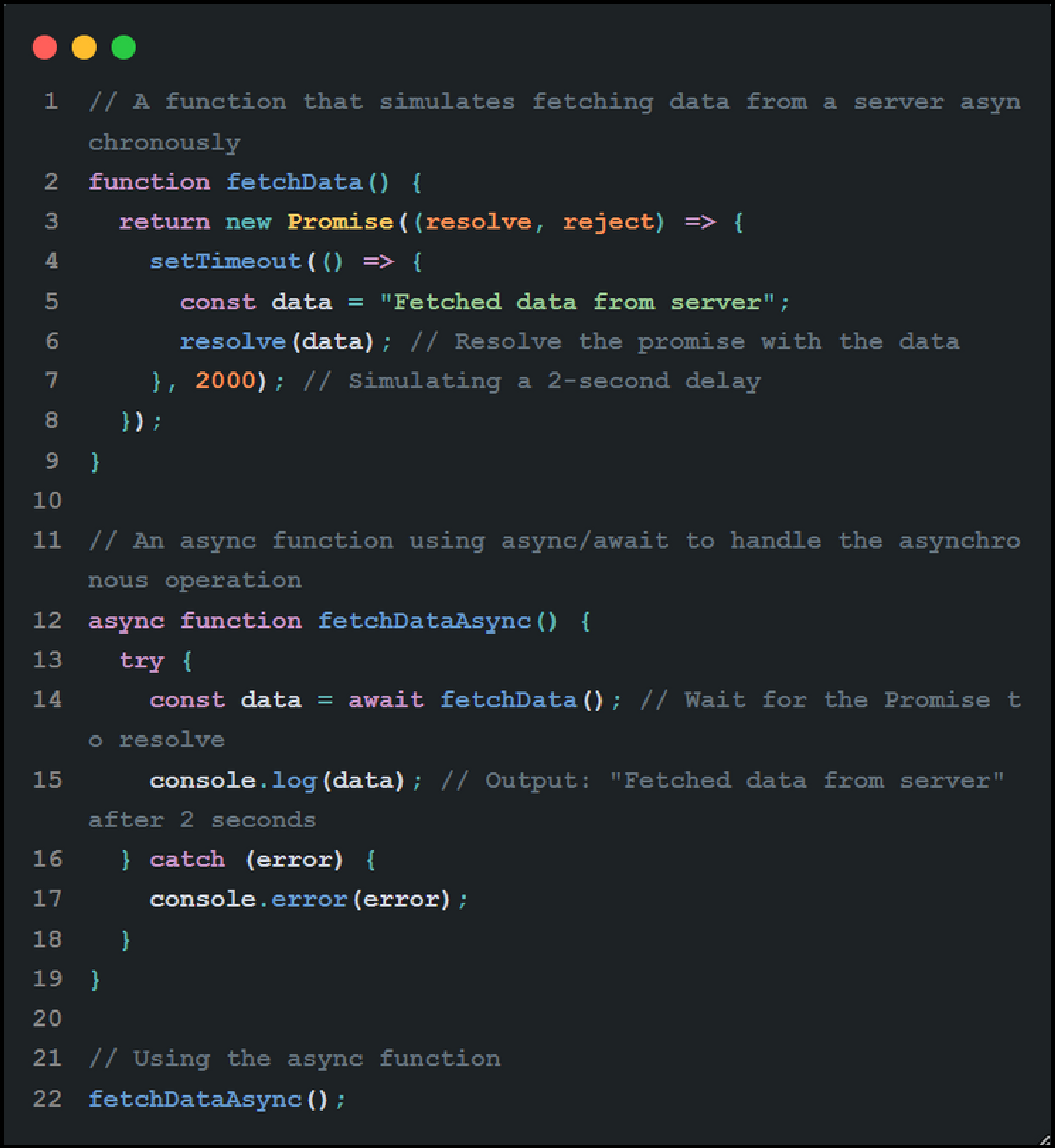
Go Through QR

Jimmy Ramani

@jimmyramani

Async - Await

async/await is a modern JavaScript feature that provides a cleaner and more concise way to work with asynchronous code compared to using callbacks or Promises directly. It makes asynchronous code look more like synchronous code, improving readability and maintainability.



```javascript
// A function that simulates fetching data from a server asynchronously
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = "Fetched data from server";
      resolve(data); // Resolve the promise with the data
    }, 2000); // Simulating a 2-second delay
  });
}

// An async function using async/await to handle the asynchronous operation
async function fetchDataAsync() {
  try {
    const data = await fetchData(); // Wait for the Promise to resolve
    console.log(data); // Output: "Fetched data from server" after 2 seconds
  } catch (error) {
    console.error(error);
  }
}

// Using the async function
fetchDataAsync();
```

More about Promises :

https://www.linkedin.com/feed/update/urn:li:activity:7102726137457344512?utm_source=share&utm_medium=member_desktop

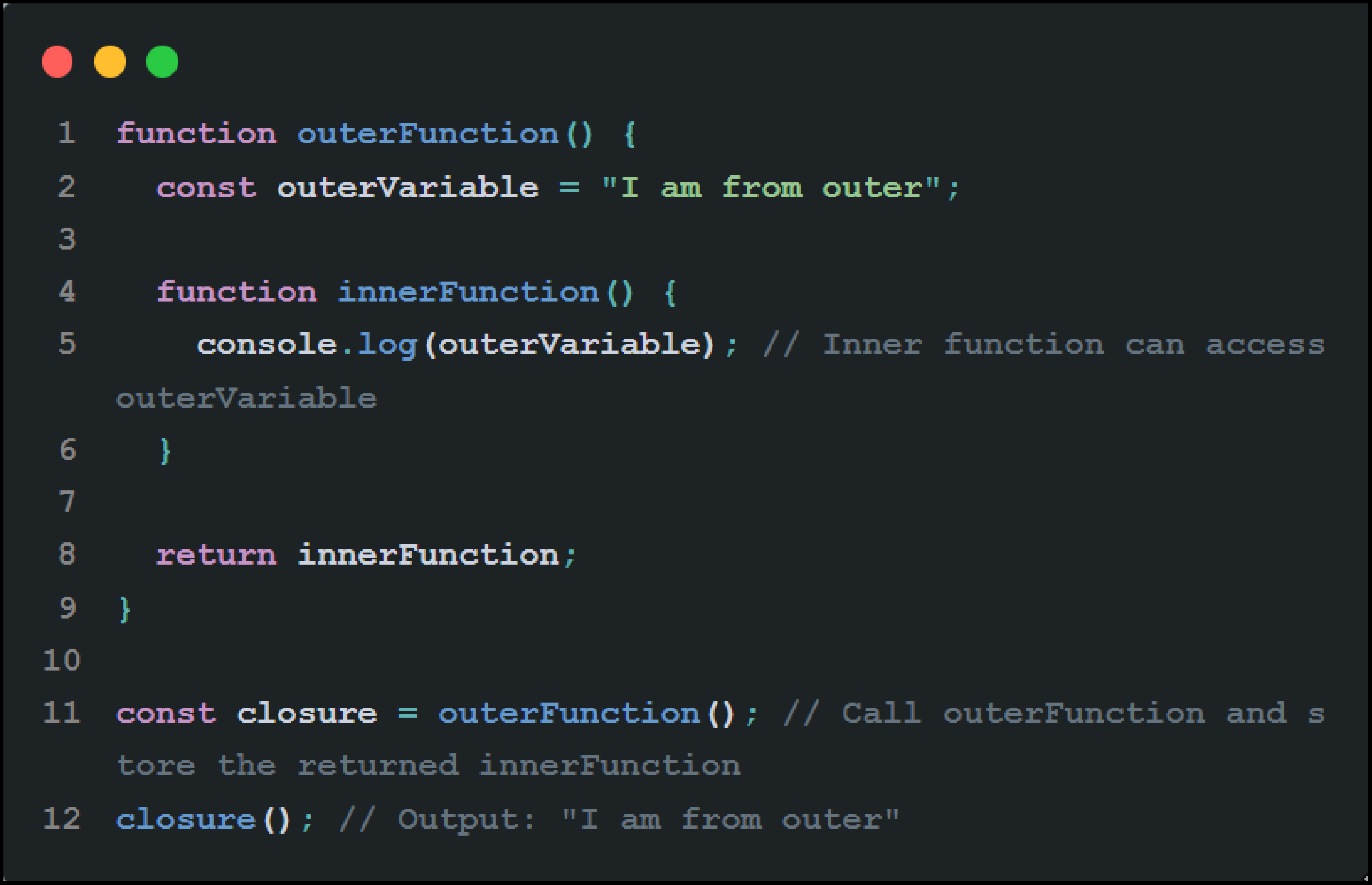
Go Through QR

Jimmy Ramani

@jimmyramani

Clousure

A **closure** in JavaScript is a function that "remembers" its lexical scope even when it's executed outside that scope. In simpler terms, a closure allows a function to access variables from its outer (enclosing) function even after that outer function has finished executing.



```javascript
function outerFunction() {
  const outerVariable = "I am from outer";

  function innerFunction() {
    console.log(outerVariable); // Inner function can access outerVariable
  }

  return innerFunction;
}

const closure = outerFunction(); // Call outerFunction and store the returned innerFunction
closure(); // Output: "I am from outer"
```

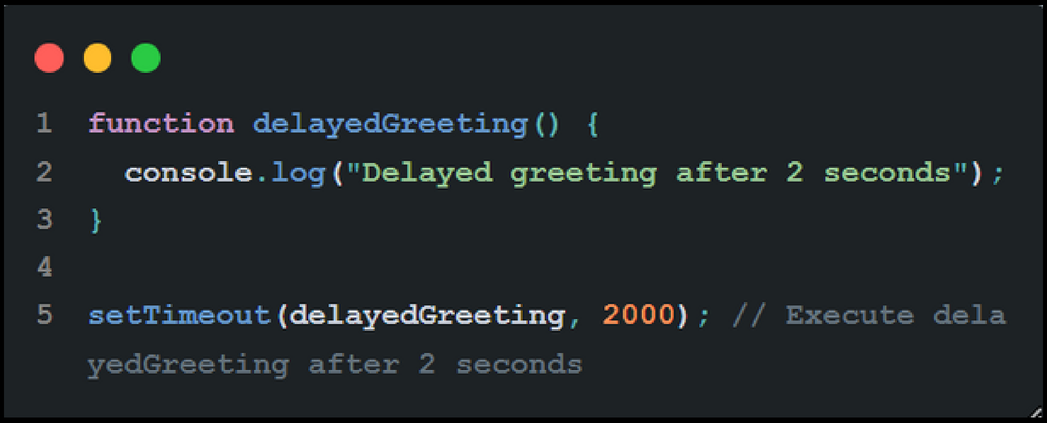
Jimmy Ramani
@jimmyramani

Timer

Timers in JavaScript are used to schedule the execution of a function or a piece of code after a certain amount of time has passed. There are three main timer functions available in JavaScript: setTimeout, setInterval, and clearTimeout.

✓ setTimeout :

The setTimeout function is used to execute a function or code block after a specified delay (in milliseconds).

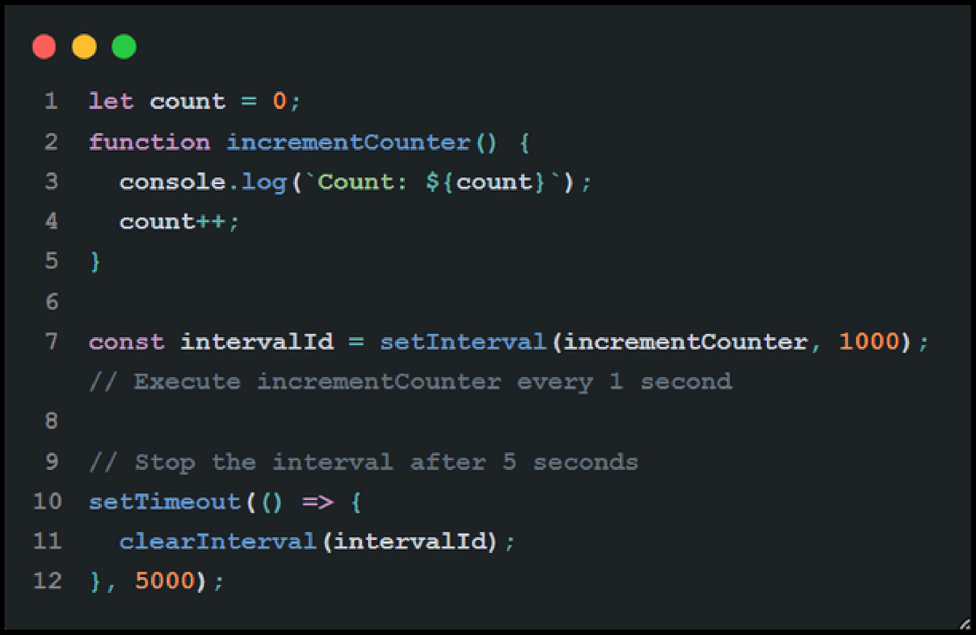


```javascript
function delayedGreeting() {
  console.log("Delayed greeting after 2 seconds");
}

setTimeout(delayedGreeting, 2000); // Execute delayedGreeting after 2 seconds
```

✓ setInterval & ClearTimeout :

The setInterval function repeatedly executes a function or code block at a specified interval

The clearTimeout function is used to cancel a timeout that was previously set using setTimeout.



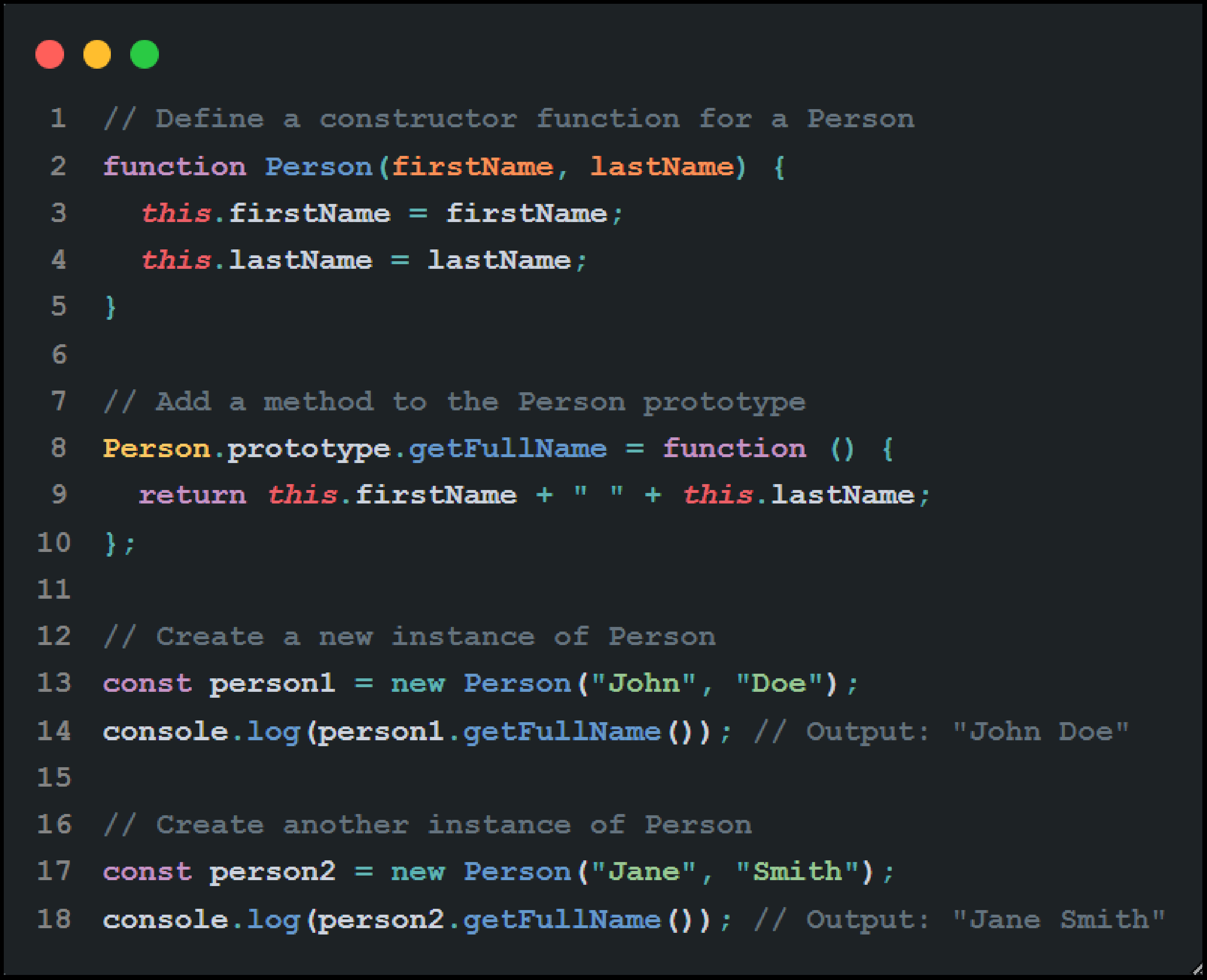
```javascript
let count = 0;
function incrementCounter() {
  console.log(`Count: ${count}`);
  count++;
}

const intervalId = setInterval(incrementCounter, 1000); // Execute incrementCounter every 1 second

// Stop the interval after 5 seconds
setTimeout(() => {
  clearInterval(intervalId);
}, 5000);
```

Jimmy Ramani

@jimmyramani

Proto-Typing

prototyping is a mechanism that allows objects to inherit properties and methods from other objects. Objects in JavaScript have a prototype chain that defines their inheritance hierarchy. Every object is linked to a prototype object, and this chain continues until the base object, which has null as its prototype.



```javascript
// Define a constructor function for a Person
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

// Add a method to the Person prototype
Person.prototype.getFullName = function () {
  return this.firstName + " " + this.lastName;
};

// Create a new instance of Person
const person1 = new Person("John", "Doe");
console.log(person1.getFullName()); // Output: "John Doe"

// Create another instance of Person
const person2 = new Person("Jane", "Smith");
console.log(person2.getFullName()); // Output: "Jane Smith"
```

Jimmy Ramani

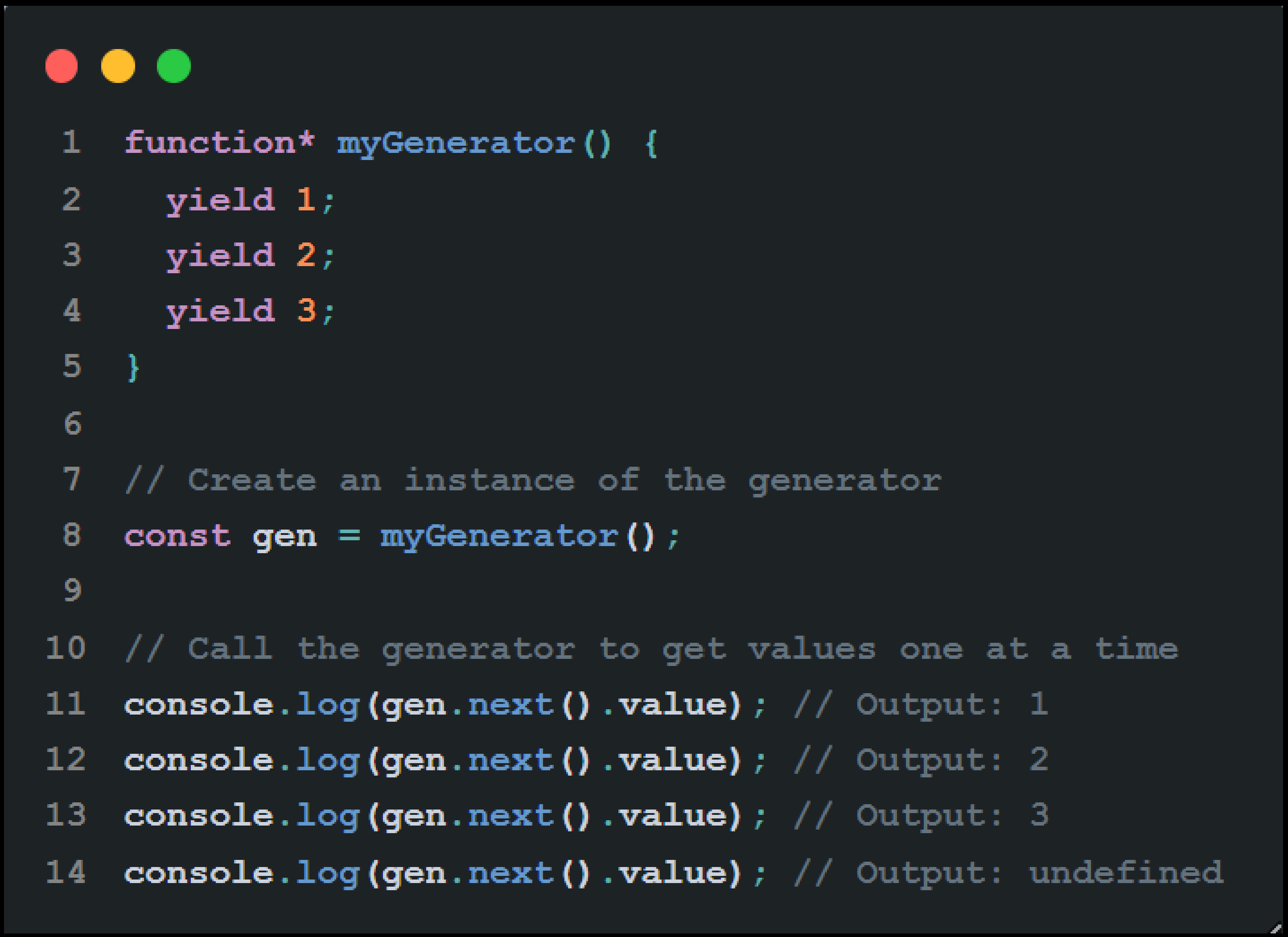
@jimmyramani

When you access a property or method on an object, JavaScript checks if that property or method is directly available on the object. If not, it looks up the prototype chain to find it on the prototype object. If it's not found on the prototype, it continues up the chain until it reaches the base object, where the prototype is null.

This prototypical inheritance allows for efficient sharing of common properties and methods among objects, and it's a fundamental part of JavaScript's object-oriented programming paradigm. However, modern JavaScript also provides classes and extends for a more class-like syntax for creating and inheriting from objects, making prototyping less common in newer codebases.

Generators

Generators are a unique feature introduced in ECMAScript 6 (ES6) that allow you to pause and resume the execution of a function. They are created using a special kind of function called a generator function. Generator functions are defined using an asterisk (*) after the function keyword, and they contain one or more yield expressions that indicate where the function can be paused and resumed.



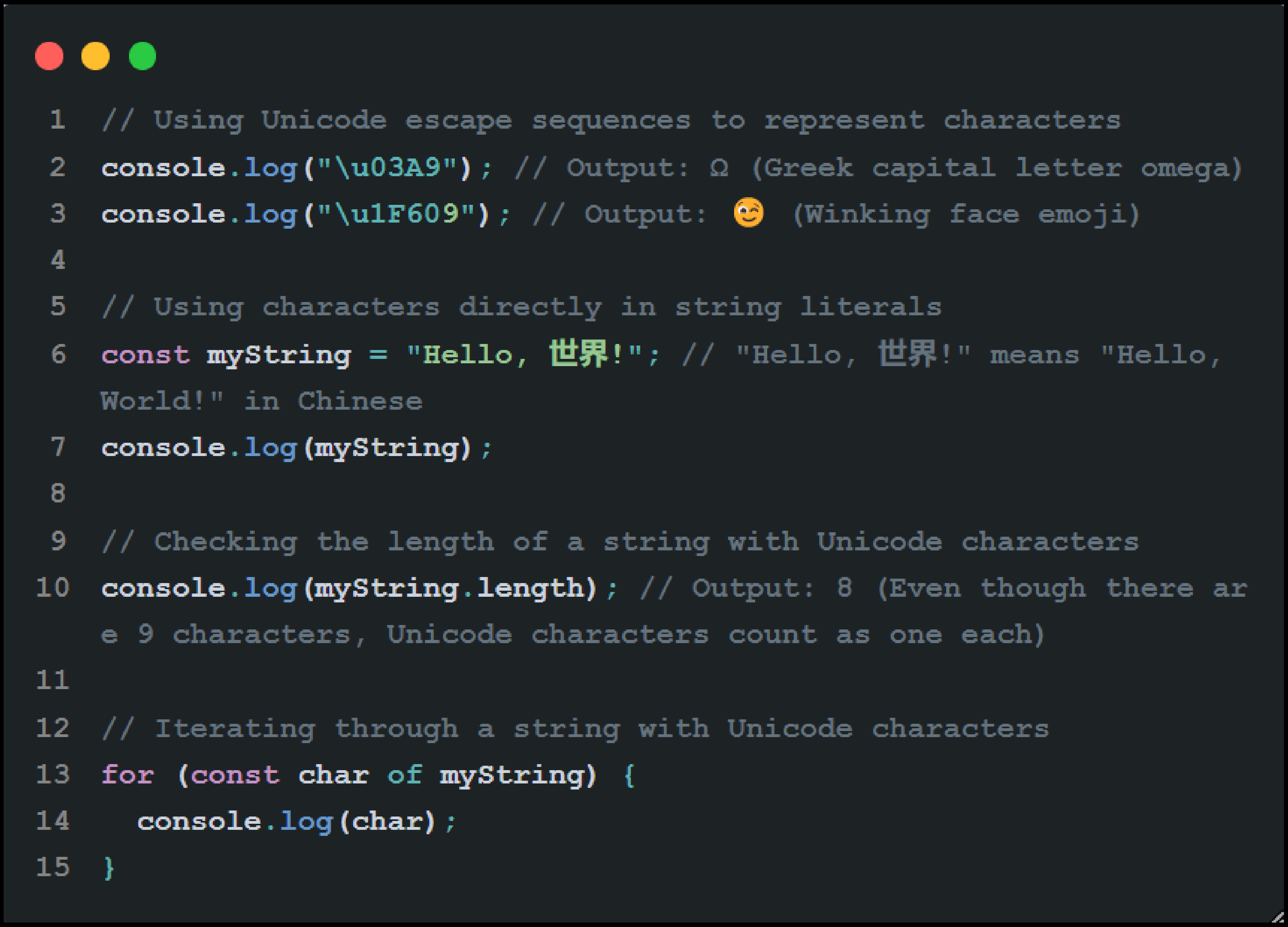
```
function* myGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

// Create an instance of the generator
const gen = myGenerator();

// Call the generator to get values one at a time
console.log(gen.next().value); // Output: 1
console.log(gen.next().value); // Output: 2
console.log(gen.next().value); // Output: 3
console.log(gen.next().value); // Output: undefined
```

Try Generators for
An infinite sequence of Fibonacci numbers

Jimmy Ramani
@jimmyramani

Unicodes

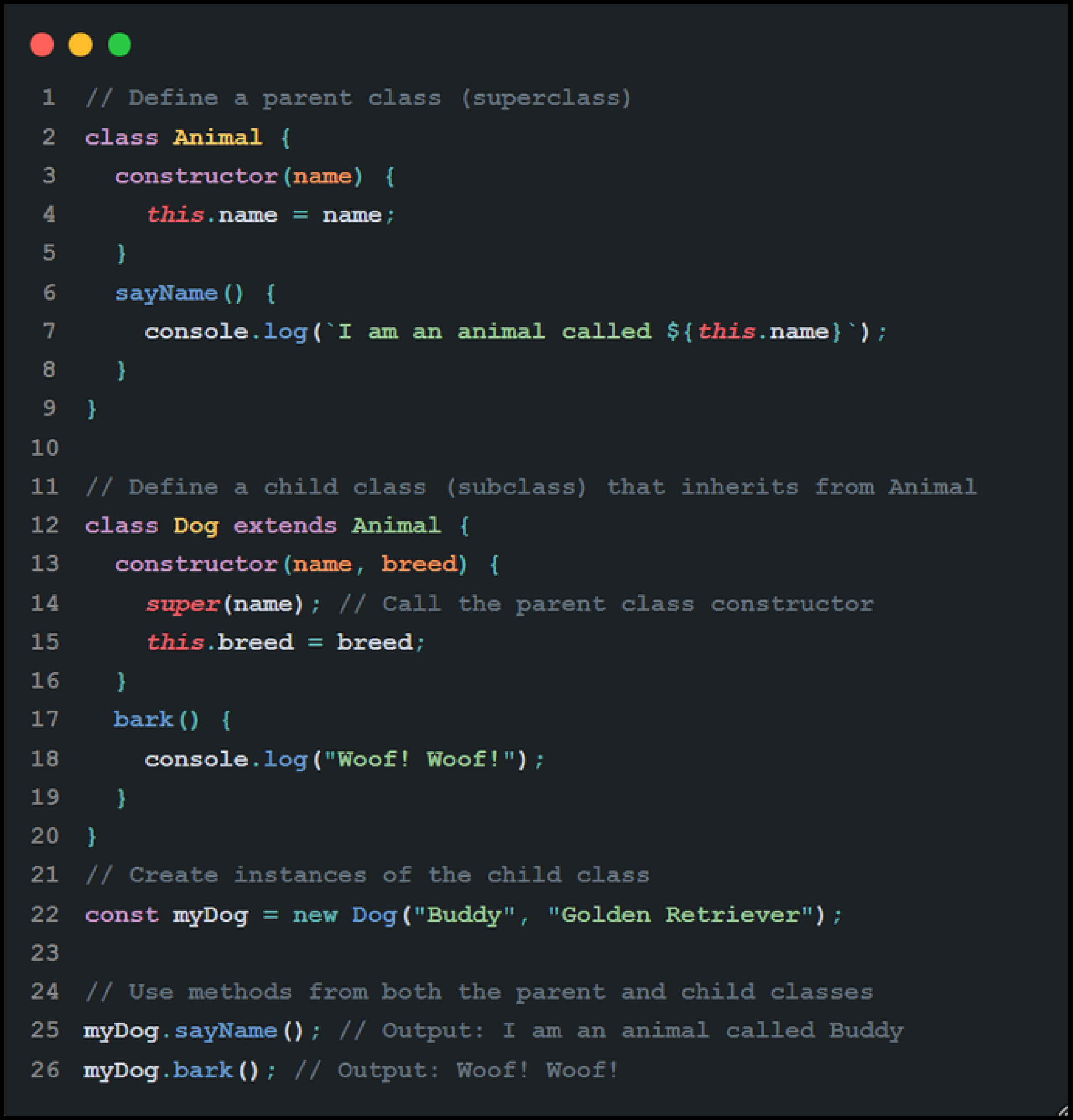
Unicode is a character encoding standard that aims to provide a unique code point (an integer) for every character, symbol, and emoji used in human writing systems. JavaScript fully supports Unicode, allowing you to work with a wide range of characters in your applications.



```
// Using Unicode escape sequences to represent characters
console.log("\u03A9"); // Output: Ω (Greek capital letter omega)
console.log("\u1F609"); // Output: 😉 (Winking face emoji)

// Using characters directly in string literals
const myString = "Hello, 世界!"; // "Hello, 世界!" means "Hello, World!" in Chinese
console.log(myString);

// Checking the length of a string with Unicode characters
console.log(myString.length); // Output: 8 (Even though there are 9 characters, Unicode characters count as one each)

// Iterating through a string with Unicode characters
for (const char of myString) {
  console.log(char);
}
```

Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows you to create new classes (subclasses or child classes) based on existing classes (superclasses or parent classes). In JavaScript, inheritance is implemented using prototypes and constructor functions.



```javascript
// Define a parent class (superclass)
class Animal {
  constructor(name) {
    this.name = name;
  }
  sayName() {
    console.log(`I am an animal called ${this.name}`);
  }
}

// Define a child class (subclass) that inherits from Animal
class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Call the parent class constructor
    this.breed = breed;
  }
  bark() {
    console.log("Woof! Woof!");
  }
}
// Create instances of the child class
const myDog = new Dog("Buddy", "Golden Retriever");

// Use methods from both the parent and child classes
myDog.sayName(); // Output: I am an animal called Buddy
myDog.bark(); // Output: Woof! Woof!
```

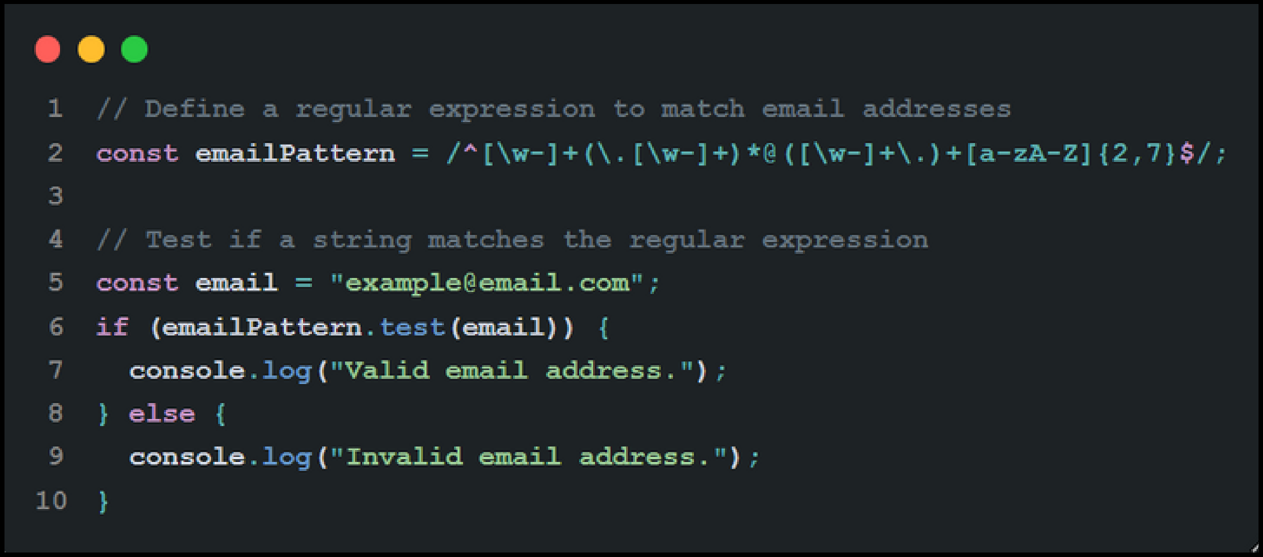
Go Through
QR

For other OOP concepts
like
Abstraction ,
Encapsulation ...

Jimmy Ramani
@jimmyramani

Regular Expressions

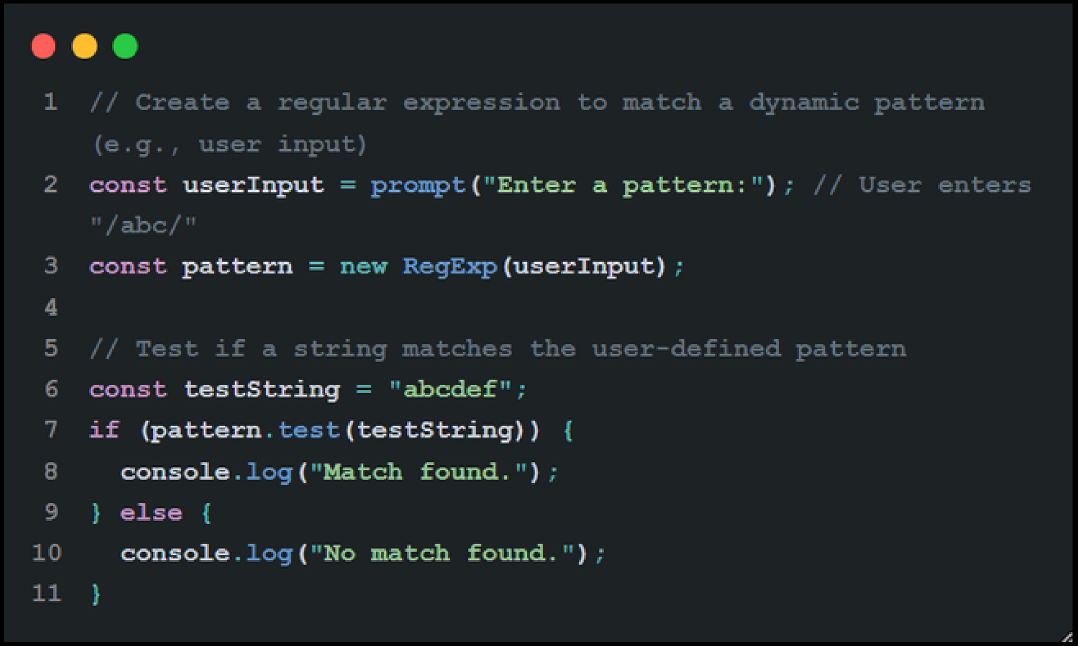
Regular Expressions, often referred to as RegEx or RegExp, are powerful patterns used for matching character combinations in strings. JavaScript provides built-in support for regular expressions through the RegExp object and regular expression literals.



```
// Define a regular expression to match email addresses
const emailPattern = /^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$/;

// Test if a string matches the regular expression
const email = "example@email.com";
if (emailPattern.test(email)) {
  console.log("Valid email address.");
} else {
  console.log("Invalid email address.");
}
```

Using Regular Expression Literals

Using the RegExp Object:



```
// Create a regular expression to match a dynamic pattern (e.g., user input)
const userInput = prompt("Enter a pattern:"); // User enters "/abc/"
const pattern = new RegExp(userInput);

// Test if a string matches the user-defined pattern
const testString = "abcdef";
if (pattern.test(testString)) {
  console.log("Match found.");
} else {
  console.log("No match found.");
}
```

Projects

- ✓ Form Validation
- ✓ Carousel
- ✓ Accoridan
- ✓ Pagination
- ✓ Modal & Overlay
- ✓ Tabs Component
- ✓ Mobile Navbar
- ✓ Tabs Component

Expense Tracker App

Quiz App

Try With Your Self

Scan & find all Projects Here

WAS THIS HELPFUL?

Share with a friend who needs it!

Jimmy Ramani

@jimmyramani