JAVASCRIPT INTERVIEW QUESTIONS & ANSWERS

Important JavaScript questions and answers for technical interview

DEVELOPER UPDATES

- **What is JavaScript?**

  JavaScript is a programming language that enables you to create dynamic web pages. It is often used to create interactivity and animation on web pages, as well as to add functionality to websites.

  JavaScript is used in a wide range of web development tasks, including front-end development, back-end development, game development, and mobile app development.

- **What is the difference between let, const, and var?**

  In JavaScript, there are three different types of variables: let const and var.

  let is a type of variable that allows you to declare a variable within a block of code. This means that the variable will be local to the block and will not be shared with any other blocks.

  const is like a let but that can't change its value once it's been set.

  var allows you to declare a global variable that will be accessible from anywhere in the code.

```
var varVariable = 10;

testVariable();

function testVariable() {
    let letVariable = 20;

    console.log('letVariable: ' + letVariable); //output: letVariable: 20
    console.log('varVariable: ' + varVariable); //output: varVariable: 10
}

console.log('letVariable: ' + letVariable); //output: ReferenceError: letVariable is not defined

const name = 'John';

name = 'Jane'; //output: TypeError: Assignment to constant variable.
```

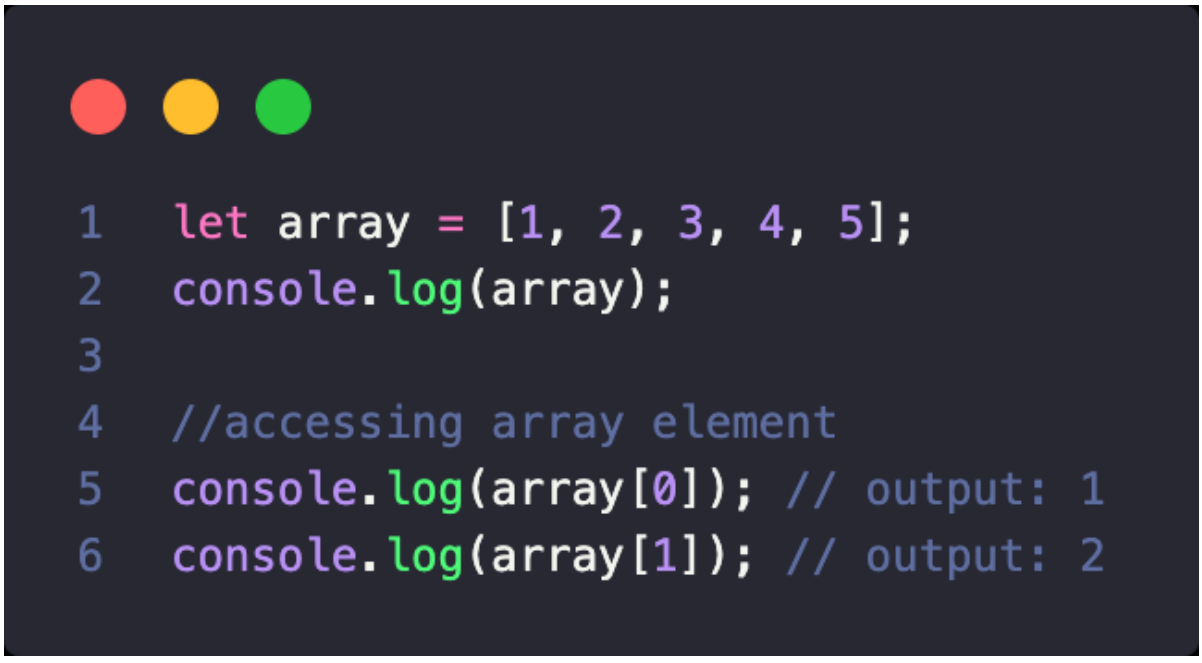
- **What are the different types of data types in JavaScript?**
  - Number
  - String
  - Object
  - Undefined
  - Boolean
  - Null

- **Is JavaScript a case-sensitive language?**

  Yes

- **What is an Array? and How to create an array in JavaScript?**

  An array is a type of data structure that stores multiple elements of the same type. In JavaScript, arrays are created using the square bracket notation [].



```
let array = [1, 2, 3, 4, 5];
console.log(array);

//accessing array element
console.log(array[0]); // output: 1
console.log(array[1]); // output: 2
```

- **What are the different methods of Array available in JavaScript?**
  - push()
  - pop()
  - join()
  - length
  - concat()
  - at()
  - filter()
  - find()
  - map()
  - reduce()
  - includes()
  - indexOf()
  - isArray()
  - slice()
  - values()

- **What is a function? And How to declare a function in JavaScript?**

  Functions are one of the most important features of JavaScript. A function is a piece of code that can take one or more parameters and return a result.

  Functions can be used to perform specific tasks or to make your code cleaner and easier to read. With function, we can reuse the same code in our project.

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

console.log(add(1, 2)); // output: 3
```

- **What is the difference between function and method?**

  The major difference between methods and functions in JavaScript is that methods are called on objects while functions are not.

  Methods are called on objects with a dot operator while functions are called using parentheses.

  Method accepts arguments through an object parameter, and function accepts arguments through a parenthesis parameter.

- **What is the use of void(0) in JavaScript?**

  void(0) is a special keyword in JavaScript that's used to declare a function that doesn't return any value. When you use void(0), it tells the interpreter that this function should not return anything.

  When you don't want a link to navigate to another page or reload a page. You can use javascript: void(0) to run code that doesn't change the current page.

- **What is an Object in JavaScript?**

  An object is a data structure that stores various information. An object can contain properties and methods.

  Properties are simple variables that store data, while methods are functions that you can call.

- **List out the different Object built-in methods.**
  - assign()
  - entries()
  - values()
  - keys()
  - is()
  - toString()
  - valueOf()
  - hasOwn()
- **What is the difference between localStorage and sessionStorage()?**

  localStorage is a storage mechanism that is saved in the browser's local storage. It stores data for the current website and can be accessed later and multiple sessions.

  sessionStorage is a different storage mechanism that is saved in the browser's session storage. It stores data only for the current website session, which means that it won't be accessible after closing the browser or the next session.
- **What is DOM?**

  DOM stands for Document Object Model, which is a core API for JavaScript. It enables you to access and manipulate the elements of a document. For example, you can change the text in an element, add new elements, or remove elements.
- **What is the difference between Cookies and Local Storage?**

  Cookies are small pieces of information that are sent by a website to the browser of a visitor. They're used to store information about the visitor's preferences so that the website can remember things like which language is selected or how long they've been viewing the page.

LocalStorage is a feature of JavaScript that allows you to save data on the client side. This means that the data is kept in local storage on the user's device, instead of on a server.

Cookies can be accessed on the server and localStorage data is not accessed on the server.

- **Why “use strict” is used in JavaScript?**

'Use strict' is used in JavaScript to make sure that your code is written correctly. Strict mode enforces the following rules

  - Variables must be declared before they are used,
  - All comparisons must be done between literal values
  - No function calls can be made until the function has been defined.

This helps to catch errors early on in the development process and keeps your code more organised.

- **What is the difference between null and undefined?**

The difference between null and undefined is that null represents a nonexistent value, while undefined represents a value that hasn't been defined yet.

- **What are promises?**

In JavaScript, promises are used to handle asynchronous operations. They are objects that represent the successful or unsuccessful completion of an asynchronous operation, allowing you to write more concise and readable code when working with asynchronous APIs.

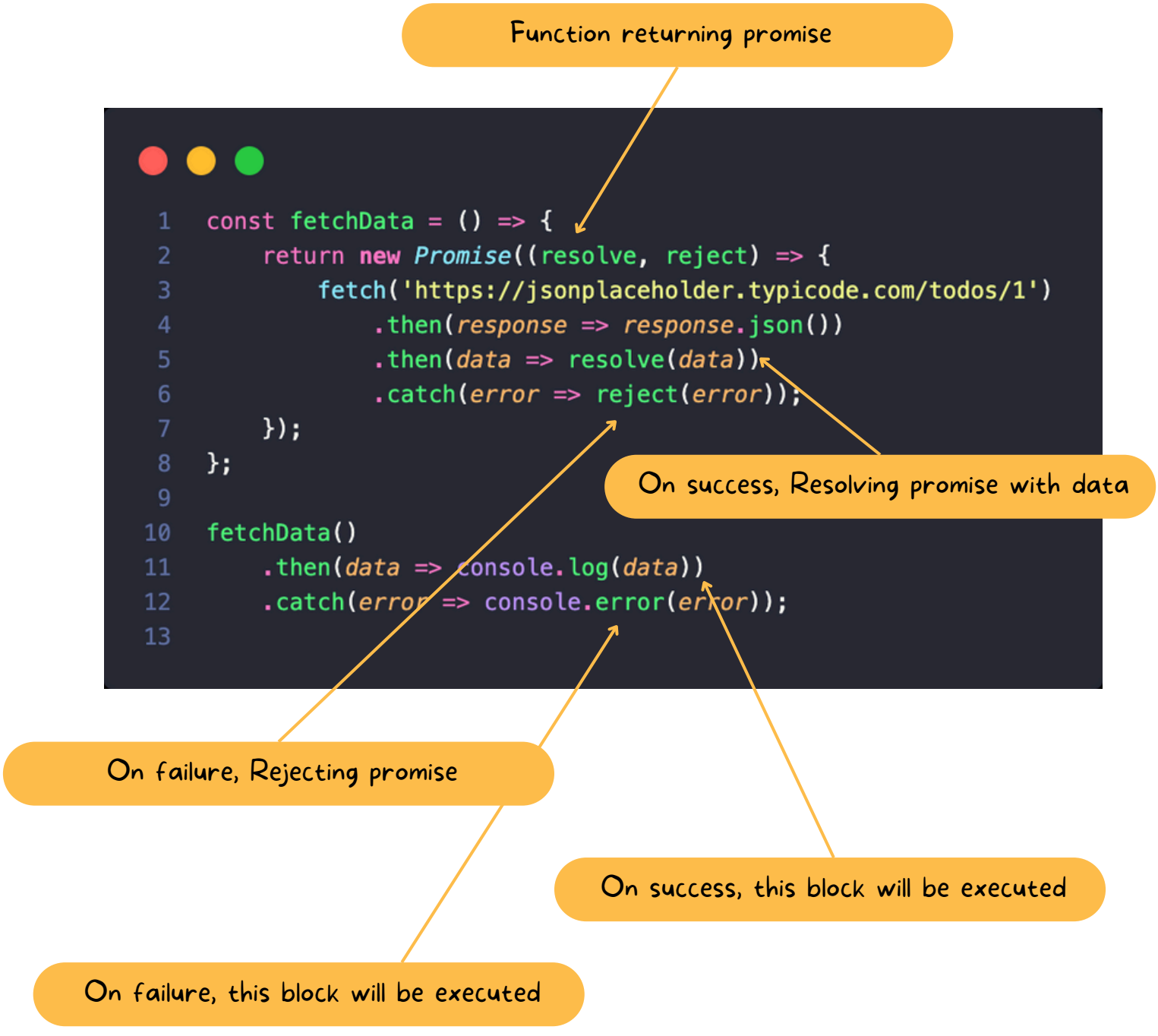
A Promise has three states: pending, fulfilled, or rejected. A Promise is said to be settled when it is fulfilled or rejected, and its value cannot be changed.

Promises simplify code execution, avoid callback hell, and improve readability.

- **Write a code to demonstrate promises.**

  Let's assume, if you're creating a web application that sends data requests to a server, you must handle those requests asynchronously to keep the application responsive to user interactions.

  Promises are an important part of this type of task because they let you handle operations that happen at different times in a clean and effective way.




```
const fetchData = () => {
    return new Promise((resolve, reject) => {
        fetch('https://jsonplaceholder.typicode.com/todos/1')
            .then(response => response.json())
            .then(data => resolve(data))
            .catch(error => reject(error));
    });
};

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

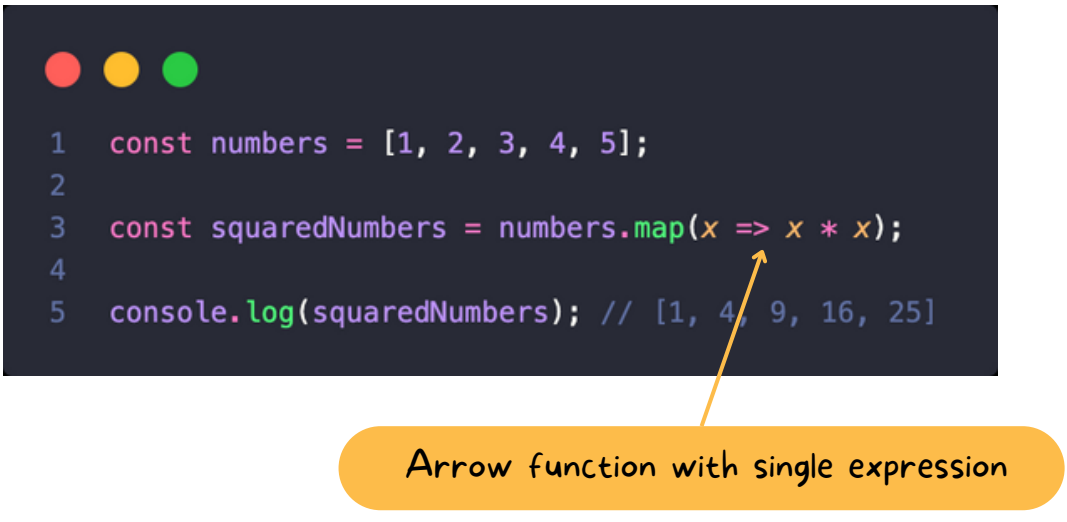
- **What is an arrow function? Give an example.**

  With arrow functions, you can write functions in a shorter way, which makes code easier to read and write. They also have some advantages over traditional functions, like being able to look at the lexical scope (Accessing variables and functions defined in the outer function or global scope).

  Arrow functions implicitly bind the this keyword to the enclosing scope, so you don't need to use bind(), call(), or apply() to bind it to the current object.

  For example, arrow functions can help you write code that is easier to read and understand when you need to write several small functions for event listeners.

  Here is the example of arrow function



- **What is NaN Property?**

  The NaN (not a number) property is used to represent a value that does not exist in the number system. The NaN property can be used to check if a number is valid or not.

```
console.log(isNaN(undefined)); // Output: true
console.log(isNaN(123)); // Output: false
console.log(isNaN("String")); // Output: true
console.log(NaN === NaN); // Output: false
console.log(isNaN(NaN)); // Output: true
```

- **What is the difference between “==” and “===”?**

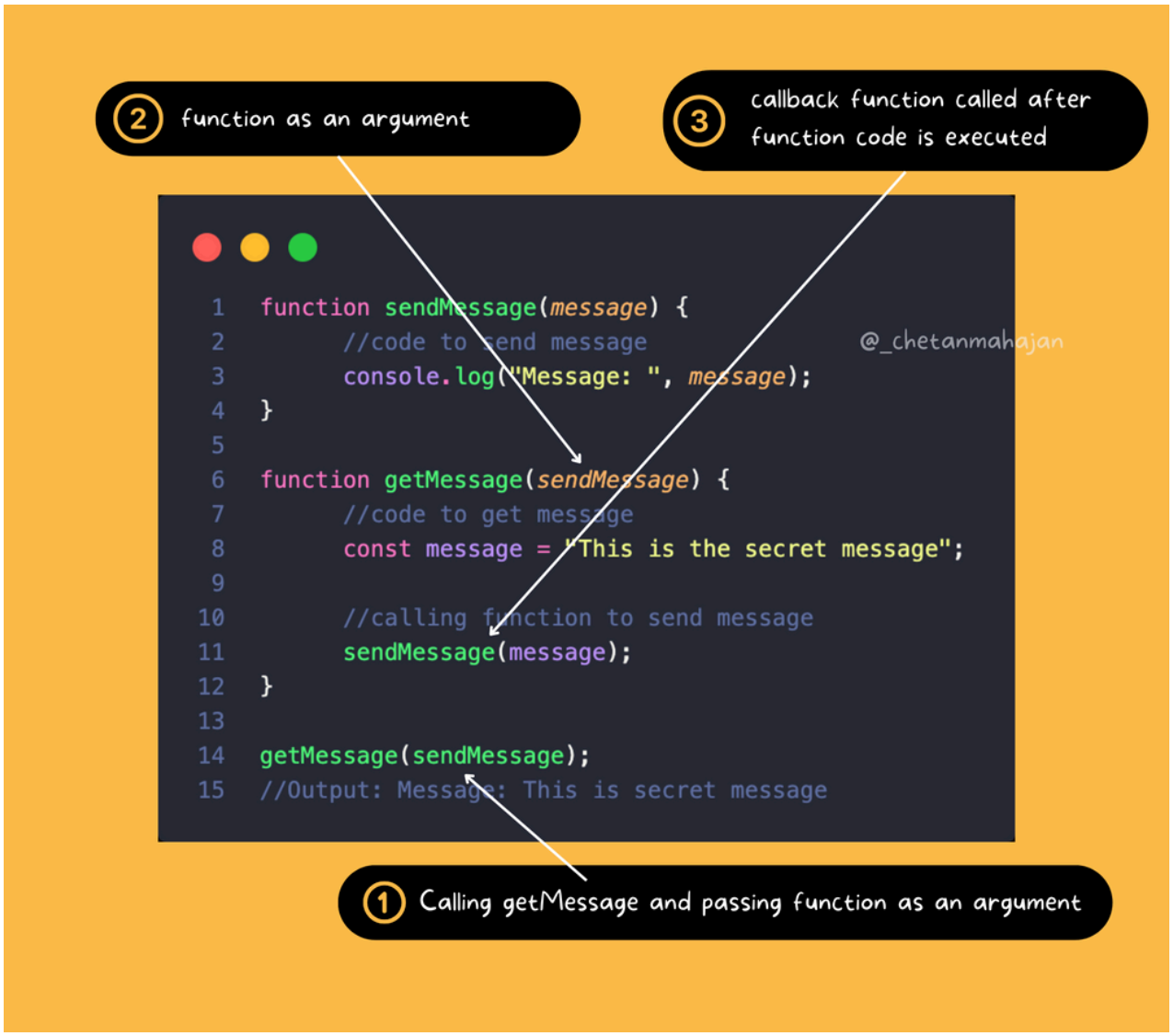
  “==” is used to check the equality of the variable and “===” is used to check equality as well as data types of the variable.

```
console.log(1 == "1"); // Output: true
console.log(1 === "1"); // Output: false
console.log(1 === 1); // Output: true
```

- **What is a callback function in JavaScript?**

  A callback function is a function that is passed as an argument to another function and it is executed after some operation/event is executed.

- **How to define a callback function?**




```
function sendMessage(message) {
      //code to send message
      console.log("Message: ", message);
}

function getMessage(sendMessage) {
      //code to get message
      const message = "This is the secret message";

      //calling function to send message
      sendMessage(message);
}

getMessage(sendMessage);
//Output: Message: This is secret message
```

In above example, we passed sendMessage as an argument to getMessage() and sendMessage() is executed after getting a message.

- **What is the scope of the let variable?**

  The scope of a let variable is the block in which it is defined. In other words, the let keyword will reserve a block of memory for the variable and it will only be accessible from within that block.

  If you want to access the let variable from outside of its block, you will need to use the var keyword.

- **What are closures?**

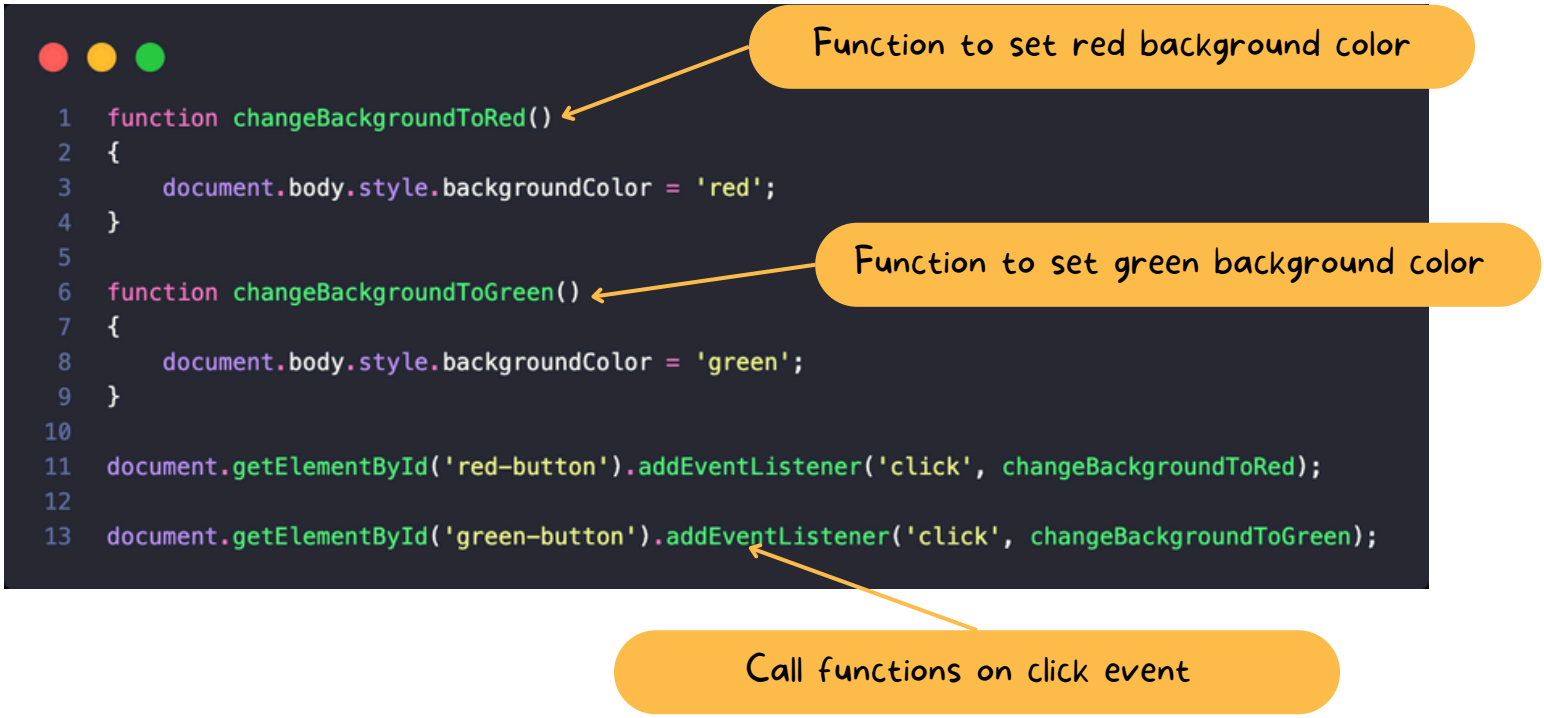
  A closure in JavaScript is a special type of function that keeps track of variables and information from the outside, or "enclosing", function even after it has finished running.

  This allows the inner function to continue to use and reference these values, even after the outer function has completed its execution.

  Closures are a useful for creating private variables and generating functions with specific attributes.

  For example, let's consider a website where a user can click on different buttons to change the background color of the page.

  Without using closures, you would need to create a separate function for each button to handle the click event and change the background color, like this:

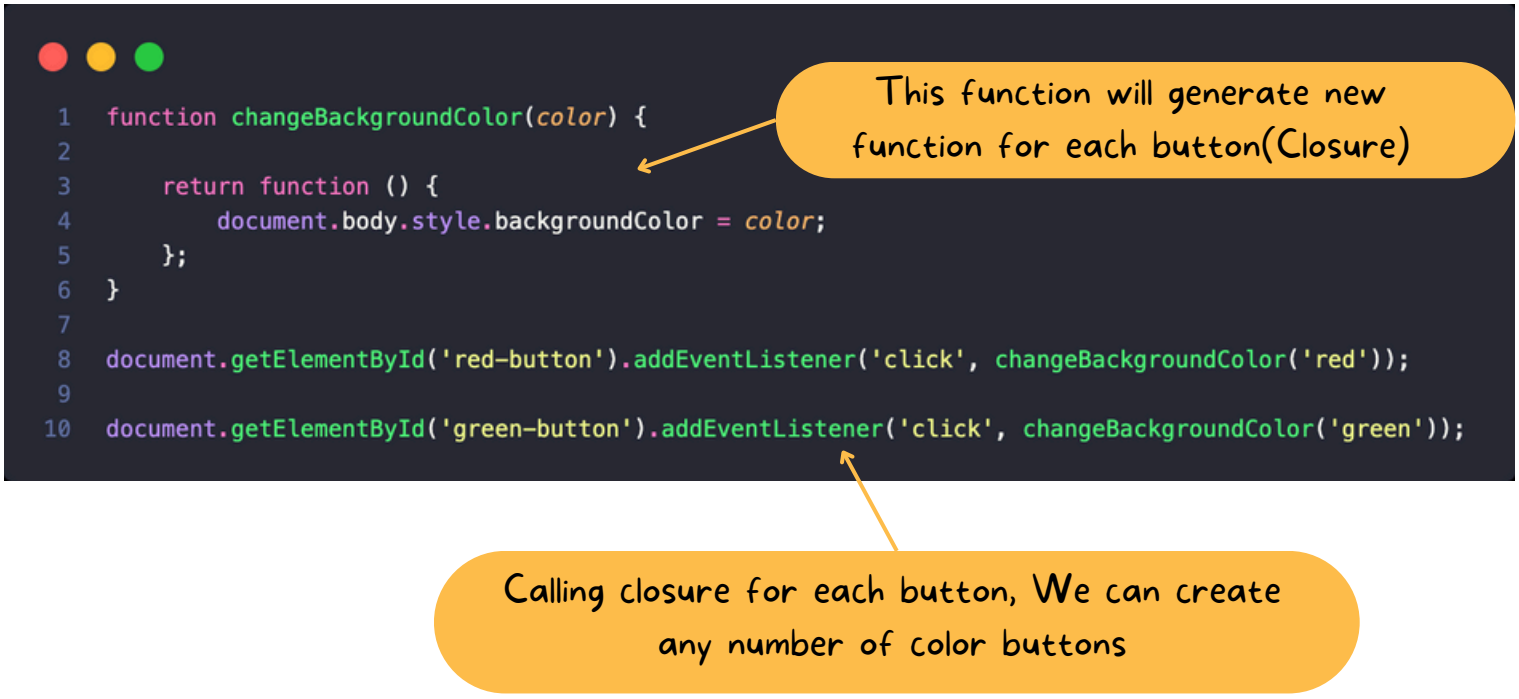



```
function changeBackgroundToRed()
{
    document.body.style.backgroundColor = 'red';
}

function changeBackgroundToGreen()
{
    document.body.style.backgroundColor = 'green';
}

document.getElementById('red-button').addEventListener('click', changeBackgroundToRed);

document.getElementById('green-button').addEventListener('click', changeBackgroundToGreen);
```




```
function changeBackgroundColor(color) {

    return function () {
        document.body.style.backgroundColor = color;
    };
}

document.getElementById('red-button').addEventListener('click', changeBackgroundColor('red'));

document.getElementById('green-button').addEventListener('click', changeBackgroundColor('green'));
```

In this example, the changeBackgroundColor function acts as a function factory, generating a new function for each button that has the desired background color "closed over" within its scope.

This way, when the generated function is executed, it has access to the correct background color value, even though the changeBackgroundColor function has already returned.

- **What is the scope?**

Scope refers to the current context of code execution. Scope determines the visibility and lifetime of variables and functions. JavaScript has two types of scope: global scope and local scope.

Global scope addresses the context in which all code is executed. Variables and functions declared in the global scope are available to all parts of your code

Local scope is the context limited to a function. Variables and functions inside a function are only visible and accessible within that function, making them invisible outside of it.

- **Write a function to set cookies using JavaScript?**

```
function setCookies(name, value, days) {
    var date = new Date();
    date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
    var expires = "expires=" + date.toUTCString();
    document.cookie = name + "=" + value + "; " + expires;
}
```

- **Write a function to read cookies values using JavaScript?**

```
function getCookie(name) {
    var nameEQ = name + "=";
    var ca = document.cookie.split(';');
    for (var i = 0; i < ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') c = c.substring(1, c.length);
        if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length, c.length);
    }
    return null;
}
```

- **How to add and remove a class to HTML Element?**

```
//add class
document.getElementById('my-div').classList.add('my-class');

//remove class
document.getElementById('my-div').classList.remove('my-class');
```

- **How to fetch data from an API?**

  There are several different methods that you can use in Javascript. The most common way is to use the fetch() method. This method accepts a URL as an input and returns the data that is located at that URL.

```
async function getData() {
    const response = await fetch('https://jsonplaceholder.typicode.com/users');
    const data = await response.json();
    console.log(data);
}

getData();
```

- **Can we use Javascript at the server-side and How?**

  Yes we can use Javascript at the server-side using Node.js.

- **Can we use JavaScript for Mobile and Desktop App Development and How?**

  Yes, We can use Javascript for Mobile Development using React-Native. And for desktop app development using the Electron.js framework.

- **List out the different types of JavaScript Frameworks and Libraries?**

1. React.js
2. Angular
3. Vue.js
4. Electron.js
5. React Native
6. Backbone.js
7. Ember.js

- **How to empty an array?**

```
//assign an empty array
let myArray = [1, 2, 3, 4, 5];
myArray = [];

//setting length property 0
let myArray = [1, 2, 3, 4, 5];
myArray.length = 0;

//splice method to remove elements
let myArray = [1, 2, 3, 4, 5];
myArray.splice(0, myArray.length);

//using loop and pop
let myArray = [1, 2, 3, 4, 5];

for(let i = myArray.length; i >= 0; i--)
{
  myArray.pop();   //pop method to remove array elements one by one
}
```

- **Explain this keyword?**

  this keyword refers to the object that is currently executing code. It can be used to access properties and methods on that object.

- **What different types of popup boxes are available in JavaScript?**

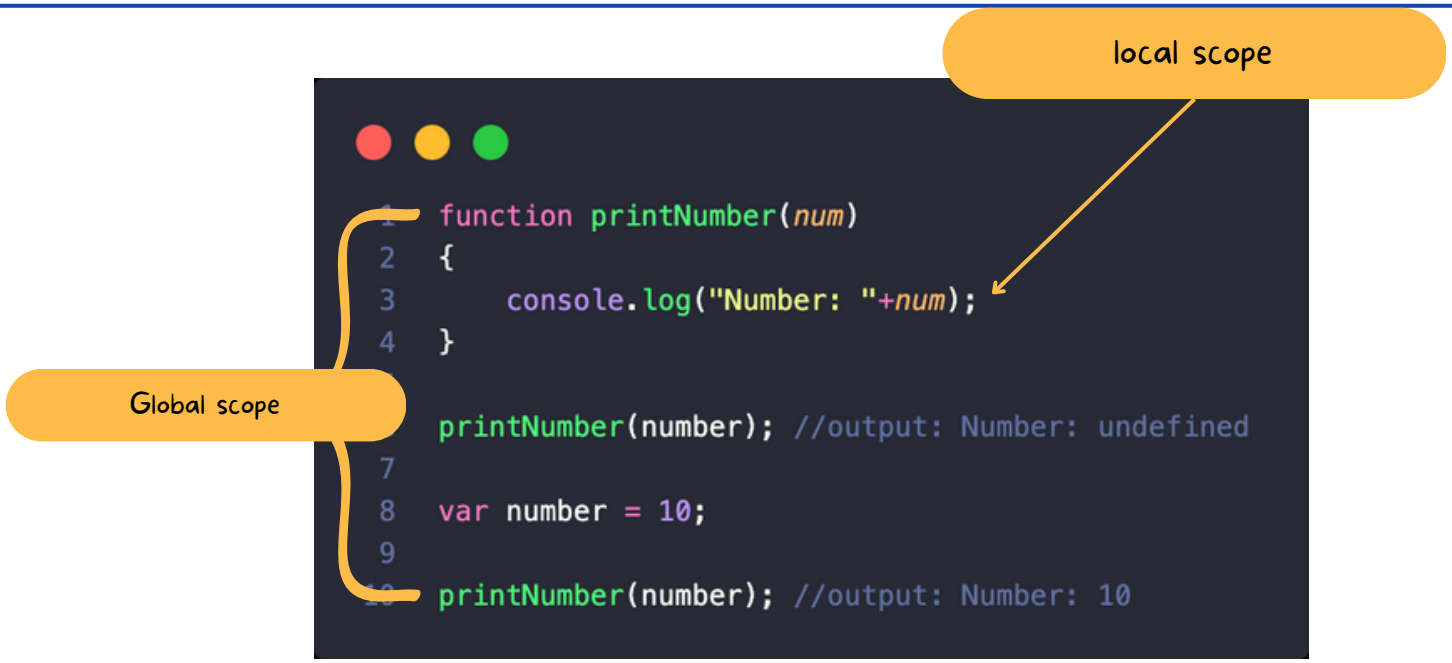
  - alert() - To show message to the user
  - prompt() - To get input from the user
  - confirm() - To get permission or confirmation from the user

- **What is the hoisting in JavaScript?**

  Hoisting simply means that all declarations are lifted to the top of the scope, which may have them declared before or after other code.

  This means that even if you declare your variable after some other code, it will be available at the top of that scope - making it accessible outside of its original location.

  For example, consider the following code:



In this code, we called printNumber function with number variable before its declaration.

However, because of hoisting, the declaration of the number variable is processed before the code is executed, so the console.log statement prints undefined rather than causing an error.

Note that hoisting only affects declarations, not assignments. This means that the value of a variable is not initialized until the code is executed, even if the declaration of the variable has been hoisted to the top of the scope.

- **What is an anonymous function?**

  Regular function has 2 parts name and a body, But in JavaScript, you can define a function without a name. You can't use this, super, or new in an anonymous function.

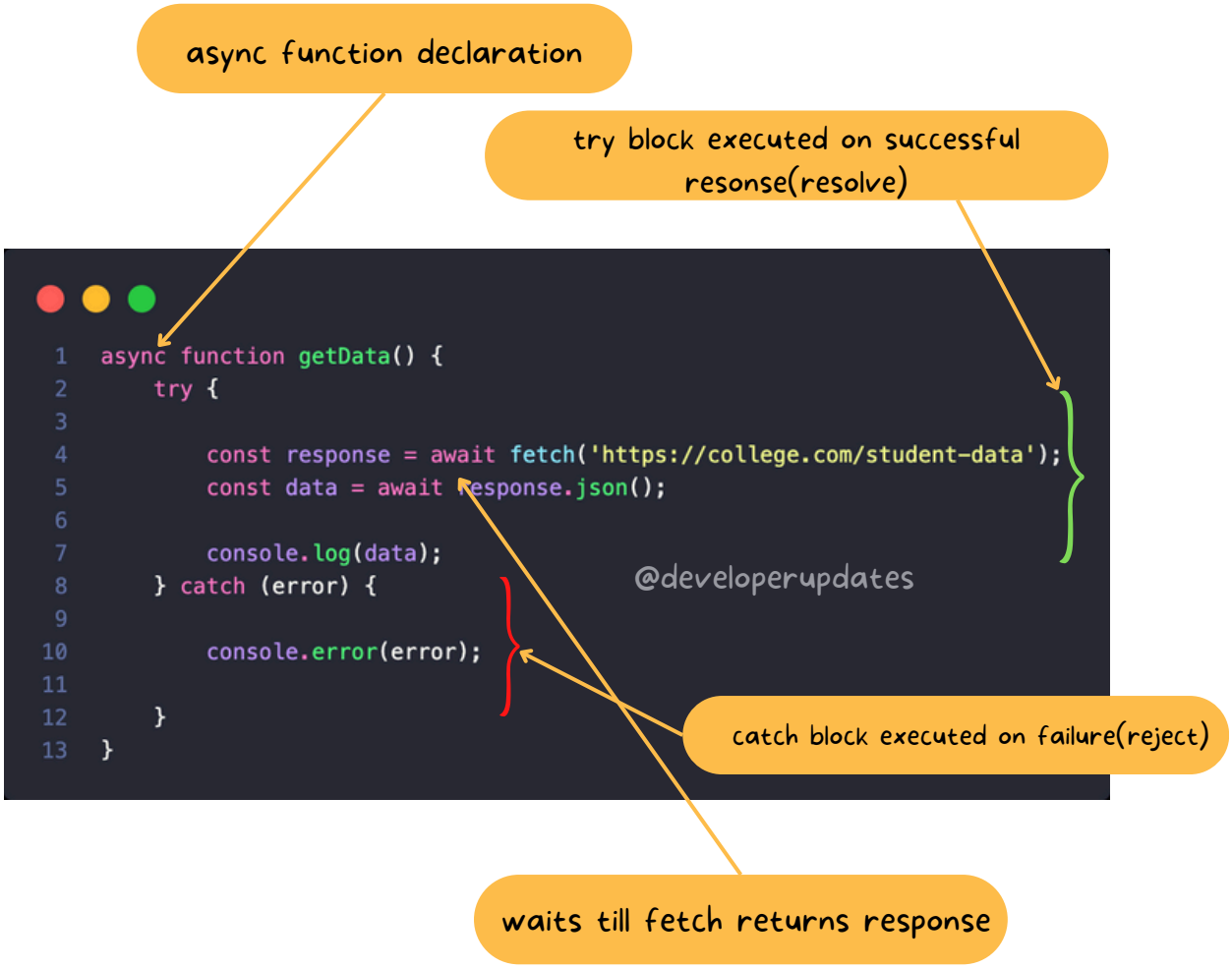
  Anonymous functions are functions that don't have a name. it can be accessed by a variable it is stored in.

- **What is async/await?**

  Async/await allows you to write asynchronous code. It is mainly used with promises.

  When a long-running task needs to be performed without blocking the main thread of execution, such as making network requests or accessing a database, at that time async/await is used.

  For example,



As you can see above function, the async keyword is used before the function, it is used to make the function return a promise, and await keyword is used to pause execution till program resolve/reject the promise.

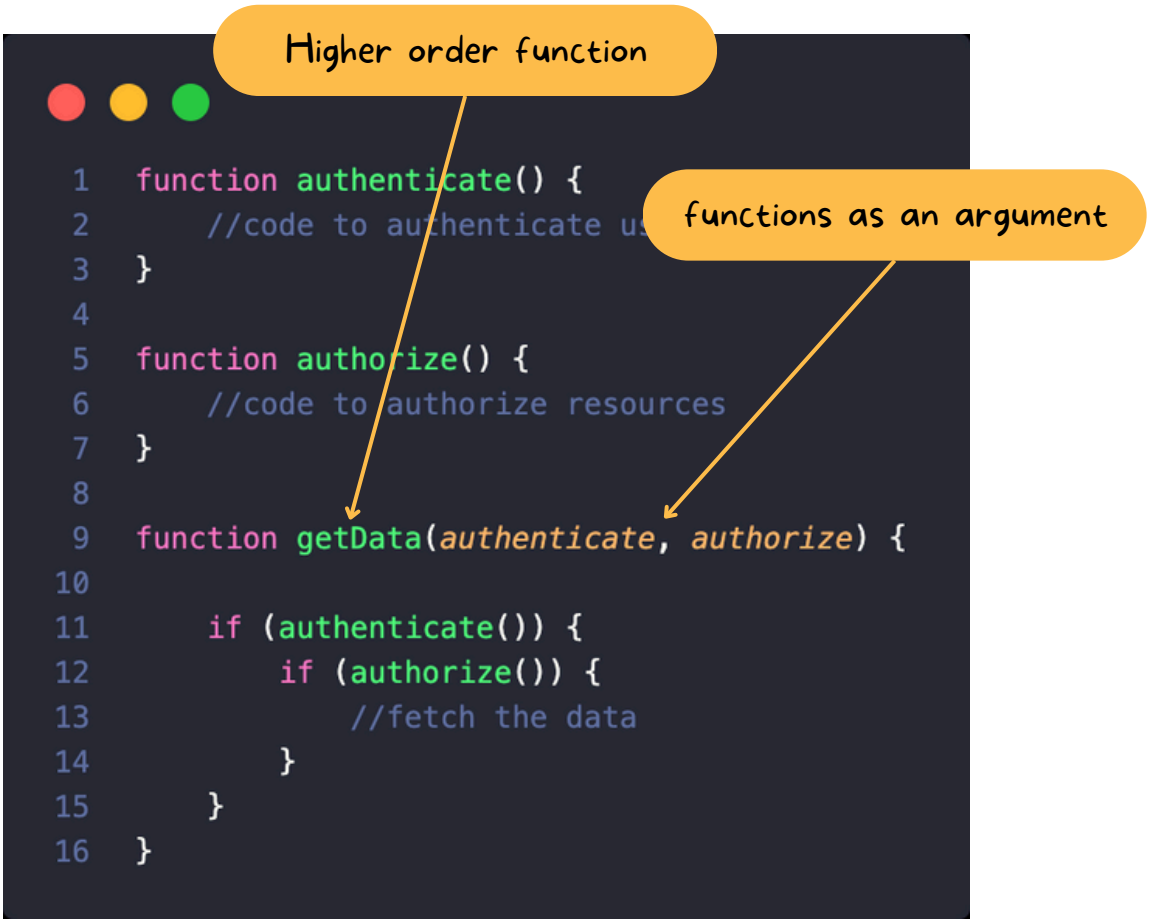
- **What are higher order functions?**

  Higher order are functions that either accept one or more functions as arguments or returns a function.

  Let's take an example,

  When a user sends a request to access data, you have to first authenticate them, then authorize them to access the data.

  So code will be as follows:



  In the above example, getData is a higher-order function as it takes two functions as arguments.

- **What is memoization? Explain it with example.**

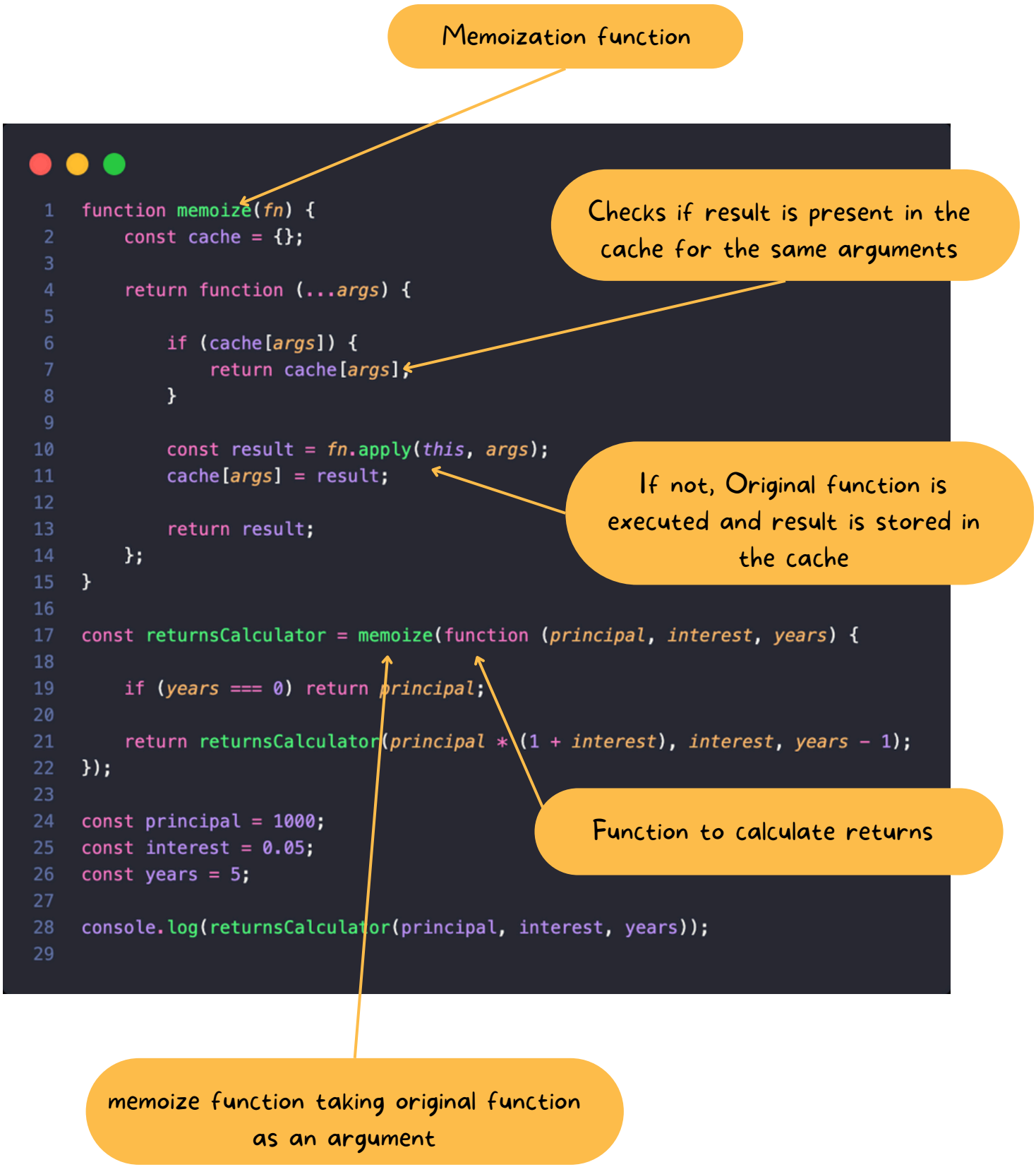
  Memoization is a technique in JavaScript where the result of a function is stored in the cache to avoid redundant computation when the function is called again with the same arguments.

  The idea is to store the results of a function in a cache, keyed by its arguments, so that the next time the function is called with the same arguments, the result can be immediately retrieved from the cache instead of being recomputed.

  Memoization is required because it optimizes the performance of slow or expensive functions by reducing the number of times they need to be executed.

  For example, in financial calculations, if the same mathematical formula is used multiple times, memoization can be used to cache the result of the formula, reducing the calculation time and increasing the overall efficiency of the application.

  Let's take an example, you are writing a program to calculate returns for your stock market application, At that time you can use memorization as follows.




```
function memoize(fn) {
    const cache = {};

    return function (...args) {

        if (cache[args]) {
            return cache[args];
        }

        const result = fn.apply(this, args);
        cache[args] = result;

        return result;
    };
}

const returnsCalculator = memoize(function (principal, interest, years) {

    if (years === 0) return principal;

    return returnsCalculator(principal * (1 + interest), interest, years - 1);
});

const principal = 1000;
const interest = 0.05;
const years = 5;

console.log(returnsCalculator(principal, interest, years));

```

- **How to add and remove object properties dynamically in Javascript?**

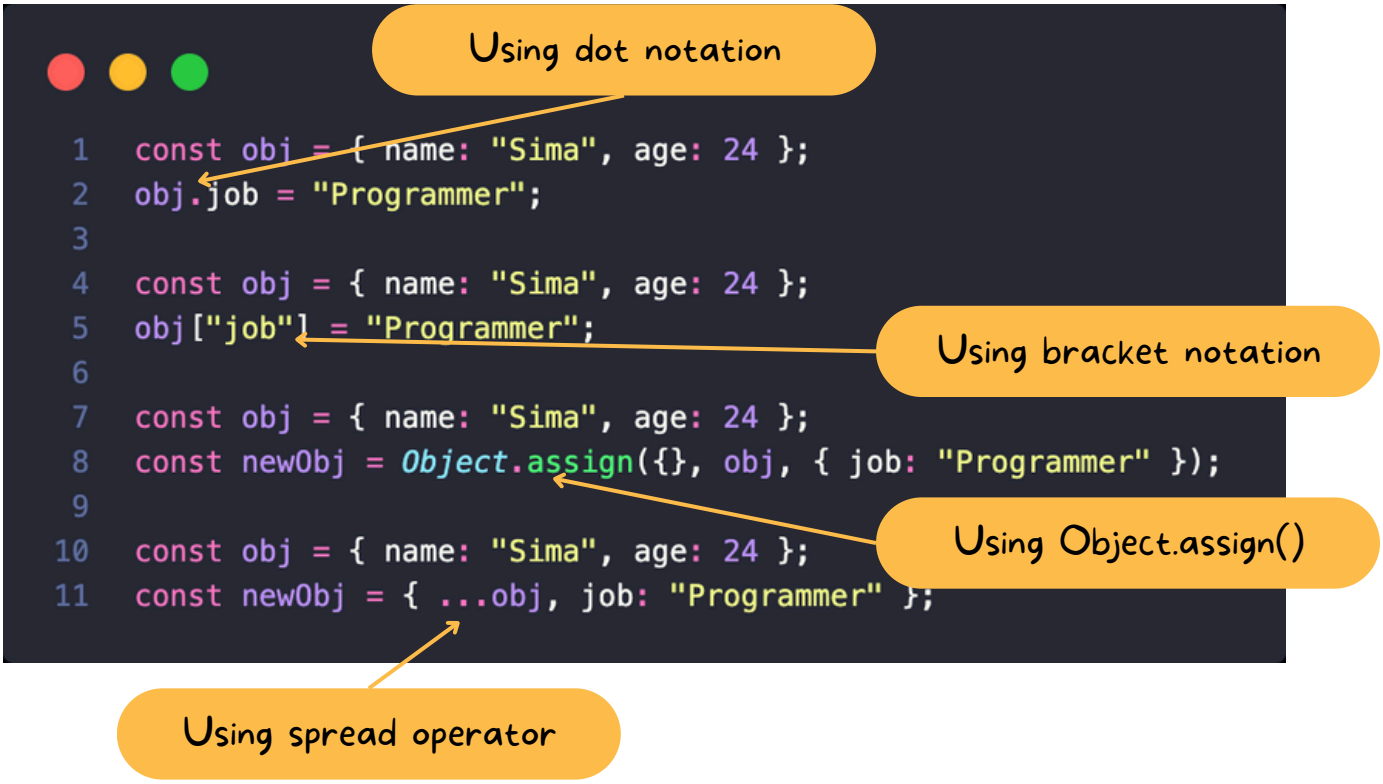
  In JavaScript, there are four ways to add new properties dynamically to an object.

  1. Dot notation: You can add a new property to an object by using dot notation and assigning a value to a new or existing key.

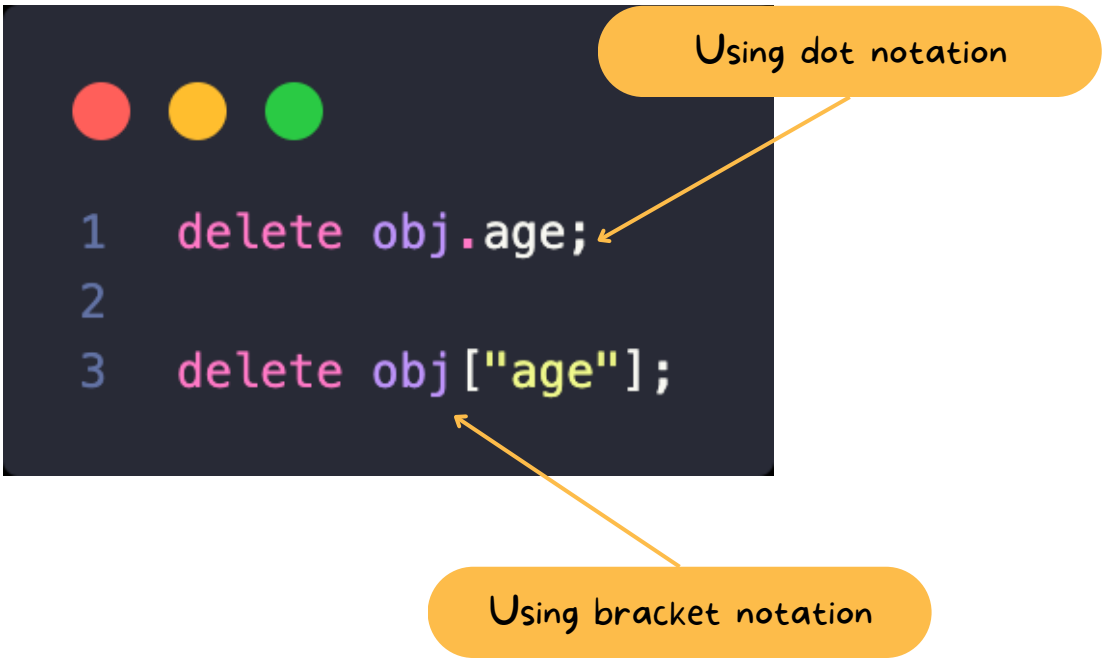
  2. Bracket notation: Using bracket notation and assigning a value to a new or existing key, you can also add a new property to an object.

  3. Object.assign(): When you merge an object with another object, you can add properties to the merged object. This method copies the properties from one or more source objects to a target object.

  4. Spread operator: You can also use the spread operator to add properties to an object. This method makes a new object that has all of the properties of the original object plus the properties that you tell it to have.



You can remove a property from an object using bracket and dot notation and the delete keyword as follows:

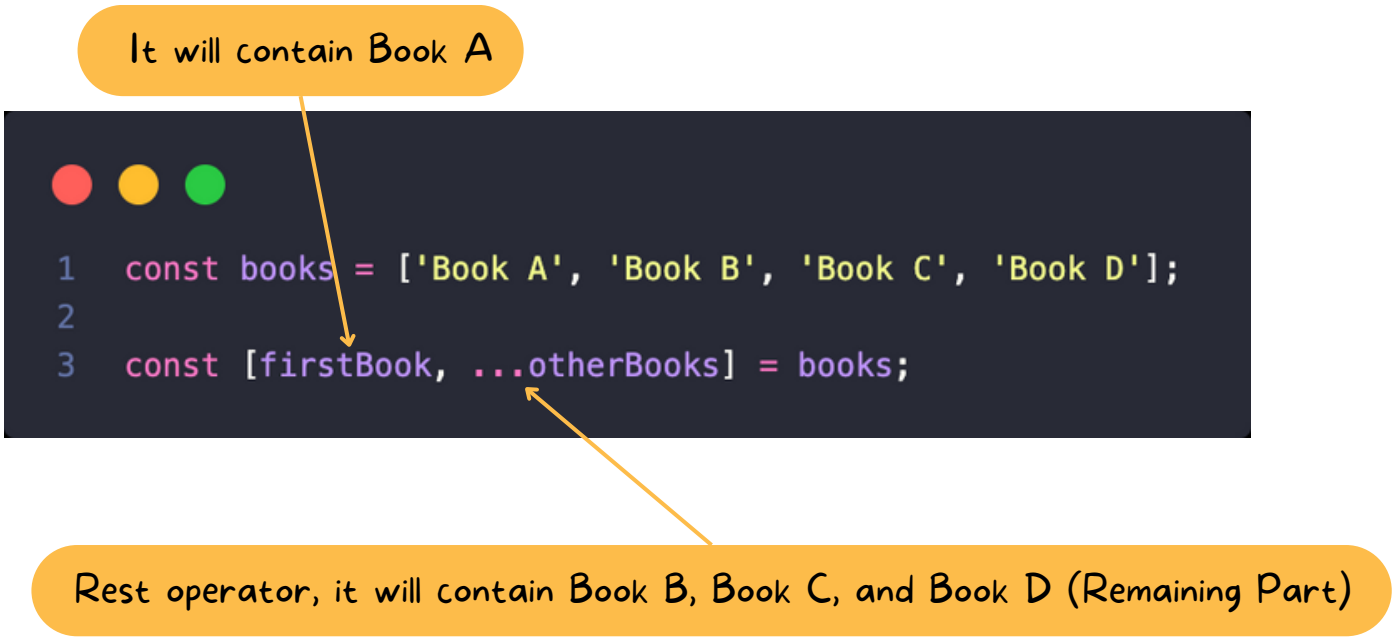


- **Explain about Rest operator.**

The Rest operator is denoted by three dots (...).

It is used to make a new array or object from the remaining parts of an array or object.

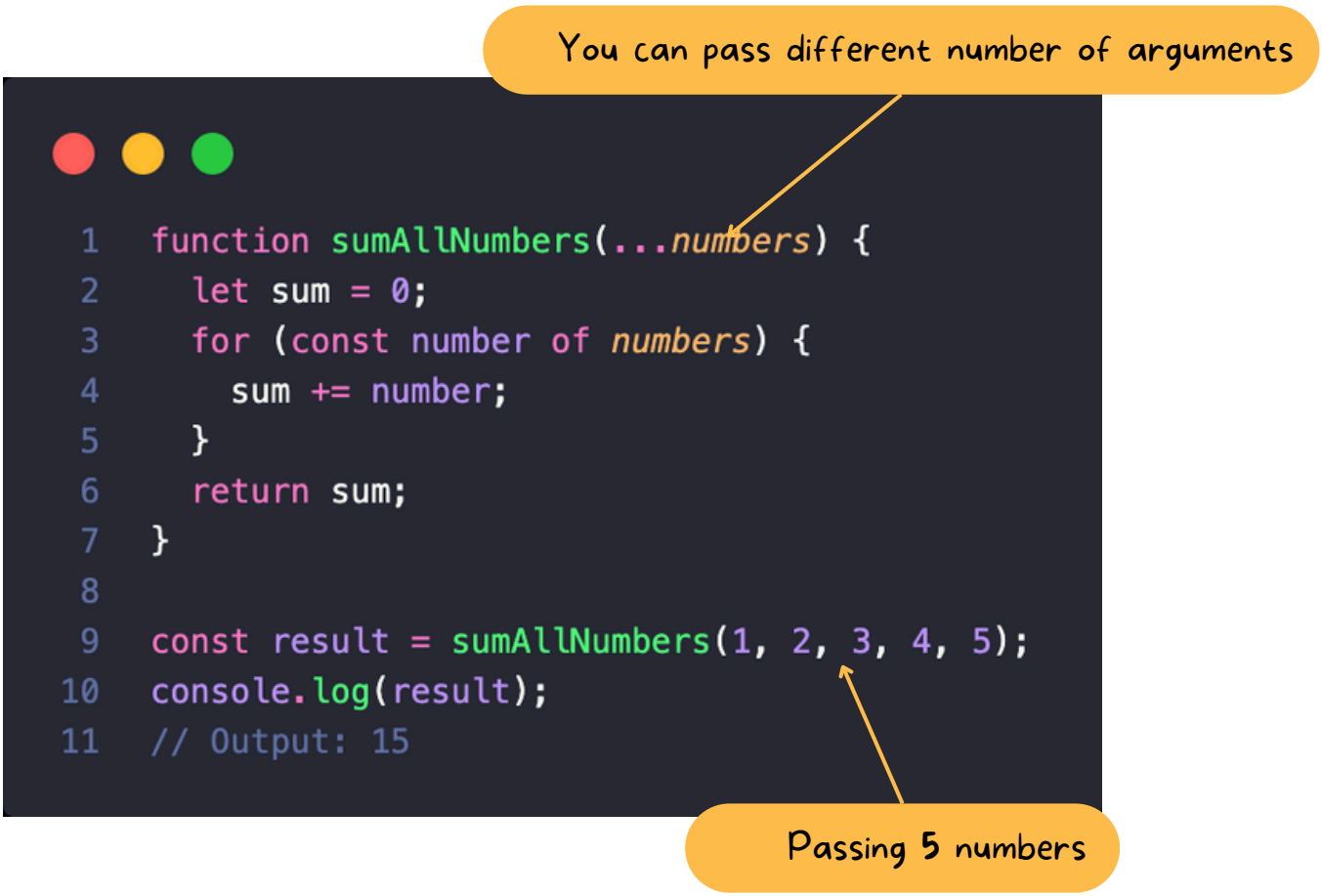
Let's take an example



Let's take the second example,

The Rest operator can also be used in functions to accept a variable number of arguments.

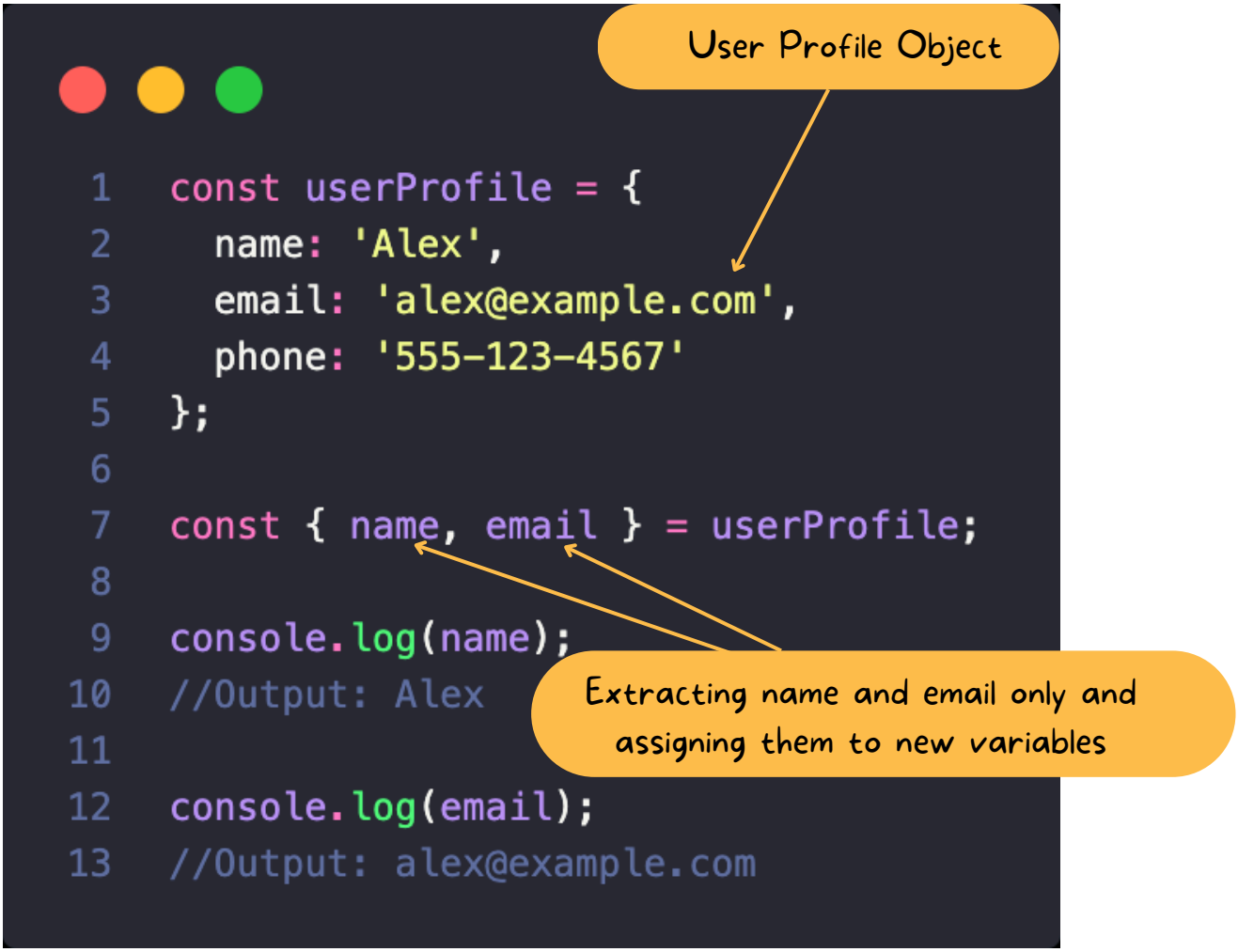
Here's an example of a function using the Rest operator:



- **What is object destructuring?**

Object destructuring is a way to extract properties from an object and assign them to variables. It makes working with objects simpler and easier to read.

Let's take an example to understand it.



- **What is Callback Hell?**

  Callback hell occurs when multiple asynchronous operations are nested, making code hard to read, understand, and maintain.

  Imagine planning a surprise party and having to order food, decorate, and hire a DJ.

  You delegate tasks to friends and ask them to call you when they're done instead of waiting. Programming calls these "callbacks".

  If each of these tasks requires a callback, such as the food order needing approval, the decorations needing confirmation, and the DJ needing your music selection, the callbacks start to become dependent on one another.

This complexity makes the overall process harder to manage. In programming, this situation is known as "callback hell."

Let's take an example to understand it better:

```
function orderFood(callback) {
    setTimeout(() => {
        console.log("Food ordered");
        callback();
    }, 1000);
}

function decorateVenue(callback) {
    setTimeout(() => {
        console.log("Venue decorated");
        callback();
    }, 1000);
}

function arrangeDJ(callback) {
    setTimeout(() => {
        console.log("DJ arranged");
        callback();
    }, 1000);
}

orderFood(() => {
    decorateVenue(() => {
        arrangeDJ(() => {
            console.log("All tasks completed");
        });
    });
});
```

In the above example, the nested callbacks make the code difficult to read and maintain.

To avoid callback hell, we can use Promises or async/await.

```javascript
function orderFood() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Food ordered");
            resolve();
        }, 1000);
    });
}

function decorateVenue() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Venue decorated");
            resolve();
        }, 1000);
    });
}

function arrangeDJ() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("DJ arranged");
            resolve();
        }, 1000);
    });
}

orderFood()
    .then(() => decorateVenue())
    .then(() => arrangeDJ())
    .then(() => console.log("All tasks completed"));
```

Using Promise to avoid callback hell

```javascript
async function runTasks() {
    await orderFood();
    await decorateVenue();
    await arrangeDJ();
    console.log("All tasks completed");
}

runTasks();
```

Using async/await

- **Explain about call(), bind() and apply() methods in JavaScript.**

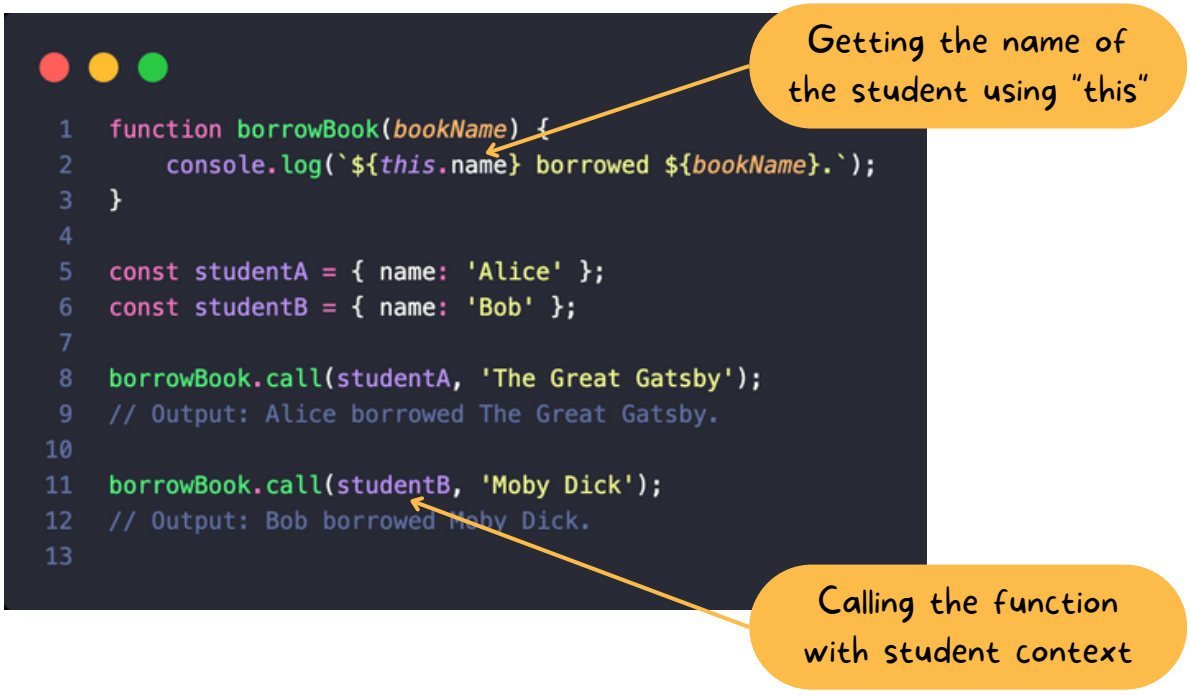
call(), apply(), and bind() are useful for setting the context (i.e., the 'this' value) in JavaScript functions. call() and apply() invoke functions immediately, while bind() creates a new function with a specified 'this' value to use later.

Let's understand them using examples.

1.call()

The call() method allows you to call a function with a specified 'this' value and arguments.

It's helpful when you want to use a method from one object in the context of another object.



2. apply():

The apply() method is similar to call(). It allows you to call a function with a specified 'this' value and arguments as an array or array-like object.

```
function borrowBooks(book1, book2) {
    console.log(`${this.name} borrowed ${book1} and ${book2}.`);
}

const books = ['The Catcher in the Rye', 'To Kill a Mockingbird'];

borrowBooks.apply(studentA, books);
// Output: Alice borrowed The Catcher in the Rye and To Kill a Mockingbird.
```

Passing the array as an argument

3. bind():

The bind() method creates a new function with a specified 'this' value and optional initial arguments.

This method doesn't call the function immediately, instead, it returns a new function with the bound context, allowing you to call it later.

```
function returnBook(bookName) {
    console.log(`${this.name} returned ${bookName}.`);
}

const studentAReturn = returnBook.bind(studentA);
const studentBReturn = returnBook.bind(studentB);

studentAReturn('The Great Gatsby');
// Output: Alice returned The Great Gatsby.
studentBReturn('Moby Dick');
// Output: Bob returned Moby Dick.
```

Accessing object value bind with function

Binding the student object

Calling the function created by bind()

- **What is Progressive Web App (PWAs)?**

  PWAs are a type of web application that combines the best features of both websites and native mobile apps.

  They are made to be fast, reliable, and interesting for users, giving them a web experience similar to that of an app.

  To accomplish this, PWAs employ modern web technologies such as Service Workers, which enable them to function offline and deliver push notifications.
  and the Web App Manifest enables users to install the PWA on their devices and make it accessible even without an internet connection, is another important feature.

  A good example of a PWA is Twitter Lite. Twitter recognized that many users had slow internet connections or limited data plans, so they created a lightweight version of their platform that loads quickly and uses less data.

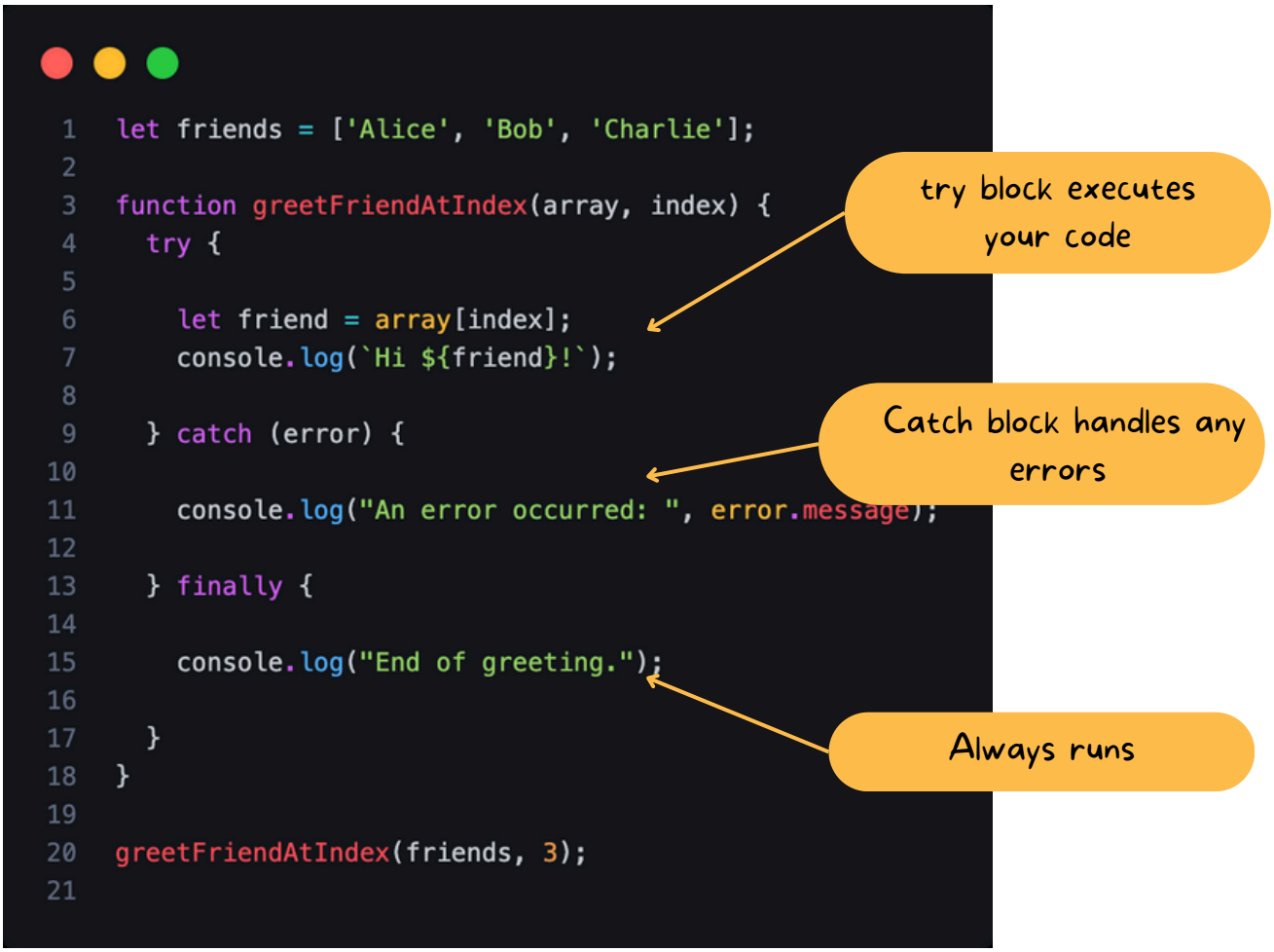
  So, Twitter Lite offers a smooth, app-like experience that can be used on any device. This makes it easier for millions of people around the world to use and gives them more options.

- **How to handle exceptions/errors in JavaScript?**

  You handle exceptions or errors by wrapping your code in a **try** block, then handling any errors that occur in a **catch** block, with an optional **finally** block for cleanup tasks that should always run.

  let's say you have an array of your friends' names and a function that attempts to say "Hi" to a friend at a specific index.

  If you try to greet a friend at an index that doesn't exist in the array, an error may occur. We'll handle this using try-catch.

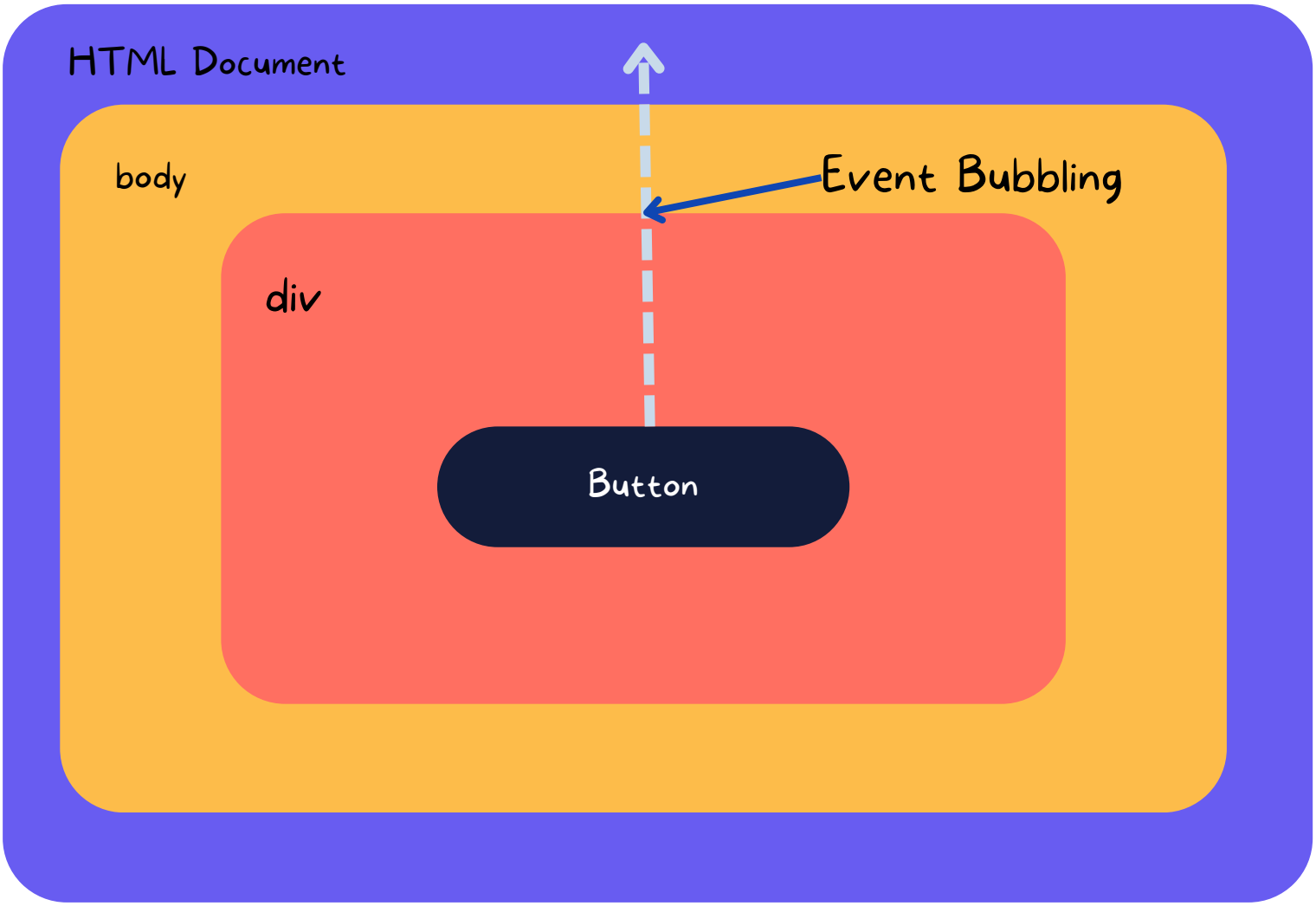


- **What is Event Bubbling in JavaScript?**

  Event bubbling is a way that events spread from the element where they occur to its parent elements, and then to their parent elements, and so on.

  For example, let's say we have a button inside a div. If we click on the button, the onclick event will be called first on the button element, then on the div element, and then on the document element.

  This can be useful if you want to handle an event on all of the elements in a hierarchy, but it can also be a problem if you only want to handle the event on a specific element.



To prevent an event from bubbling, you can use the stopPropagation() method. This will stop the event from propagating to any of the ancestor elements.

- **what is the difference between prototype and __proto__**

  To explain the difference between prototype and __proto__, let's start with a real technical example:

```
function Person(name) {
    this.name = name;
}

Person.prototype.greet = function () {
    console.log(`Hello, my name is ${this.name}!`);
};

const john = new Person("John Doe");
```

In this example, we have a Person constructor function. The Person.prototype property is an object that contains all of the properties and methods that will be shared by all instances of the Person class.

In this case, we have added a greet() method to the prototype.

When we create a new instance of the Person class using the new keyword, the JavaScript engine creates a new object and sets its **__proto__** property to the Person.prototype object.

This means that the new object will inherit all of the properties and methods from the **prototype**.

In the example above, the Person.prototype object is the **prototype** of the john object. The john object's **__proto__** property points to the Person.prototype object.

Here is a simplified analogy:

- prototype is the blueprint for an object.
- __proto__ is the pointer to the blueprint.

- **What is the difference between synchronous and asynchronous code?**

Synchronous Code	Asynchronous Code
Executes sequentially.	Executes out of order, not waiting for one operation to complete before starting the next.
Can lead to a blocking behavior in applications, potentially making the UI unresponsive if a task takes a long time.	Non-blocking behavior, allowing other tasks to run simultaneously.

Example
`const result = someFunction();`
`console.log(result);`
`console.log('After function');`
	Example
	`fetch('https://api.example.com/data')`
	`.then(response => response.json())`
	`.then(data => console.log(data))`
	`.catch(error => console.error(error));`
	`console.log('After fetch call');`

Not well-suited for tasks that can take an unpredictable amount of time, such as network requests or reading from disk.	Ideal for tasks like network requests, timers, and other operations that shouldn't block the main thread.
Common Methods/Functions: Array.forEach(), Math.max(), etc.	**Common Methods/Functions:** setTimeout(), fetch(), XMLHttpRequest, Promises, async/await.

- **What is the difference between debouncing and throttling, and when would you use each one?**

What is Debouncing?
Debouncing waits to execute a function until a certain amount of time has passed since the last time it was called.

This is useful for preventing unnecessary function calls when the user is interacting with an element rapidly, such as when typing in a search bar or scrolling through a list.

What is Throttling?

Throttling ensures that a function is only executed at most once within a given time period, regardless of how many times it is called.

This is useful for limiting the load on a server or other resource when the user is repeatedly performing an action, such as clicking a button or submitting a form.

For example, if you have a button that submits a form, you could use throttling to prevent the form from being submitted more than once per second.

This would help to protect your server from being overloaded and would also prevent the user from accidentally submitting the form multiple times.

Here are some specific examples of when to use debouncing and throttling:

Debouncing:

- Search bar: Prevent the search results from updating every time the user types a letter.
- Infinite scroll: Prevent the function that loads new content as the user scrolls down the page from being called too often.
- Autocomplete: Prevent the autocomplete suggestions from updating every time the user types a letter.

Throttling:

- Button click: Prevent a button from being clicked more than once per second.
- Form submission: Prevent a form from being submitted more than once per minute.
- API calls: Prevent making too many API calls in a short period of time.

- **List out and explain mouse events**

1. **click**: Occurs when the mouse clicks on an element (mouse button pressed and released).
2. **dblclick**: Happens when the mouse double-clicks on an element.
3. **mousedown**: Triggers when the mouse button is pressed down over an element.
4. **mouseup**: Fires when the mouse button is released over an element.
5. **mousemove**: Occurs when the mouse is moving while over an element.
6. **mouseover**: Triggered when the mouse comes over an element.
7. **mouseout**: Fires when the mouse leaves an element.
8. **mouseenter**: Similar to mouseover, but it does not bubble and does not react to the mouse coming from a child element.
9. **mouseleave**: Similar to mouseout, but it does not bubble and does not react to the mouse going to a child element.
10. **contextmenu**: Occurs when the right button of the mouse is clicked (before the context menu is displayed).

- **List out and explain keyboard events**

1. **keydown** - Fired when a key is pressed down, and repeats while the key is kept down.
2. **keyup** - Triggered when a key is released after being pressed.
3. **keypress** - Occurs when a key is pressed and released, and a character is actually produced (Note: This event is deprecated and not recommended for use in new code).
4. **input** - Fires when an input event occurs (like entering text in a field), not strictly a keyboard event but often used in conjunction with them to handle input.

- **Explain about event propagation.**

Event propagation is about how events like clicks, keypresses, or mouse movements travel through your webpage's elements. Let's break it down in a simple way:

Imagine you have a webpage with a button inside a div. If you click on the button, not only does the button "know" about the click, but the **div enclosing it also gets the information about that click.** This is event propagation at work.

There are two main phases in event propagation:

1. **Capturing Phase:** This is the less commonly used phase. It starts from the top of the document (the root) and goes down to the target element. In our example, it would start from the document object, move to the div, and finally reach the button you clicked.
2. **Bubbling Phase:** This is the default phase used in most web development scenarios. Here, the event starts from the target element and bubbles up to the top of the document. Using our example, after clicking the button, the event first occurs on the button and then bubbles up to the div, and further up to the document object.

Now, why is this important? Understanding event propagation allows you to manage how events are handled and reacted to on your page.

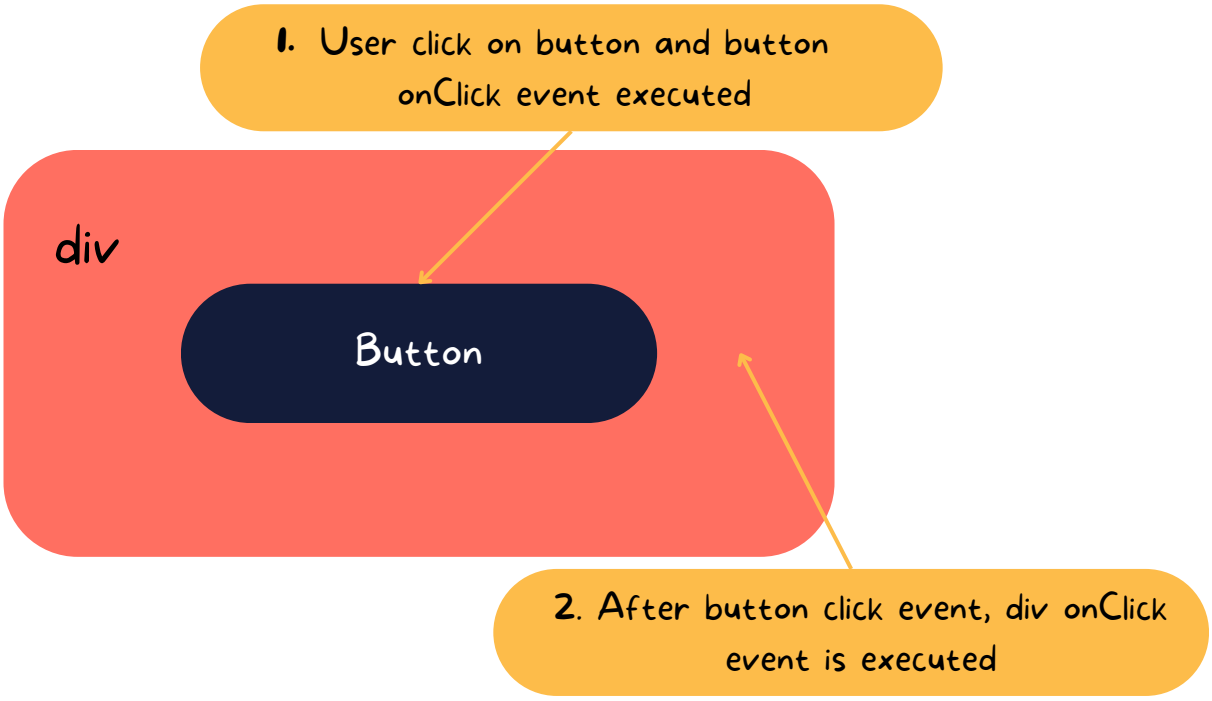
For instance, you might want a click on the button to do one thing, but a click on the div that contains the button to do something else.

```html
<body>
    <div id="myDiv">
        <button id="myButton">Click Me!</button>
    </div>

    <script>
        document.getElementById('myButton').addEventListener('click', function (event) {
            alert('Button was clicked!');
        });

        document.getElementById('myDiv').addEventListener('click', function (event) {
            alert('Div was clicked!');
        });
    </script>
</body>
```

This code is also executed after button click

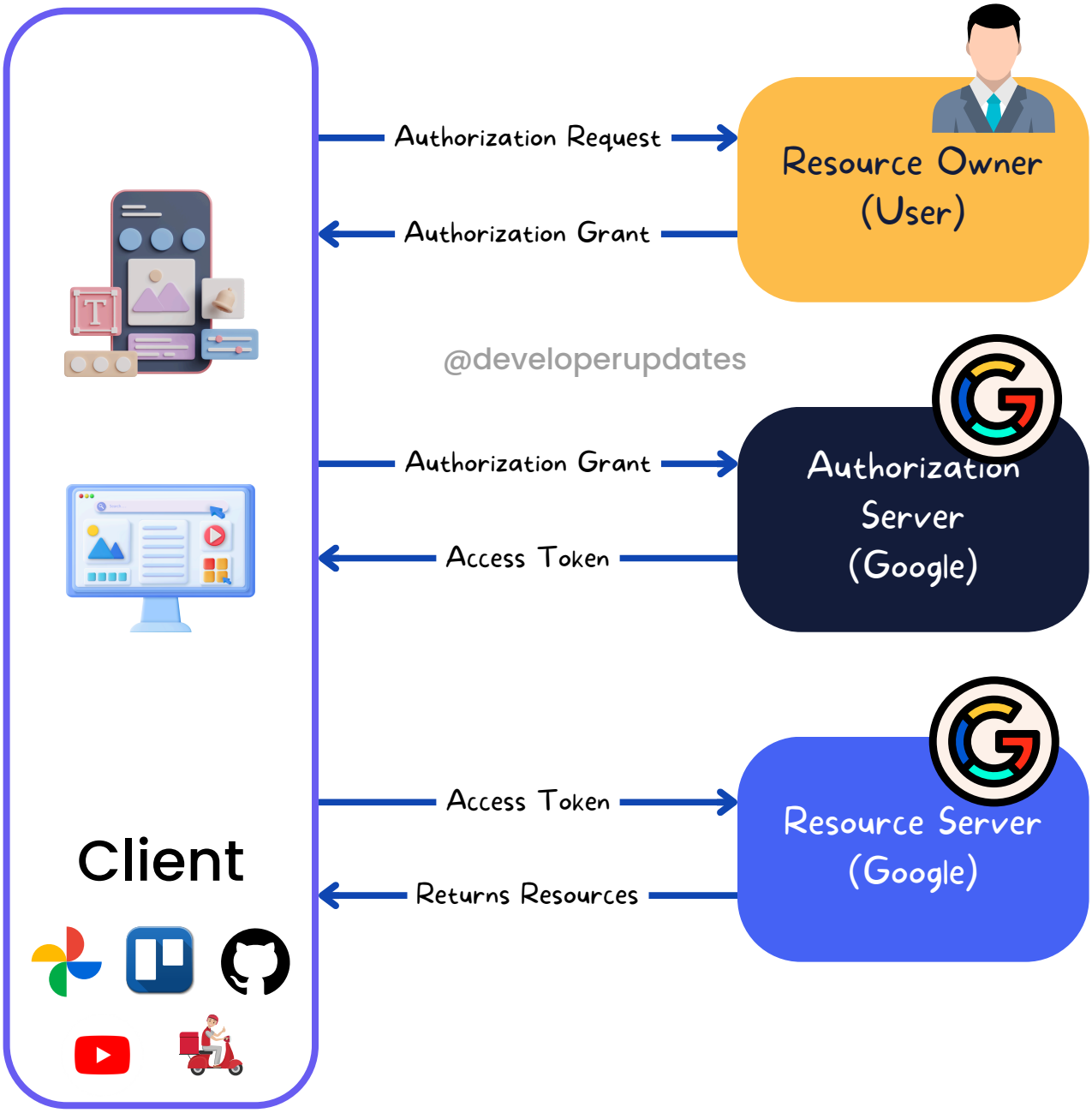


JavaScript gives you control over this process. You can stop an event from propagating using methods like **event.stopPropagation()**. This is often used to prevent an event from "bubbling up" to parent elements or "capturing down" to child elements.

- **Explain the OAuth 2.0 authorization flow for a web application.**

  OAuth 2.0 is a protocol for secure authorization, enabling apps to access user data from services like Facebook, Google, and Microsoft without needing the user's password, providing limited and controlled permissions.

  Let's go through the OAuth 2.0 authorization flow using Google's authentication services as an example. Here's how it typically works step by step:



Suppose You're using a delivery app and choose to log in with your Google account.

1. **App Requests Permission**: The app redirects you to a Google login page. This is the app asking for permission to access some of your Google account data (like your email address).
2. **User Grants Permission**: You enter your Google credentials and grant the requested permissions. Note that you are giving these details to Google, not the delivery app.
3. **Google Provides Authorization Code**: Once you grant permission, Google sends you back to the delivery app with an authorization code. This code is like a temporary passkey.
4. **App Requests Access Token**: The delivery app takes this authorization code and asks Google for an access token. This is done in the background.
5. **Google Issues Access Token**: If the authorization code is valid, Google gives the delivery app an access token and sometimes a refresh token.
6. **Access Token Used**: The delivery app uses this access token to access your Google account data as permitted.

- ## What are the key differences between prototypal and classical inheritance models?

  ### 1. Conceptual Foundation

  - **Classical Inheritance:** It is based on classes. You define a class (like a blueprint), and then create objects (instances) from that class. **It's similar to creating various objects from a specific template.**
  - **Prototypal Inheritance:** This model is based on existing objects rather than theoretical classes. You start with an object (which is already functional) and then **create new objects that inherit properties from this existing object.**

2. Hierarchy and Structure

- **Classical Inheritance:** Involves a clear hierarchical structure. You have base classes and derived classes. The derived **classes inherit from the base classes and can also have their own unique properties.**
- **Prototypal Inheritance:** There is no strict hierarchy. **Any object can inherit properties from any other object.** This approach is more flexible but can be less structured, which might lead to complexity in large systems.

3. Inheritance Mechanism

- **Classical Inheritance**: The mechanism is through the creation of instances of classes. It follows a top-down approach where the **child class inherits from the parent class.**
- **Prototypal Inheritance**: The inheritance is achieved by linking objects. A new object is **created by essentially cloning an existing object**, known as the prototype.

4. Flexibility and Dynamism

- **Classical Inheritance**: It's more rigid. Once a class is defined, **you can't change its structure at runtime**. Modifications in the class hierarchy can be complex.
- **Prototypal Inheritance**: Offers more dynamism. **You can modify the prototype object at runtime, and all objects inheriting from this prototype will see these changes.** This makes it highly flexible but requires careful management to avoid unintended side-effects.

- ## What is design pattern in javascript

  Design patterns in JavaScript are essentially reusable solutions to common programming problems.

  They serve as templates for writing efficient and optimized code, enabling developers to solve complex problems more effectively.

There are three primary categories of design patterns:

1. **Creational Patterns** focus on the object creation process. They aim to make the instantiation process more flexible and straightforward.
2. **Structural Patterns** deal with the composition of classes or objects. They help ensure that if one part of a system changes, the entire system doesn't need to do the same.
3. **Behavioral Patterns** are about improving communication between objects. They help objects cooperate and fulfill tasks.

- **How does the Module design pattern work in JavaScript?**

The Module design pattern in JavaScript is a popular and powerful pattern used to organize and structure your code in a modular way, making it more maintainable, scalable, and understandable.

It allows you to **encapsulate private variables and functions**, exposing only the parts you want to be public, thus shielding parts of your code from the **global scope and preventing potential naming conflicts.**

The essence of the Module pattern is to create self-contained modules or pieces of code, each with its own private and public parts.

This is achieved by taking advantage of JavaScript's function scope and closure capabilities.

Here's a simple way to think about it:

1. **Encapsulation**: The Module pattern encapsulates "private" variables and functions inside a function scope. **This is not accessible from the outside world,** only to functions defined within the same scope.
2. **Public API Exposure:** It then explicitly exposes a public interface. Typically, this is done by returning an object from the module's enclosing function, where this object contains references to the functionalities you wish to expose. This can include functions and variables that internally access and return the private variables/functions defined within the scope.

Here's a basic example to illustrate:

```
var myModule = (function() {
    // Private variables and functions
    var privateVar = 'I am private';
    function privateFunction() {
        console.log(privateVar);
    }

    // Public part
    return {
        // Public variables and functions
        publicMethod: function() {
            console.log('Accessing private part: ');
            privateFunction();
        }
    };
})();

myModule.publicMethod();
// Outputs: Accessing private part: I am private
// Trying to access privateVar directly will result in an error
console.log(myModule.privateVar); // undefined
```

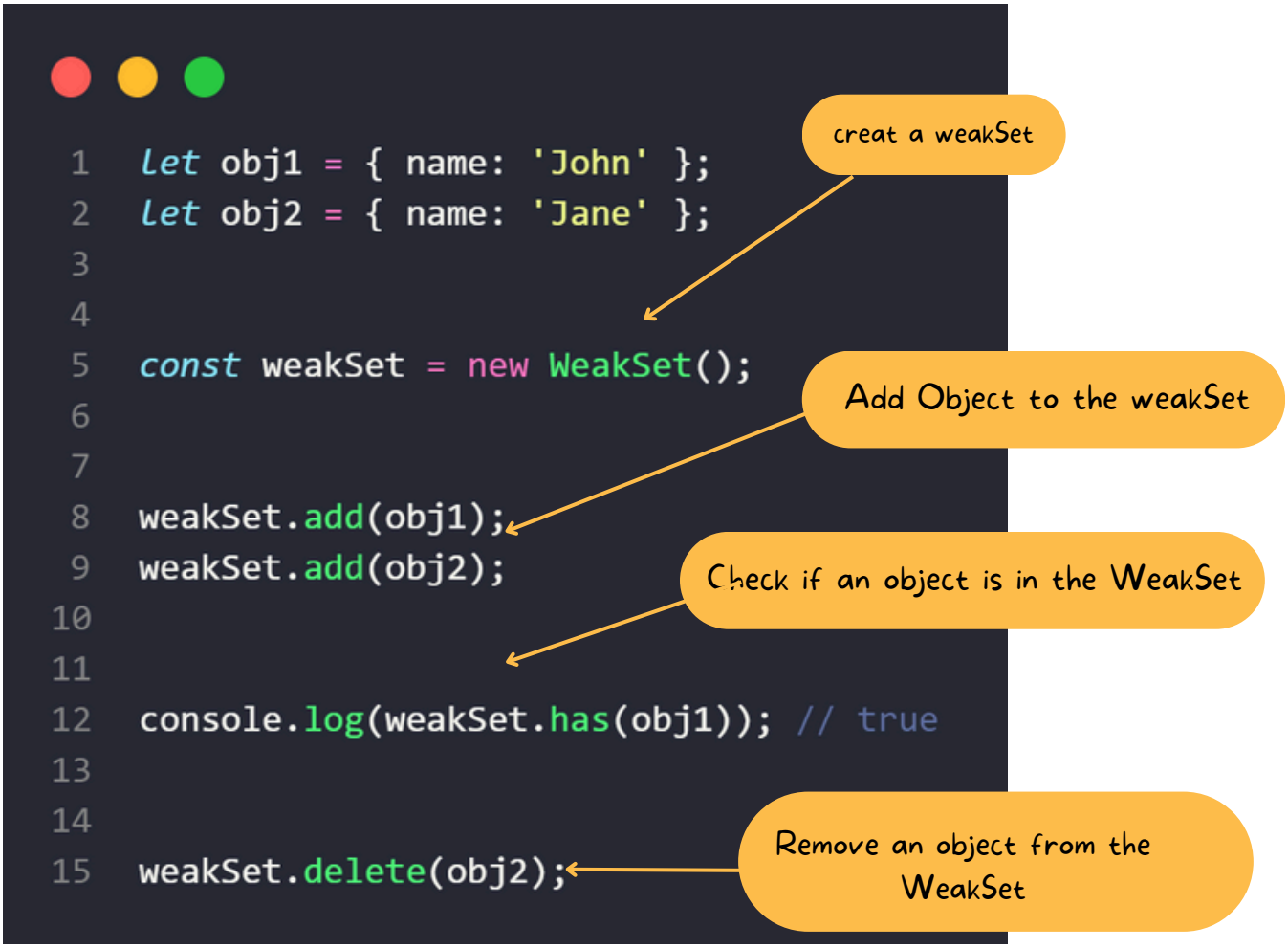
- **What is WeakSet and WeakMap?**

  1. WeakSet:

  A WeakSet is a special type of Set object in JavaScript. It holds only objects and not primitive values like numbers or strings. The main difference is that objects in a WeakSet are weakly referenced.

  This means that if an object is only referenced by WeakSet instances, it will be garbage collected if there are no other references to it.

  WeakSets are useful for storing objects that you don't want to keep alive artificially. They have limited functionality compared to regular Sets.



You cannot iterate over a WeakSet or get its size. Objects can only be added and removed from it.

2. WeakMap:

A WeakMap is similar to a regular Map object, but with a different type of key.

In a WeakMap, the keys must be objects, and they are weakly referenced. Values can be any type, including objects or primitive values.

Like WeakSets, if an object used as a key in a WeakMap has no other references, it will be garbage collected, and the entry in the WeakMap will be removed.

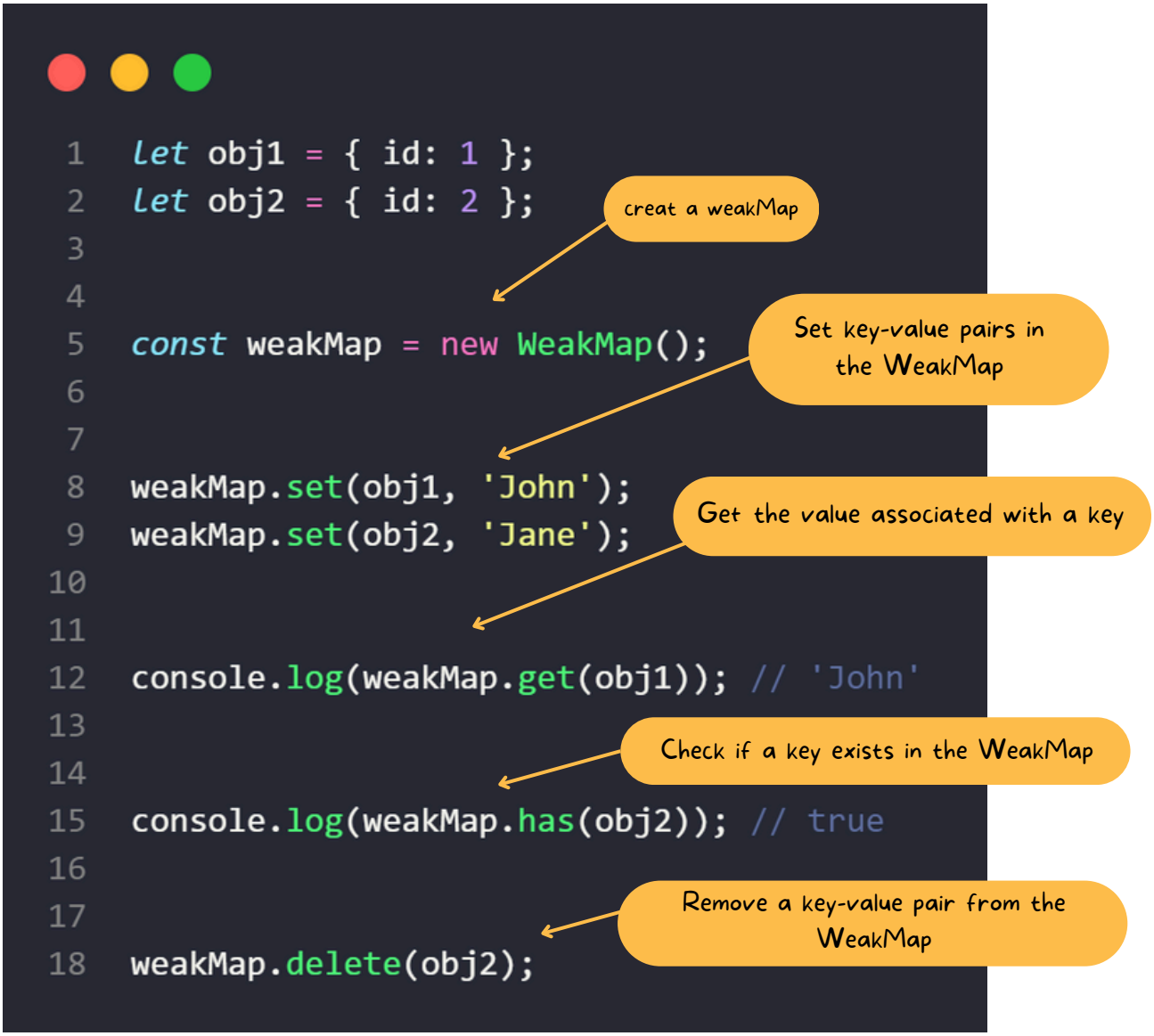
This helps in preventing memory leaks.

WeakMaps are particularly useful for associating data with objects without having to worry about manually cleaning up that data.

They have limited functionality compared to regular Maps.

You cannot iterate over a WeakMap or get its size.

The only methods available are set(), get(), has(), and delete().



- **What is Factory Pattern?**

  The Factory Pattern is a creational design pattern in object-oriented programming. It provides a way to create objects without exposing the creation logic.

  The main idea behind the Factory Pattern is to define an interface or abstract class for creating objects.

  This interface or abstract class is then implemented by one or more concrete classes.

The client code interacts with the Factory and requests objects from it. The Factory then creates and returns the requested objects without revealing the actual classes involved.

This pattern promotes loose coupling between the client and the concrete classes. It also allows introducing new concrete classes without modifying the existing client code.

The Factory Pattern is useful in scenarios where:

- The creation process of objects is complex or involves multiple steps.
- The client code should not know the details of how objects are created.
- The type of object to be created is determined at runtime.

Some key components of the Factory Pattern are:

- Product interface/abstract class: Defines the interface for objects created by the Factory.
- Concrete Product classes: Implements the Product interface/abstract class.
- Factory interface/abstract class: Declares the factory method for creating objects.
- Concrete Factory classes: Implements the factory method and creates specific Product objects.

By using the Factory Pattern, the code becomes more maintainable, extensible, and testable.

It also follows the Open/Closed Principle, as new concrete classes can be added without modifying existing code.

- **Describe a time when you had to troubleshoot and debug a complex front-end issue. How did you approach the problem-solving process?**

1. Give Introduction:
Briefly set the context by mentioning a specific project or application where I encountered a complex front-end issue.

Example: "During the development of a large-scale e-commerce web application, I encountered a challenging bug related to rendering performance."

2. Give Problem Description:
Describe the issue in more detail, highlighting its complexity and impact on the application.

Example:
"The application had a product listing page with hundreds of items. As the user scrolled down, the page became increasingly sluggish and unresponsive, negatively impacting the user experience."

3. Troubleshooting Approach:
Explain the systematic approach you took to identify and resolve the issue.

"I began by analyzing the application's code and identifying potential bottlenecks. I utilized browser developer tools like the JavaScript profiler and network panel to monitor performance and isolate the root cause.

After thorough investigation, I discovered that the issue stemmed from inefficient rendering of product images and their associated data.

The application was unnecessarily re-rendering the entire product list instead of only updating the visible components on the screen."

4. Explain Solution:
Describe the solution you implemented and how it addressed the problem.
"To optimize performance, I implemented virtual scrolling techniques using popular front-end libraries like React Virtual or ngx-virtual-scroller.

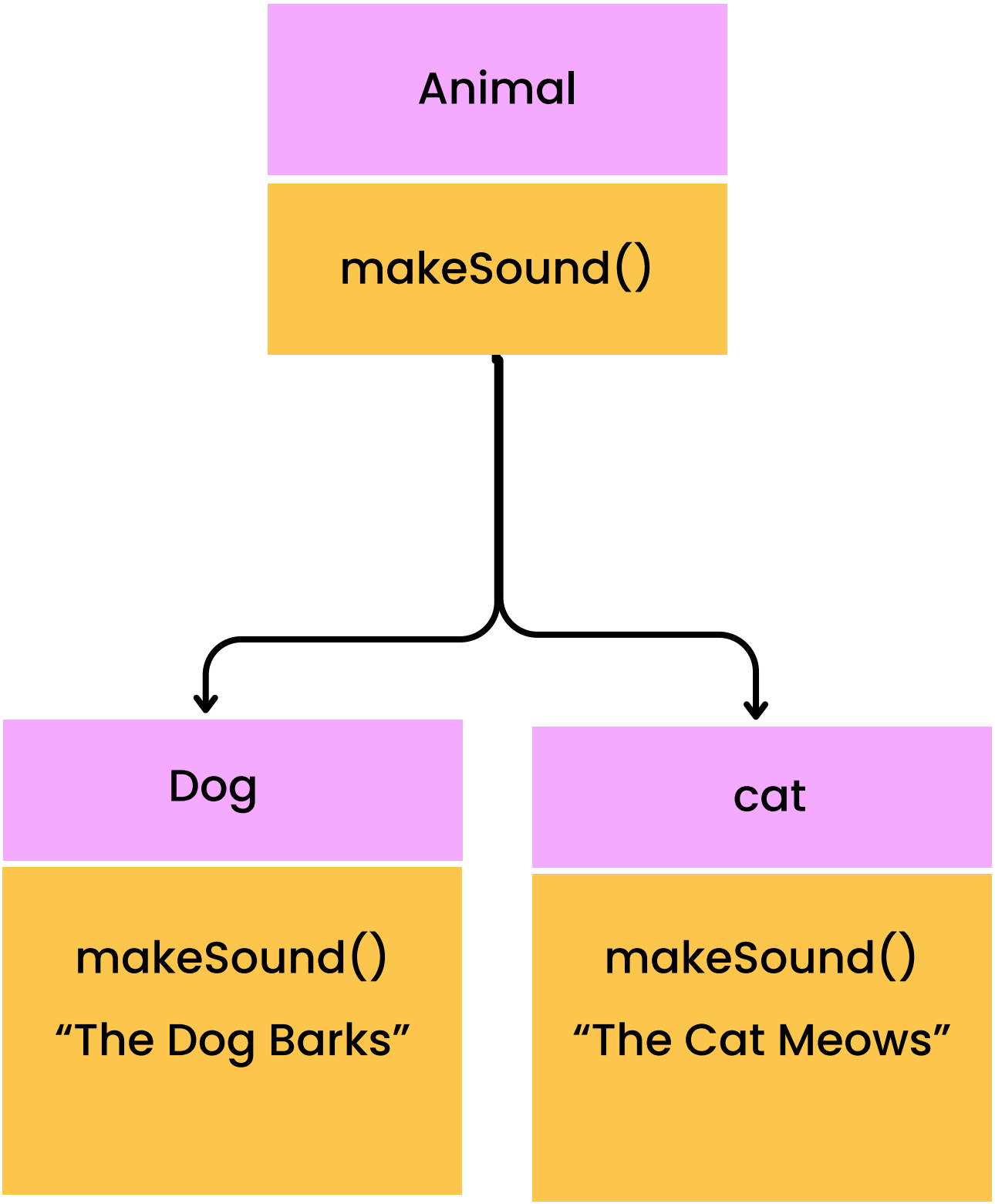
This approach rendered only the visible components on the screen, significantly reducing the workload and improving overall performance.

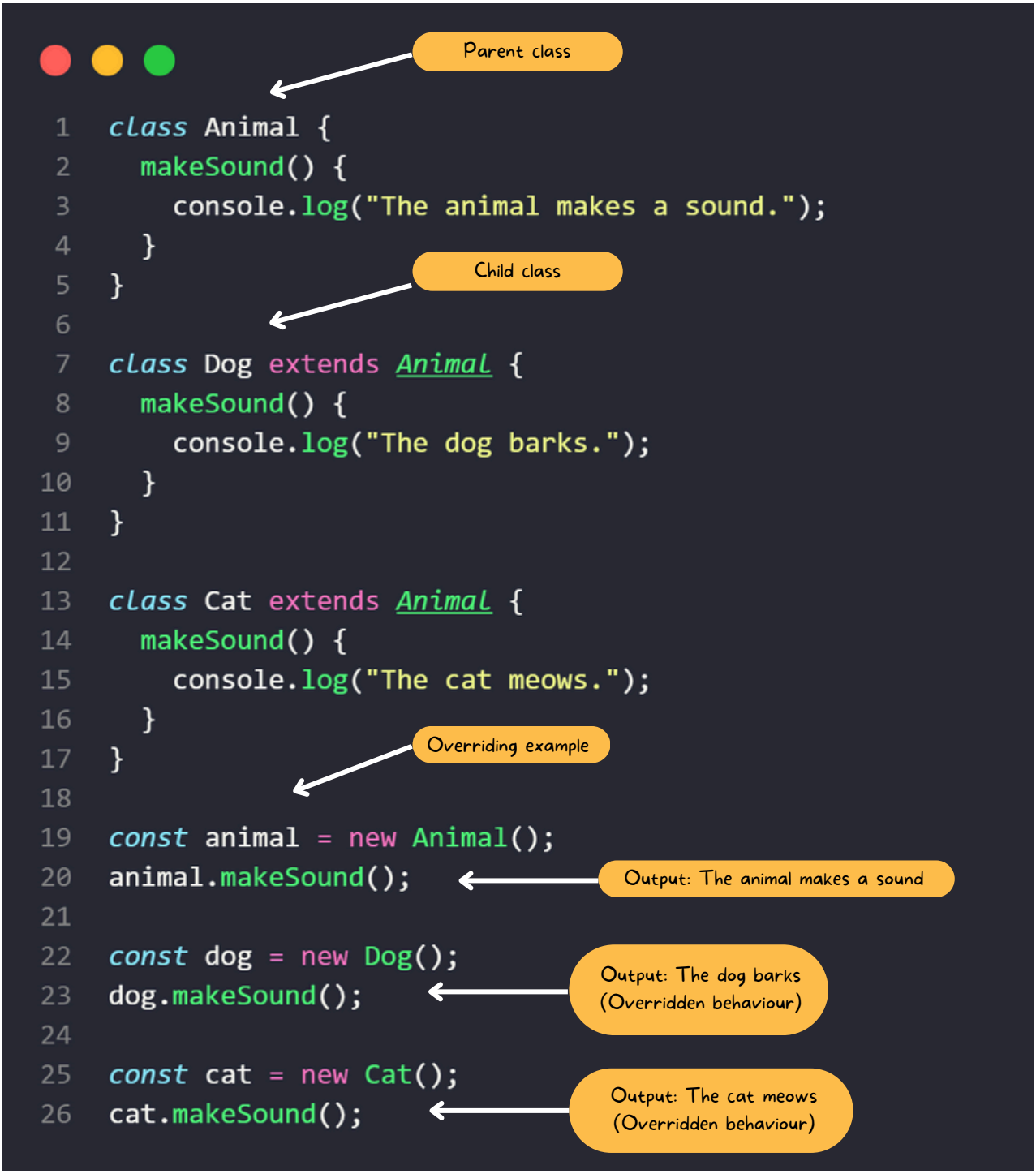
Additionally, I implemented code splitting and lazy loading to minimize the initial bundle size and load times."

- **Can you describe the concept of polymorphism in object-oriented programming?**

Imagine we have different types of animals (classes) that can make sounds (methods).

However, the way they make sounds is different for each animal.


Animal
makeSound()
Dog
makeSound()
“The Dog Barks”
cat
makeSound()
“The Cat Meows”




```
class Animal {
  makeSound() {
    console.log("The animal makes a sound.");
  }
}

class Dog extends Animal {
  makeSound() {
    console.log("The dog barks.");
  }
}

class Cat extends Animal {
  makeSound() {
    console.log("The cat meows.");
  }
}

const animal = new Animal();
animal.makeSound();

const dog = new Dog();
dog.makeSound();

const cat = new Cat();
cat.makeSound();
```

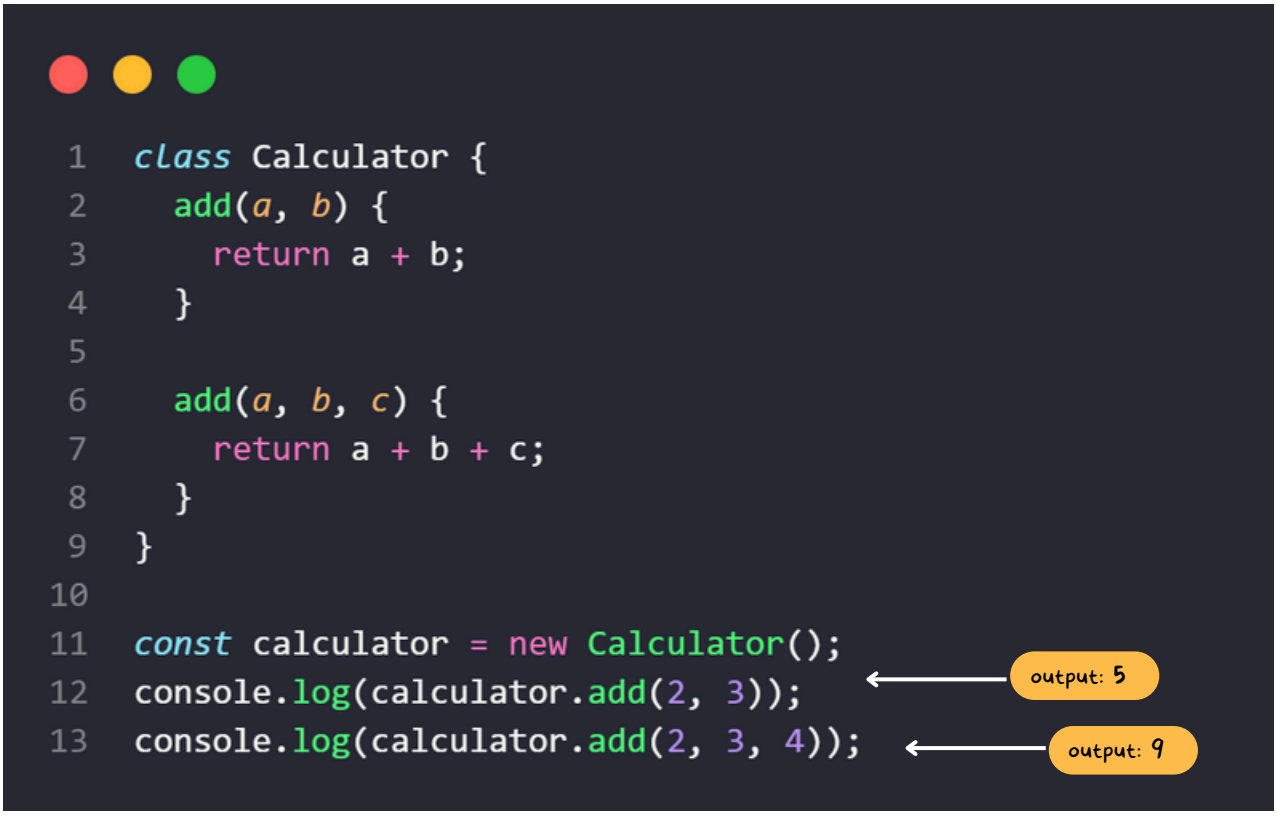
In this example, we have a base class Animal with a makeSound method.

The Dog and Cat classes inherit from Animal and override the makeSound method with their own implementation.

When we create instances of Animal, Dog, and Cat, and call the makeSound method on them, we get different outputs based on the actual type of the object at runtime.

This is an example of **runtime polymorphism (overriding)**.

Now, let's look at an example of overloading (compile-time polymorphism) in JavaScript:



In this example, the Calculator class has two add methods with different parameter lists.

When we call the add method with two arguments, the first implementation is invoked.

When we call it with three arguments, the second implementation is invoked.
However, it's important to note that JavaScript doesn't have true method overloading.

Instead, it uses method overwriting, where the last defined method with the same name overwrites the previous ones.

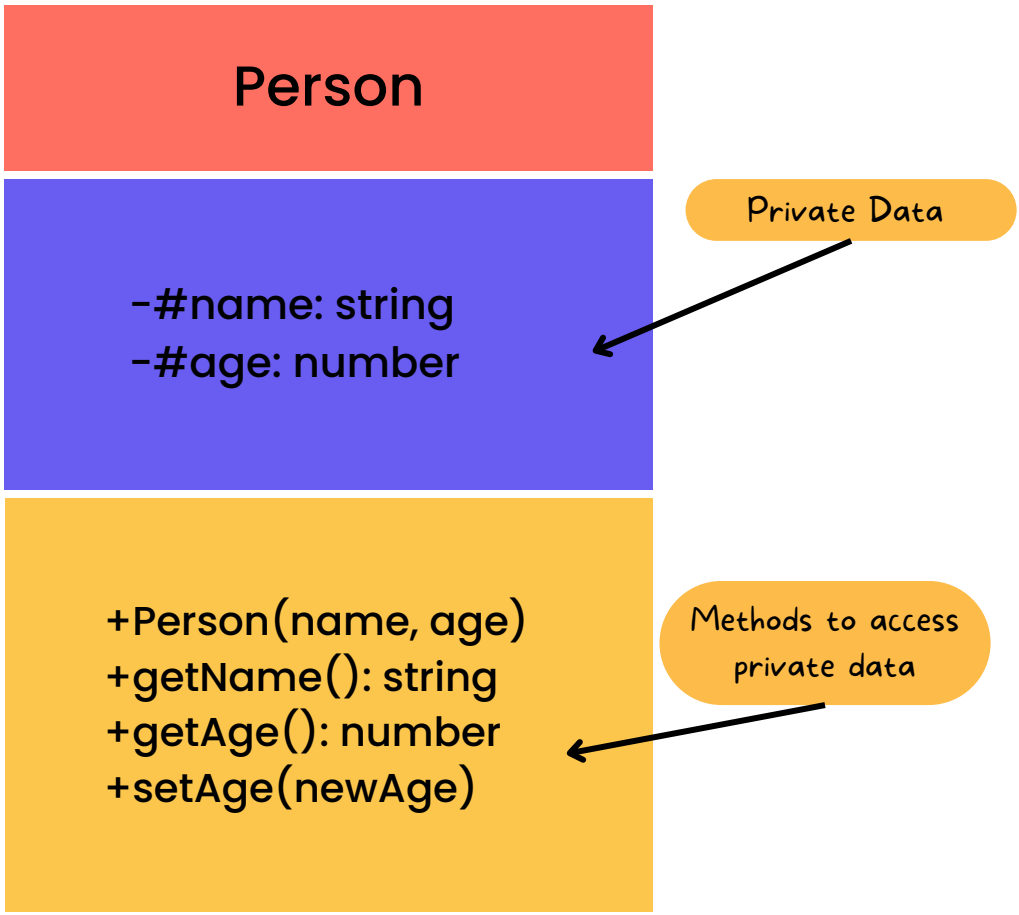
The example above works because JavaScript treats methods with different parameter lists as separate methods.

To simulate overloading in JavaScript, a common approach is to use a single method that checks the number and types of arguments passed, and then executes the appropriate logic based on those arguments.

- **Explain the concept of Encapsulation in object-oriented programming.**

Encapsulation is like a locked box where you can put your valuable items.

The box has a key (interface) that allows you to access the items inside, **but it doesn't let anyone else directly access or modify the items without using the key.**



In object-oriented programming, an object is like this locked box. It contains data (properties) and methods (functions) that operate on that data.

The data is kept private, just like the valuable items in the box.

The methods act as the key, allowing controlled access and modification of the data.

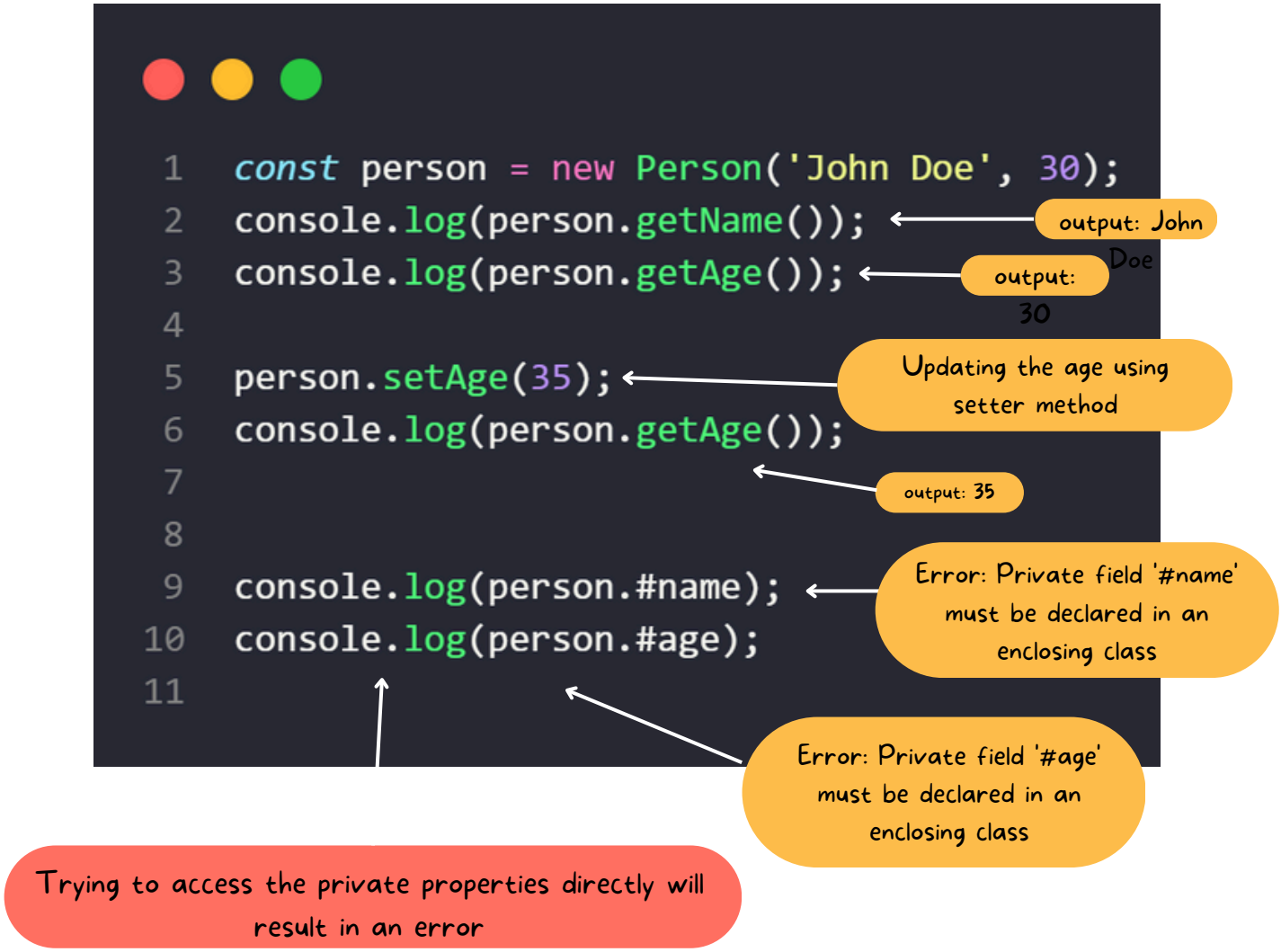

```
class Person {
  #name;
  #age;

  constructor(name, age) {
    this.#name = name;
    this.#age = age;
  }

  getName() {
    return this.#name;
  }

  getAge() {
    return this.#age;
  }

  setAge(newAge) {
    if (newAge > 0) {
      this.#age = newAge;
    } else {
      console.log('Invalid age');
    }
  }
}
```



In this example, the Person class has two private properties: #name and #age.

These properties are like valuable items in a locked box, and we can't access or modify them directly from outside the class.

The class provides getter methods (getName and getAge) to retrieve the values of these private properties.

These methods act like the key that allows us to access the data inside the box.

The class also has a setter method (setAge) to modify the #age property.

This method encapsulates the logic for validating the new age before updating it, ensuring data integrity.

Stay ahead with daily updates!

Follow us on social media for useful web development content.

@richwebdeveloper

@new_javascript

@developerupdates

@developerupdates

@_chetanmahajan

Note: This kit is continuously updated. Please continue to check Gumroad or our website (where you purchased this) for updated questions and answers. You will receive all updates for FREE

Download from our Website

WWW.DEVELOPERUPDATES.COM