{WDM} Learn To Code Web Dev Mastery

"JavaScript Most Important Interview Questions"

Basic Level:

1. What is JavaScript? How is it different from Java?
2. Explain the difference between `var`, `let`, and `const`.
3. What are the data types in JavaScript?
4. What is hoisting in JavaScript?
5. What are closures in JavaScript?
6. What is the difference between == and === in JavaScript?
7. Explain how this keyword works in JavaScript.
8. What is an event in JavaScript? How do you prevent default behavior?
9. What are arrow functions, and how are they different from regular functions?
10. What is NaN in JavaScript? How do you check if a value is NaN?

Intermediate Level:

1. What is the difference between synchronous and asynchronous programming in JavaScript?
2. Explain the concept of promises in JavaScript.
3. What are callback functions? How are they used?
4. What are JavaScript prototypes and how do they work?
5. What is the event loop in JavaScript? How does it work with asynchronous code?
6. Explain how `map()`, `filter()`, and `reduce()` functions work.
7. What is the purpose of async and `await` in JavaScript?
8. How does JavaScript handle memory management and garbage collection?

9. **What is the debounce function in JavaScript? How is it different from `throttle`?**
10. **What is destructuring in JavaScript? Provide examples of array and object destructuring.**

Advanced Level:

1. **Explain the concept of event delegation in JavaScript.**
2. **What are generators in JavaScript, and how do they differ from regular functions?**
3. **What are JavaScript modules, and how do you use `import` and `export`?**
4. **What is the `Proxy` object in JavaScript, and what are its use cases?**
5. **Explain currying in JavaScript.**
6. **What are the differences between deep cloning and shallow cloning in JavaScript? How would you implement both?**
7. **What is memoization, and how would you implement it in JavaScript?**
8. **Explain the concept of weak maps and weak sets in JavaScript.**
9. **What is the difference between function declarations and function expressions?**
10. **What is a JavaScript `Symbol`, and when would you use it?**
11. **How does the `apply`, `call`, and `bind` methods work in JavaScript? Provide examples.**
12. **What is the purpose of the async iterator, and how does it differ from a regular iterator?**
13. **How can you prevent object mutation in JavaScript?**
14. **Print all elements of the nested array `arr = [1, [2, [3, 4], 5], 6]`.**
15. **Reverse Words in a String In JavaScript.**
16. **Check for Balanced Parentheses.**
17. **Remove Duplicates from an Array.**
18. **Find the Largest Sum of Contiguous Subarray (Kadane's Algorithm)**
19. **Find the First Non-Repeated Character in a String**
20. **Given an array of n consecutive integers from 1 to n, where one integer is missing, write a JavaScript function to find the missing number with a time complexity of O(n).**

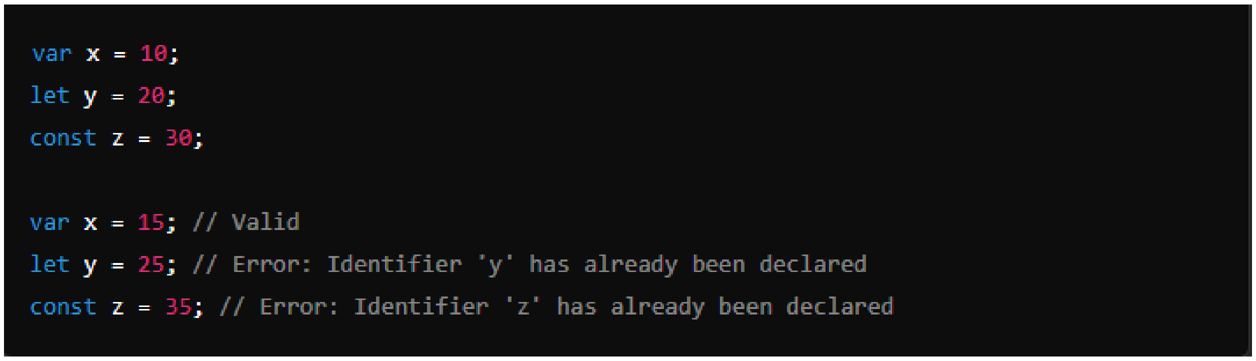
Basic Level:

1. What is JavaScript? How is it different from Java?

- **JavaScript** is a lightweight, interpreted programming language primarily used for enhancing interactivity on websites. It is a core technology of the World Wide Web, alongside HTML and CSS.
- **Differences from Java**:
    - **Type**: Java is a statically typed, compiled language, while JavaScript is a dynamically typed, interpreted language.
    - **Syntax**: Java uses class-based object-oriented programming, while JavaScript uses prototype-based object-oriented programming.
    - **Execution Environment**: Java runs on the Java Virtual Machine (JVM), while JavaScript runs in web browsers or on servers (like Node.js).

2. Explain the difference between `var`, `let`, and `const`.

- `var`:
    - Function-scoped or globally scoped.
    - Can be re-declared and updated.
- `let`:
    - Block-scoped.
    - Can be updated but not re-declared in the same scope.
- `const`:
    - Block-scoped.
    - Cannot be updated or re-declared. However, it can hold mutable objects.



```
var x = 10;
let y = 20;
const z = 30;

var x = 15; // Valid
let y = 25; // Error: Identifier 'y' has already been declared
const z = 35; // Error: Identifier 'z' has already been declared
```

3. What are the data types in JavaScript?

JavaScript has two main categories of data types:

- **Primitive Data Types**:
  - **Undefined**: A variable that has been declared but has not been assigned a value.
  - **Null**: Represents a deliberate non-value.
  - **Boolean**: Represents a logical entity and can have two values: `true` and `false`.
  - **Number**: Represents both integer and floating-point numbers.
  - **BigInt**: An arbitrary precision integer.
  - **String**: Represents a sequence of characters.
  - **Symbol**: A unique and immutable value often used as object property keys.
- **Reference Data Types**:
  - **Object**: A collection of key-value pairs.
  - **Array**: A special type of object used to store lists of values.
  - **Function**: A callable object.

4. What is hoisting in JavaScript?

- **Hoisting** is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during the compile phase. This means you can use variables and functions before they are declared in the code.

```
console.log(a); // Outputs: undefined (due to hoisting)
var a = 5;

foo(); // Outputs: "Hello!"
function foo() {
  console.log("Hello!");
}
```

5. What are closures in JavaScript?

- A **closure** is a function that retains access to its outer scope, even when the function is executed outside that scope. Closures are created every time a function is created, allowing for data encapsulation.

```
function outerFunction() {
  let outerVariable = 'I am outside!';

  return function innerFunction() {
    console.log(outerVariable); // Accesses outerVariable
  };
}

const inner = outerFunction();
inner(); // Outputs: "I am outside!"
```

6. What is the difference between `==` and `===` in JavaScript?

- `==` (Abstract Equality Operator):
  - Performs type coercion if the operands are of different types.
  - Example: `0 == '0'` evaluates to `true`.
- `===` (Strict Equality Operator):

- Checks for both value and type without coercion.
- Example: `0 === '0'` evaluates to `false`.

```
console.log(0 == '0');  // Outputs: true
console.log(0 === '0'); // Outputs: false
```

7. Explain how the `this` keyword works in JavaScript.

- The **`this`** keyword refers to the context in which a function is called. Its value is determined by how the function is invoked:
  - In a method, it refers to the object the method is called on.
  - In a regular function, it refers to the global object (or `undefined` in strict mode).
  - In an arrow function, `this` is lexically inherited from the enclosing scope.

```
const obj = {
  name: 'Alice',
  greet: function() {
    console.log(`Hello, ${this.name}`);
  }
};

obj.greet(); // Outputs: "Hello, Alice"

const greetFunc = obj.greet;
greetFunc(); // Outputs: "Hello, undefined" (in non-strict mode, refers to global object)

const arrowGreet = () => {
  console.log(`Hello, ${this.name}`); // 'this' is inherited from the outer scope
};
arrowGreet(); // Outputs: "Hello, undefined"
```

8. What is an event in JavaScript? How do you prevent default behavior?

- An **event** is an action or occurrence that happens in the browser, such as clicks, key presses, or page loads. You can prevent the default behavior of an event using the `event.preventDefault()` method.

```
document.querySelector('form').addEventListener('submit', function(event) {
  event.preventDefault(); // Prevents form submission
  console.log('Form submission prevented!');
});
```

9. What are arrow functions, and how are they different from regular functions?

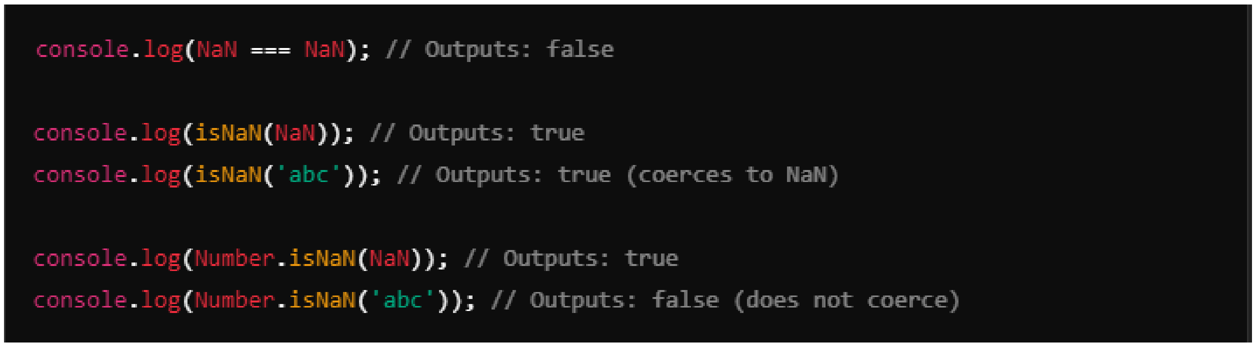
- **Arrow functions** are a more concise syntax for writing function expressions. They differ from regular functions in several ways:
    - Do not have their own `this` context (inherited from the enclosing scope).
    - Cannot be used as constructors (i.e., cannot use the `new` keyword).
    - Do not have the `arguments` object.

```
const regularFunction = function() {
  console.log(this); // Refers to the global object or undefined in strict mode
};

const arrowFunction = () => {
  console.log(this); // Inherits from the enclosing scope
};

regularFunction();
arrowFunction();
```

10. What is `NaN` in JavaScript? How do you check if a value is `NaN`?

- `NaN` stands for "Not-a-Number" and indicates that a value is not a valid number. It is a special numeric value resulting from invalid mathematical operations (like dividing zero by zero).

You can check if a value is `NaN` using the `isNaN()` function or `Number.isNaN()` for a more reliable check.



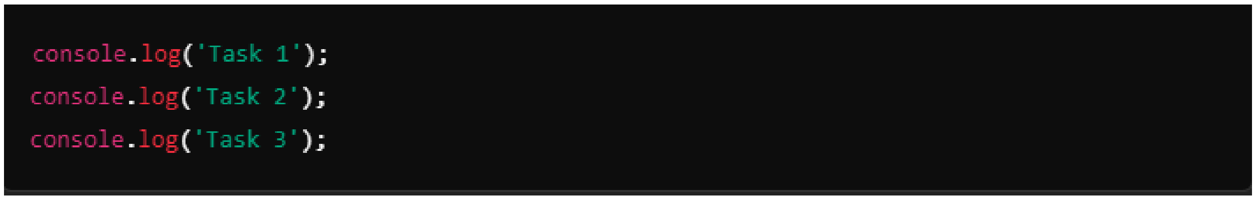
```javascript
console.log(NaN === NaN); // Outputs: false

console.log(isNaN(NaN)); // Outputs: true
console.log(isNaN('abc')); // Outputs: true (coerces to NaN)

console.log(Number.isNaN(NaN)); // Outputs: true
console.log(Number.isNaN('abc')); // Outputs: false (does not coerce)
```

Intermediate Level:

1. What is the difference between synchronous and asynchronous programming in JavaScript?

- **Synchronous programming** means that tasks are executed one after the other. Each task must finish before the next one starts. This can block the execution if a task takes a long time.
- **Asynchronous programming** allows tasks to run concurrently. This means that while one task is waiting (like fetching data), other tasks can continue executing without waiting.

Example of Synchronous Code:



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

Example of Asynchronous Code:

```
console.log('Task 1');
setTimeout(() => {
  console.log('Task 2 (after 2 seconds)');
}, 2000);
console.log('Task 3');
```

2. Explain the concept of promises in JavaScript.

- A **promise** is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to write cleaner code without nesting multiple callbacks.

States of a Promise:

- **Pending**: The initial state, neither fulfilled nor rejected.
- **Fulfilled**: The operation completed successfully.
- **Rejected**: The operation failed.

Example of a Promise:

```
const fetchData = new Promise((resolve, reject) => {
  const success = true; // Simulate success or failure

  if (success) {
    resolve('Data fetched successfully!'); // Fulfills the promise
  } else {
    reject('Error fetching data'); // Rejects the promise
  }
});

// Using the promise
fetchData
  .then((data) => console.log(data))  // If fulfilled
  .catch((error) => console.log(error)); // If rejected
```

3. What are callback functions? How are they used?

- A **callback function** is a function that is passed as an argument to another function and is executed after some operation has been completed. This is commonly used in asynchronous programming.

Example of a Callback Function:

```javascript
function fetchData(callback) {
  setTimeout(() => {
    const data = 'Data fetched!';
    callback(data); // Call the callback with data
  }, 2000);
}

// Using the callback function
fetchData((result) => {
  console.log(result); // Outputs: Data fetched!
});
```

4. What are JavaScript prototypes and how do they work?

- **Prototypes** are a mechanism by which JavaScript objects inherit features from one another. Every JavaScript object has a prototype object from which it can inherit properties and methods.

Example of Prototypes:

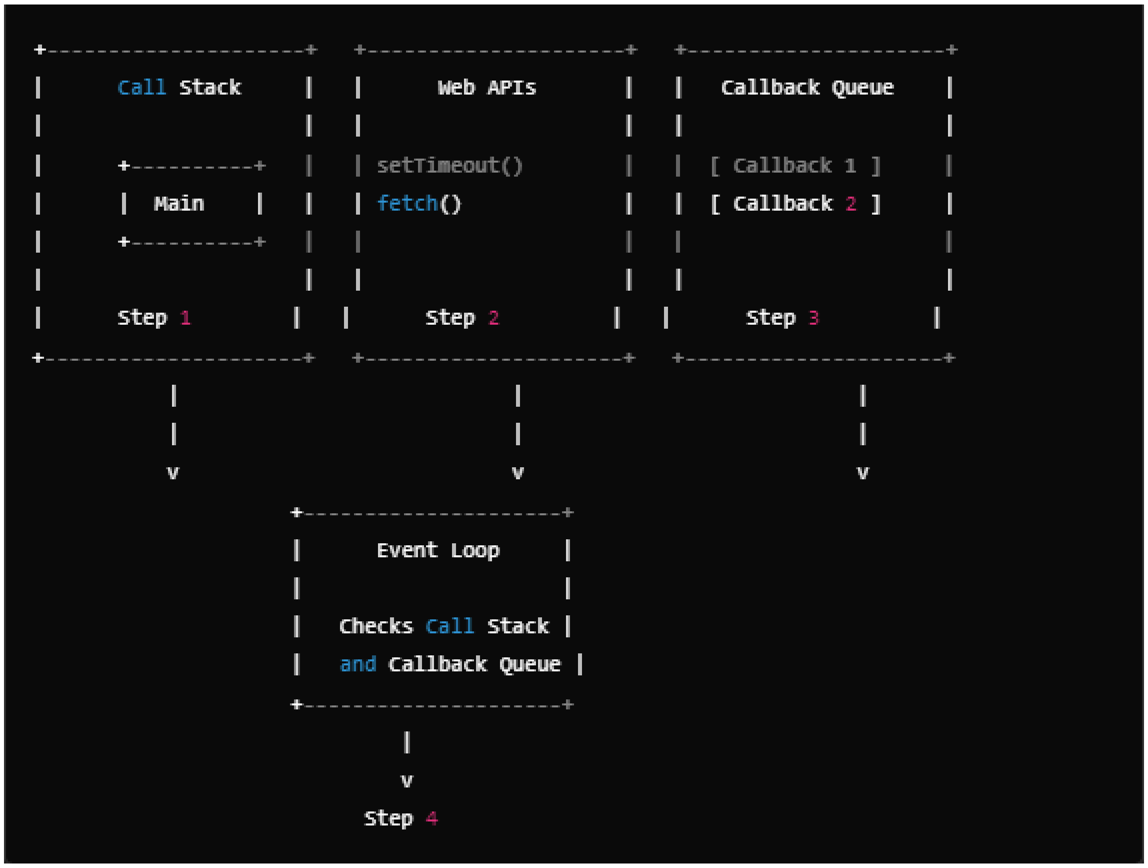
```javascript
function Person(name) {
  this.name = name;
}

// Adding a method to the prototype
Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name}`);
};

const alice = new Person('Alice');
alice.greet(); // Outputs: Hello, my name is Alice
```

5. What is the event loop in JavaScript? How does it work with asynchronous code?

- The **event loop** is a mechanism that allows JavaScript to perform non-blocking operations despite being single-threaded. It manages the execution of code, collects and processes events, and executes queued sub-tasks.
- When asynchronous tasks (like fetching data) are completed, their callbacks are added to the **callback queue**. The event loop continuously checks the call stack and the callback queue to execute tasks.

Illustration of the Event Loop:

1. Call Stack: Where functions are executed.
2. Web APIs: For handling asynchronous tasks (like `setTimeout`, `fetch`).
3. Callback Queue: Holds the callbacks from completed asynchronous tasks.
4. Event Loop: Checks if the call stack is empty, then moves the first callback from the callback queue to the call stack.

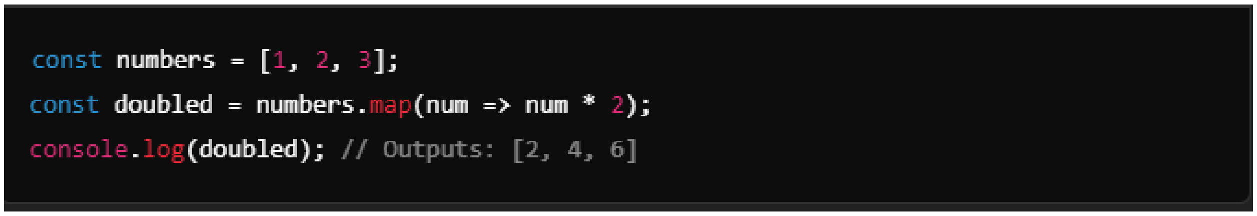


```
+---------------------+   +---------------------+   +---------------------+
|      Call Stack     |   |       Web APIs      |   |    Callback Queue   |
|                     |   |                     |   |                     |
|      +----------+   |   | setTimeout()        |   | [ Callback 1 ]      |
|      |  Main    |   |   | fetch()             |   | [ Callback 2 ]      |
|      +----------+   |   |                     |   |                     |
|                     |   |                     |   |                     |
|      Step 1        |   |      Step 2        |   |      Step 3        |
+---------------------+   +---------------------+   +---------------------+
           |                          |                          |
           |                          |                          |
           v                          v                          v
                     +---------------------+
                     |      Event Loop     |
                     |                     |
                     |   Checks Call Stack |
                     |   and Callback Queue |
                     +---------------------+
                              |
                              v
                           Step 4
```

6. Explain how `map()`, `filter()`, and `reduce()` functions work.

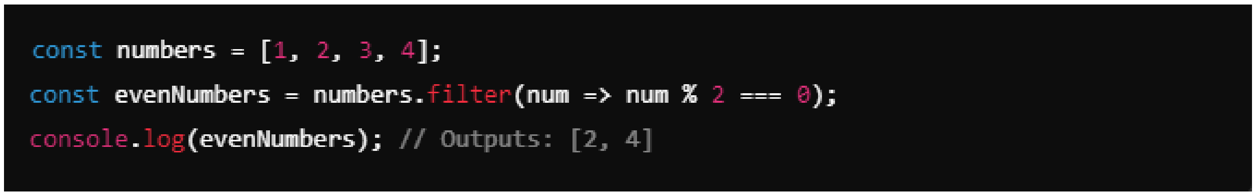
- These are higher-order functions for manipulating arrays:

`map()`: Creates a new array by applying a function to each element of the original array.
Example:



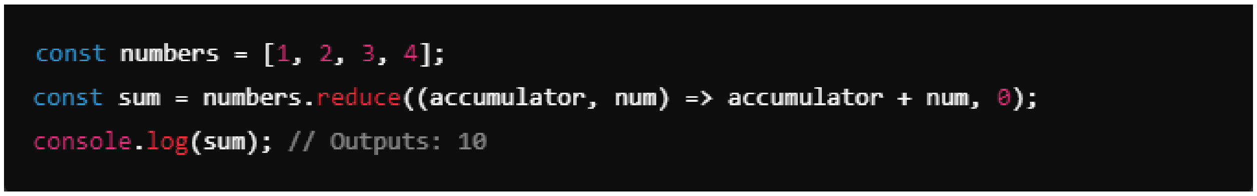
```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Outputs: [2, 4, 6]
```

`filter()`: Creates a new array with all elements that pass the test implemented by the provided function.
Example:



```
const numbers = [1, 2, 3, 4];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Outputs: [2, 4]
```

`reduce()`: Executes a reducer function on each element of the array, resulting in a single output value.
Example:

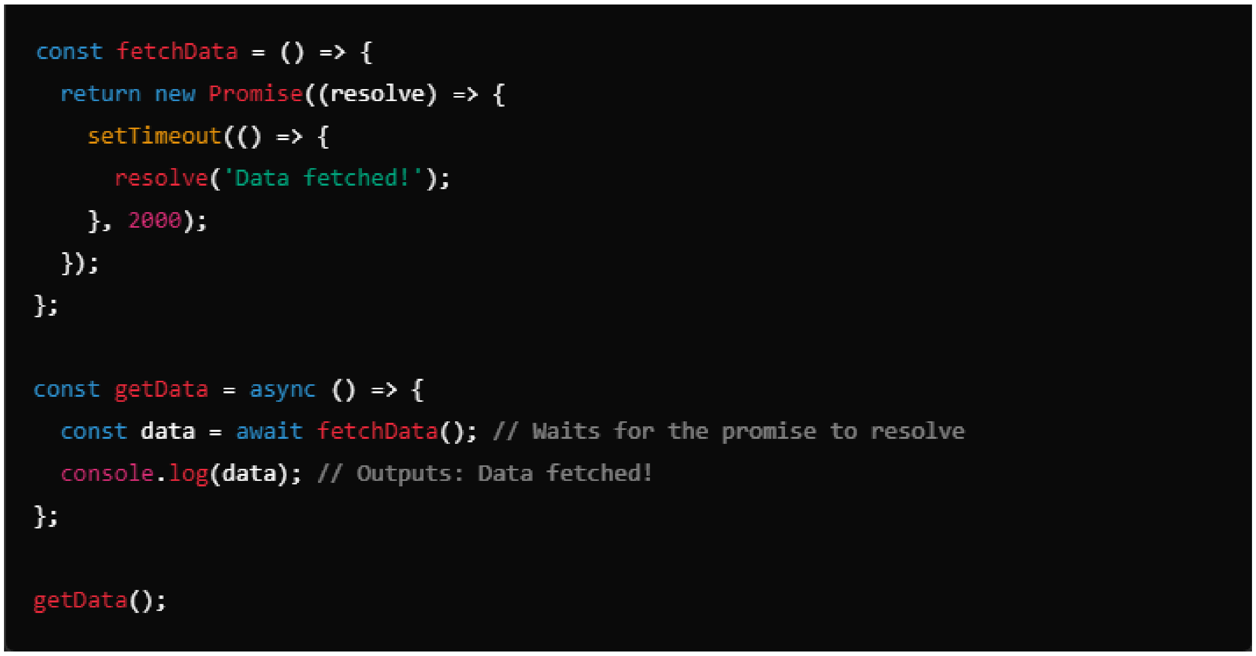


```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, num) => accumulator + num, 0);
console.log(sum); // Outputs: 10
```

7. What is the purpose of `async` and `await` in JavaScript?

- **`async`** and **`await`** are used to work with promises in a more readable way.
- `async` is a keyword that you can place before a function declaration to indicate that the function will return a promise.
- `await` is used inside an `async` function to pause execution until the promise is resolved.

Example:



```javascript
const fetchData = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Data fetched!');
    }, 2000);
  });
};

const getData = async () => {
  const data = await fetchData(); // Waits for the promise to resolve
  console.log(data); // Outputs: Data fetched!
};

getData();
```

8. How does JavaScript handle memory management and garbage collection?

- JavaScript has automatic memory management, which means that developers do not need to manually allocate and free memory.
- **Garbage Collection** is the process of automatically reclaiming memory that is no longer in use. It tracks objects that are no longer referenced in the code and frees their memory.
- JavaScript primarily uses **mark-and-sweep** garbage collection, where it marks reachable objects and sweeps away unmarked (unreachable) objects.

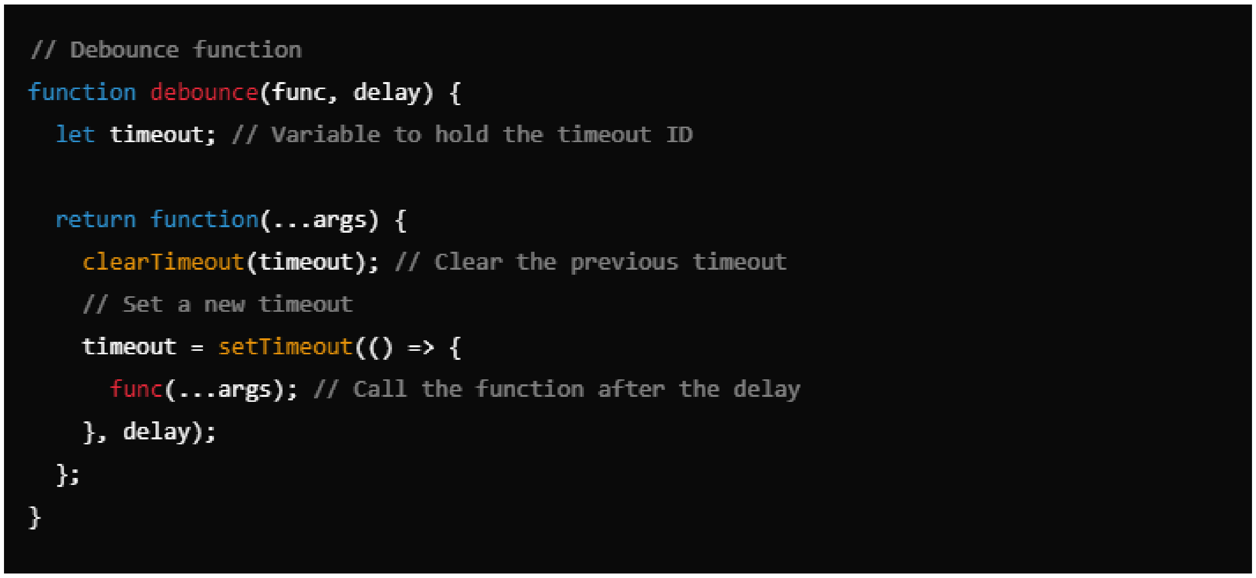
9. What is the debounce function in JavaScript? How is it different from throttle?

Debounce

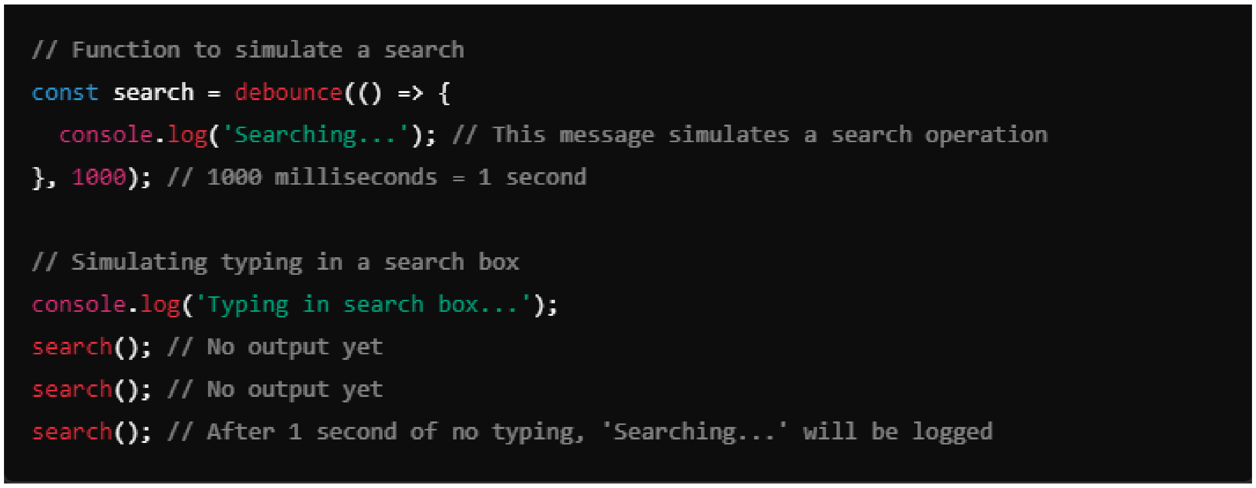
Main Definition: **Debounce** ensures that a function is executed only after a specified time has passed since it was last invoked. This is helpful when you want to avoid running a function too frequently, such as during user input.

Real-Life Example: Imagine you are typing in a search box. You don't want to search every time a letter is typed; instead, you want the search to occur after you stop typing for a moment.

Code Example:



```javascript
// Debounce function
function debounce(func, delay) {
  let timeout; // Variable to hold the timeout ID

  return function(...args) {
    clearTimeout(timeout); // Clear the previous timeout
    // Set a new timeout
    timeout = setTimeout(() => {
      func(...args); // Call the function after the delay
    }, delay);
  };
}
```



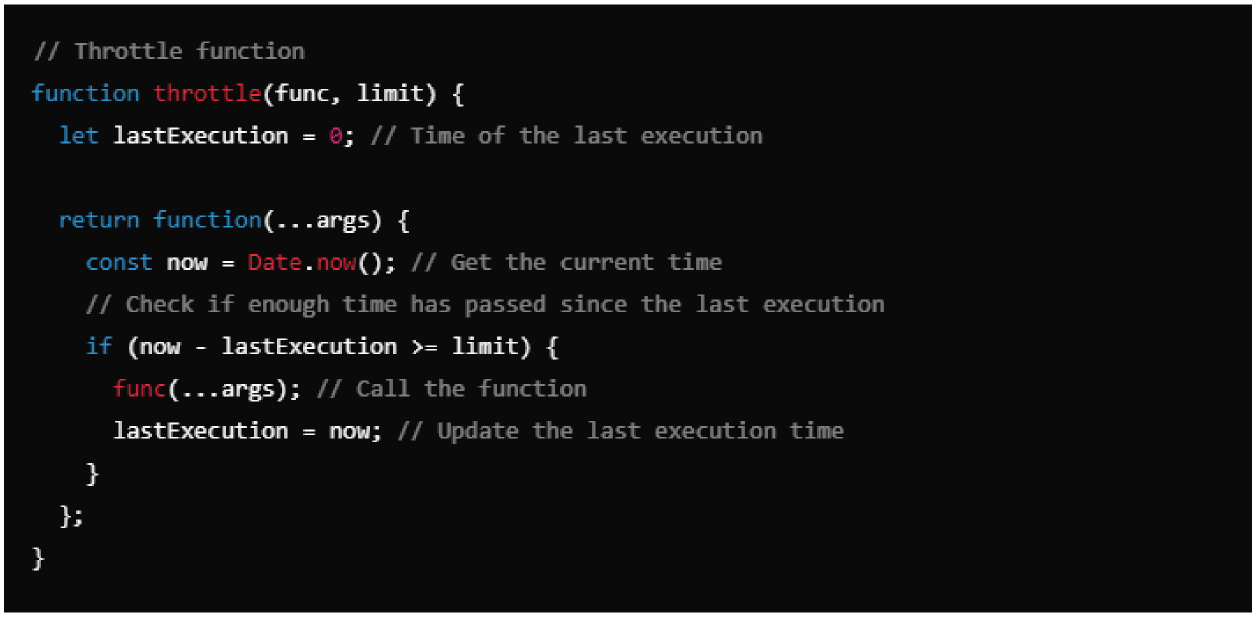
```javascript
// Function to simulate a search
const search = debounce(() => {
  console.log('Searching...'); // This message simulates a search operation
}, 1000); // 1000 milliseconds = 1 second

// Simulating typing in a search box
console.log('Typing in search box...');
search(); // No output yet
search(); // No output yet
search(); // After 1 second of no typing, 'Searching...' will be logged
```

Throttle

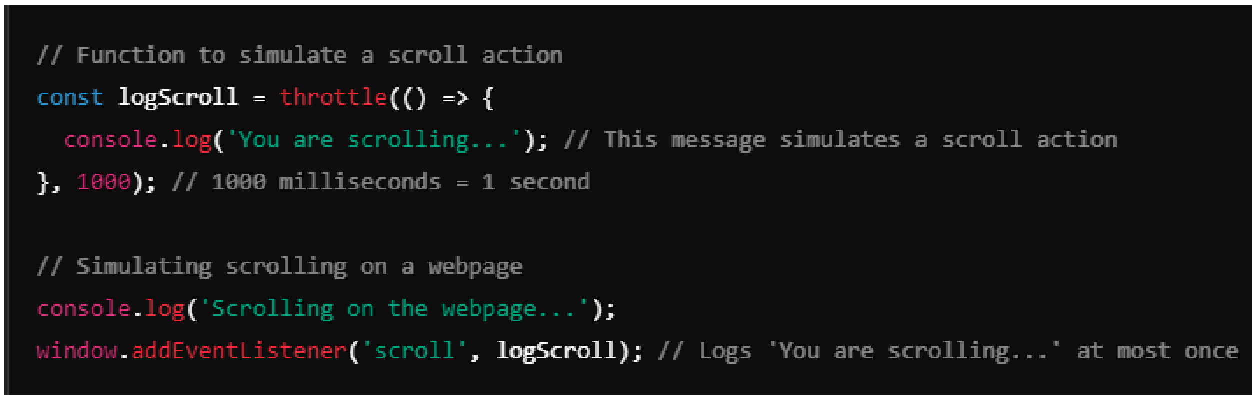
Main Definition: **Throttle** ensures that a function is executed at most once in a specified period of time. This is useful for events that can occur continuously, such as scrolling or resizing a window.

Real-Life Example: Think about a traffic light. It changes color at set intervals, regardless of whether cars are still arriving. Throttling means allowing an action to happen at a limited rate.

Code Example:



```javascript
// Throttle function
function throttle(func, limit) {
  let lastExecution = 0; // Time of the last execution

  return function(...args) {
    const now = Date.now(); // Get the current time
    // Check if enough time has passed since the last execution
    if (now - lastExecution >= limit) {
      func(...args); // Call the function
      lastExecution = now; // Update the last execution time
    }
  };
}
```



```javascript
// Function to simulate a scroll action
const logScroll = throttle(() => {
  console.log('You are scrolling...'); // This message simulates a scroll action
}, 1000); // 1000 milliseconds = 1 second

// Simulating scrolling on a webpage
console.log('Scrolling on the webpage...');
window.addEventListener('scroll', logScroll); // Logs 'You are scrolling...' at most once
```

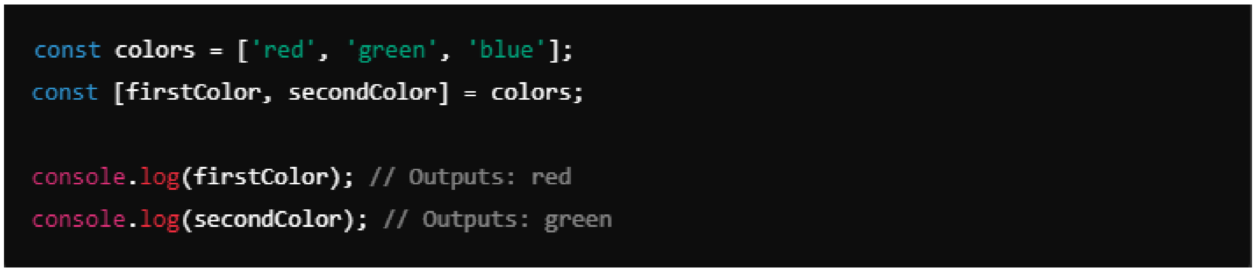
Summary

- **Debounce**: "Only search when I'm done typing!" It waits until you stop typing for a specified time before executing the search function.
- **Throttle**: "You can only drive this fast!" It limits how often a function can be executed, ensuring it runs at most once every specified interval.

10. What is destructuring in JavaScript? Provide examples of array and object destructuring.

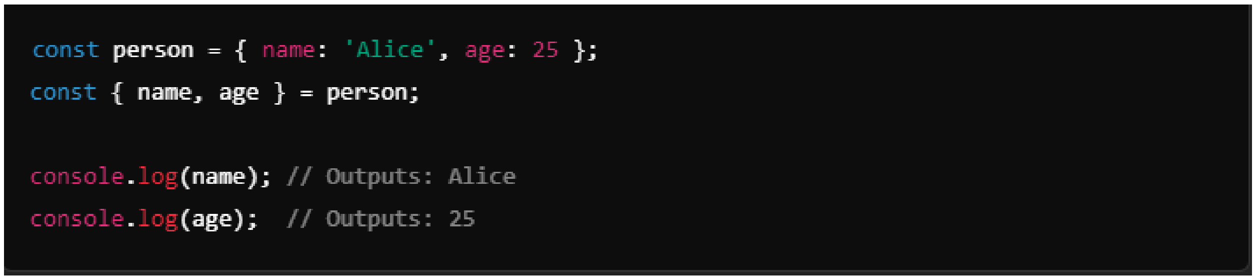
- **Destructuring** is a syntax that allows you to unpack values from arrays or properties from objects into distinct variables.

Array Destructuring Example:



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

console.log(firstColor); // Outputs: red
console.log(secondColor); // Outputs: green
```

Object Destructuring Example:



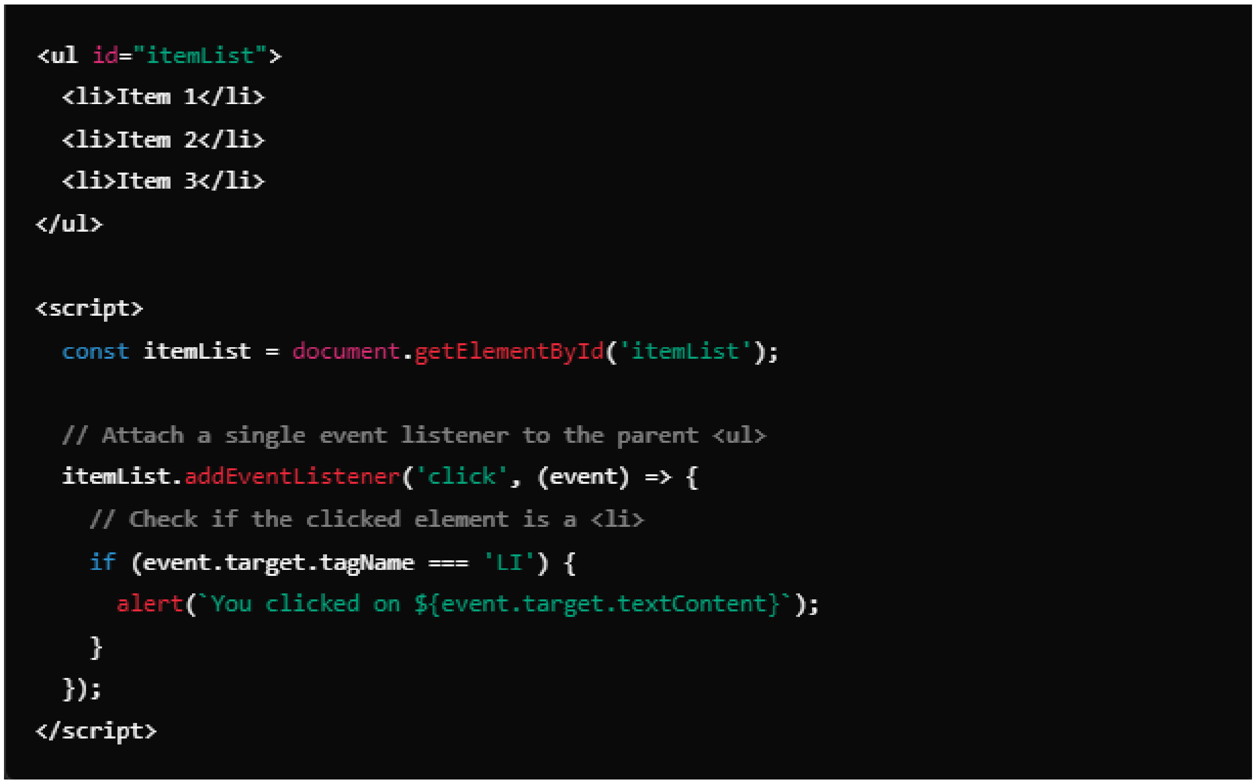
```javascript
const person = { name: 'Alice', age: 25 };
const { name, age } = person;

console.log(name); // Outputs: Alice
console.log(age);  // Outputs: 25
```

Advanced Level:

1. Explain the concept of event delegation in JavaScript.

- **Event delegation** is a technique where you attach a single event listener to a parent element instead of multiple listeners to individual child elements. This improves performance and simplifies event handling, especially when dealing with dynamically added elements.

Example:



```
<ul id="itemList">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>

<script>
  const itemList = document.getElementById('itemList');

  // Attach a single event listener to the parent <ul>
  itemList.addEventListener('click', (event) => {
    // Check if the clicked element is a <li>
    if (event.target.tagName === 'LI') {
      alert(`You clicked on ${event.target.textContent}`);
    }
  });
</script>
```

2. What are generators in JavaScript, and how do they differ from regular functions?

- **Generators** are a special type of function that can pause and resume execution. They are defined using the `function*` syntax and use the `yield` keyword to return values one at a time.
- Regular functions run to completion when called, while generators can yield multiple values over time.

Example of a Generator:

```
function* numberGenerator() {
  yield 1; // Pause and return 1
  yield 2; // Pause and return 2
  yield 3; // Pause and return 3
}

const generator = numberGenerator();
console.log(generator.next()); // Outputs: { value: 1, done: false }
console.log(generator.next()); // Outputs: { value: 2, done: false }
console.log(generator.next()); // Outputs: { value: 3, done: false }
console.log(generator.next()); // Outputs: { value: undefined, done: true }
```

3. What are JavaScript modules, and how do you use import and export?

- **JavaScript modules** allow you to break your code into separate files, making it easier to manage and reuse. You can export functions, objects, or variables from one module and import them into another.

Export Example:

```
// myModule.js
export const greeting = 'Hello, World!';
export function sayHello() {
  console.log(greeting);
}
```

Import Example:

```
// main.js
import { greeting, sayHello } from './myModule.js';

console.log(greeting); // Outputs: Hello, World!
sayHello(); // Outputs: Hello, World!
```

4. What is the Proxy object in JavaScript, and what are its use cases?

Definition: A **Proxy** is a JavaScript object that allows you to create a wrapper around another object, enabling you to intercept and redefine fundamental operations for that object. This includes property access, assignment, enumeration, and function invocation. Proxies are useful for implementing custom behaviors such as logging, validation, and more.

Real-Life Example: Bank Account

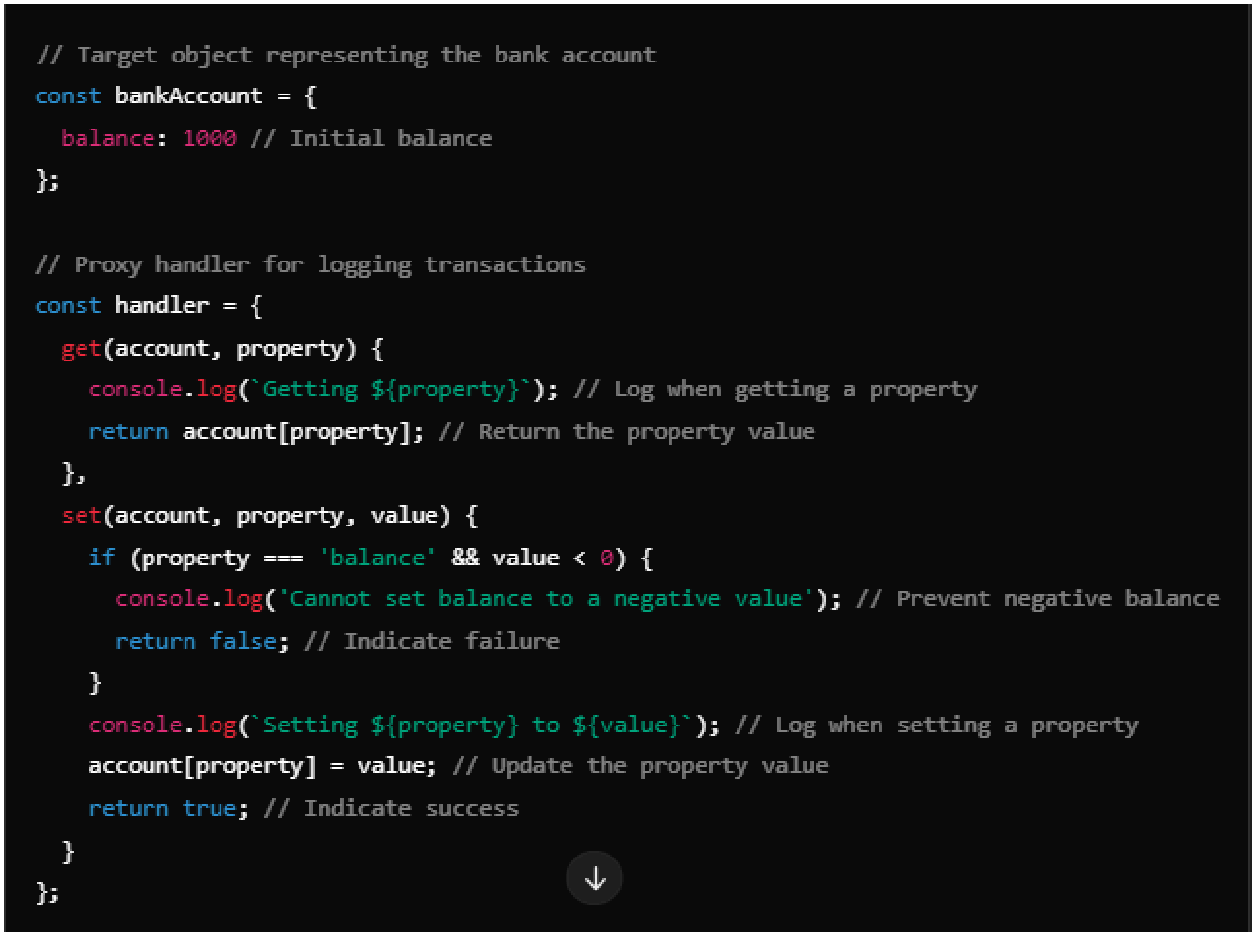
Let's consider a bank account system where you want to track deposits and withdrawals for security purposes. By using a Proxy, you can log these actions whenever they occur.

Definitions of `get` and `set`

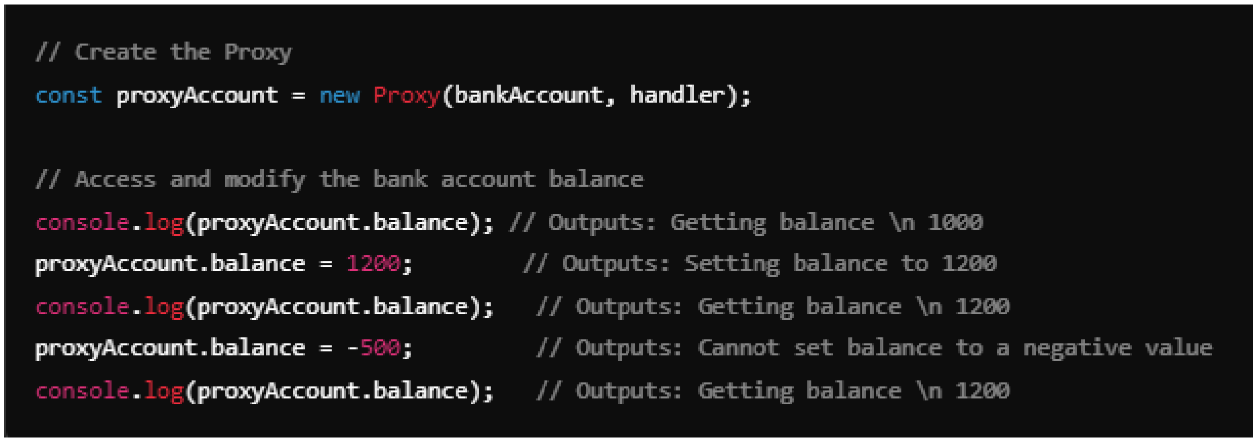
- `get`: Intercepts property access, allowing you to define custom behavior when reading a property.
- `set`: Intercepts property assignment, enabling you to define custom behavior when updating a property.

Code Example

Here's how you could implement a simple bank account with a Proxy:



```javascript
// Target object representing the bank account
const bankAccount = {
  balance: 1000 // Initial balance
};

// Proxy handler for logging transactions
const handler = {
  get(account, property) {
    console.log(`Getting ${property}`); // Log when getting a property
    return account[property]; // Return the property value
  },
  set(account, property, value) {
    if (property === 'balance' && value < 0) {
      console.log('Cannot set balance to a negative value'); // Prevent negative balance
      return false; // Indicate failure
    }
    console.log(`Setting ${property} to ${value}`); // Log when setting a property
    account[property] = value; // Update the property value
    return true; // Indicate success
  }
};
```



```javascript
// Create the Proxy
const proxyAccount = new Proxy(bankAccount, handler);

// Access and modify the bank account balance
console.log(proxyAccount.balance); // Outputs: Getting balance \n 1000
proxyAccount.balance = 1200;        // Outputs: Setting balance to 1200
console.log(proxyAccount.balance);   // Outputs: Getting balance \n 1200
proxyAccount.balance = -500;         // Outputs: Cannot set balance to a negative value
console.log(proxyAccount.balance);   // Outputs: Getting balance \n 1200
```

5. Explain currying in JavaScript.

- **Currying** is a functional programming technique where a function is transformed into a sequence of functions, each taking a single argument. This allows you to create more reusable and configurable functions.

Example:

```
function multiply(a) {
  return function(b) {
    return a * b;
  };
}

const double = multiply(2); // Creates a function that doubles a number
console.log(double(5)); // Outputs: 10
console.log(multiply(3)(4)); // Outputs: 12
```

6. What are the differences between deep cloning/copy and shallow cloning/copy in JavaScript? How would you implement both?

- **Shallow Cloning** creates a new object but only copies the references of nested objects, not the actual objects. Changes to nested objects in the clone affect the original object.

Shallow Clone Example:

```
const original = { name: 'Alice', address: { city: 'Wonderland' } };
const shallowClone = { ...original };

shallowClone.address.city = 'New City'; // Affects original
console.log(original.address.city); // Outputs: New City
```

- **Deep Cloning** creates a new object and recursively copies all objects and values, meaning that changes to nested objects in the clone do not affect the original object.

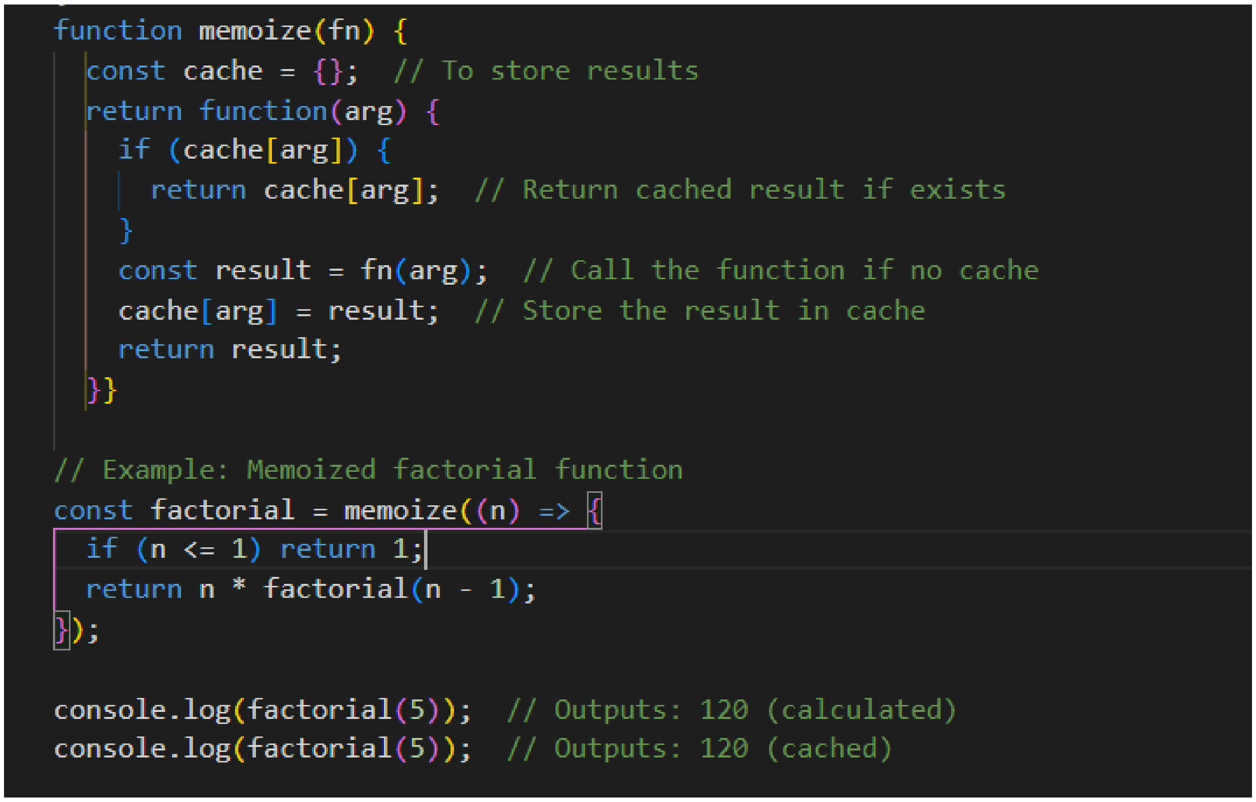
Deep Clone Example:

```
const original = { name: 'Alice', address: { city: 'Wonderland' } };
const deepClone = JSON.parse(JSON.stringify(original)); // Simple deep clone

deepClone.address.city = 'New City'; // Does not affect original
console.log(original.address.city); // Outputs: Wonderland
```

7. What is memoization, and how would you implement it in JavaScript?

- **Memoization** is an optimization technique used to speed up functions by caching their results based on input arguments. If the function is called again with the same arguments, it returns the cached result instead of recalculating it.

Example:

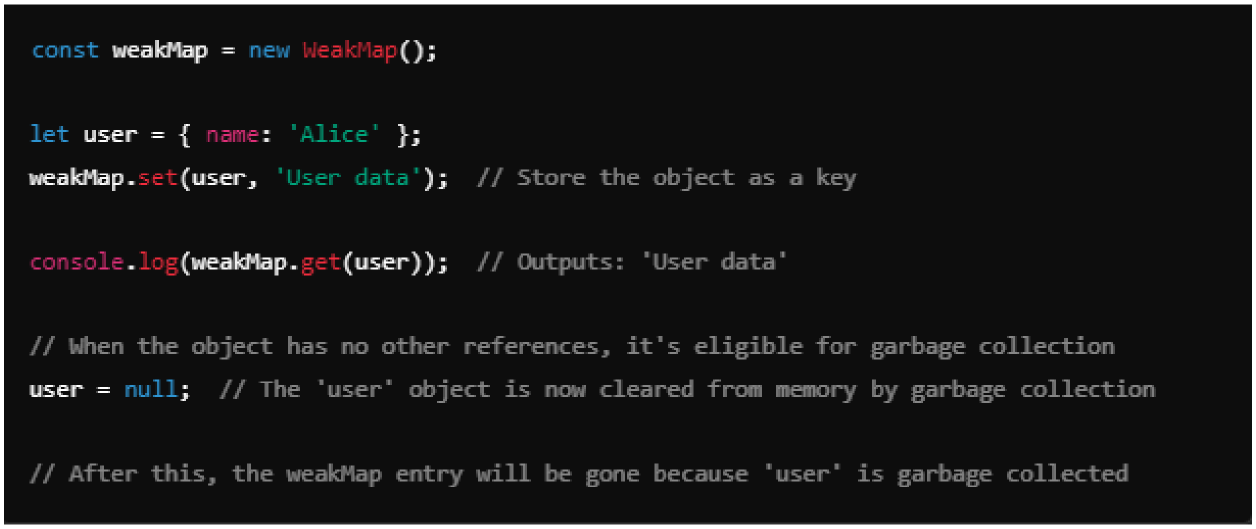


```
function memoize(fn) {
  const cache = {};  // To store results
  return function(arg) {
    if (cache[arg]) {
      return cache[arg];  // Return cached result if exists
    }
    const result = fn(arg);  // Call the function if no cache
    cache[arg] = result;  // Store the result in cache
    return result;
  }}

// Example: Memoized factorial function
const factorial = memoize((n) => {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
});

console.log(factorial(5));  // Outputs: 120 (calculated)
console.log(factorial(5));  // Outputs: 120 (cached)
```

8. Explain the concept of weak maps and weak sets in JavaScript.

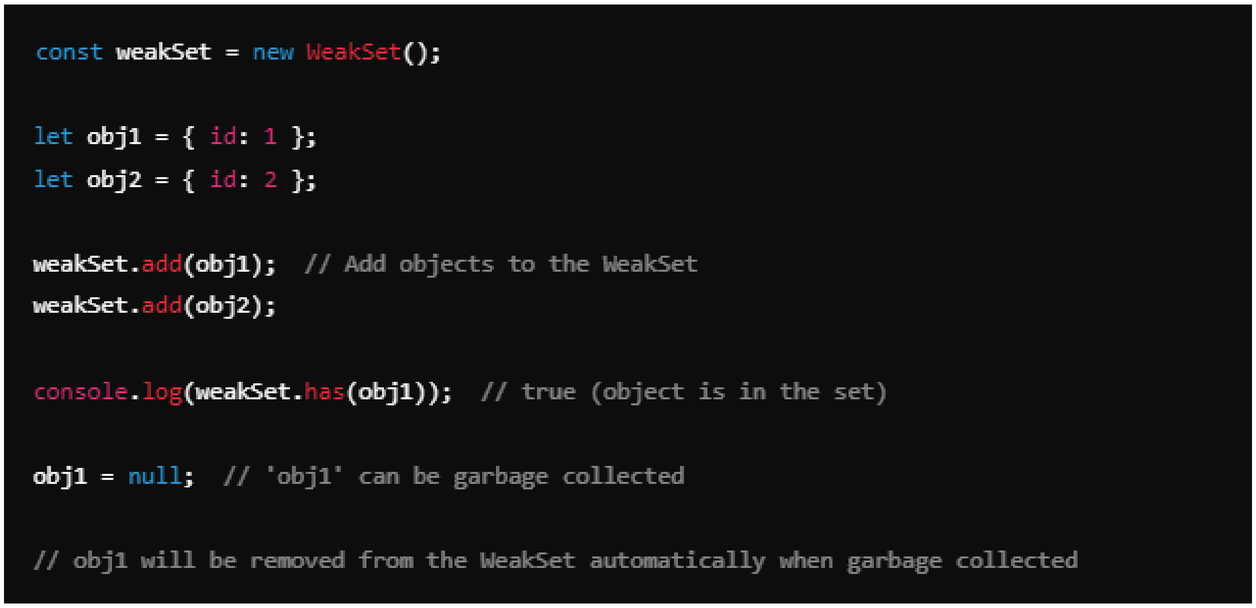
- **WeakMap** and **WeakSet** are collections that allow you to store objects weakly. This means that if there are no other references to the objects stored in a WeakMap or WeakSet, they can be garbage collected, preventing memory leaks.
- **WeakMap**: Stores key-value pairs where keys are objects and values can be any type.
- **WeakSet**: Stores unique objects (no primitive values).

Example of WeakMap:



```javascript
const weakMap = new WeakMap();

let user = { name: 'Alice' };
weakMap.set(user, 'User data');  // Store the object as a key

console.log(weakMap.get(user));  // Outputs: 'User data'

// When the object has no other references, it's eligible for garbage collection
user = null;  // The 'user' object is now cleared from memory by garbage collection

// After this, the weakMap entry will be gone because 'user' is garbage collected
```

Example of WeakSet:



```javascript
const weakSet = new WeakSet();

let obj1 = { id: 1 };
let obj2 = { id: 2 };

weakSet.add(obj1);  // Add objects to the WeakSet
weakSet.add(obj2);

console.log(weakSet.has(obj1));  // true (object is in the set)

obj1 = null;  // 'obj1' can be garbage collected

// obj1 will be removed from the WeakSet automatically when garbage collected
```

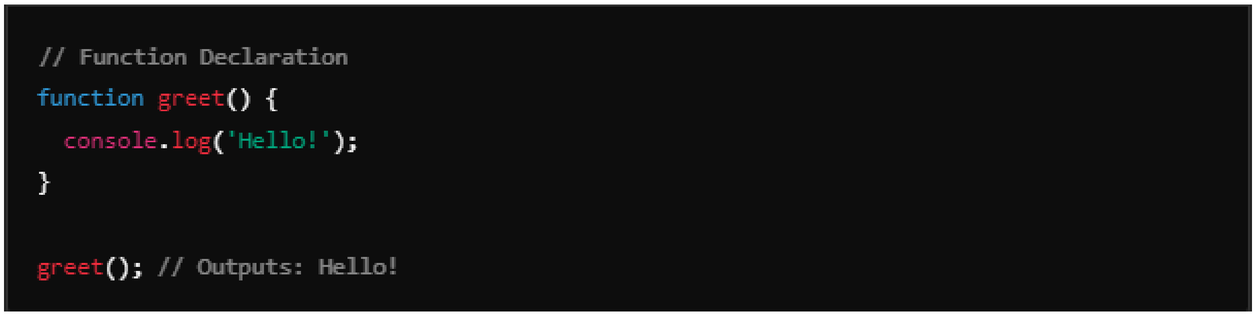
Real-Life Analogy:

- Imagine you have a sticky note (the object) that you temporarily attach to a fridge (WeakMap/WeakSet). Once you no longer need the note, you can throw it away. Similarly, WeakMap/WeakSet will automatically remove the object when it's no longer needed or referenced.

WeakMap and WeakSet are particularly useful in cases where you want to manage objects without worrying about memory leaks, such as managing DOM elements or caching data for objects that may be removed or no longer used.

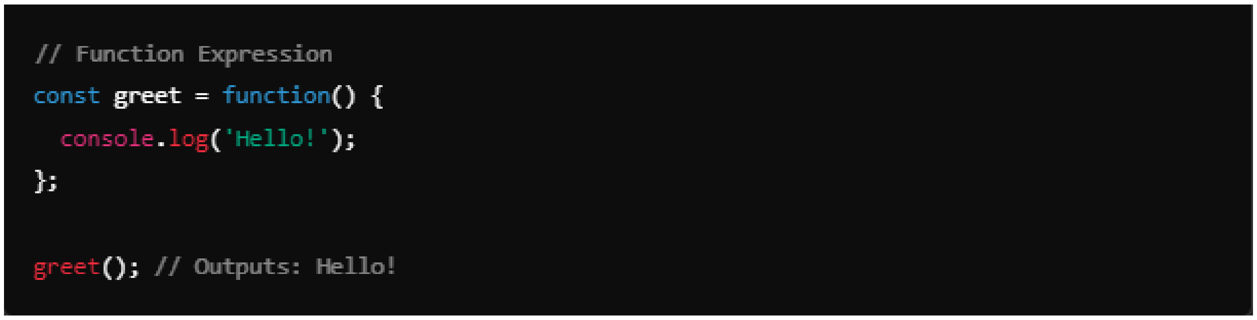
9. What is the difference between function declarations and function expressions?

- **Function Declaration**: A function defined using the `function` keyword followed by a name. It is hoisted, meaning it can be called before its definition in the code.



```javascript
// Function Declaration
function greet() {
  console.log('Hello!');
}

greet(); // Outputs: Hello!
```

- **Function Expression**: A function defined within an expression, often assigned to a variable. It is not hoisted, meaning it can only be called after its definition.



```javascript
// Function Expression
const greet = function() {
  console.log('Hello!');
};

greet(); // Outputs: Hello!
```

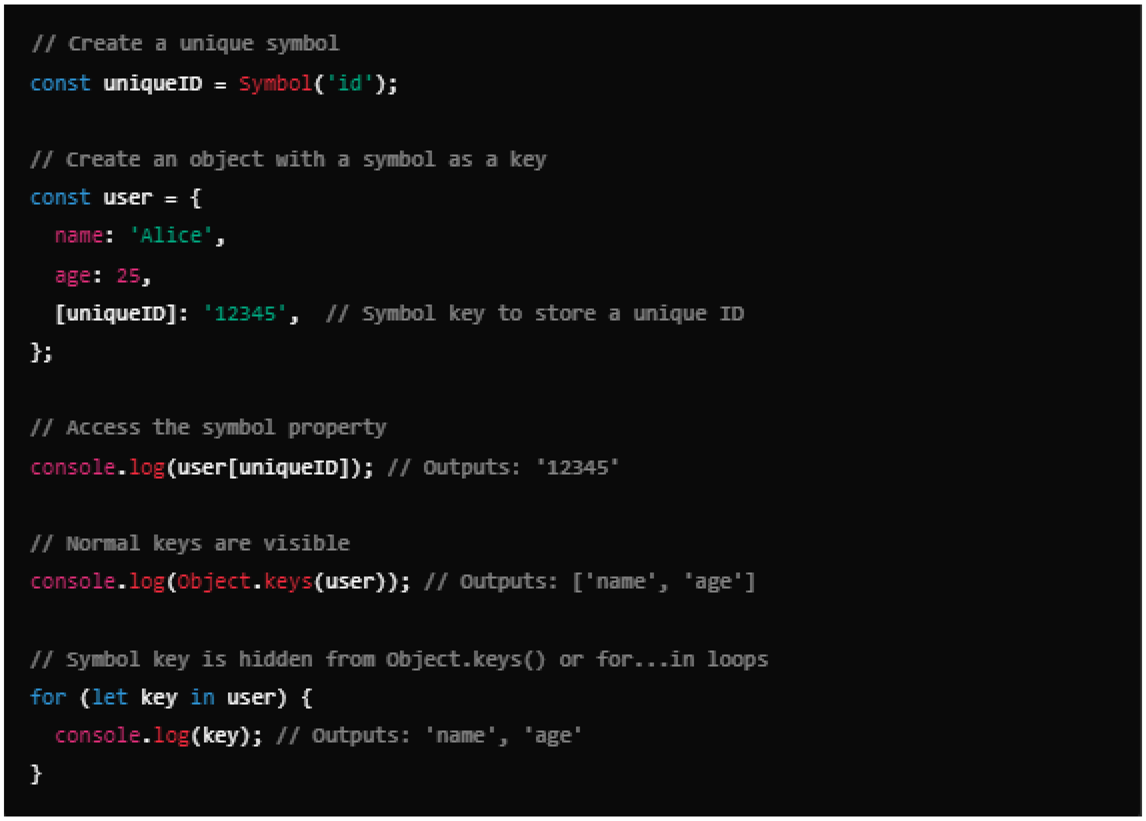
10. What is a JavaScript Symbol, and when would you use it?

A **Symbol** is a unique, immutable primitive value in JavaScript, typically used as a key for object properties. Each Symbol is guaranteed to be unique, even if they have the same description. They are useful when you want to create private or hidden properties that won't clash with other properties, even if they share the same name.

Key Points:

- **Uniqueness**: Every Symbol is unique.
- **Use case**: Symbols are often used as keys in objects to avoid property name collisions and ensure that properties are hidden from methods like `Object.keys()`.

Example Code:



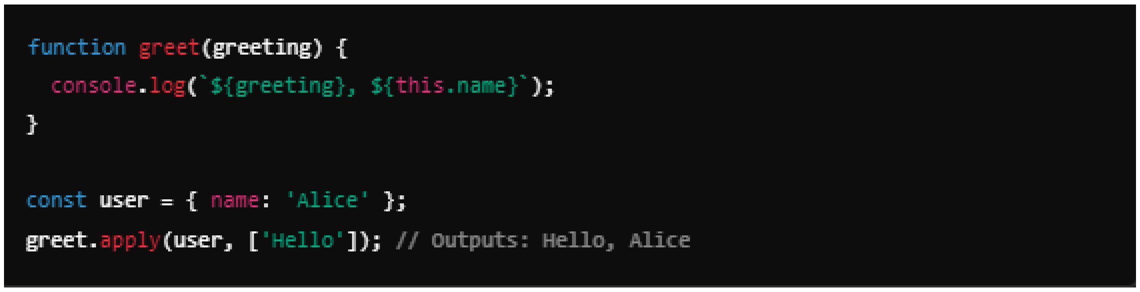
```javascript
// Create a unique symbol
const uniqueID = Symbol('id');

// Create an object with a symbol as a key
const user = {
  name: 'Alice',
  age: 25,
  [uniqueID]: '12345',  // Symbol key to store a unique ID
};

// Access the symbol property
console.log(user[uniqueID]); // Outputs: '12345'

// Normal keys are visible
console.log(Object.keys(user)); // Outputs: ['name', 'age']

// Symbol key is hidden from Object.keys() or for...in loops
for (let key in user) {
  console.log(key); // Outputs: 'name', 'age'
}
```

Where would you use Symbols?

- **Avoiding Name Clashes**: When multiple parts of your codebase may use the same property names, Symbols ensure there are no collisions.
- **Creating Hidden or Private Properties**: Since Symbols are not enumerable, they can be used to store metadata or hidden values in objects that you don't want to expose accidentally.
- **Implementing Protocols**: Symbols are useful in libraries where you want to implement custom behaviors without conflicting with existing object properties (e.g., iterators using `Symbol.iterator`).

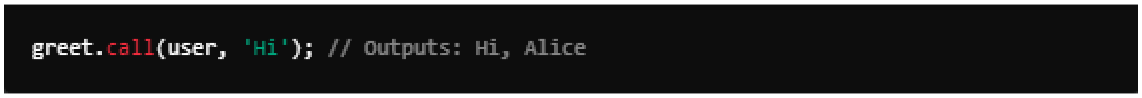
11. How do the apply, call, and bind methods work in JavaScript? Provide examples.

- **apply()**: Calls a function with a given `this` value and arguments provided as an array.



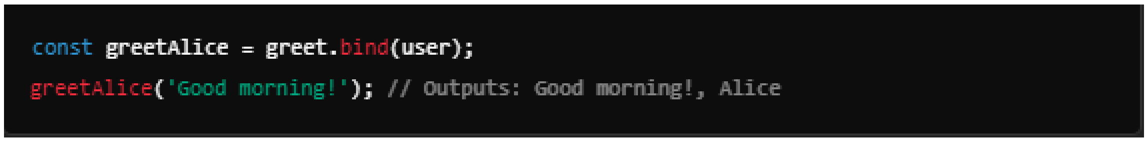
```
function greet(greeting) {
  console.log(`${greeting}, ${this.name}`);
}

const user = { name: 'Alice' };
greet.apply(user, ['Hello']); // Outputs: Hello, Alice
```

- **call()**: Similar to `apply()`, but takes arguments individually.



```
greet.call(user, 'Hi'); // Outputs: Hi, Alice
```

- **bind()**: Creates a new function that, when called, has its `this` keyword set to the provided value.



```
const greetAlice = greet.bind(user);
greetAlice('Good morning!'); // Outputs: Good morning!, Alice
```

12. What is the purpose of the async iterator, and how does it differ from a regular iterator?

- An **async iterator** is used to iterate over data asynchronously, such as when fetching data from an API. It allows you to work with promises in a `for...of` loop.
- A **regular iterator** works synchronously and cannot handle promises.

Example of an Async Iterator:

```
async function* asyncGenerator() {
  const data = ['Alice', 'Bob', 'Charlie'];
  for (const name of data) {
    // Simulate an asynchronous operation
    await new Promise(resolve => setTimeout(resolve, 1000));
    yield name;
  }
}

(async () => {
  for await (const name of asyncGenerator()) {
    console.log(name); // Outputs names with a 1-second delay between each
  }
})();
```

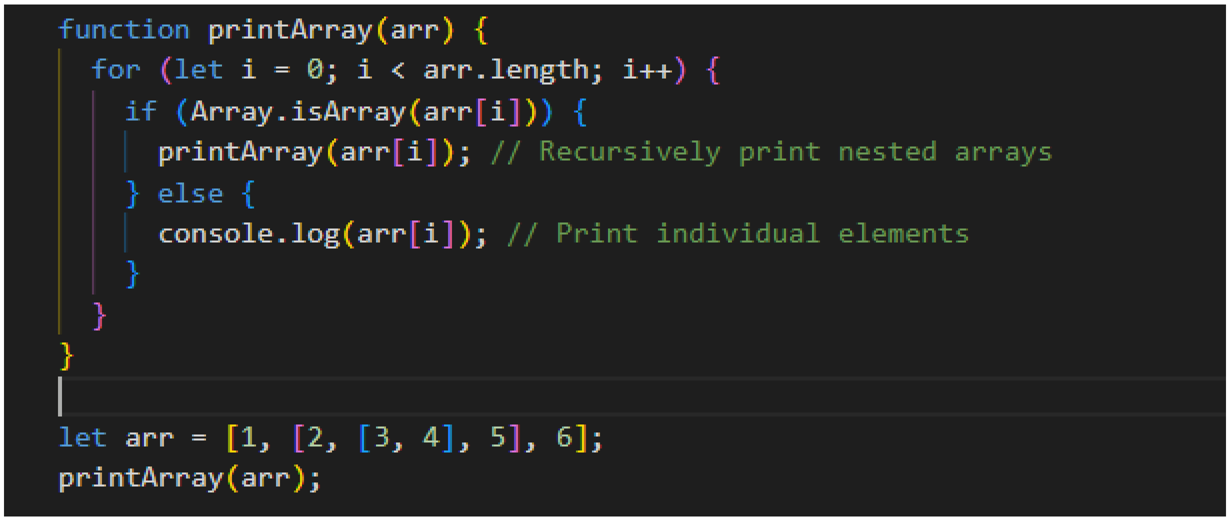
13. How can you prevent object mutation in JavaScript?

- To prevent object mutation, you can use the following techniques:
  - Use `Object.freeze()` to make an object immutable.
  - Use spread operators or methods like `Object.assign()` to create shallow copies.
  - For deep immutability, you can create utility functions that recursively freeze objects.

Example with Object.freeze():

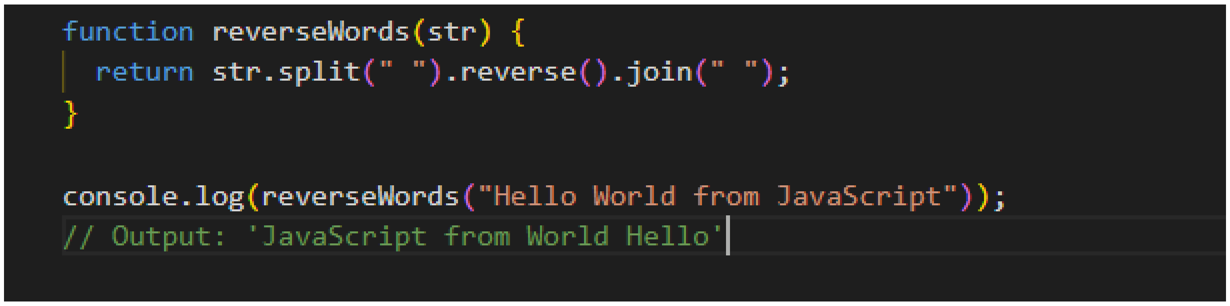
```
const original = { name: 'Alice' };
const frozenObject = Object.freeze(original);

frozenObject.name = 'Bob'; // This will not change the name property

console.log(frozenObject.name); // Outputs: Alice
```

14. **Print all elements of the nested array `arr = [1, [2, [3, 4], 5], 6]`.**



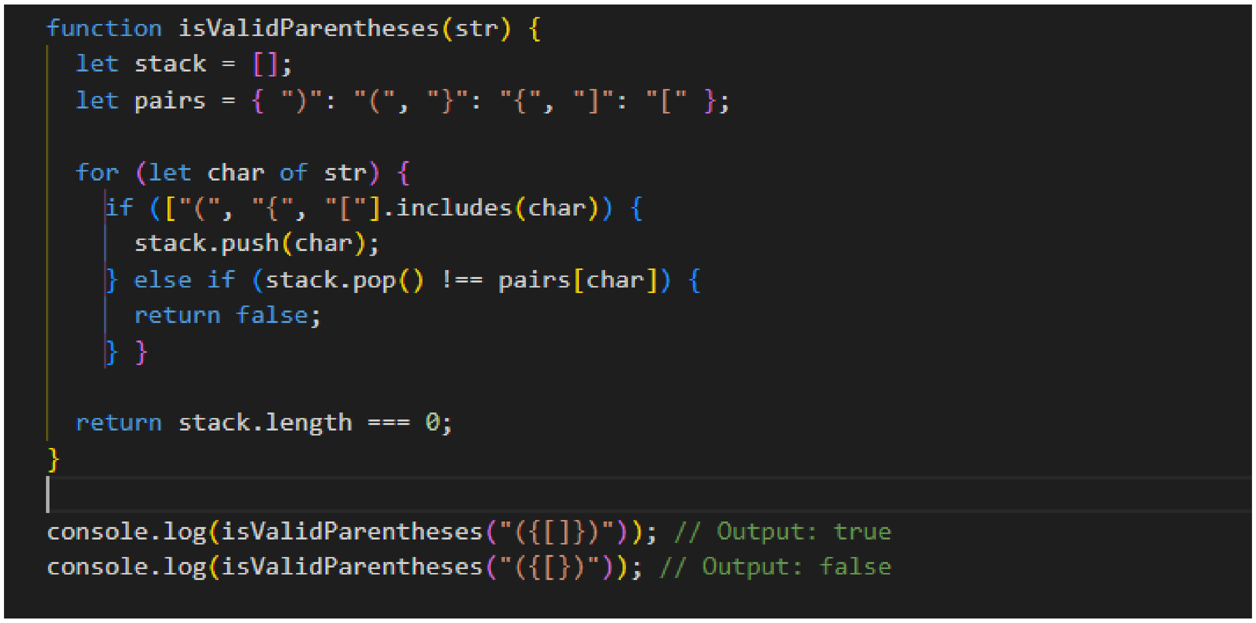
```
function printArray(arr) {
  for (let i = 0; i < arr.length; i++) {
    if (Array.isArray(arr[i])) {
      printArray(arr[i]); // Recursively print nested arrays
    } else {
      console.log(arr[i]); // Print individual elements
    }
  }
}

let arr = [1, [2, [3, 4], 5], 6];
printArray(arr);
```

15. Reverse Words in a String In JavaScript.



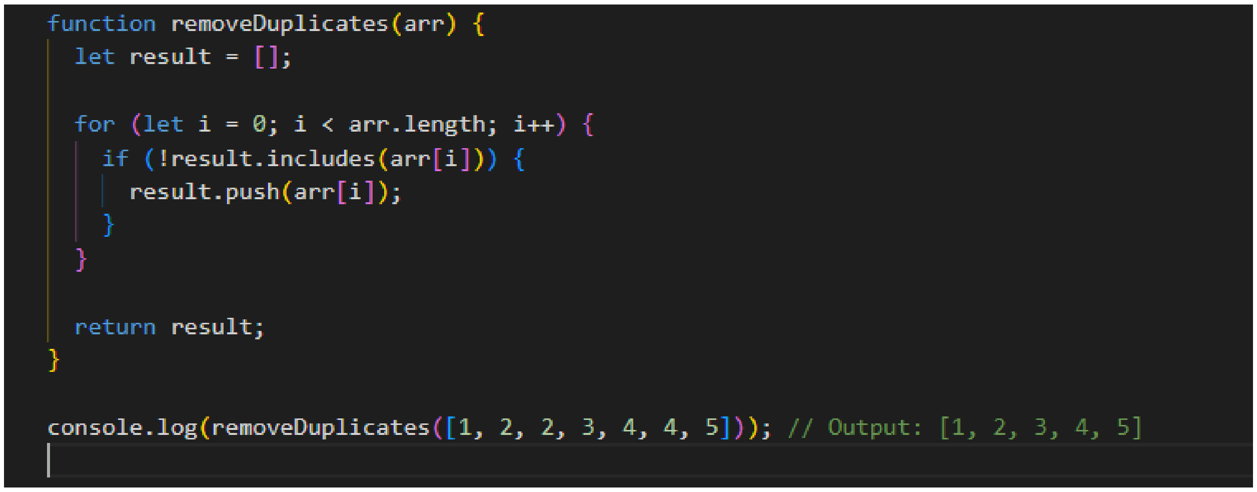
```
function reverseWords(str) {
  return str.split(" ").reverse().join(" ");
}

console.log(reverseWords("Hello World from JavaScript"));
// Output: 'JavaScript from World Hello'
```

16. Check for Balanced Parentheses.



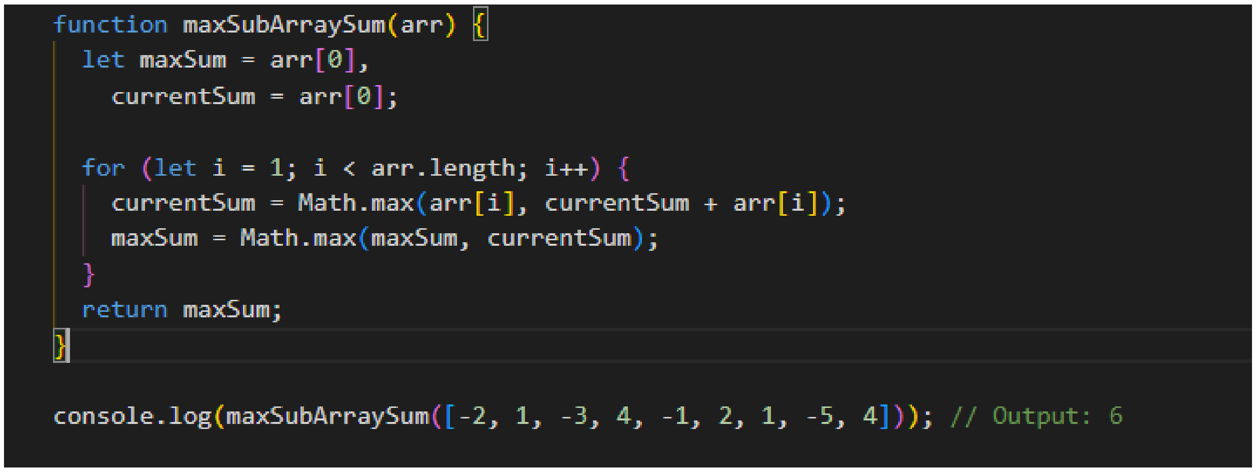
```
function isValidParentheses(str) {
  let stack = [];
  let pairs = { ")": "(", "}": "{", "]": "[" };

  for (let char of str) {
    if (["(", "{", "["].includes(char)) {
      stack.push(char);
    } else if (stack.pop() !== pairs[char]) {
      return false;
    } }

  return stack.length === 0;
}

console.log(isValidParentheses("({[]})")); // Output: true
console.log(isValidParentheses("({[})")); // Output: false
```

17. Remove Duplicates from an Array.



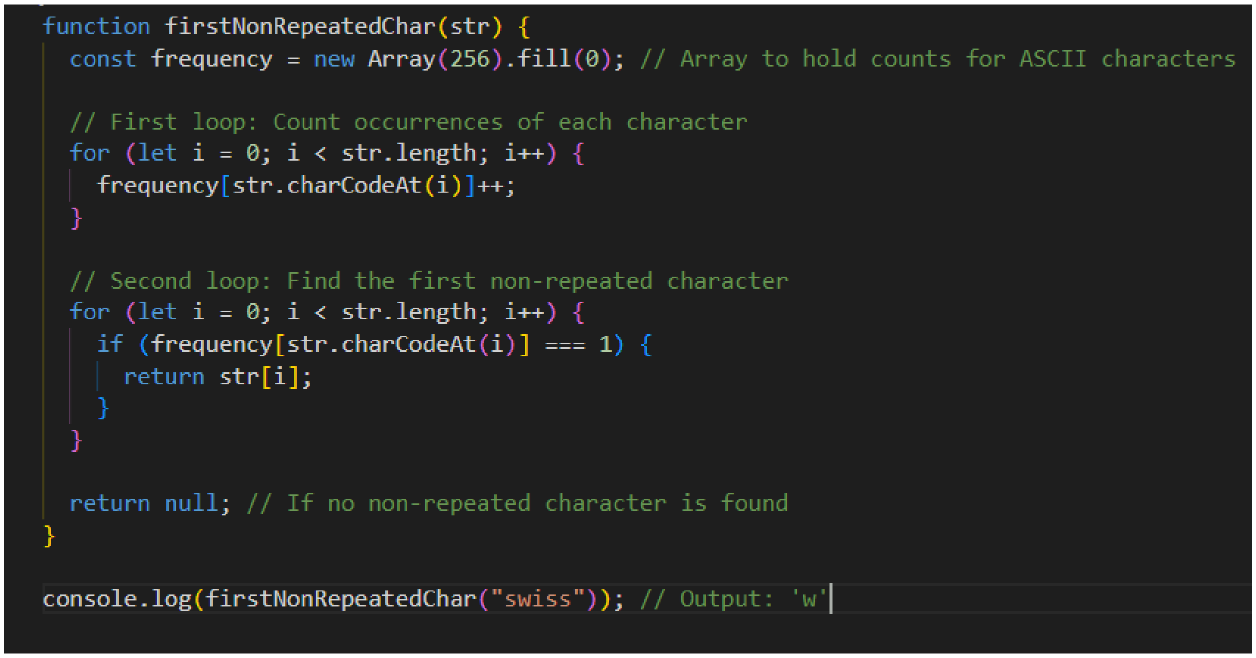
```
function removeDuplicates(arr) {
  let result = [];

  for (let i = 0; i < arr.length; i++) {
    if (!result.includes(arr[i])) {
      result.push(arr[i]);
    }
  }

  return result;
}

console.log(removeDuplicates([1, 2, 2, 3, 4, 4, 5])); // Output: [1, 2, 3, 4, 5]
```

18. Find the Largest Sum of Contiguous Subarray (Kadane's Algorithm)



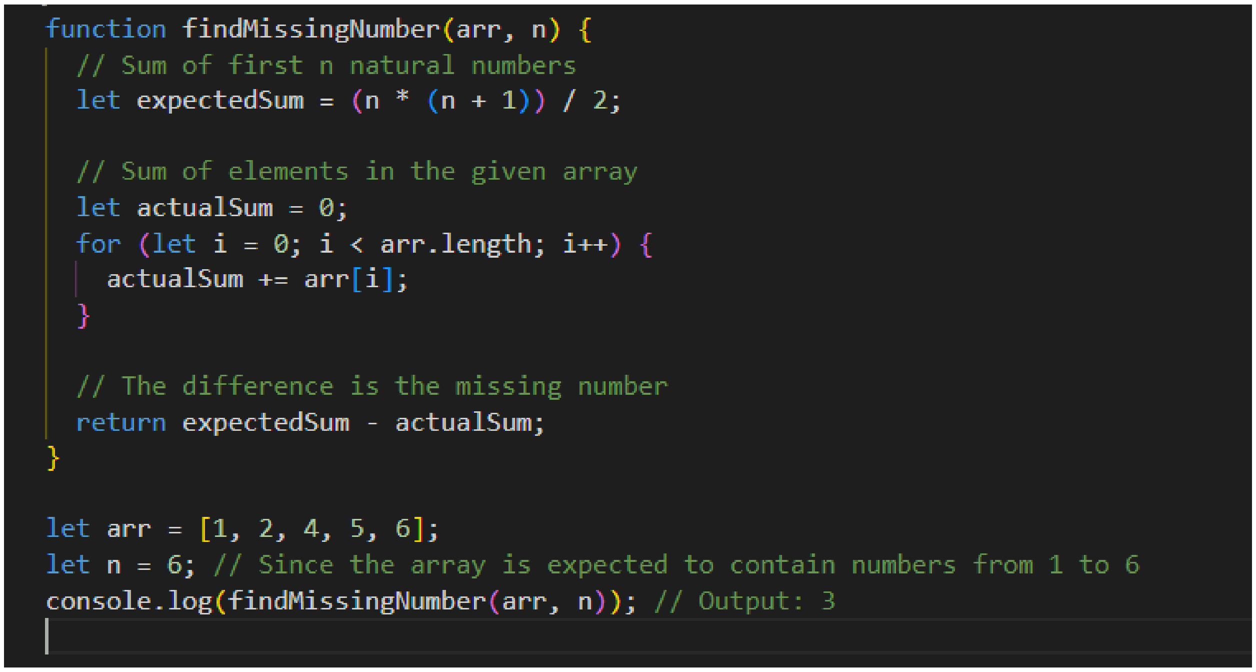
```javascript
function maxSubArraySum(arr) {
  let maxSum = arr[0],
    currentSum = arr[0];

  for (let i = 1; i < arr.length; i++) {
    currentSum = Math.max(arr[i], currentSum + arr[i]);
    maxSum = Math.max(maxSum, currentSum);
  }
  return maxSum;
}

console.log(maxSubArraySum([-2, 1, -3, 4, -1, 2, 1, -5, 4])); // Output: 6
```

19. Find the First Non-Repeated Character in a String.



```javascript
function firstNonRepeatedChar(str) {
  const frequency = new Array(256).fill(0); // Array to hold counts for ASCII characters

  // First loop: Count occurrences of each character
  for (let i = 0; i < str.length; i++) {
    frequency[str.charCodeAt(i)]++;
  }

  // Second loop: Find the first non-repeated character
  for (let i = 0; i < str.length; i++) {
    if (frequency[str.charCodeAt(i)] === 1) {
      return str[i];
    }
  }

  return null; // If no non-repeated character is found
}

console.log(firstNonRepeatedChar("swiss")); // Output: 'w'
```

20. Given an array of n consecutive integers from 1 to n, where one integer is missing, write a JavaScript function to find the missing number with a time complexity of O(n).



```javascript
function findMissingNumber(arr, n) {
  // Sum of first n natural numbers
  let expectedSum = (n * (n + 1)) / 2;

  // Sum of elements in the given array
  let actualSum = 0;
  for (let i = 0; i < arr.length; i++) {
    actualSum += arr[i];
  }

  // The difference is the missing number
  return expectedSum - actualSum;
}

let arr = [1, 2, 4, 5, 6];
let n = 6; // Since the array is expected to contain numbers from 1 to 6
console.log(findMissingNumber(arr, n)); // Output: 3
```