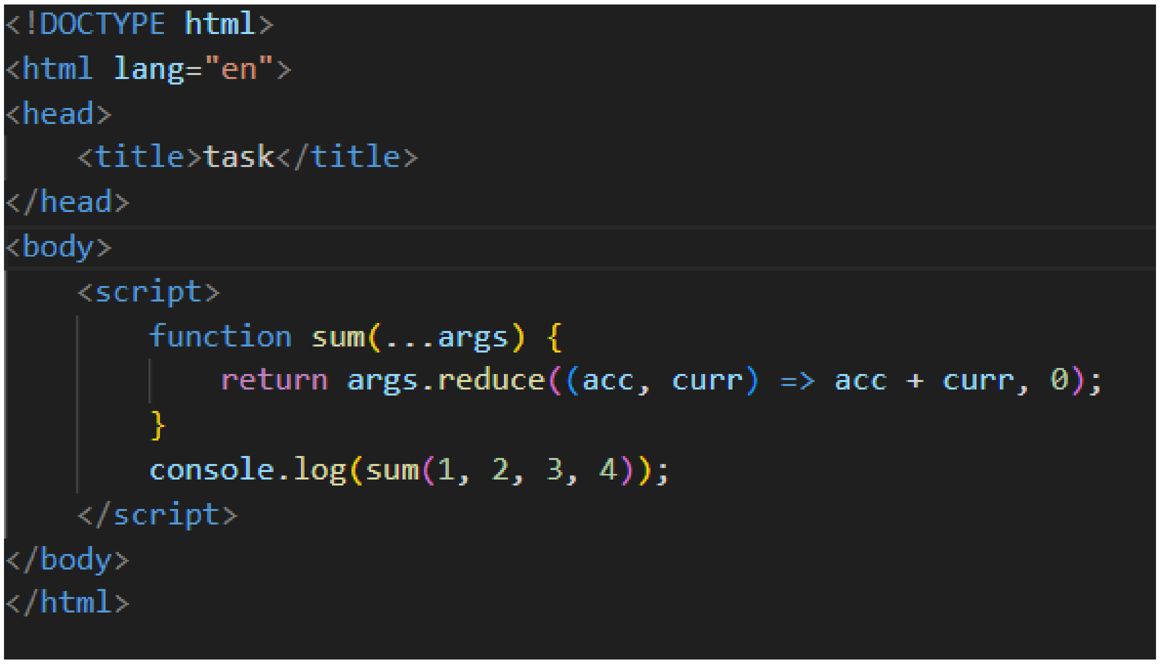
2. JSON and variable length arguments/spread syntax:

Task 1:

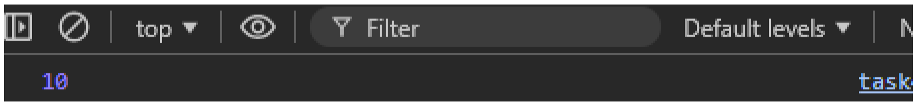
Write a function that takes an arbitrary number of arguments and returns their sum.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>task</title>
</head>
<body>
    <script>
        function sum(...args) {
            return args.reduce((acc, curr) => acc + curr, 0);
        }
        console.log(sum(1, 2, 3, 4));
    </script>
</body>
</html>
```

output:

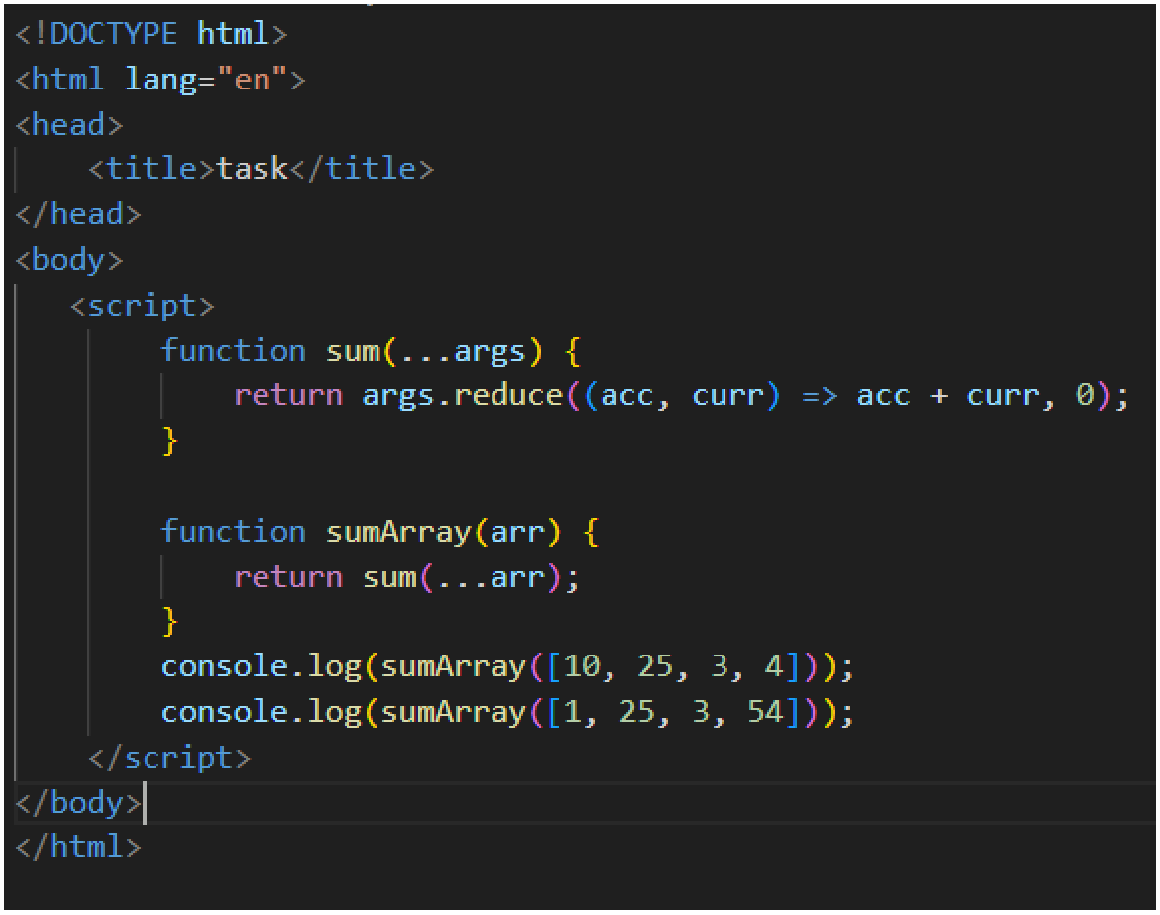


```
10                                                      task
```

Task 2:

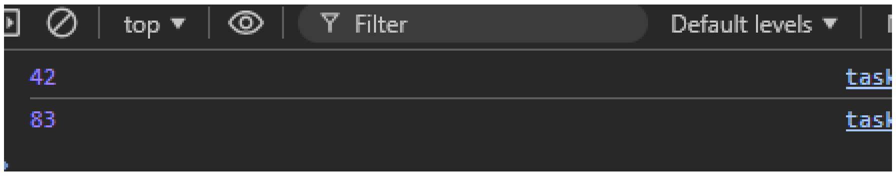
Modify a function to accept an array of numbers and return their sum using the spread syntax.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>task</title>
</head>
<body>
   <script>
        function sum(...args) {
            return args.reduce((acc, curr) => acc + curr, 0);
        }

        function sumArray(arr) {
            return sum(...arr);
        }
        console.log(sumArray([10, 25, 3, 4]));
        console.log(sumArray([1, 25, 3, 54]));
    </script>
</body>
</html>
```

output:

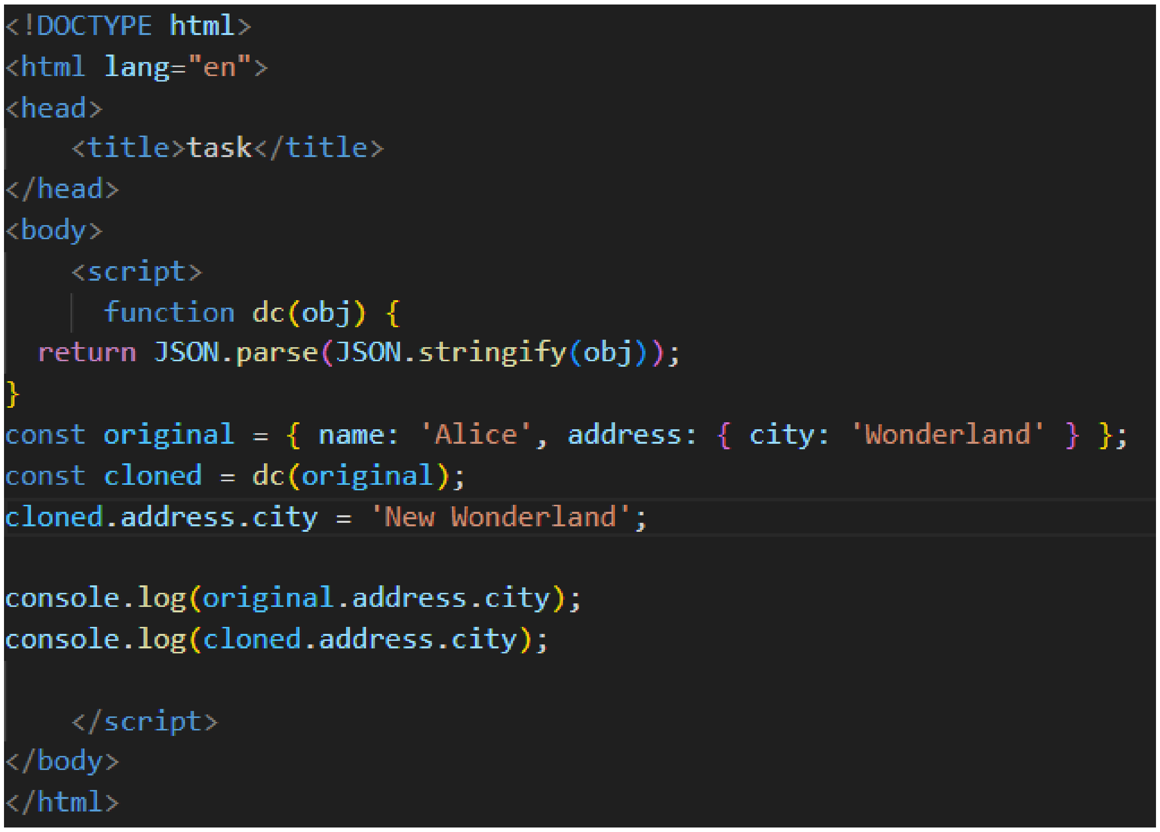


```
42                                                                    task
83                                                                    task
```

Task 3:

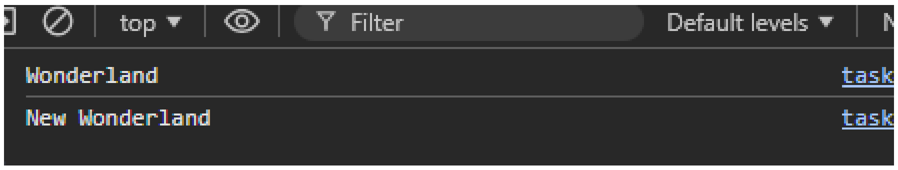
Create a deep clone of an object using JSON methods.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>task</title>
</head>
<body>
    <script>
      function dc(obj) {
  return JSON.parse(JSON.stringify(obj));
}
const original = { name: 'Alice', address: { city: 'Wonderland' } };
const cloned = dc(original);
cloned.address.city = 'New Wonderland';

console.log(original.address.city);
console.log(cloned.address.city);

    </script>
</body>
</html>
```

output:



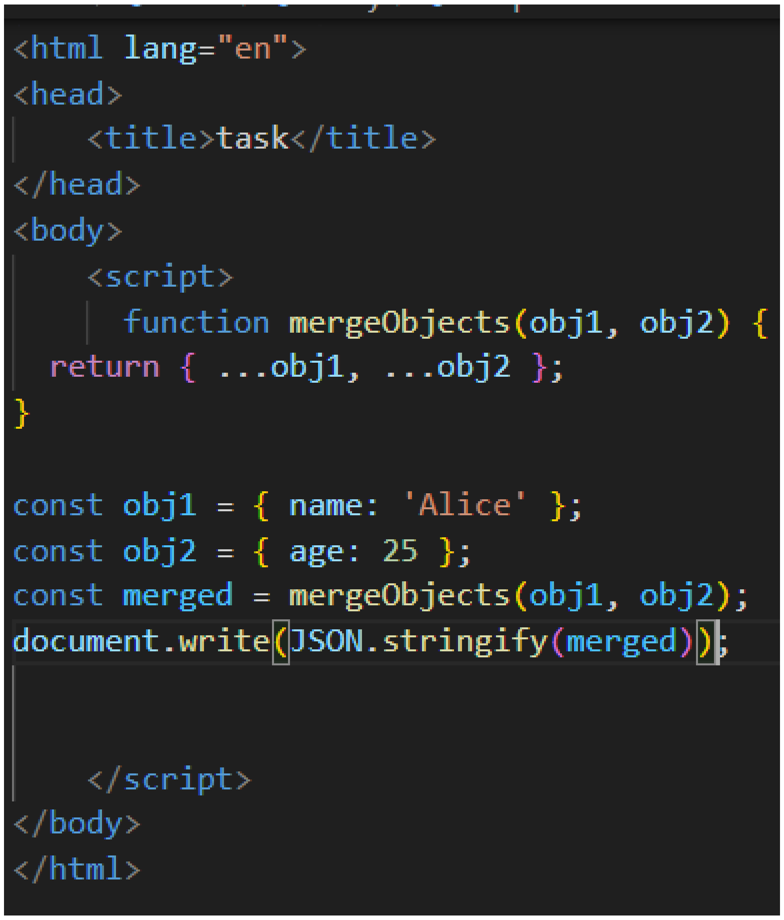
```
Wonderland
New Wonderland
```

Task 4:

Write a function that returns a new object, merging two provided objects using

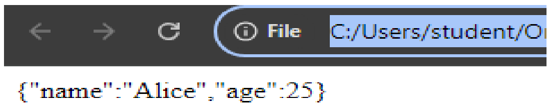
the spread syntax.

code:



```
<html lang="en">
<head>
    <title>task</title>
</head>
<body>
    <script>
      function mergeObjects(obj1, obj2) {
  return { ...obj1, ...obj2 };
}

const obj1 = { name: 'Alice' };
const obj2 = { age: 25 };
const merged = mergeObjects(obj1, obj2);
document.write(JSON.stringify(merged));


    </script>
</body>
</html>
```

output:

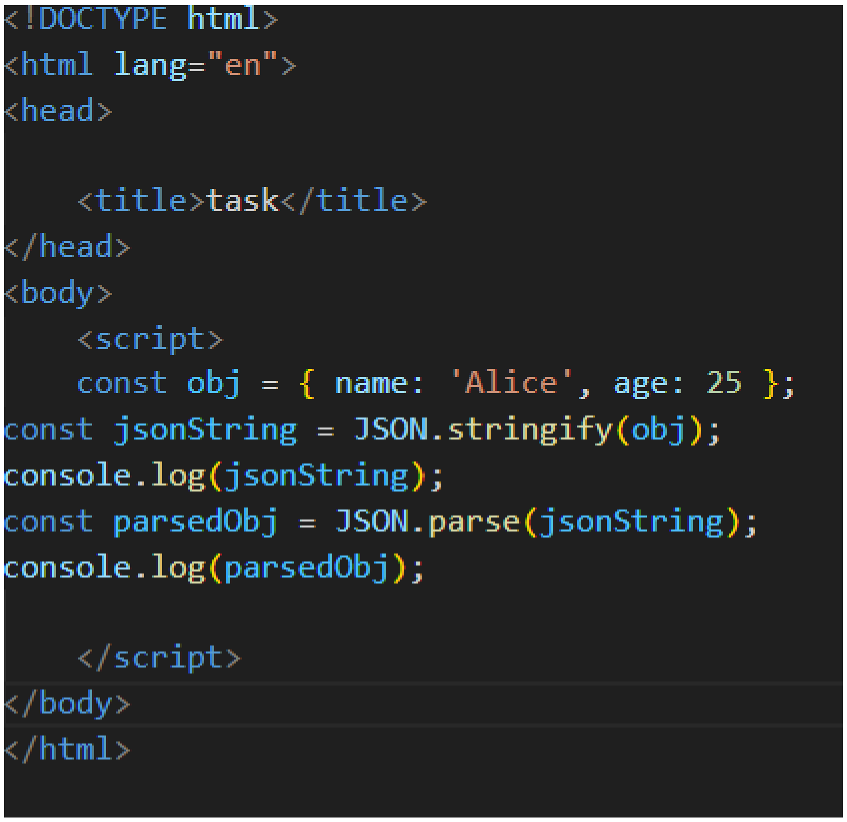


```
C:/Users/student/O
{"name":"Alice","age":25}
```

Task 5:

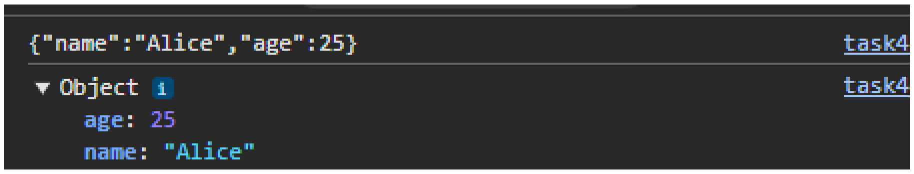
Serialize a JavaScript object into a JSON string and then parse it back into an object.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>

    <title>task</title>
</head>
<body>
    <script>
    const obj = { name: 'Alice', age: 25 };
const jsonString = JSON.stringify(obj);
console.log(jsonString);
const parsedObj = JSON.parse(jsonString);
console.log(parsedObj);

    </script>
</body>
</html>
```

output:



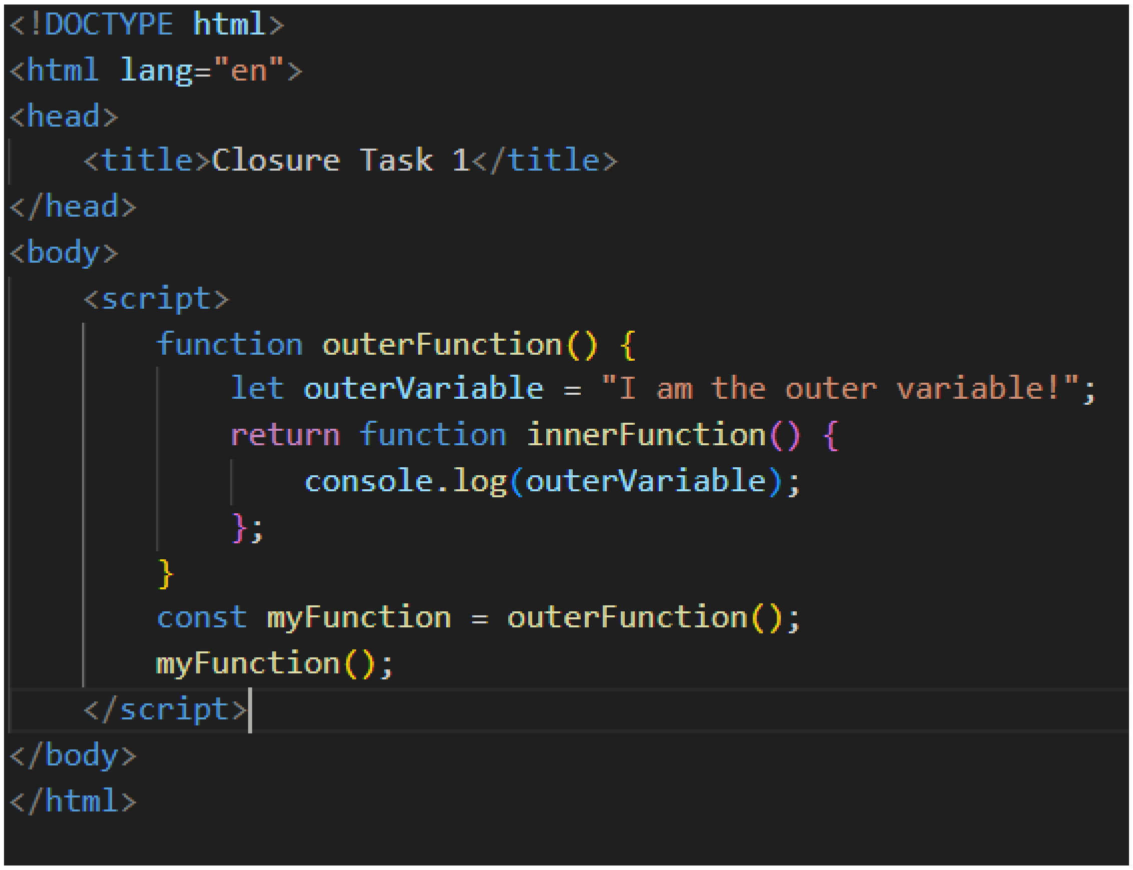
```
{"name":"Alice","age":25}                                   task4
▼ Object i                                                  task4
    age: 25
    name: "Alice"
```

3. Closure:

Task 1:

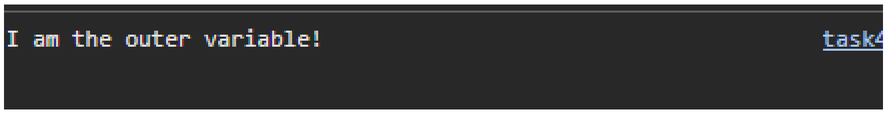
Create a function that returns another function, capturing a local variable.

code:



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Closure Task 1</title>
</head>
<body>
    <script>
        function outerFunction() {
            let outerVariable = "I am the outer variable!";
            return function innerFunction() {
                console.log(outerVariable);
            };
        }
        const myFunction = outerFunction();
        myFunction();
    </script>
</body>
</html>
```

output:

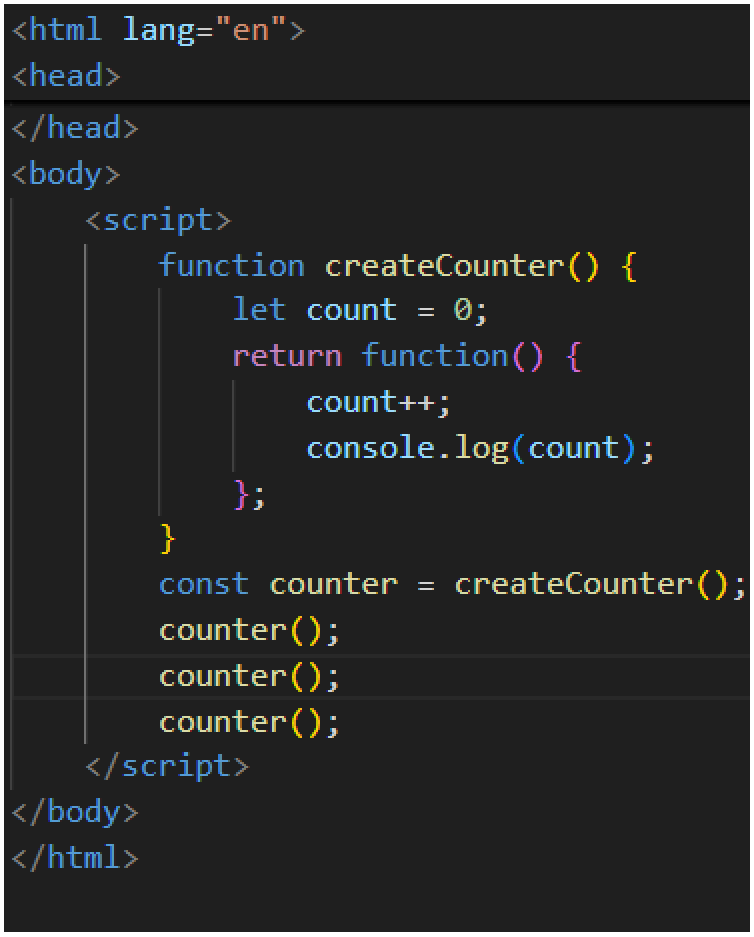


```
I am the outer variable!                                    task4
```

Task 2:

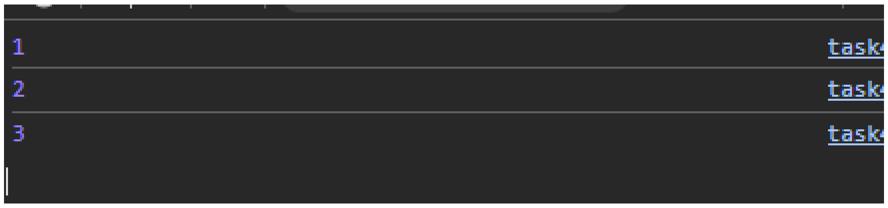
Implement a basic counter function using closure, allowing incrementing and displaying the current count.

code:



```
<html lang="en">
<head>
</head>
<body>
    <script>
        function createCounter() {
            let count = 0;
            return function() {
                count++;
                console.log(count);
            };
        }
        const counter = createCounter();
        counter();
        counter();
        counter();
    </script>
</body>
</html>
```

output:

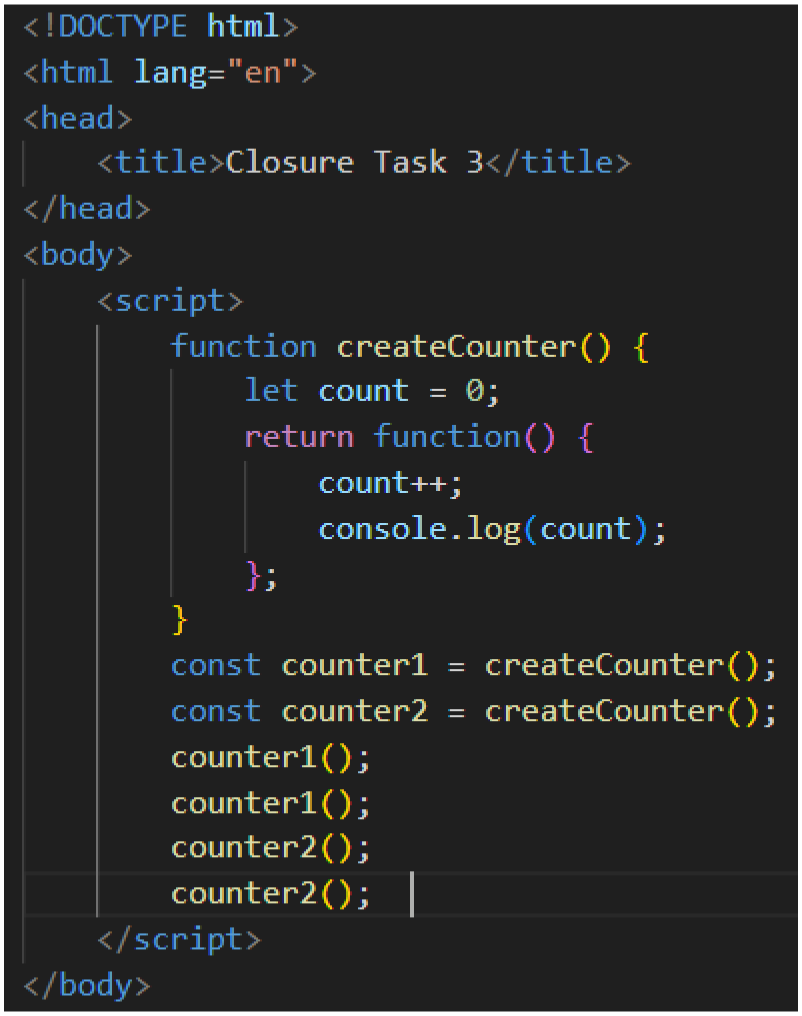


```
1                                                          task
2                                                          task
3                                                          task
```

Task 3:

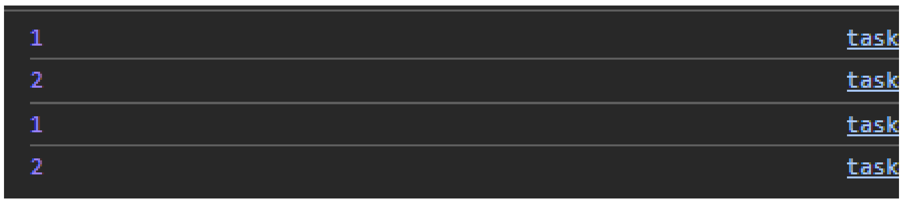
Write a function to create multiple counters, each with its own separate count.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Closure Task 3</title>
</head>
<body>
    <script>
        function createCounter() {
            let count = 0;
            return function() {
                count++;
                console.log(count);
            };
        }
        const counter1 = createCounter();
        const counter2 = createCounter();
        counter1();
        counter1();
        counter2();
        counter2();
    </script>
</body>
```

output:

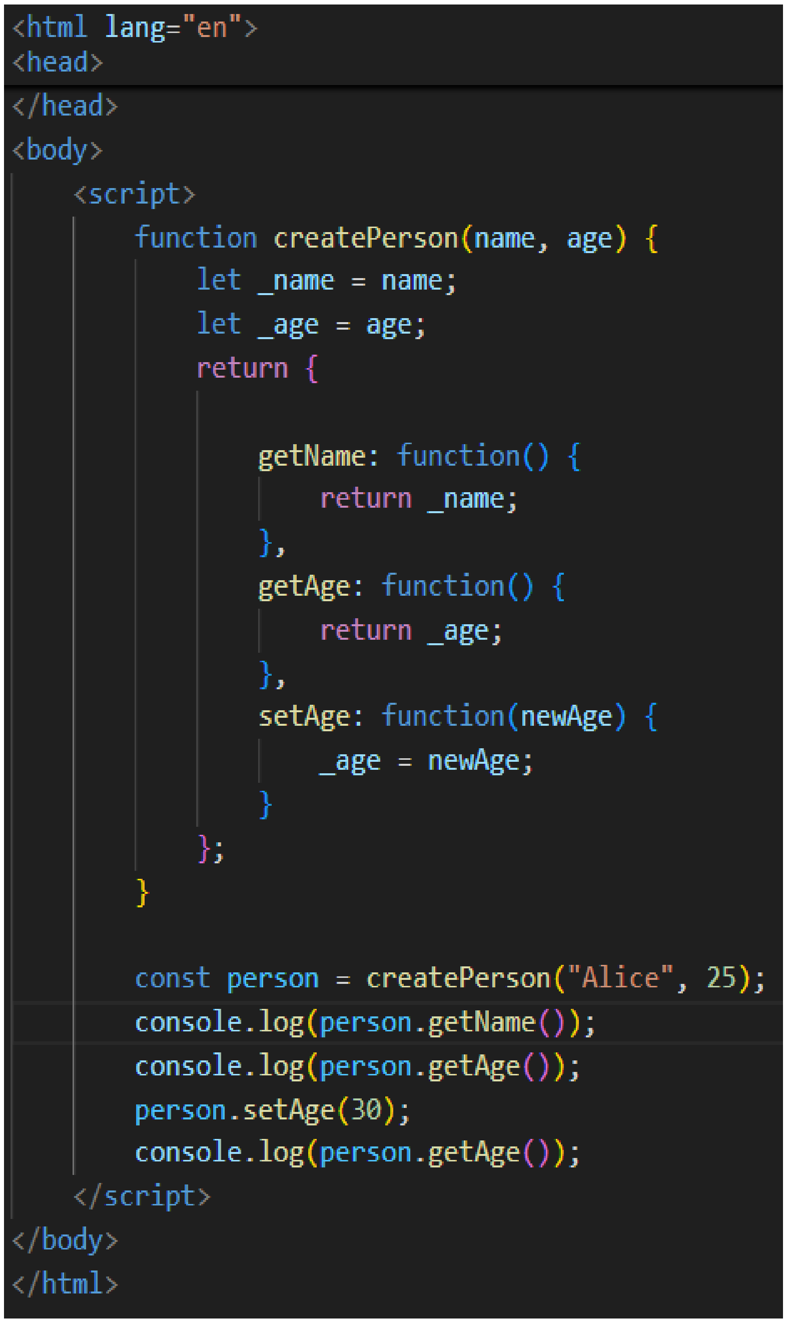


```
1                                                        task
2                                                        task
1                                                        task
2                                                        task
```

Task 4:

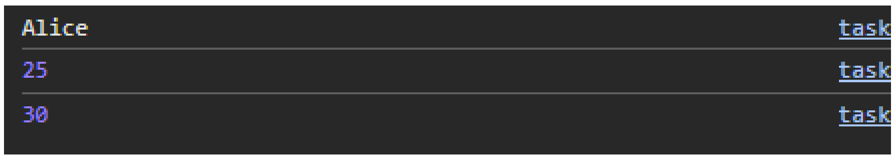
Use closures to create private variables within a function.

code:



```
<html lang="en">
<head>
</head>
<body>
    <script>
        function createPerson(name, age) {
            let _name = name;
            let _age = age;
            return {

                getName: function() {
                    return _name;
                },
                getAge: function() {
                    return _age;
                },
                setAge: function(newAge) {
                    _age = newAge;
                }
            };
        }

        const person = createPerson("Alice", 25);
        console.log(person.getName());
        console.log(person.getAge());
        person.setAge(30);
        console.log(person.getAge());
    </script>
</body>
</html>
```

output:

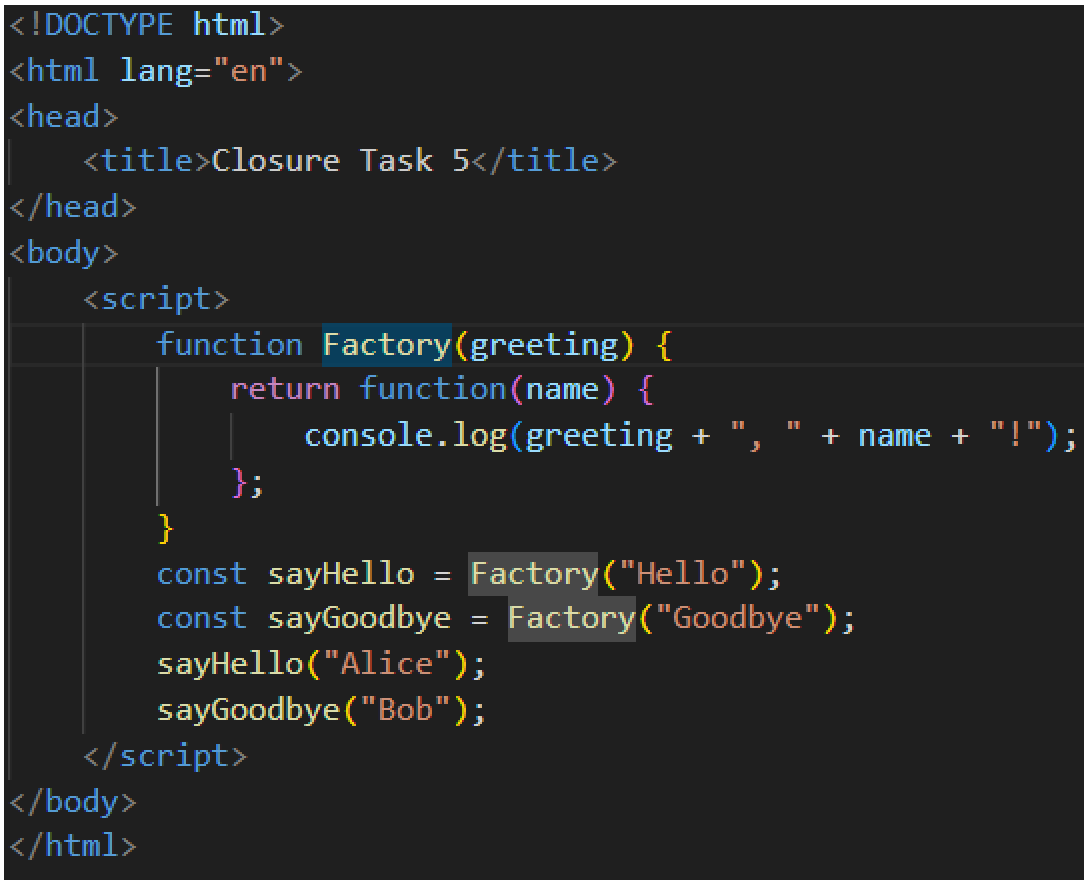


```
Alice                                              task
25                                                 task
30                                                 task
```

Task 5:

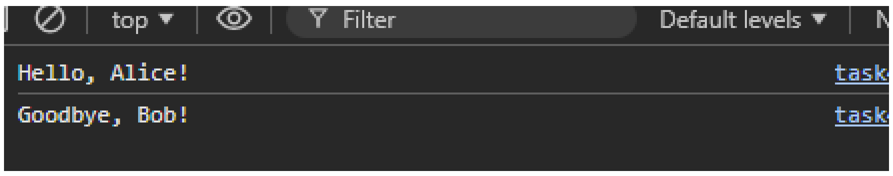
Build a function factory that generates functions based on some input using closures.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Closure Task 5</title>
</head>
<body>
    <script>
        function Factory(greeting) {
            return function(name) {
                console.log(greeting + ", " + name + "!");
            };
        }
        const sayHello = Factory("Hello");
        const sayGoodbye = Factory("Goodbye");
        sayHello("Alice");
        sayGoodbye("Bob");
    </script>
</body>
</html>
```

output:



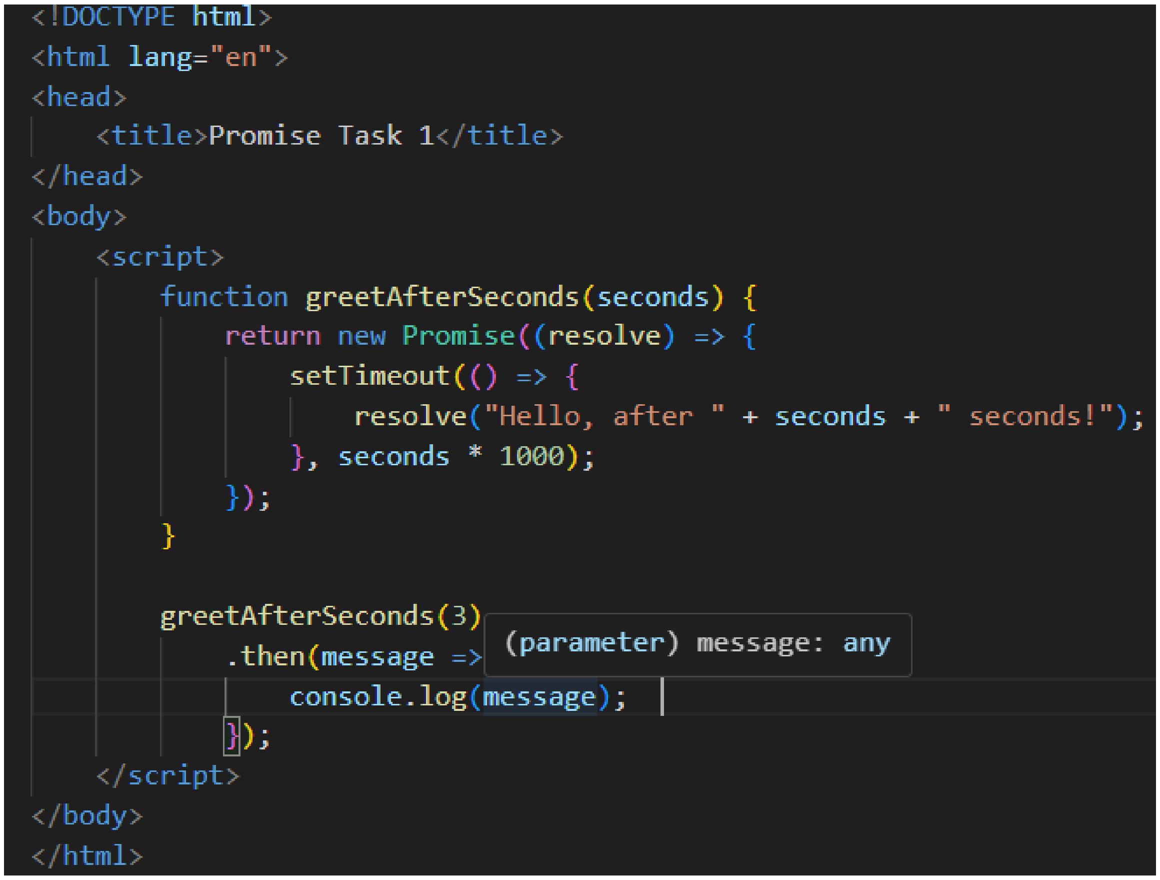
```
Hello, Alice!                                                task
Goodbye, Bob!                                                task
```

4. Promise, Promises chaining:

Task 1:

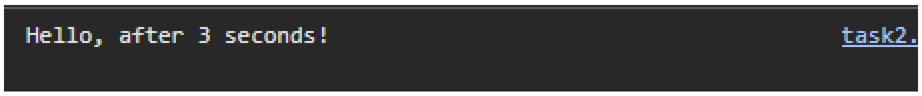
Create a new promise that resolves after a set number of seconds and returns a greeting.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Promise Task 1</title>
</head>
<body>
    <script>
        function greetAfterSeconds(seconds) {
            return new Promise((resolve) => {
                setTimeout(() => {
                    resolve("Hello, after " + seconds + " seconds!");
                }, seconds * 1000);
            });
        }

        greetAfterSeconds(3)
            .then(message =>
                console.log(message);
            });
    </script>
</body>
</html>
```

output:

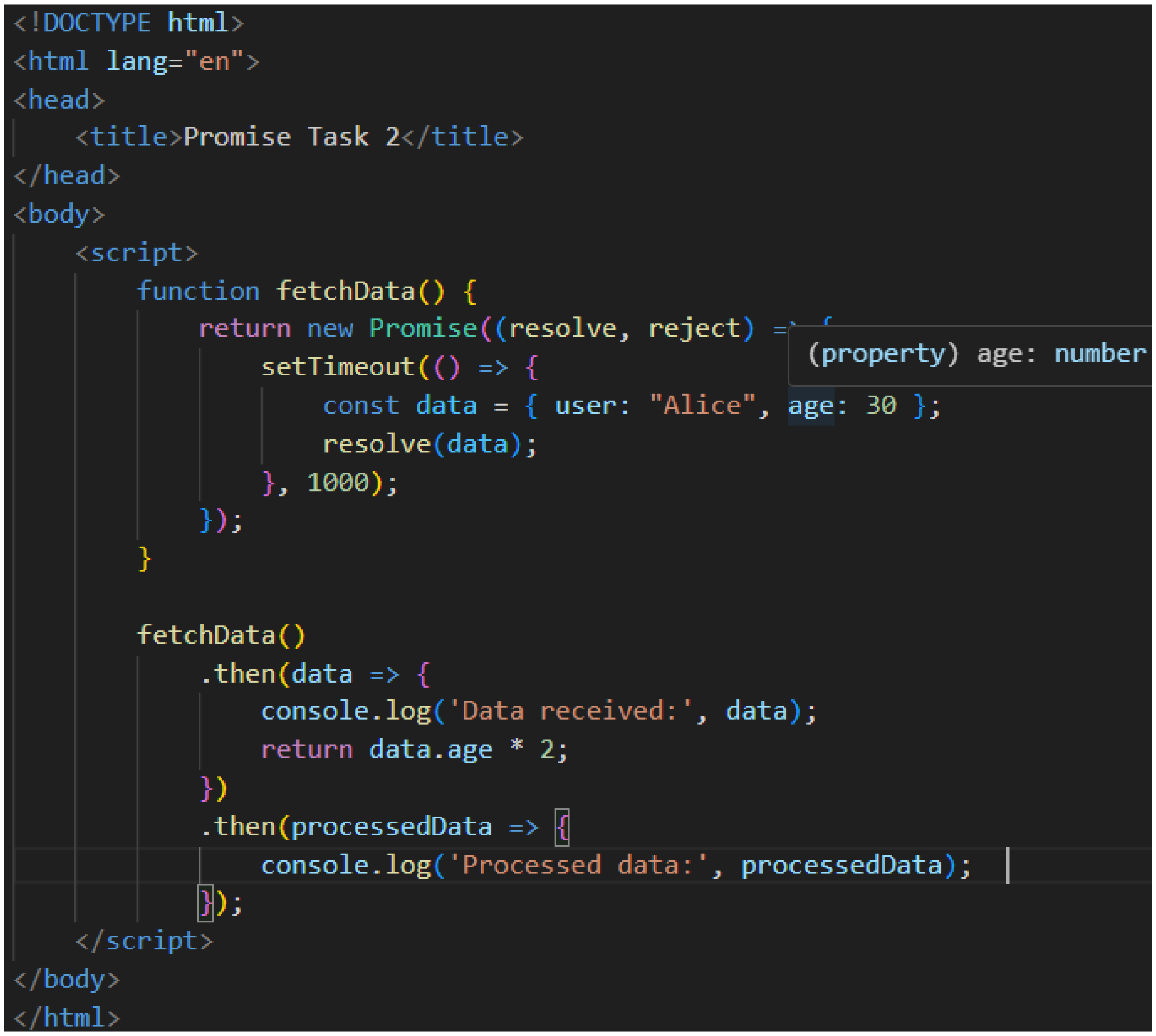


```
Hello, after 3 seconds!                                    task2.
```

Task 2:

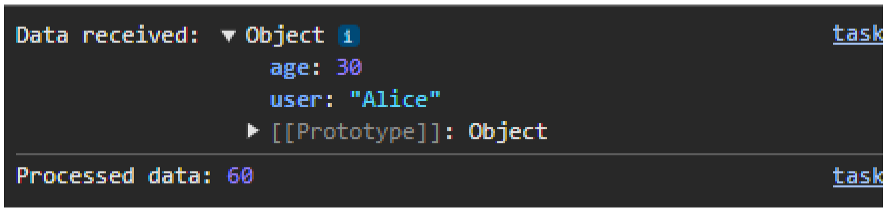
Fetch data from an API using promises, and then chain another promise to process this data.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Promise Task 2</title>
</head>
<body>
    <script>
        function fetchData() {
            return new Promise((resolve, reject) => {
                setTimeout(() => {
                    const data = { user: "Alice", age: 30 };
                    resolve(data);
                }, 1000);
            });
        }

        fetchData()
            .then(data => {
                console.log('Data received:', data);
                return data.age * 2;
            })
            .then(processedData => {
                console.log('Processed data:', processedData);
            });
    </script>
</body>
</html>
```

output:

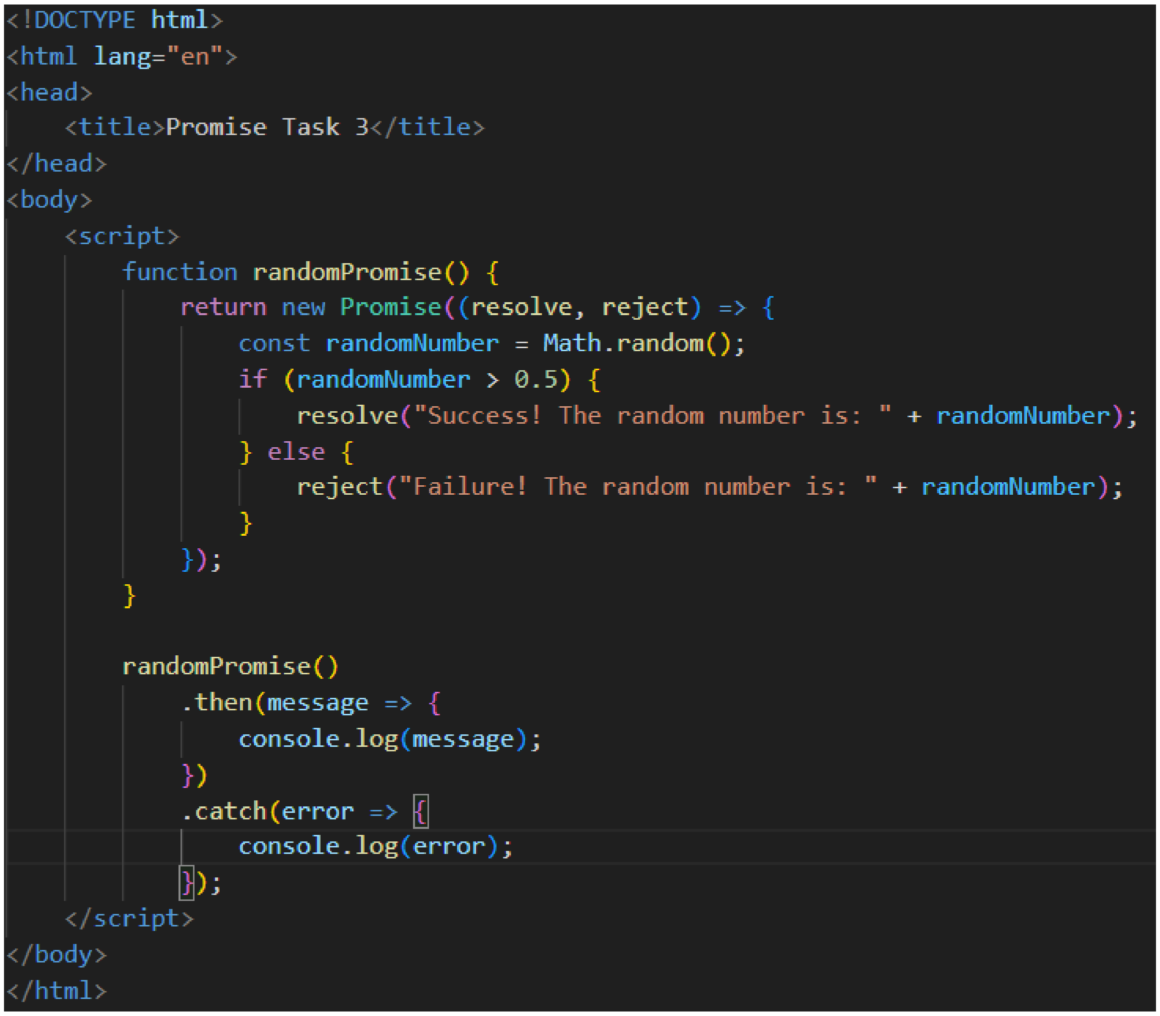


```
Data received:  ▼ Object i                                    task
                    age: 30
                    user: "Alice"
                  ▶ [[Prototype]]: Object
Processed data: 60                                            task
```

Task 3:

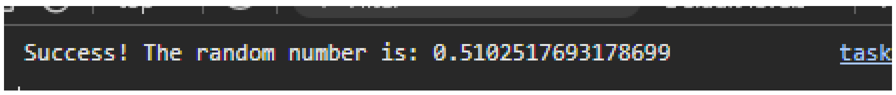
Create a promise that either resolves or rejects based on a random number.

code:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Promise Task 3</title>
</head>
<body>
    <script>
        function randomPromise() {
            return new Promise((resolve, reject) => {
                const randomNumber = Math.random();
                if (randomNumber > 0.5) {
                    resolve("Success! The random number is: " + randomNumber);
                } else {
                    reject("Failure! The random number is: " + randomNumber);
                }
            });
        }

        randomPromise()
            .then(message => {
                console.log(message);
            })
            .catch(error => {
                console.log(error);
            });
    </script>
</body>
</html>
```

output:

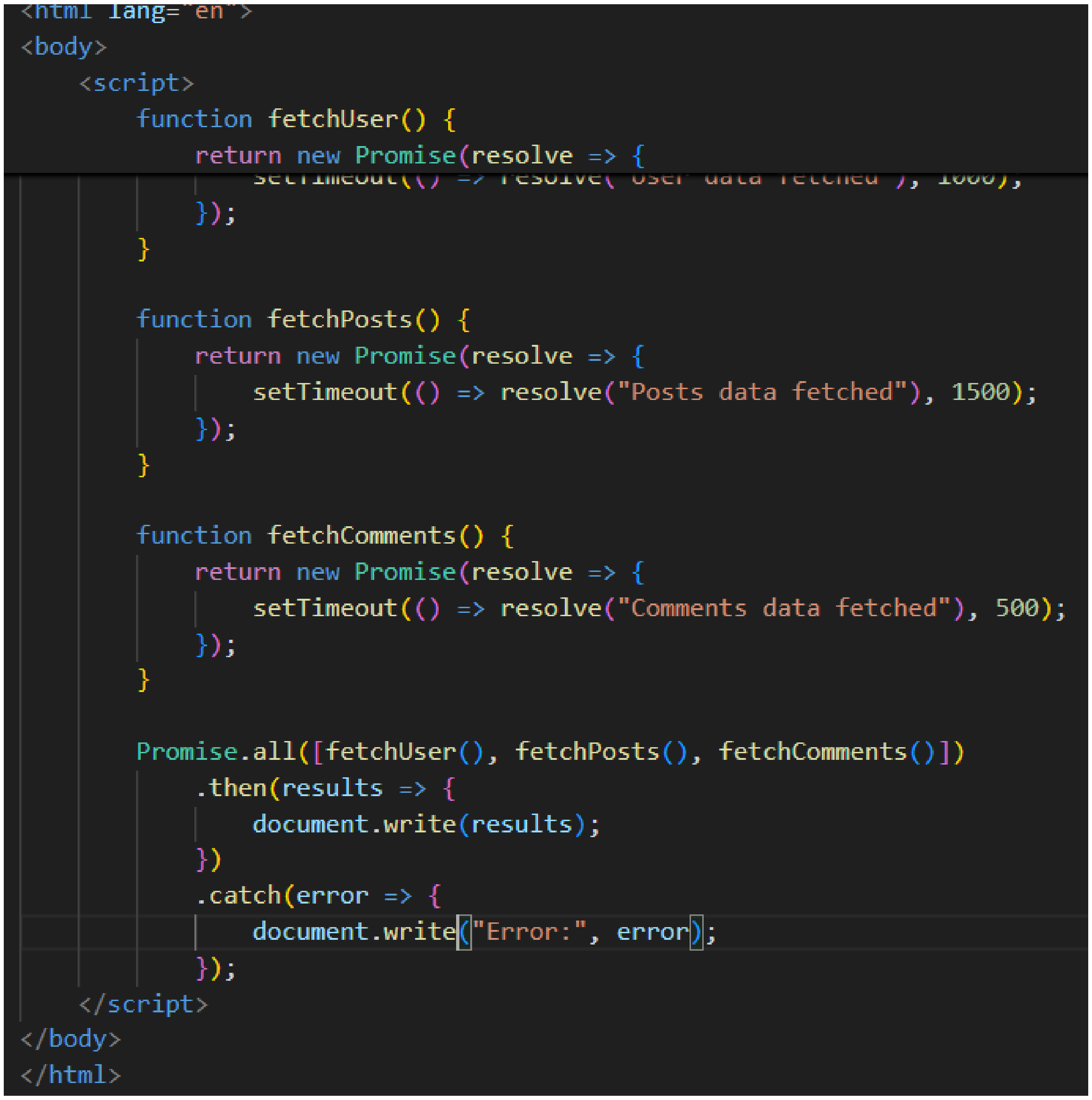


```
Success! The random number is: 0.5102517693178699          task
```

Task 4:

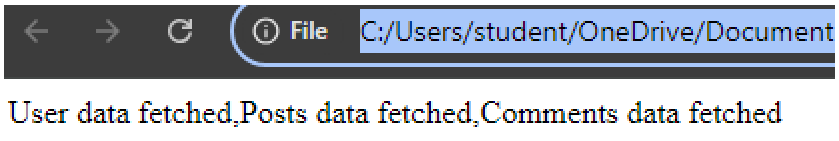
Use Promise.all to fetch multiple resources in parallel from an API.

code:



```
<html lang="en">
<body>
    <script>
        function fetchUser() {
            return new Promise(resolve => {
                setTimeout(() => resolve("User data fetched"), 1000);
            });
        }

        function fetchPosts() {
            return new Promise(resolve => {
                setTimeout(() => resolve("Posts data fetched"), 1500);
            });
        }

        function fetchComments() {
            return new Promise(resolve => {
                setTimeout(() => resolve("Comments data fetched"), 500);
            });
        }

        Promise.all([fetchUser(), fetchPosts(), fetchComments()])
            .then(results => {
                document.write(results);
            })
            .catch(error => {
                document.write("Error:", error);
            });
    </script>
</body>
</html>
```

output:

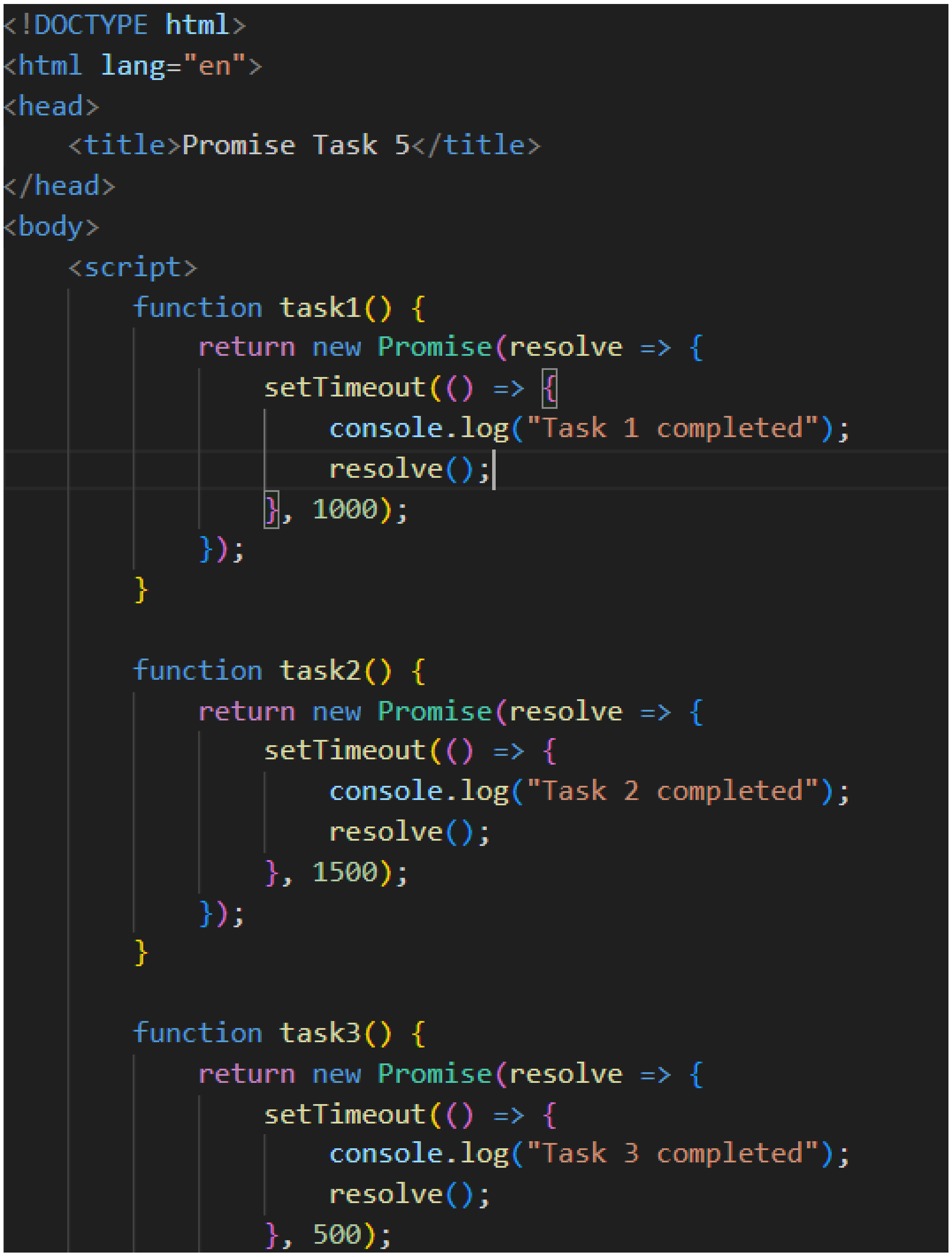


```
User data fetched,Posts data fetched,Comments data fetched
```

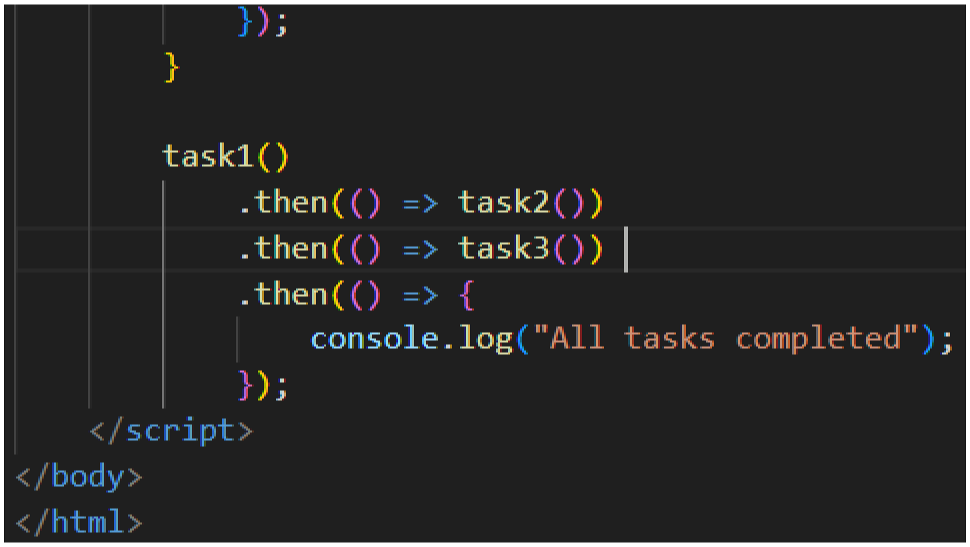
Task 5:

Chain multiple promises to perform a series of asynchronous actions in sequence.

code:

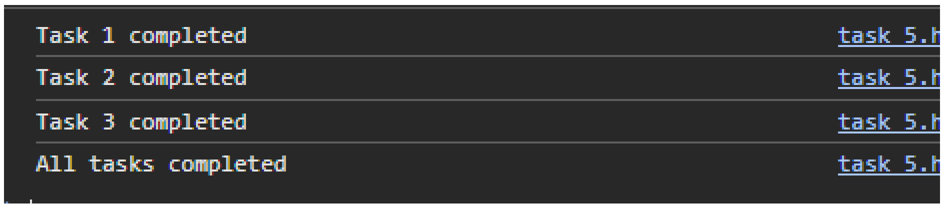


```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Promise Task 5</title>
</head>
<body>
    <script>
        function task1() {
            return new Promise(resolve => {
                setTimeout(() => {
                    console.log("Task 1 completed");
                    resolve();
                }, 1000);
            });
        }

        function task2() {
            return new Promise(resolve => {
                setTimeout(() => {
                    console.log("Task 2 completed");
                    resolve();
                }, 1500);
            });
        }

        function task3() {
            return new Promise(resolve => {
                setTimeout(() => {
                    console.log("Task 3 completed");
                    resolve();
                }, 500);
```



```
            });
        }

        task1()
            .then(() => task2())
            .then(() => task3())
            .then(() => {
                console.log("All tasks completed");
            });
    </script>
</body>
</html>
```

output:



```
Task 1 completed                                                task 5.h
Task 2 completed                                                task 5.h
Task 3 completed                                                task 5.h
All tasks completed                                             task 5.h
```