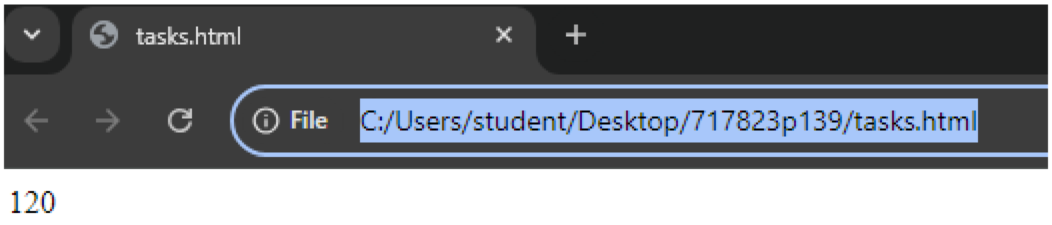
1. Recursion and stack:

Task 1: Implement a function to calculate the factorial of a number using recursion.

CODE:

```
<html>
  <body>
    <script>
      function factorial(n) {
        if (n == 0 || n === 1) {
          return 1;
        }
        return n * factorial(n - 1);
      }
      document.writeln(factorial(5));
    </script>
  </body>
</html>
```

OUTPUT:



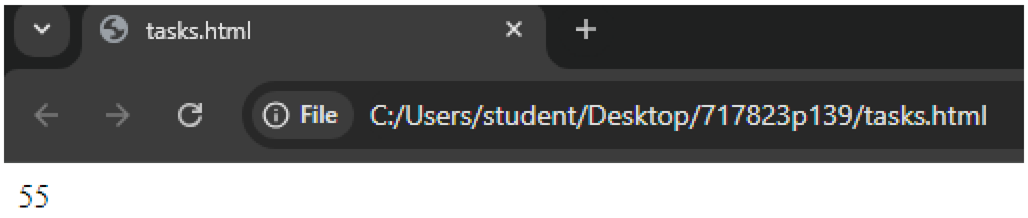
```
120
```

Task 2: Write a recursive function to find the nth Fibonacci number.

CODE:

```
<html>
  <body>
    <script>
      function fibonacci(n) {
        if (n == 0) return 0;
        else if (n == 1) return 1;
        else {
          return fibonacci(n - 1) + fibonacci(n - 2);
        }
      }
      document.writeln(fibonacci(10));
    </script>
  </body>
</html>
```

OUTPUT:



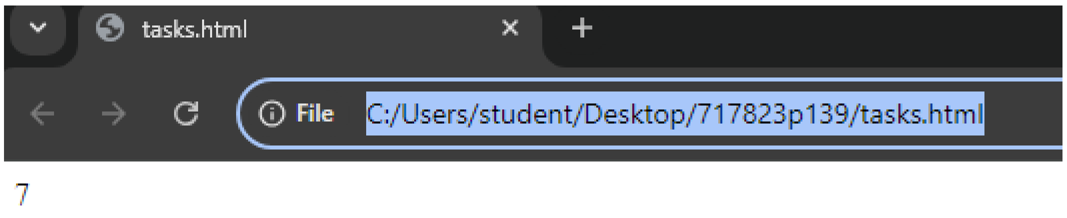
```
55
```

Task 3: Create a function to determine the total number of ways one can climb a staircase with 1, 2, or 3 steps at a time using recursion.

CODE:

```
<html>
  <body>
    <script>
      function ways(n) {
        if (n == 0) return 1 ;
        else if (n < 0) return 0 ;
        return ways(n - 1) + ways(n - 2)+ ways(n - 3);
      }
      document.writeln(ways(4));
    </script>
  </body>
</html>
```

OUTPUT:

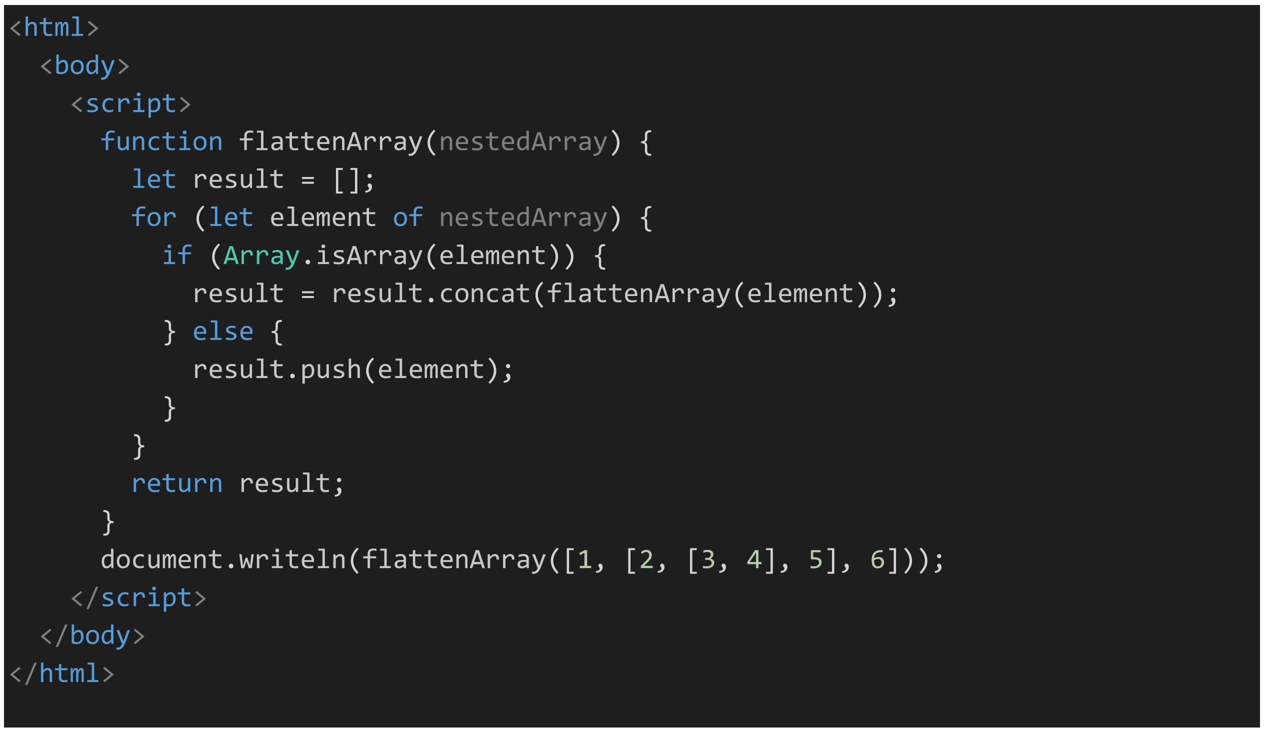

tasks.html

File C:/Users/student/Desktop/717823p139/tasks.html

```
7
```

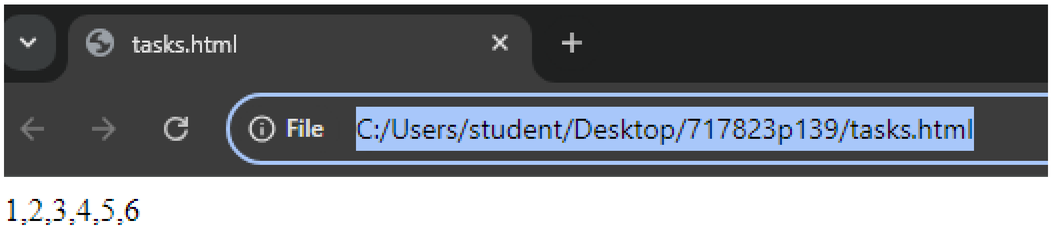
Task 4: Write a recursive function to flatten a nested array structure.

CODE:



```
<html>
  <body>
    <script>
      function flattenArray(nestedArray) {
        let result = [];
        for (let element of nestedArray) {
          if (Array.isArray(element)) {
            result = result.concat(flattenArray(element));
          } else {
            result.push(element);
          }
        }
        return result;
      }
      document.writeln(flattenArray([1, [2, [3, 4], 5], 6]));
    </script>
  </body>
</html>
```

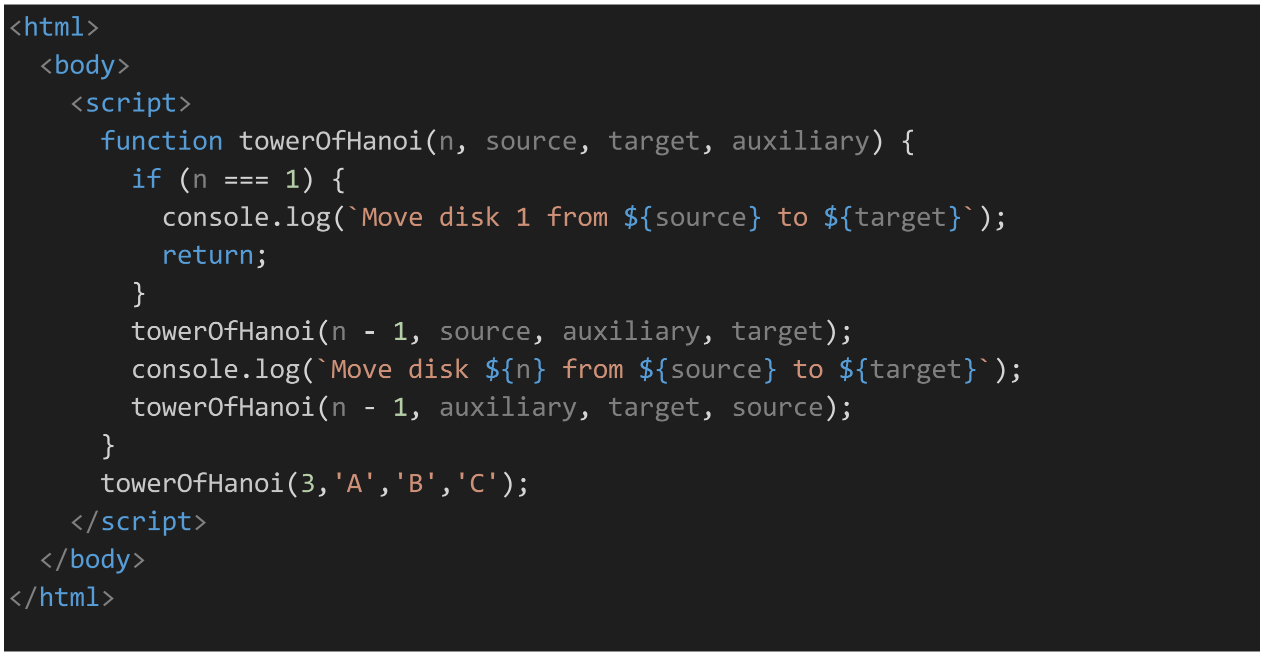
OUTPUT:



```
1,2,3,4,5,6
```

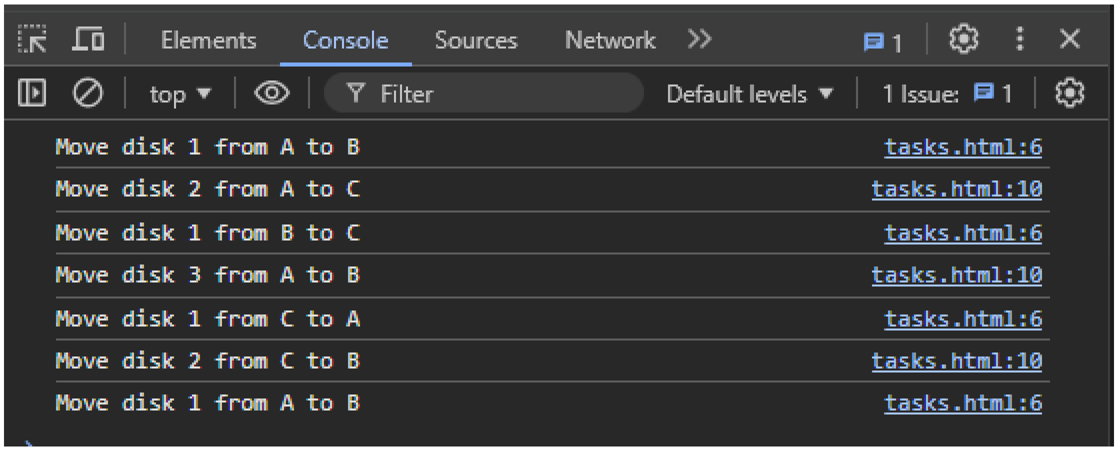
Task 5: Implement the recursive Tower of Hanoi solution.

CODE:



```
<html>
  <body>
    <script>
      function towerOfHanoi(n, source, target, auxiliary) {
        if (n === 1) {
          console.log(`Move disk 1 from ${source} to ${target}`);
          return;
        }
        towerOfHanoi(n - 1, source, auxiliary, target);
        console.log(`Move disk ${n} from ${source} to ${target}`);
        towerOfHanoi(n - 1, auxiliary, target, source);
      }
      towerOfHanoi(3,'A','B','C');
    </script>
  </body>
</html>
```

OUTPUT:



```
Move disk 1 from A to B                                    tasks.html:6
Move disk 2 from A to C                                    tasks.html:10
Move disk 1 from B to C                                    tasks.html:6
Move disk 3 from A to B                                    tasks.html:10
Move disk 1 from C to A                                    tasks.html:6
Move disk 2 from C to B                                    tasks.html:10
Move disk 1 from A to B                                    tasks.html:6
```

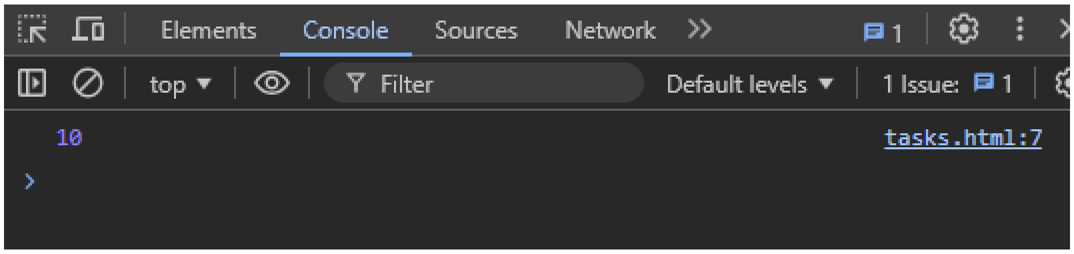
2. JSON and variable length arguments/spread syntax:

Task 1: Write a function that takes an arbitrary number of arguments and returns their sum.

CODE:

```
<html>
  <body>
    <script>
      function sum(...args) {
        return args.reduce((total, num) => total + num, 0);
      }
      console.log(sum(1, 2, 3, 4));
    </script>
  </body>
</html>
```

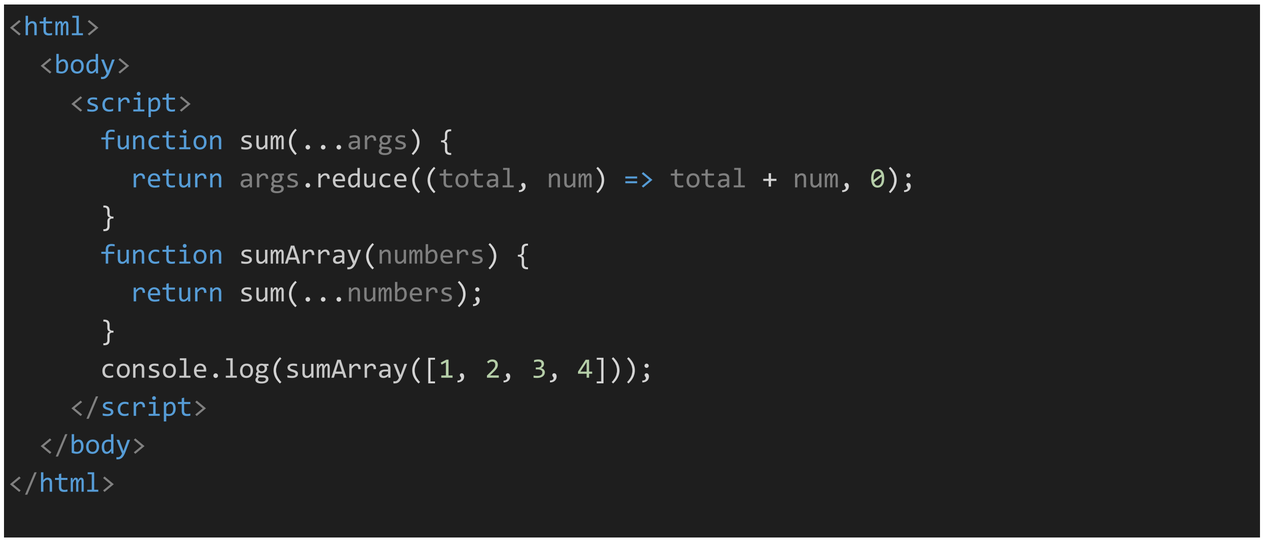
OUTPUT:



```
10                                                         tasks.html:7
```

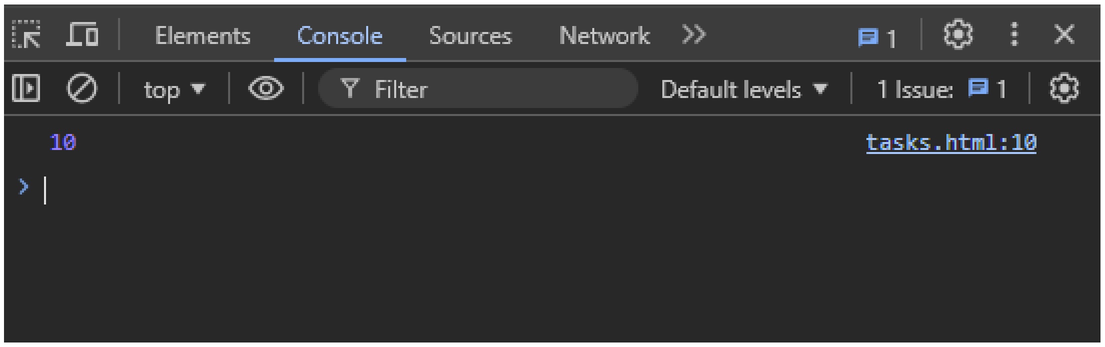
Task 2: Modify a function to accept an array of numbers and return their sum using the spread syntax.

CODE:



```
<html>
  <body>
    <script>
      function sum(...args) {
        return args.reduce((total, num) => total + num, 0);
      }
      function sumArray(numbers) {
        return sum(...numbers);
      }
      console.log(sumArray([1, 2, 3, 4]));
    </script>
  </body>
</html>
```

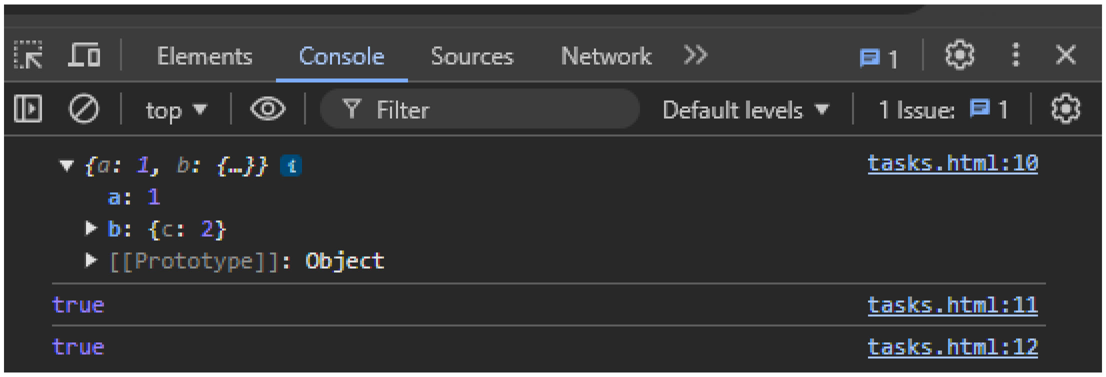
OUTPUT:



```
10                                                    tasks.html:10
>
```

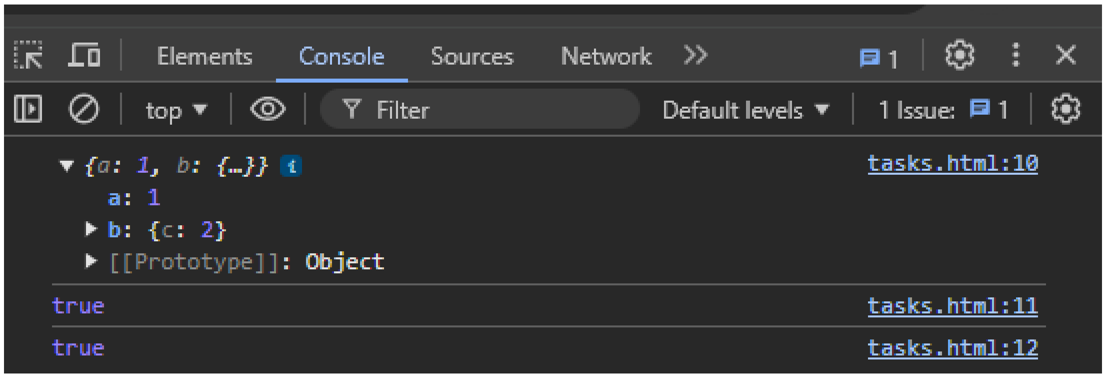
Task 3: Create a deep clone of an object using JSON methods.

CODE:



```
▼ {a: 1, b: {…}}                                    tasks.html:10
    a: 1
  ▶ b: {c: 2}
  ▶ [[Prototype]]: Object
true                                                tasks.html:11
true                                                tasks.html:12
```

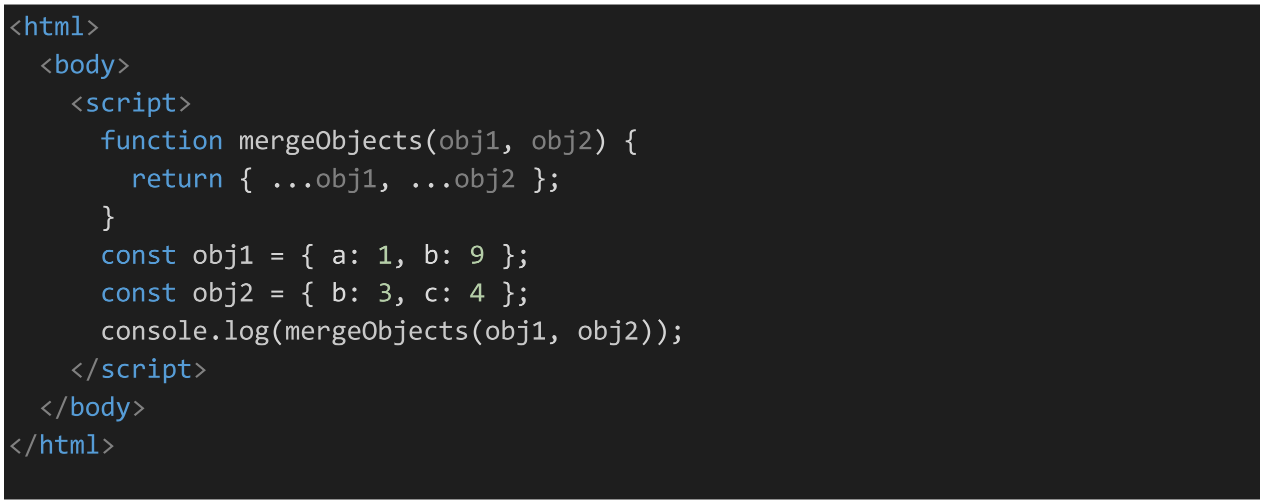
OUTPUT:



```
▼ {a: 1, b: {…}}                                    tasks.html:10
    a: 1
  ▶ b: {c: 2}
  ▶ [[Prototype]]: Object
true                                                tasks.html:11
true                                                tasks.html:12
```

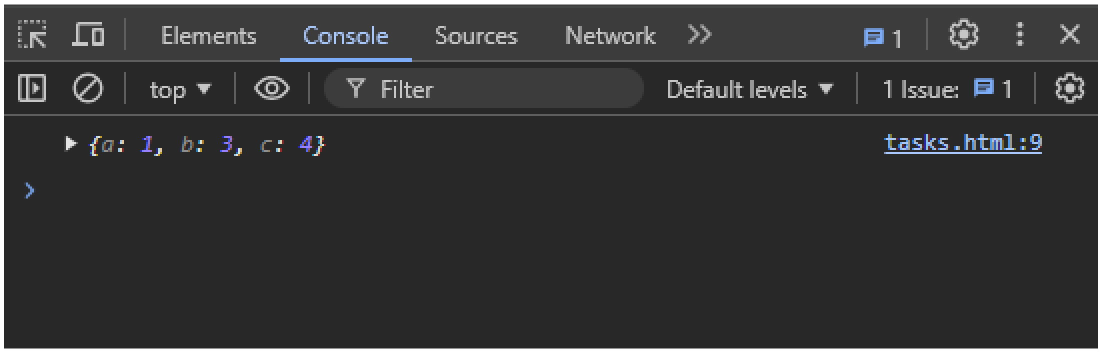
Task 4: Write a function that returns a new object, merging two provided objects using the spread syntax.

CODE:



```
<html>
  <body>
    <script>
      function mergeObjects(obj1, obj2) {
        return { ...obj1, ...obj2 };
      }
      const obj1 = { a: 1, b: 9 };
      const obj2 = { b: 3, c: 4 };
      console.log(mergeObjects(obj1, obj2));
    </script>
  </body>
</html>
```

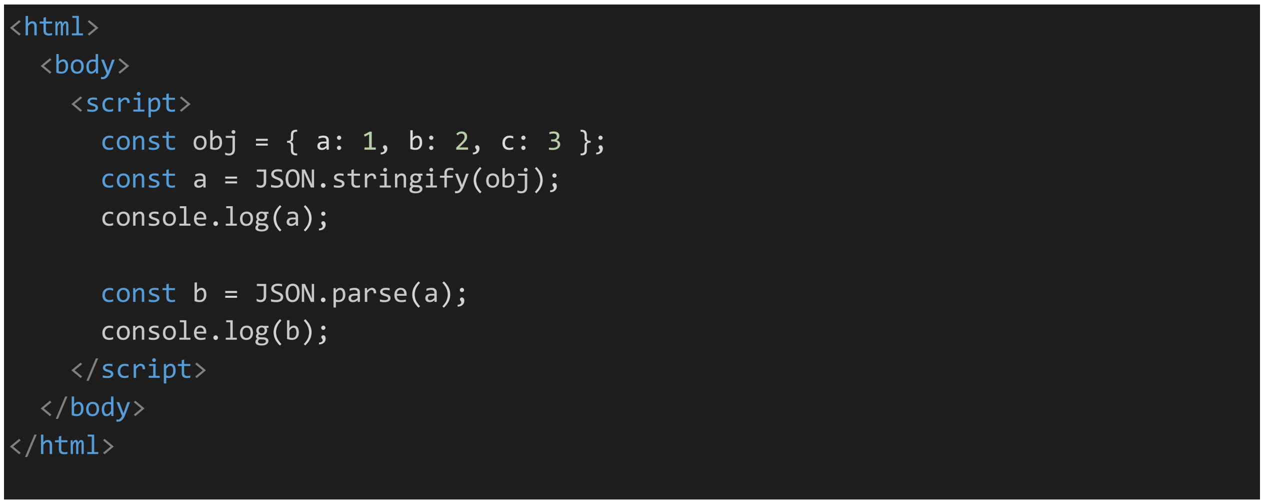
OUTPUT:



```
▸ {a: 1, b: 3, c: 4}                                   tasks.html:9
```

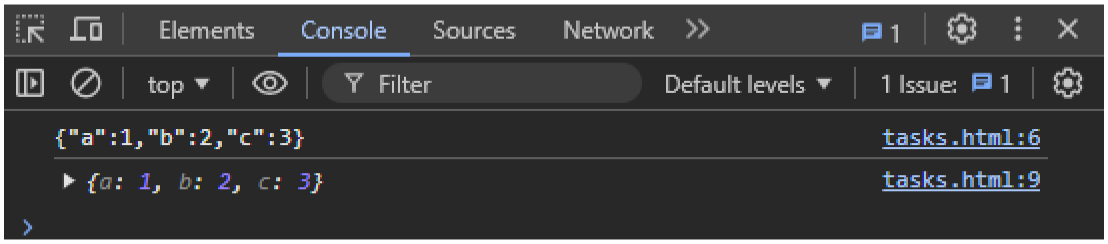
Task 5: Serialize a JavaScript object into a JSON string and then parse it back into an object.

CODE:



```
<html>
  <body>
    <script>
      const obj = { a: 1, b: 2, c: 3 };
      const a = JSON.stringify(obj);
      console.log(a);

      const b = JSON.parse(a);
      console.log(b);
    </script>
  </body>
</html>
```

OUTPUT:



```
{"a":1,"b":2,"c":3}                                    tasks.html:6
▸ {a: 1, b: 2, c: 3}                                   tasks.html:9
```

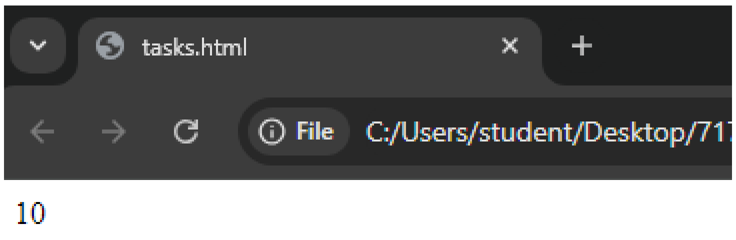
Closure:

Task 1: Create a function that returns another function, capturing a local variable.

CODE:

```
<html>
  <body>
    <script>
      function outer() {
        let a = 10;
        return function inner() {
          document.writeln(a);
        };
      }
      const exam = outer();
      exam();
    </script>
  </body>
</html>
```

OUTPUT:



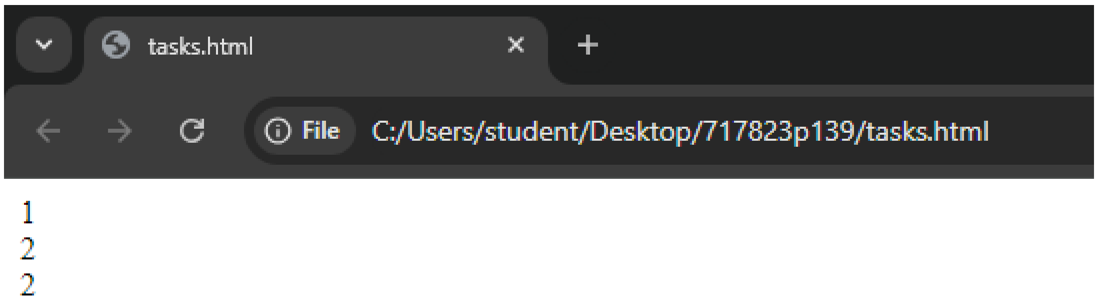
```
10
```

Task 2: Implement a basic counter function using closure, allowing incrementing and displaying the current count.

CODE:

```
<html>
  <body>
    <script>
      function createCounter() {
        let count = 0;
        return {
          increment: function () {
            count++;
            document.writeln(count + "<br>");
          },
          display: function () {
            document.writeln(count + "<br>");
          },
        };
      }
      const counter = createCounter();
      counter.increment();
      counter.increment();
      counter.display();
    </script>
  </body>
</html>
```

OUTPUT:



tasks.html

File C:/Users/student/Desktop/717823p139/tasks.html

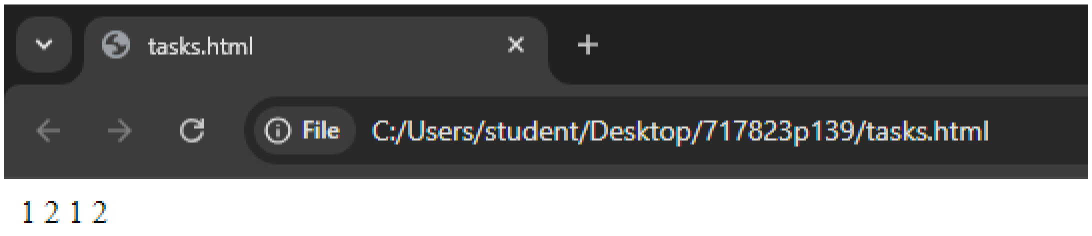
```
1
2
2
```

Task 3: Write a function to create multiple counters, each with its own separate count.

CODE:

```
<html>
  <body>
    <script>
      function createCounter() {
        let count = 0;
        return function () {
          count++;
          document.writeln(count);
        };
      }
      const counter1 = createCounter();
      const counter2 = createCounter();
      counter1();
      counter1();
      counter2();
      counter2();
    </script>
  </body>
</html>
```

OUTPUT:


tasks.html

C:/Users/student/Desktop/717823p139/tasks.html

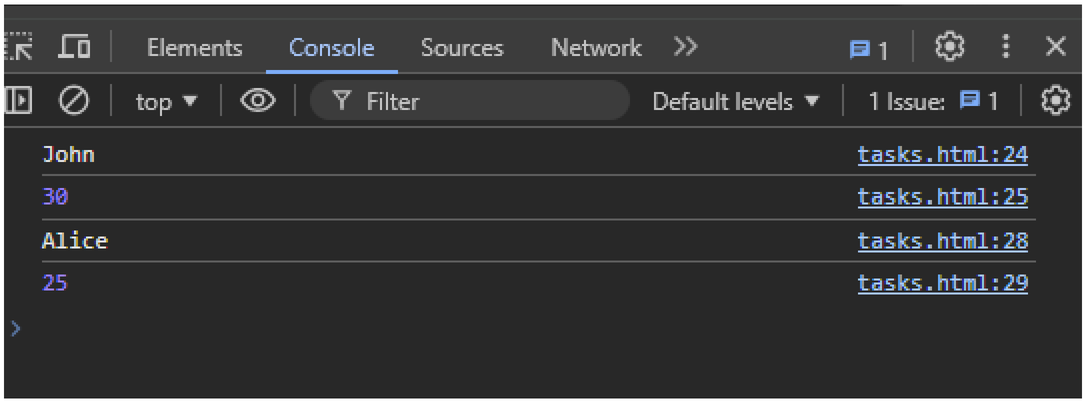
```
1 2 1 2
```

Task 4: Use closures to create private variables within a function.

CODE:

```
<html>
  <body>
    <script>
      function createPerson(name, age) {
        let _name = name;
        let _age = age;

        return {
          getName: function () {
            return _name;
          },
          getAge: function () {
            return _age;
          },
          setName: function (newName) {
            _name = newName;
          },
          setAge: function (newAge) {
            _age = newAge;
          },
        };
      }
      const person = createPerson("John", 30);
      console.log(person.getName());
      console.log(person.getAge());
      person.setName("Alice");
      person.setAge(25);
      console.log(person.getName());
      console.log(person.getAge());
    </script>
  </body>
</html>
```

OUTPUT:



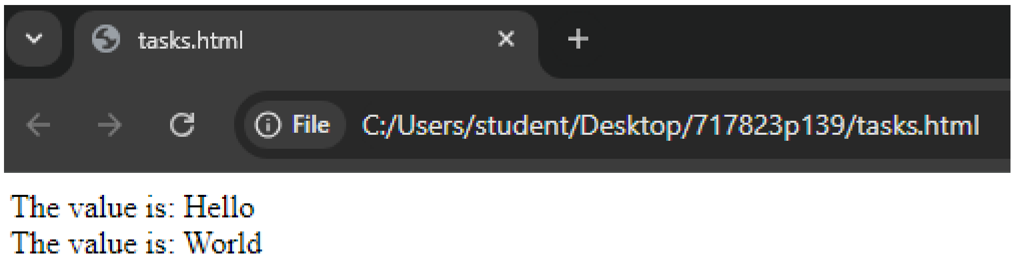
```
John                                              tasks.html:24
30                                                tasks.html:25
Alice                                             tasks.html:28
25                                                tasks.html:29
```

Task 5: Build a function factory that generates functions based on some input using closures.

CODE:

```
<html>
  <body>
    <script>
      function functionFactory(value) {
        return function () {
          document.writeln(`The value is: ${value}` + "<br>");
        };
      }
      const func1 = functionFactory("Hello");
      const func2 = functionFactory("World");
      func1();
      func2();
    </script>
  </body>
</html>
```

OUTPUT:



tasks.html

C:/Users/student/Desktop/717823p139/tasks.html

```
The value is: Hello
The value is: World
```

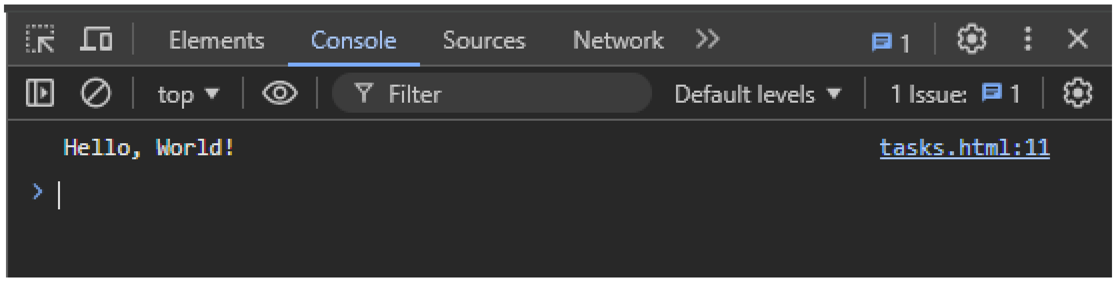
4. Promise, Promises chaining:

Task 1: Create a new promise that resolves after a set number of seconds and returns a greeting.

CODE:

```
<html>
  <body>
    <script>
      function greet(seconds) {
        return new Promise((resolve, reject) => {
          setTimeout(() => {
            resolve("Hello, World!");
          }, seconds * 1000);
        });
      }
      greet(2).then((message) => console.log(message));
    </script>
  </body>
</html>
```

OUTPUT:



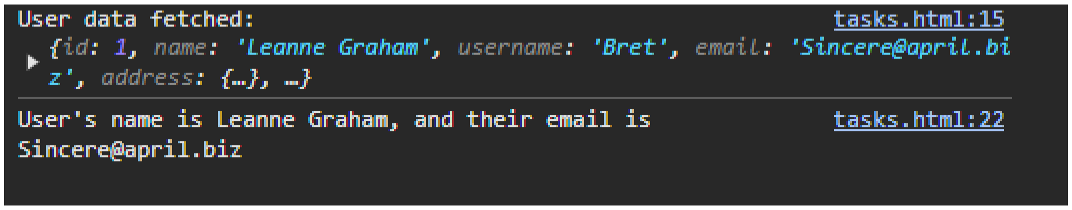
```
Hello, World!                                    tasks.html:11
```

Task 2: Fetch data from an API using promises, and then chain another promise to process this data.

CODE:

```
<html>
  <body>
    <script>
      function fetchUserData() {
        return new Promise((resolve, reject) => {
          fetch("https://jsonplaceholder.typicode.com/users/1")
            .then((response) => response.json())
            .then((data) => resolve(data))
            .catch((err) => reject(err));
        });
      }

      fetchUserData()
        .then((user) => {
          console.log("User data fetched:", user);
          return new Promise((resolve) => {
            const userInfo = `User's name is ${user.name}, and their email is
${user.email}`;
            resolve(userInfo);
          });
        })
        .then((userInfo) => {
          console.log(userInfo);
        })
        .catch((err) => console.error("Error:", err));
    </script>
  </body>
</html>
```

OUTPUT:



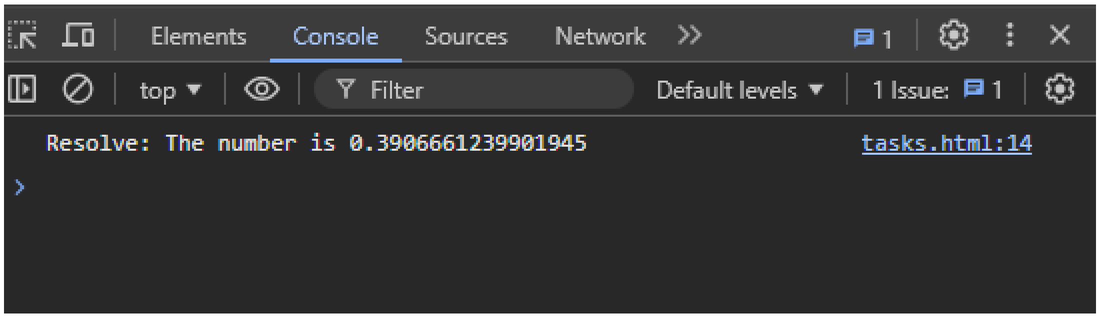
```
User data fetched:                                            tasks.html:15
▸ {id: 1, name: 'Leanne Graham', username: 'Bret', email: 'Sincere@april.bi
  z', address: {…}, …}
User's name is Leanne Graham, and their email is              tasks.html:22
Sincere@april.biz
```

Task 3: Create a promise that either resolves or rejects based on a random number.

CODE:

```
<html>
  <body>
    <script>
      function randomPromise() {
        return new Promise((resolve, reject) => {
          const ran = Math.random();
          if (ran > 0) {
            resolve("Resolve: The number is " + ran);
          } else {
            reject("Rejected " + ran);
          }
        });
      }
      randomPromise().then((message) => console.log(message));
    </script>
  </body>
</html>
```

OUTPUT:



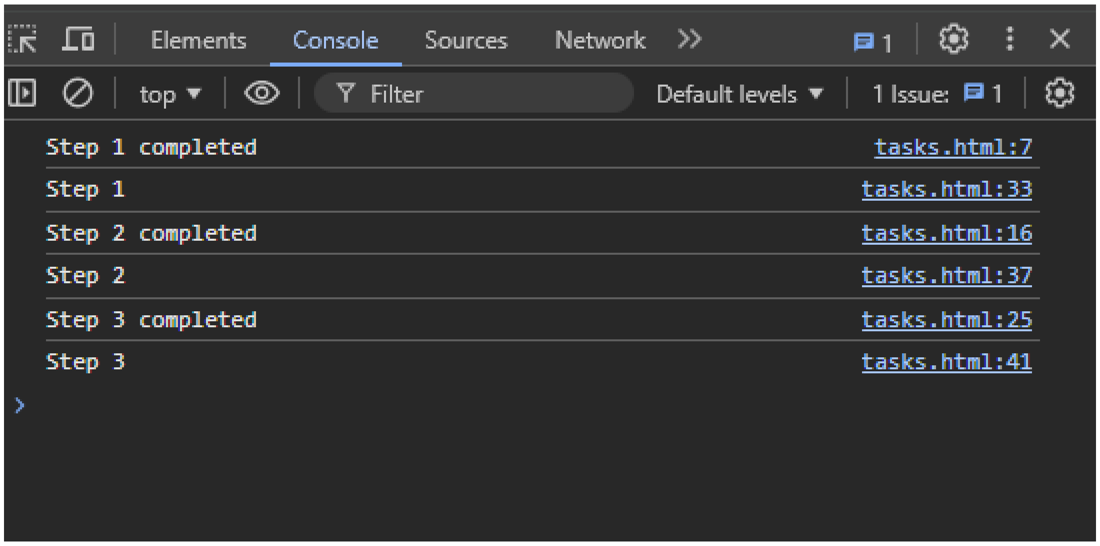
```
Resolve: The number is 0.3906661239901945                  tasks.html:14
>
```

Task 4: Use Promise.all to fetch multiple resources in parallel from an API.

CODE:

```
<html>
  <head>
    <title>My Webpage</title>
  </head>

  <body>
    <script>
      function fetchMultipleResources() {
        const urls = [
          "https://jsonplaceholder.typicode.com/posts",

          "https://jsonplaceholder.typicode.com/users",

          "https://jsonplaceholder.typicode.com/comments",
        ];

        const fetchPromises = urls.map((url) =>
          fetch(url).then((response) => response.json())
        );

        Promise.all(fetchPromises)

          .then((results) => {
            console.log("Posts:", results[0]);

            console.log("Users:", results[1]);

            console.log("Comments:", results[2]);
          })

          .catch((error) => {
            console.error("Error fetching data:", error);
          });
      }

      fetchMultipleResources();
    </script>
  </body>
</html>
```

OUTPUT:



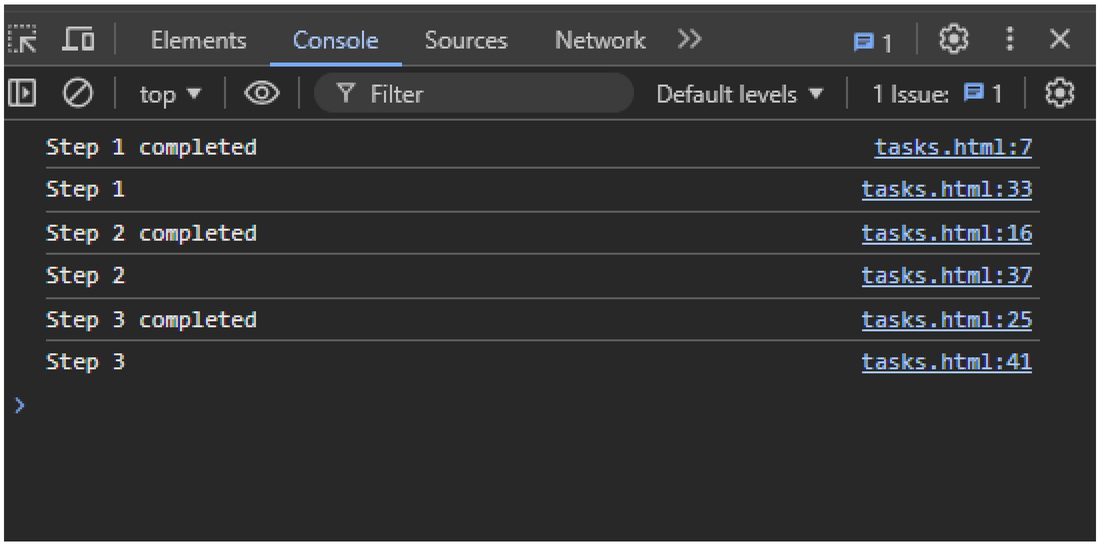
```
Step 1 completed                                tasks.html:7
Step 1                                         tasks.html:33
Step 2 completed                               tasks.html:16
Step 2                                         tasks.html:37
Step 3 completed                               tasks.html:25
Step 3                                         tasks.html:41
```

Task 5: Chain multiple promises to perform a series of asynchronous actions in sequence.

CODE:

```
<html>
  <body>
    <script>
      function step1() {
        return new Promise((resolve) => {
          setTimeout(() => {
            console.log("Step 1 completed");
            resolve("Step 1");
          }, 1000);
        });
      }

      function step2() {
        return new Promise((resolve) => {
          setTimeout(() => {
            console.log("Step 2 completed");
            resolve("Step 2");
          }, 1000);
        });
      }
      function step3() {
        return new Promise((resolve) => {
          setTimeout(() => {
            console.log("Step 3 completed");
            resolve("Step 3");
          }, 1000);
        });
      }

      step1()
        .then((result) => {
          console.log(result);
          return step2();
        })
        .then((result) => {
          console.log(result);
          return step3();
        })
        .then((result) => {
          console.log(result);
        })
        .catch((err) => console.error("Error:", err));
    </script>
  </body>
</html>
```

OUTPUT:



```
Step 1 completed                                   tasks.html:7
Step 1                                             tasks.html:33
Step 2 completed                                   tasks.html:16
Step 2                                             tasks.html:37
Step 3 completed                                   tasks.html:25
Step 3                                             tasks.html:41
```

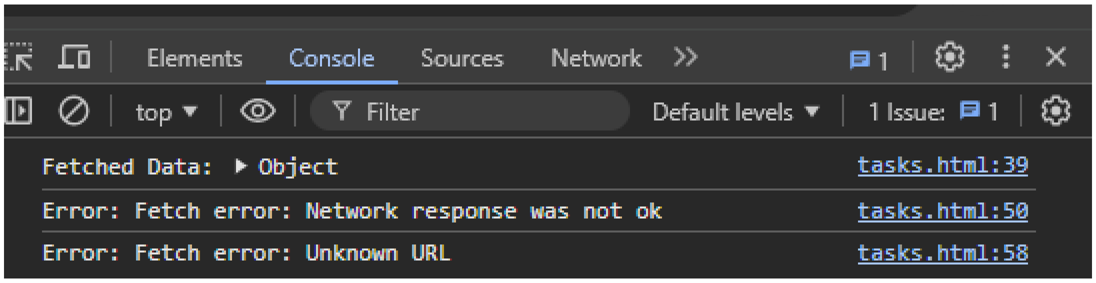
5. Async/await:

Task 1: Rewrite a promise-based function using async/await.

CODE:

```
<html>
  <head>
    <title>My Webpage</title>
  </head>

  <body>
    <script>
      async function fetchData(url) {
        try {
          const response = await fetch(url);
          if (!response.ok) {
            throw new Error("Network response was not ok");
          }
          const data = await response.json();
          return data;
        } catch (error) {
          throw new Error("Fetch error: " + error);
        }
      }

      const apiUrl = "https://jsonplaceholder.typicode.com/posts";

      fetchData(apiUrl)
        .then((data) => {
          console.log("Fetched Data:", data);
        })
        .catch((error) => {
          console.log("Error:", error.message);
        });
    </script>
  </body>
</html>
```

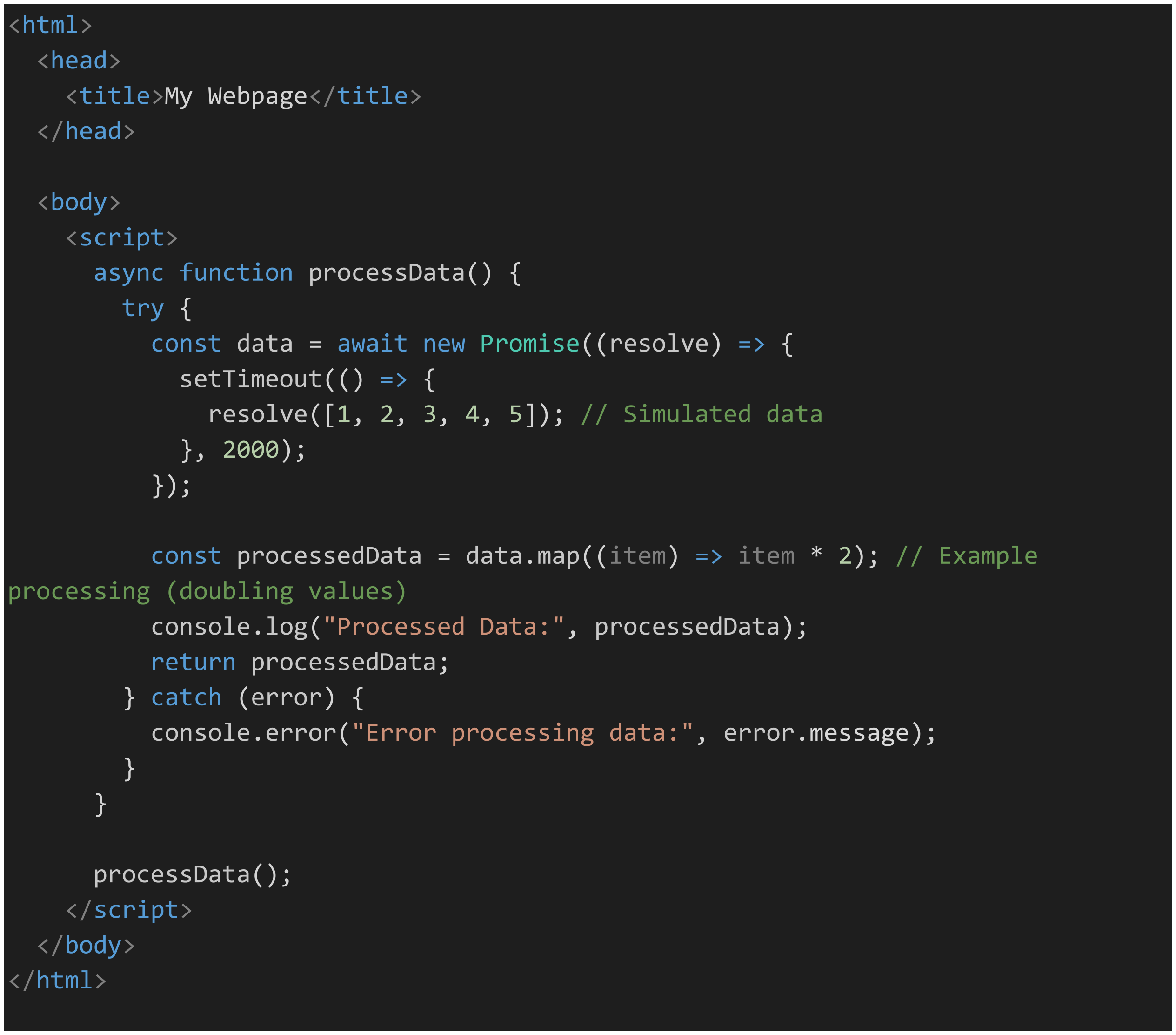
OUTPUT:



```
Fetched Data: ▶ Object                                      tasks.html:39
Error: Fetch error: Network response was not ok             tasks.html:50
Error: Fetch error: Unknown URL                             tasks.html:58
```

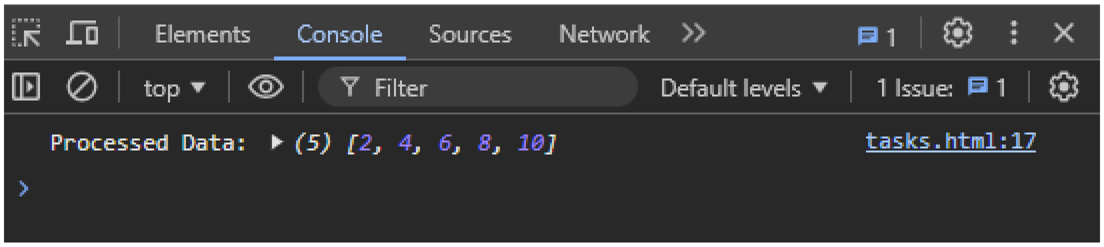
Task 2: Create an async function that fetches data from an API and processes it.

CODE:



```
<html>
  <head>
    <title>My Webpage</title>
  </head>

  <body>
    <script>
      async function processData() {
        try {
          const data = await new Promise((resolve) => {
            setTimeout(() => {
              resolve([1, 2, 3, 4, 5]); // Simulated data
            }, 2000);
          });

          const processedData = data.map((item) => item * 2); // Example
processing (doubling values)
          console.log("Processed Data:", processedData);
          return processedData;
        } catch (error) {
          console.error("Error processing data:", error.message);
        }
      }

      processData();
    </script>
  </body>
</html>
```

OUTPUT:



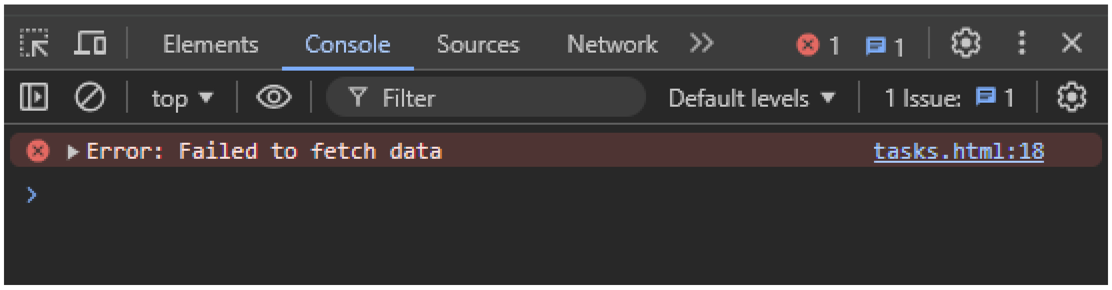
```
Processed Data:  ▶ (5) [2, 4, 6, 8, 10]                tasks.html:17
>
```

Task 3: Implement error handling in an async function using try/catch.

CODE:

```
<html>
  <head>
    <title>My Webpage</title>
  </head>

  <body>
    <script>
      async function fetchData() {
        try {
          const data = await new Promise((resolve, reject) => {
            setTimeout(() => {
              reject("Failed to fetch data");
            }, 1500);
          });

          console.log("Data:", data);
        } catch (error) {
          console.error("Error:", error);
        }
      }

      fetchData();
    </script>
  </body>
</html>
```

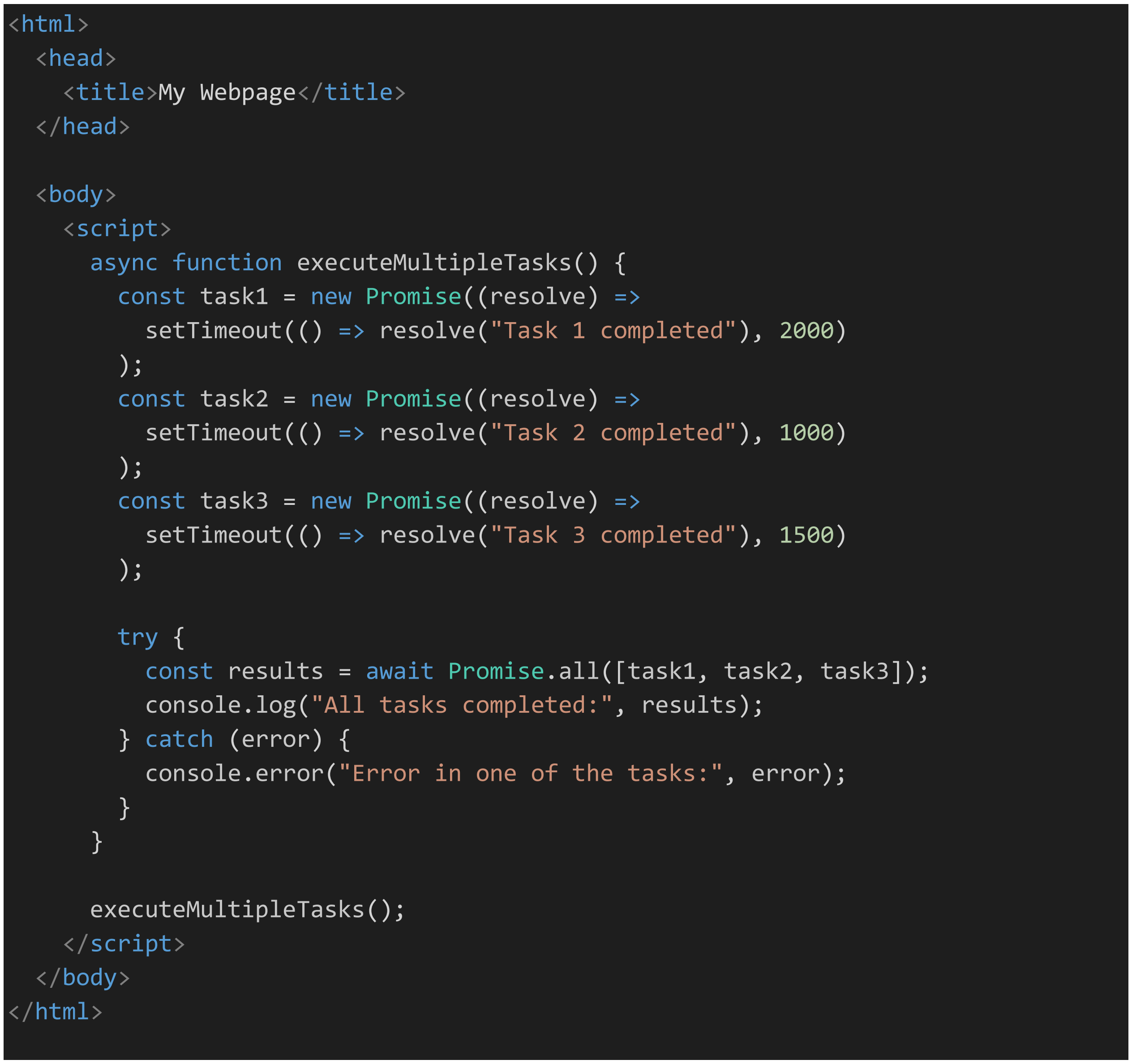
OUTPUT:



```
Error: Failed to fetch data                                    tasks.html:18
```

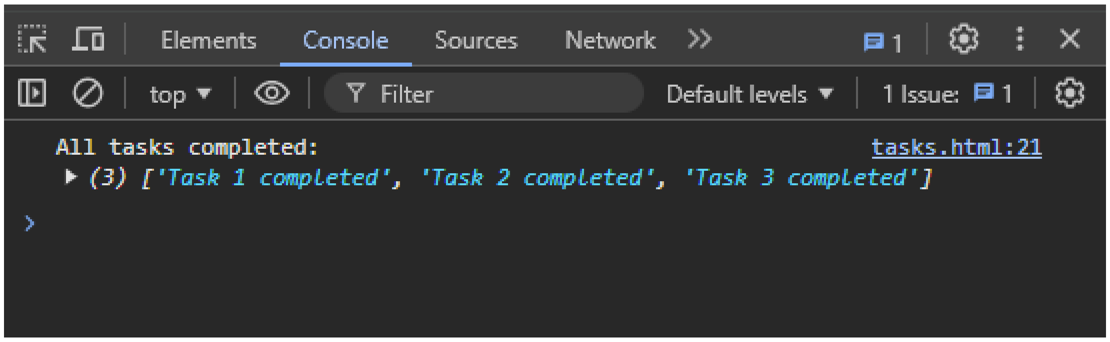
Task 4: Use async/await in combination with Promise.all.

CODE:



```html
<html>
  <head>
    <title>My Webpage</title>
  </head>

  <body>
    <script>
      async function executeMultipleTasks() {
        const task1 = new Promise((resolve) =>
          setTimeout(() => resolve("Task 1 completed"), 2000)
        );
        const task2 = new Promise((resolve) =>
          setTimeout(() => resolve("Task 2 completed"), 1000)
        );
        const task3 = new Promise((resolve) =>
          setTimeout(() => resolve("Task 3 completed"), 1500)
        );

        try {
          const results = await Promise.all([task1, task2, task3]);
          console.log("All tasks completed:", results);
        } catch (error) {
          console.error("Error in one of the tasks:", error);
        }
      }

      executeMultipleTasks();
    </script>
  </body>
</html>
```

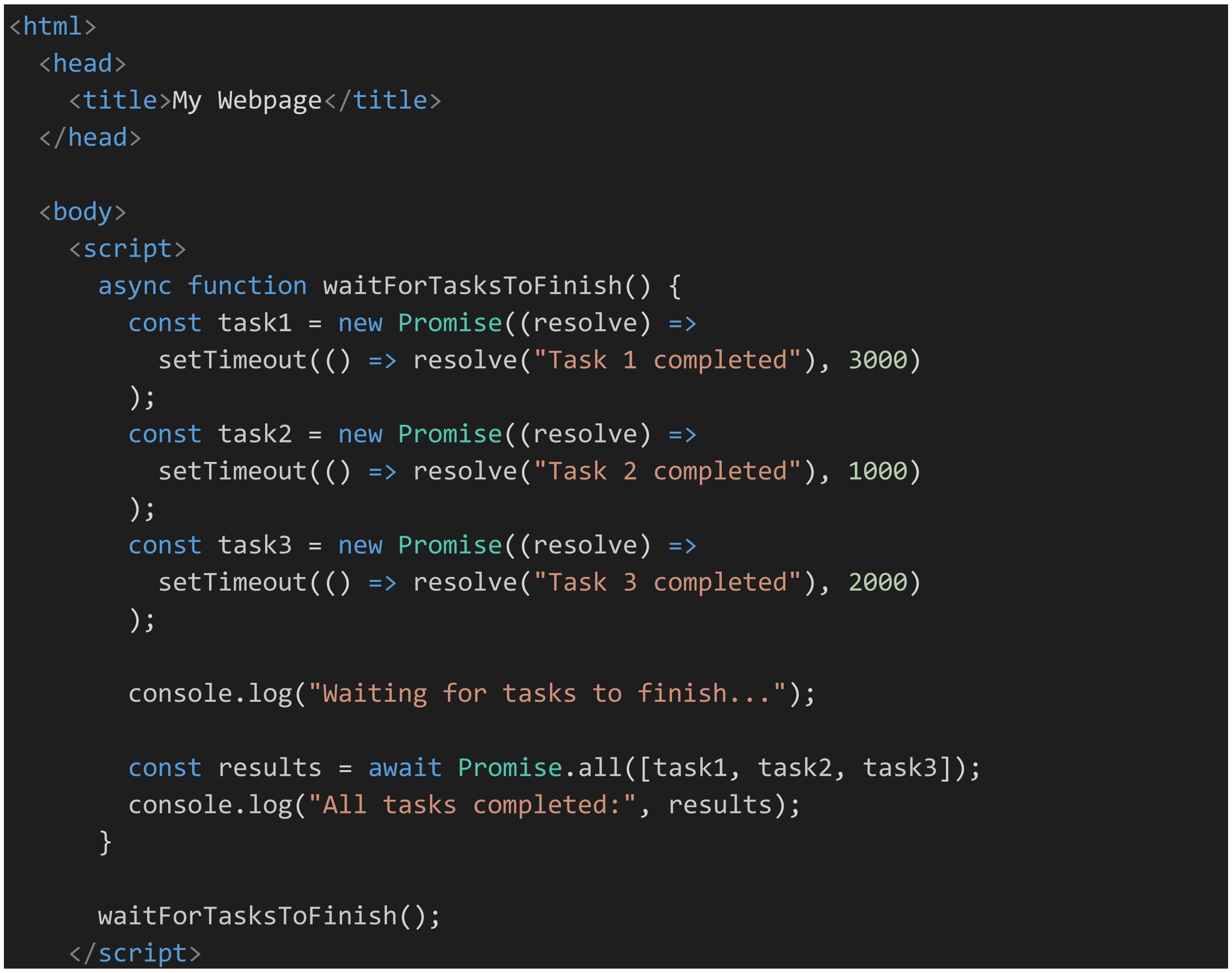
OUTPUT:



```
All tasks completed:                                          tasks.html:21
▸ (3) ['Task 1 completed', 'Task 2 completed', 'Task 3 completed']
```

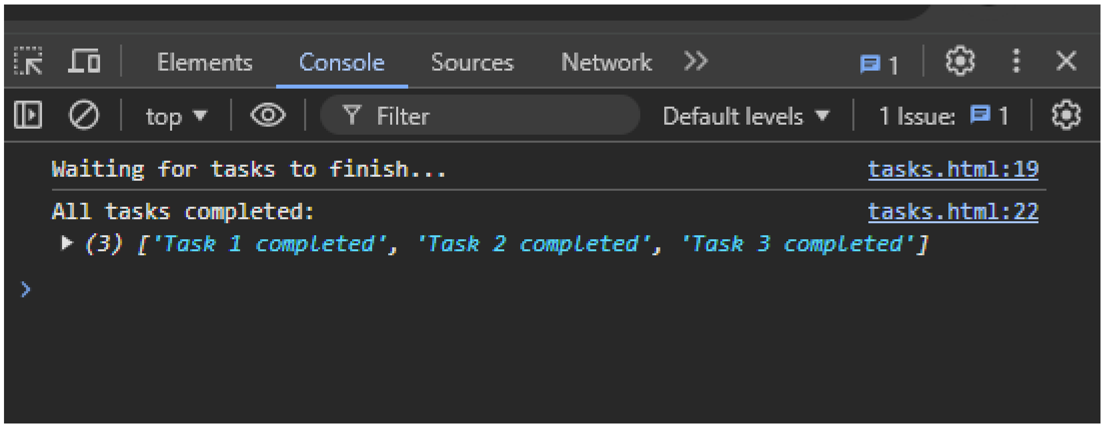
Task 5: Create an async function that waits for multiple asynchronous operations to complete before proceeding.

CODE:



```
<html>
  <head>
    <title>My Webpage</title>
  </head>

  <body>
    <script>
      async function waitForTasksToFinish() {
        const task1 = new Promise((resolve) =>
          setTimeout(() => resolve("Task 1 completed"), 3000)
        );
        const task2 = new Promise((resolve) =>
          setTimeout(() => resolve("Task 2 completed"), 1000)
        );
        const task3 = new Promise((resolve) =>
          setTimeout(() => resolve("Task 3 completed"), 2000)
        );

        console.log("Waiting for tasks to finish...");

        const results = await Promise.all([task1, task2, task3]);
        console.log("All tasks completed:", results);
      }

      waitForTasksToFinish();
    </script>
  </body>
</html>
```

OUTPUT:



```
Waiting for tasks to finish...                                tasks.html:19
All tasks completed:                                          tasks.html:22
 ▶ (3) ['Task 1 completed', 'Task 2 completed', 'Task 3 completed']
>
```