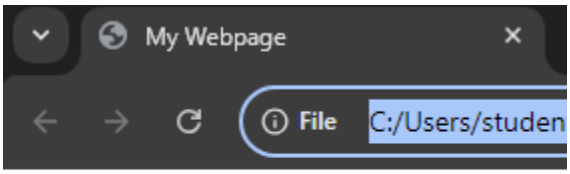
1.Recursion and stack:

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

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
    function fact(n){
if(n==0||n==1)
return n;
else
return n*fact(n-1);
    }
   document.writeln(fact(4));
</script>
</body>
</html>
```

Output:



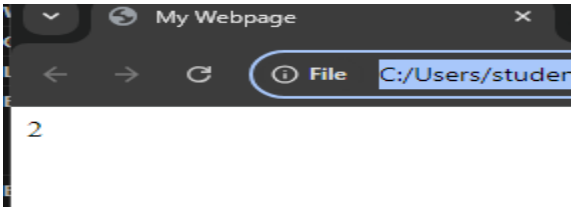
24

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

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
    function fib(n){
if(n==0||n==1)
return n;
else
return fib(n-1)+fib(n-2);
    }
   document.writeln(fib(3));
</script>
</body>
</html>
```

Output:

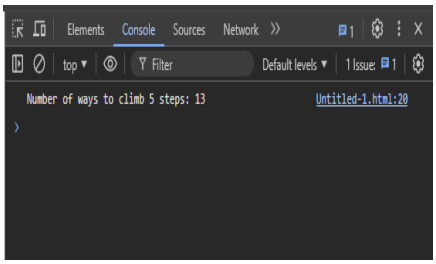


Task 3: Create a function to determine the total number of ways one can climb a staircase with 1, 2, or 3 steps at a time using recursion.

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
   function ways(n) {
    if (n === 0) {
        return 1;
    } else if (n === 1) {
        return 1;
    } else if (n === 2) {
        return 2;
    } else if (n === 3) {
        return 4;
    } else {
        return ways(n - 1) + ways(n - 2) + ways(n - 3);
    }
}
let n = 5;
console.log(`Number of ways to climb ${n} steps: ${ways(n)}`);
</script>
</body>
</html>
```

Output:



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

Code:

```
<html><head>
```

```
    <title>My Webpage</title>
</head>
<body>
<script>
function flattenArray(arr) {
    let result = [];

    for (let i = 0; i < arr.length; i++) {
        if (Array.isArray(arr[i])) {
            result = result.concat(flattenArray(arr[i]));
        } else {
            result.push(arr[i]);
        }
    }
    return result;
}
let nestedArray = [1, [2, [3, 4]], 5, [6, 7], 8];
console.log(flattenArray(nestedArray));
</script>
</body>
</html>
```

Output:

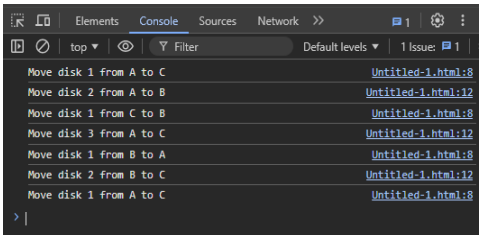
Task 5: Implement the recursive Tower of Hanoi solution.

Code:

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

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

Output:



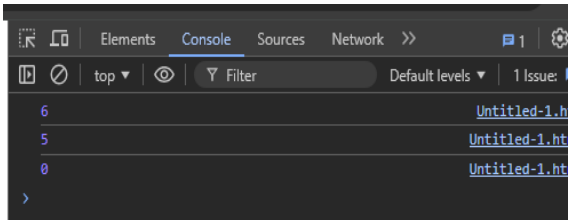
2. JSON and variable length arguments/spread syntax:

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

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function sumOfArguments(...args) {
    return args.reduce((sum, currentValue) => sum + currentValue, 0);
}
console.log(sumOfArguments(1, 2, 3));
console.log(sumOfArguments(5));
console.log(sumOfArguments());
</script>
</body>
</html>
```

Output:



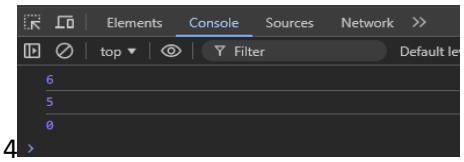
Task 2: Modify a function to accept an array of numbers and return their sum using the spread syntax.

Code:

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

```
function sumOfArguments(...args) {
    return args.reduce((sum, currentValue) => sum + currentValue, 0);
}
console.log(sumOfArguments(...[1, 2, 3]));
console.log(sumOfArguments(...[5]));
console.log(sumOfArguments(...[]));
</script>
</body>
</html>
```

Output:



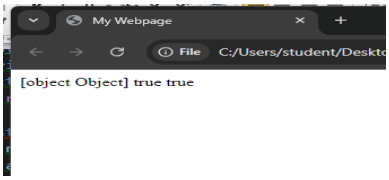
4

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

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function deepClone(obj) {
    return JSON.parse(JSON.stringify(obj));
}
const originalObject = {
    name: 'John',
    age: 30,
    address: {
        city: 'New York',
        zip: '10001'
    }
};
const clonedObject = deepClone(originalObject);
document.writeln(clonedObject);
document.writeln(clonedObject !== originalObject);
document.writeln(clonedObject.address !== originalObject.address);
</script>
</body>
</html>
```

Output:

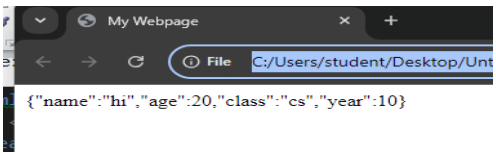


Task 4: Write a function that returns a new object, merging two provided objects using the spread syntax.

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
let obj1={
    name:"hi",
    age:20
}
let obj2={
    class:"cs",
    year:10
}
let obj={...obj1,...obj2}
document.writeln(JSON.stringify(obj))
</script>
</body>
</html>
```

Output:



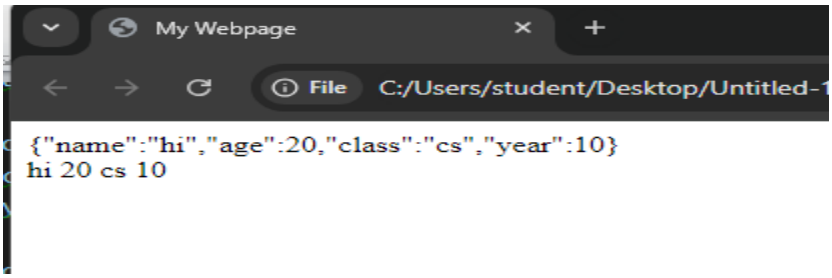
Task 5: Serialize a JavaScript object into a JSON string and then parse it back into an object.

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
let obj1={
    name:"hi",
    age:20
}
let obj2={
    class:"cs",
    year:10
```

```
}
let obj={...obj1,...obj2}
let aaa=JSON.stringify(obj)
document.writeln(aaa)
document.writeln("<br>")
let bbb=JSON.parse(aaa)
document.writeln(bbb.name+" "+bbb.age+" "+bbb.class+" "+bbb.year)
</script>
</body>
</html>
```

Output:



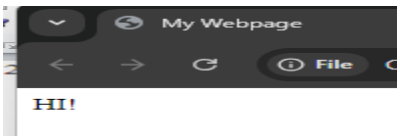
3. Closure:

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

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function fun(message) {
    return function fun2(){
        document.writeln(message);
    }
}
let great=fun("HI!")
great();
</script>
</body>
</html>
```

Output:

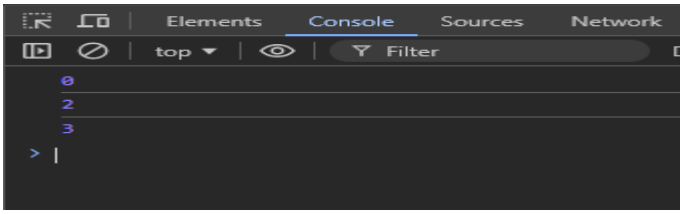


Task 2: Implement a basic counter function using closure, allowing incrementing and displaying the current count.

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function createCounter() {
    let count = 0;
    return {
        increment: function() {
            count++;
        },
        getCurrentCount: function() {
            return count;
        }
    };
}
const counter = createCounter();
console.log(counter.getCurrentCount());
counter.increment();
counter.increment();
console.log(counter.getCurrentCount());
counter.increment();
console.log(counter.getCurrentCount());
</script>
</body>
</html>
```

Output:



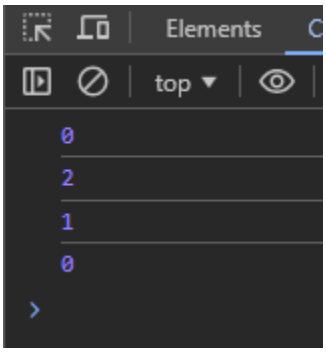
Task 3: Write a function to create multiple counters, each with its own separate count.

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function createCounter() {
    let count = 0;
    return {
        increment: function() {
            count++;
```

```
        },
        decrement: function() {
            count--;
        },
        getCurrentCount: function() {
            return count;
        }
    };
}
const counter1 = createCounter();
const counter2 = createCounter();
console.log(counter1.getCurrentCount());
counter1.increment();
counter1.increment();
console.log(counter1.getCurrentCount());
counter2.increment();
console.log(counter2.getCurrentCount());
counter2.decrement();
console.log(counter2.getCurrentCount());
</script>
</body>
</html>
```

Output:



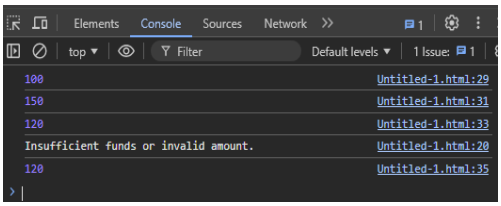
Task 4: Use closures to create private variables within a function.

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function createBankAccount(initialBalance) {
    let balance = initialBalance;
    return {
        deposit: function(amount) {
            if (amount > 0) {
                balance += amount;
            } else {
```

```
                console.log("Deposit amount must be positive.");
            }
        },
        withdraw: function(amount) {
            if (amount > 0 && amount <= balance) {
                balance -= amount;
            } else {
                console.log("Insufficient funds or invalid amount.");
            }
        },
        getBalance: function() {
            return balance;
        }
    };
}
const account = createBankAccount(100);
console.log(account.getBalance())
account.deposit(50);
console.log(account.getBalance());
account.withdraw(30);
console.log(account.getBalance());
account.withdraw(200);
console.log(account.getBalance());
</script>
</body>
</html>
```

Output:



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

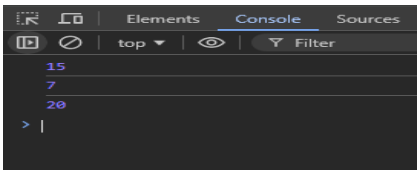
Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function createMathFunction(operation) {
    return function(num) {
        if (operation === 'add') {
            return function(x) { return x + num; };
        } else if (operation === 'subtract') {
            return function(x) { return x - num; };
        } else if (operation === 'multiply') {
```

```
            return function(x) { return x * num; };
        } else {
            return function() { return 'Invalid operation'; };
        }
    };
}
const add5 = createMathFunction('add')(5);
console.log(add5(10));
const subtract3 = createMathFunction('subtract')(3);
console.log(subtract3(10));
const multiply2 = createMathFunction('multiply')(2);
console.log(multiply2(10));

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

Output:



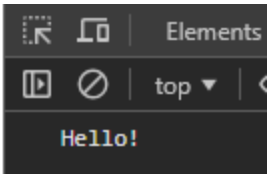
4. Promise, Promises chaining:

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

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function greetAfterSeconds() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Hello!");
        }, 1000);
    });
}
greetAfterSeconds().then((greeting) => {
    console.log(greeting);
});
</script>
</body>
</html>
```

Output:



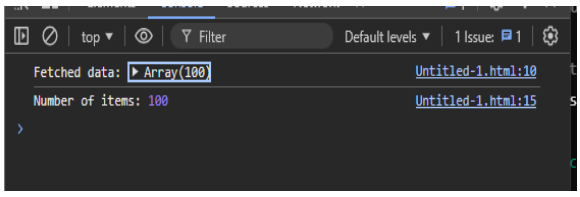
Task 2: Fetch data from an API using promises, and then chain another promise to process this data.

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function fetchData(url) {
    return fetch(url)
        .then(response => response.json())
        .then(data => {
            console.log('Fetched data:', data);
            return data;
        })
        .then(data => {
            const count = data.length;
            console.log('Number of items:', count);
        })
        .catch(error => {
            console.log('Error:', error);
        });
}
const apiUrl = 'https://jsonplaceholder.typicode.com/posts';
fetchData(apiUrl);

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

Output:



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

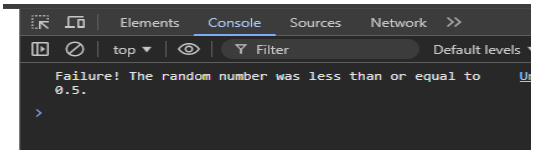
Code:

```
<html><head>

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

```
<script>
function randomPromise() {
    return new Promise((resolve, reject) => {
        const randomNumber = Math.random();
        if (randomNumber > 0.5) {
            resolve("Success! The random number was greater than 0.5.");
        } else {
            reject("Failure! The random number was less than or equal to 0.5.");
        }
    });
}
randomPromise()
    .then((message) => {
        console.log(message);
    })
    .catch((error) => {
        console.log(error);
    });
</script>
</body>
</html>
```

Output:



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

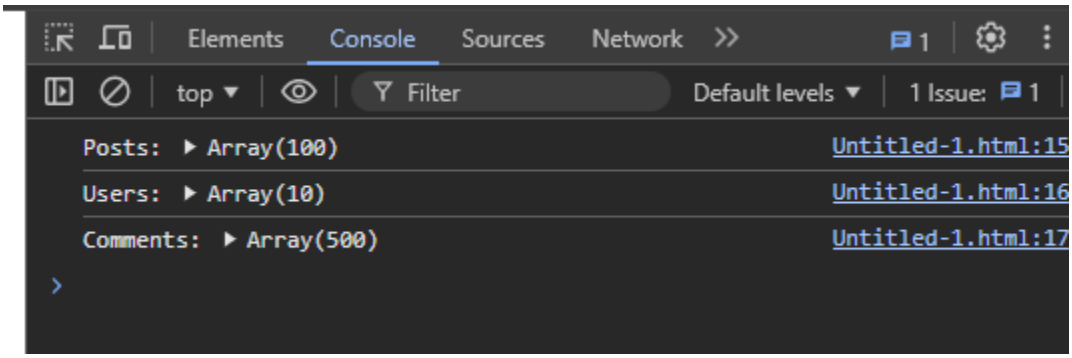
Code:

```
<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:



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

Code:

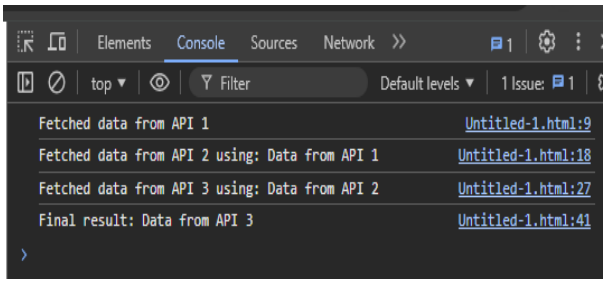
```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function fetchDataFromAPI1() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Fetched data from API 1");
            resolve("Data from API 1");
        }, 1000);
    });
}

function fetchDataFromAPI2(data) {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log(`Fetched data from API 2 using: ${data}`);
            resolve("Data from API 2");
        }, 1000);
        });
```

```
        }

function fetchDataFromAPI3(data) {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log(`Fetched data from API 3 using: ${data}`);
            resolve("Data from API 3");
        }, 1000);
    });
}
function chainPromises() {
    fetchDataFromAPI1()
        .then((data1) => {
            return fetchDataFromAPI2(data1);
        })
        .then((data2) => {
            return fetchDataFromAPI3(data2);
        })
        .then((data3) => {
            console.log(`Final result: ${data3}`);
        })
        .catch((error) => {
            console.error("Error:", error);})
}
chainPromises();
</script>
</body>
</html>
```

Output:



5.Async/await:

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

Code:

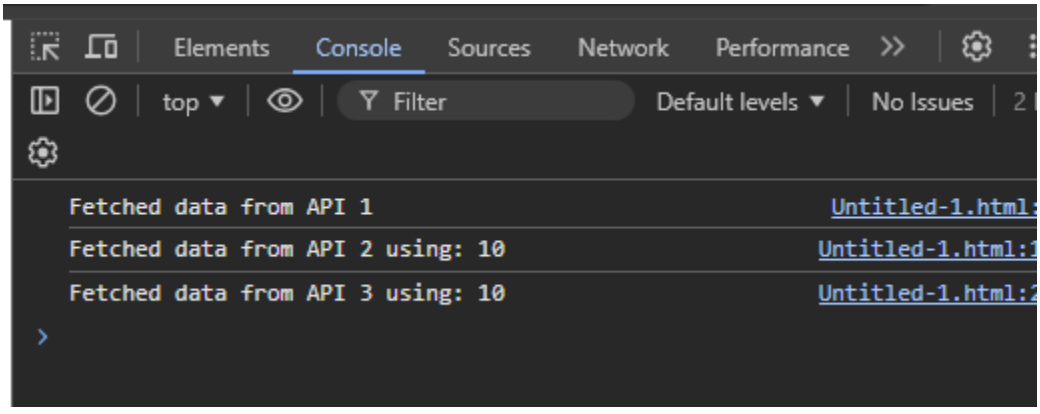
```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
function fetchDataFromAPI1() {
```

```
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Fetched data from API 1");
            resolve("Data from API 1");
        }, 1000);
    });
}

function fetchDataFromAPI2(data) {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log(`Fetched data from API 2 using: ${data}`);
            resolve("Data from API 2");
        }, 1000);
        });
        }

function fetchDataFromAPI3(data) {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log(`Fetched data from API 3 using: ${data}`);
            resolve("Data from API 3");
        }, 1000);
    });
}
async function chainPromises() {
    let data1=10;
    const a=await fetchDataFromAPI1();
   const b=await fetchDataFromAPI2(data1);
   const c=await fetchDataFromAPI3(data1);
   document.writeln(status)
}
chainPromises();
</script>
</body>
</html>
```

Output

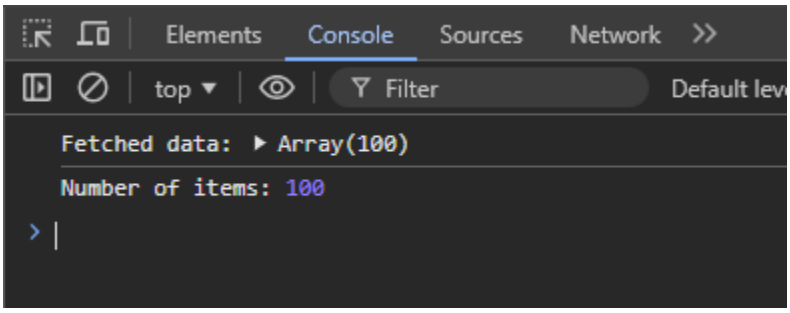


Task 2: Create an async function that fetches data from an API and processes it.

Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
async function fetchDataAndProcess(url) {
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error("Network response was not ok");
        }
        const data = await response.json();
        console.log("Fetched data:", data);
        return data.length;
    } catch (error) {
        console.error("Error fetching data:", error);
    }
}
const apiUrl = 'https://jsonplaceholder.typicode.com/posts';
fetchDataAndProcess(apiUrl).then((result) => {
    if (result !== undefined) {
        console.log('Number of items:', result);
    }
});
</script>
</body>
</html>
```

Output:



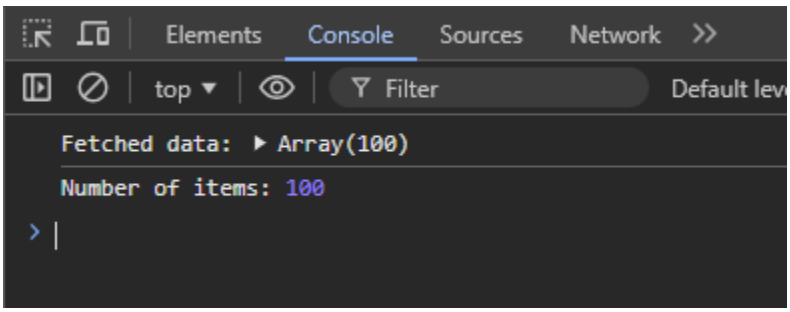
Task 3: Implement error handling in an async function using try/catch.

Code:

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

```
<script>
async function fetchDataAndProcess(url) {
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error("Network response was not ok");
        }
        const data = await response.json();
        console.log("Fetched data:", data);
        return data.length;
    } catch (error) {
        console.error("Error fetching data:", error);
    }
}
const apiUrl = 'https://jsonplaceholder.typicode.com/posts';
fetchDataAndProcess(apiUrl).then((result) => {
    if (result !== undefined) {
        console.log('Number of items:', result);
    }
});
</script>
</body>
</html>
```

Output:



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

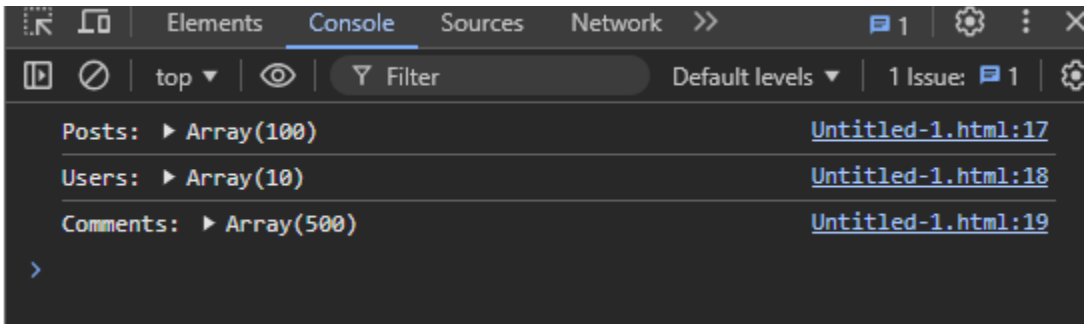
Code:

```
<html><head>
    <title>My Webpage</title>
</head>
<body>
<script>
async function fetchMultipleResources() {
    const urls = [
        'https://jsonplaceholder.typicode.com/posts',
        'https://jsonplaceholder.typicode.com/users',
        'https://jsonplaceholder.typicode.com/comments'
    ];
```

```
    try {
        const fetchPromises = urls.map(url => fetch(url).then(response => response.json()));
        const results = await Promise.all(fetchPromises);

        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:



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 waitForMultipleOperations() {
    try {
        const operation1 = new Promise((resolve) => setTimeout(() => resolve('Operation 1 Complete'), 2000));
        const operation2 = new Promise((resolve) => setTimeout(() => resolve('Operation 2 Complete'), 3000));
        const operation3 = new Promise((resolve) => setTimeout(() => resolve('Operation 3 Complete'), 1000));

        const results = await Promise.all([operation1, operation2, operation3]);
        console.log('All operations completed:');
        results.forEach(result => console.log(result));

    } catch (error) {
```

```
        console.error('Error:', error);
    }
}
waitForMultipleOperations();
</script>
</body>
</html>
```

Output: