

Task-1

Aim: Basic Promises

- Create a function `fetchData` that simulates fetching data from an API using a Promise.
- Inside the function, use `setTimeout` to simulate a delay of 2 seconds.
- If the data is successfully fetched, resolve the Promise with the fetched data. If there is an error, reject the Promise with an appropriate error message.
- Implement a function `handleData` that uses `fetchData` and handles both the resolved and rejected cases. Log the data or error message to the console accordingly.

Description:

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

Source Code:

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Data successfully fetched");
      } else {
        reject("Error fetching data");
      }
    }, 2000);
  });
}

function handleData() {
  fetchData()
    .then((data) => {
      console.log("Resolved:", data);
    })
    .catch((error) => {
      console.error("Rejected:", error);
    });
}

handleData();
```

Output:



Task-2

Aim: Chaining Promises

- Create three functions: `fetchFirstData`, `fetchSecondData`, and `fetchThirdData`, each simulating fetching data with a 2-second delay.
- Use Promise chaining to fetch data sequentially: fetch the first data, then the second data, and finally the third data.
- Log the combined result of all three data fetches.

Description:

A common need is to execute two or more asynchronous operations back-to-back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step.

Source Code:

```
let firstData;
let secondData;

function fetchFirstData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("First Data");
    }, 2000);
  });
}

function fetchSecondData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Second Data");
    }, 2000);
  });
}

function fetchThirdData() {
```

```
return new Promise((resolve) => {
  setTimeout(() => {
    resolve("Third Data");
  }, 2000);
});
}

fetchFirstData()
  .then((data) => {
    firstData = data;
    return fetchSecondData();
  })
  .then((data) => {
    secondData=data
    return fetchThirdData()
  })
  .then((thirdData) => {
    console.log("Combined Result:", firstData, secondData, thirdData);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

Output:

Combined Result: First Data Second Data Third Data

[index.html:50](#)

Task-3

Aim: Promise.all

- Create three functions: fetchData1, fetchData2, and fetchData3, each simulating fetching data with a 2-second delay.
- Use Promise.all to fetch all three sets of data concurrently.
- Log the combined result of all three data fetches.

Description:

The Promise.all() static method takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises fulfill (including when an empty iterable is passed), with an array of the fulfillment values. It rejects when any of the input's promises rejects, with this first rejection reason.

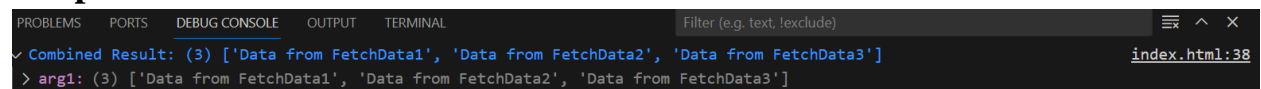
Source Code:

```
function fetchData1() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data from FetchData1");
    }, 2000);
  });
}

function fetchData2() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data from FetchData2");
    }, 2000);
  });
}

function fetchData3() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data from FetchData3");
    }, 2000);
  });
}

Promise.all([fetchData1(), fetchData2(), fetchData3()])
  .then((results) => {
    console.log("Combined Result:", results);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

Output:

PROBLEMS PORTS DEBUG CONSOLE OUTPUT TERMINAL Filter (e.g. text, !exclude) index.html:38

✓ Combined Result: (3) ['Data from FetchData1', 'Data from FetchData2', 'Data from FetchData3']

> arg1: (3) ['Data from FetchData1', 'Data from FetchData2', 'Data from FetchData3']

Task-4

Aim: Basic Async Function

- Create an asynchronous function `fetchData` that simulates fetching data from an API using `setTimeout` with a delay of 2 seconds.
- Use `async/await` to handle the asynchronous operation.
- If the data is successfully fetched, return the data. If there is an error, throw an error with an appropriate message.
- Implement a function `handleData` that calls `fetchData` and logs the data or error message to the console.

Description:

The `async` function declaration creates a binding of a new `async` function to a given name. The `await` keyword is permitted within the function body, enabling asynchronous, promise-based behavior to be written in a cleaner style and avoiding the need to explicitly configure promise chains.

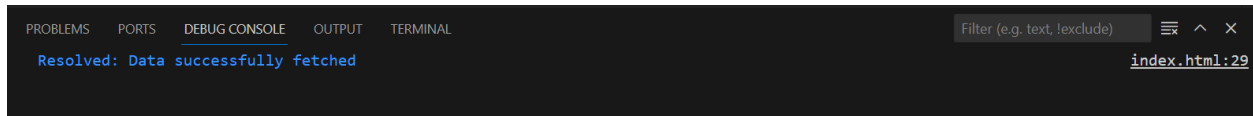
Source Code:

```
async function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Data successfully fetched");
      } else {
        reject("Error fetching data");
      }
    }, 2000);
  });
}

async function handleData() {
  try {
    const data = await fetchData();
    console.log("Resolved:", data);
  } catch (error) {
    console.error("Rejected:", error);
  }
}

handleData();
```

Output:



Task-5

Aim: Sequential Async Operations

- Create three asynchronous functions: `fetchFirstData`, `fetchSecondData`, and `fetchThirdData`, each simulating fetching data with a 2-second delay.
- Use `async/await` to fetch data sequentially: fetch the first data, then the second data, and Finally, the third data.
- Log the combined result of all three data fetches.

Description:

Sequential Async Operations refer to a scenario where asynchronous tasks are executed in a specific order, one after the other. In JavaScript, this often involves using the `async/await` syntax to handle promises in a more synchronous-looking manner.

Source Code:

```
async function fetchFirstData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("First Data");
    }, 2000);
  });
}

async function fetchSecondData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Second Data");
    }, 2000);
  });
}

async function fetchThirdData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Third Data");
    }, 2000);
  });
}
```

```
});  
}  
  
async function sequentialAsyncOperations() {  
  const firstData = await fetchFirstData();  
  const secondData = await fetchSecondData();  
  const thirdData = await fetchThirdData();  
  
  console.log("Combined Result:", firstData, secondData, thirdData);  
}  
sequentialAsyncOperations();
```

Output:



Task-6

Aim: Parallel Async Operations

- Create three asynchronous functions: fetchData1, fetchData2, and fetchData3, each simulating fetching data with a 2-second delay.
- Use Promise.all with async/await to fetch all three sets of data concurrently.
- Log the combined result of all three data fetches.

Description:

Parallel Async Operations refer to a scenario where multiple asynchronous tasks are executed concurrently or simultaneously, without waiting for the completion of one before starting the next. In JavaScript, this can be achieved using features such as Promise.all and asynchronous functions.

Source Code:

```
async function fetchData1() {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve("Data from FetchData1");  
    }, 2000);  
  });  
}
```

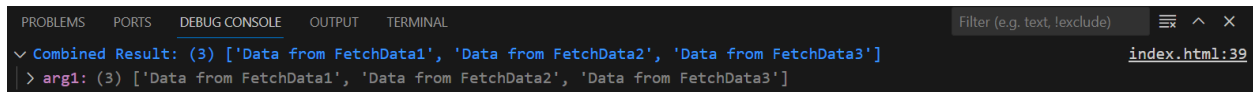
```
async function fetchData2() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data from FetchData2");
    }, 2000);
  });
}

async function fetchData3() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data from FetchData3");
    }, 2000);
  });
}

async function parallelAsyncOperations() {
  const results = await Promise.all([fetchData1(), fetchData2(), fetchData3()]);
  console.log("Combined Result:", results);
}

parallelAsyncOperations();
```

Output:



```
PROBLEMS  PORTS  DEBUG CONSOLE  OUTPUT  TERMINAL
v Combined Result: (3) ['Data from FetchData1', 'Data from FetchData2', 'Data from FetchData3']
| > arg1: (3) ['Data from FetchData1', 'Data from FetchData2', 'Data from FetchData3']
index.html:39
```

Task-7

Aim: Creating Modules

- Create a module mathOperations that exports functions for basic mathematical operations (addition, subtraction, multiplication, division).
- Create another module stringOperations that exports functions for string manipulation (concatenation, uppercase, lowercase).
- Import both modules in a main script and use functions from both modules to perform a variety of operations.

Description:

Creating modules in JavaScript involves organizing code into separate files, each containing related functionality, variables, or classes. Modules help improve code maintainability,

readability, and reusability. In JavaScript, there are different ways to create modules, and one common approach is using the ES6 module syntax.

Source Code:

```
//main.js
import * as mathOps from "./mathOperations.js";
import * as stringOps from "./stringOperations.js";

// Math Operations
const resultAdd = mathOps.add(5, 3);
const resultSubtract = mathOps.subtract(10, 4);
const resultMultiply = mathOps.multiply(2, 6);
const resultDivide = mathOps.divide(8, 2);

console.log("Math Operations:");
console.log("Addition:", resultAdd);
console.log("Subtraction:", resultSubtract);
console.log("Multiplication:", resultMultiply);
console.log("Division:", resultDivide);

// String Operations
const resultConcatenate = stringOps.concatenate("Hello", "World");
const resultUpperCase = stringOps.toUpperCase("hello");
const resultLowerCase = stringOps.toLowerCase("WORLD");

console.log("\nString Operations:");
console.log("Concatenation:", resultConcatenate);
console.log("Uppercase:", resultUpperCase);
console.log("Lowercase:", resultLowerCase);
```

```
//mathOperations.js
export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}

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

```
}  
  
export function divide(a, b) {  
  return a / b;  
}
```

```
//stringOperations.js  
export function concatenate(str1, str2) {  
  return str1 + str2;  
}  
  
export function toUpperCase(str) {  
  return str.toUpperCase();  
}  
  
export function toLowerCase(str) {  
  return str.toLowerCase();  
}
```

Output:

Live reload enabled.	index.html:39
Math Operations:	main.js:10
Addition: 8	main.js:11
Subtraction: 6	main.js:12
Multiplication: 12	main.js:13
Division: 4	main.js:14
String Operations:	main.js:21
Concatenation: HelloWorld	main.js:22
Uppercase: HELLO	main.js:23
Lowercase: world	main.js:24

Task-8

Aim: Asynchronous Module Loading

- Create a module asyncModule that exports an asynchronous function which simulates fetching data with a 3-second delay.
- In the main script, use dynamic import to load the asyncModule asynchronously.
- Call the asynchronous function from the imported module and log the result.

Description:

Asynchronous Module Loading (AML) refers to the ability to load modules dynamically, on-demand, and asynchronously in a JavaScript application. This concept is closely related to the

idea of lazy loading, where modules are only loaded when they are actually needed, rather than at the beginning when the application starts.

Source Code:

```
//main.js
(async () => {
  const asyncModule = await import("./asyncModule.js");
  const result = await asyncModule.fetchDataAsync();
  console.log("Async Module Result:", result);
})();
```

```
//asyncModule.js
export async function fetchDataAsync() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data from Async Module");
    }, 3000);
  });
}
```

Output:

Live reload enabled.

Async Module Result: Data from Async Module

[index.html:38](#)

[index.html:53](#)

Task-9

Aim: Default Export

- Create a module defaultExportModule that exports a default function to calculate the square of a number.
- Import the module in a main script and use the default exported function to calculate the square of a given number.

Description:

Default export in JavaScript allows a module to specify a single "default" export that can be imported without specifying a name. It provides a convenient way to export a primary entity or functionality from a module, and when importing, you can choose to give it any name you prefer.

Source Code:

```
//main.js
import square from "./defaultExportModule.js";

const numberToSquare = 5;
const result = square(numberToSquare);

console.log(`Square of ${numberToSquare} is:`, result);
```

```
//defaultExportModule.js
export default function square(number) {
  return number * number;
}
```

Output:

Live reload enabled.

Square of 5 is: 25

[index.html:38](#)

[main.js:7](#)

Learning Outcome:

Overall, the provided code examples cover various important aspects of web development, including HTML structure, Promises, Chaining Promises, Promise.all, Async Function, Sequential Async Operations, Parallel Async Operations, Creating Modules, Asynchronous Module Loading and Default Export JavaScript.