EXCEPTION &EXCEPTION HANDLING

What is an Exception?

An **exception** in JavaScript is an unexpected error that occurs during the execution of code. It interrupts the normal flow of the program, causing it to terminate unless the error is handled properly.

What is Exception Handling?

Exception handling is the process of catching and managing these errors in a way that allows the program to continue running or fail gracefully. In JavaScript, this is primarily done using try...catch blocks.

Key Elements of Exception Handling in JavaScript

1. **try Block**: Code that may throw an error is placed here.
2. **catch Block**: Handles the error if one is thrown in the try block.
3. **finally Block** *(Optional)*: Executes code after try and catch, regardless of the outcome.
4. **throw Statement**: Used to throw a custom error.

Syntax:

```
try {
  // Code that may throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code that runs regardless of success or failure (optional)
}
```

Example 1: Basic Exception Handling

```
try {
  let result = 10 / 0; // No error in JavaScript, result will be Infinity
  console.log(result);
  let num = parseInt("abc"); // Throws NaN but not an exception
  if (isNaN(num)) {
    throw new Error("Invalid number format");
  }
} catch (error) {
  console.log("Error caught:", error.message);
} finally {
  console.log("Execution completed.");
}
```

Output:

```
Infinity
Error caught: Invalid number format
Execution completed.
```

Example 2: Handling a Specific Error

```
try {
  let arr = [1, 2, 3];
  console.log(arr[5].toString()); // Throws a TypeError
} catch (error) {
  if (error instanceof TypeError) {
    console.log("TypeError caught:", error.message);
  } else {
    console.log("Unexpected error:", error.message);
  }
}
```

Output:

```
TypeError caught: Cannot read properties of undefined (reading 'toString')
```

Example 3: Custom Errors with throw

```
function validateAge(age) {
  if (age < 18) {
    throw new Error("Age must be 18 or older.");
  }
  console.log("Age is valid.");
}
try {
  validateAge(16);
} catch (error) {
  console.log("Validation error:", error.message);
} finally {
  console.log("Validation process complete.");
}
```

Output:

```
Validation error: Age must be 18 or older.
Validation process complete.
```

Example 4: Using finally

```
try {
  console.log("Trying...");
  let x = 10 / 0;
} catch (error) {
  console.log("An error occurred:", error.message);
} finally {
  console.log("This block always runs.");
}
```

Output :

```
Trying...
This block always runs.
```

Why Use Exception Handling?

1. Prevent application crashes.
2. Display meaningful error messages to users.
3. Allow program execution to continue after handling errors.
4. Help developers debug issues effectively.

TASKS

1. Write a program that divides two numbers. Use a try...catch block to handle division by zero and display an error message when this occurs.

2. Create a nested try...catch block to handle different types of errors (e.g., ReferenceError, TypeError) in the same program.

3. Write a function that takes a number as input and throws a custom error if the number is negative. Use try...catch to handle the error and display the error message.

4. Write a program that reads an array and tries to access an out-of-bounds index. Use finally to print a message indicating that the program has finished execution.

5. **Validation with Custom Errors**: Write a program that validates user input (e.g., age, email). Throw custom errors for invalid inputs and use try...catch to provide appropriate error messages.

CALLBACK & CALLBACK HELL

CallBack

A callback function, often simply referred to as a "callback," is a function that is passed as an argument to another function and is executed after the completion of that function. Callbacks are commonly used in asynchronous programming and event handling to ensure that certain code is executed at the appropriate time.

Example

```
function doTask(taskName, callback) {
  console.log(`Starting: ${taskName}`);
  setTimeout(() => {
    console.log(`Completed: ${taskName}`);
    callback(); // Execute the callback function after the task is done
  }, 1000);
}

doTask("Task 1", () => {
  console.log("Task 1 finished! Ready for the next task.");
});
```

OutPut

```
Starting: Task 1
Completed: Task 1
Task 1 finished! Ready for the next task.
```

Callback Hell

Callback Hell refers to a situation where multiple asynchronous operations are nested within each other, creating deeply nested and hard-to-read code. It is also known as the "pyramid of doom" due to its structure.

Example

Suppose you need to:

1. Fetch user data.
2. Fetch the user’s orders based on the user data.
3. Process a payment for the orders.
4. Send a confirmation email after payment

.Here's how the code might look with nested callbacks:

```
function getUser(userId, callback) {
  setTimeout(() => {
    console.log("Fetched user data");
    callback({ userId, name: "Alice" });
  }, 1000);
}
function getOrders(userId, callback) {
  setTimeout(() => {
    console.log("Fetched orders for user:", userId);
    callback(["order1", "order2"]);
  }, 1000);
}
function processPayment(order, callback) {
  setTimeout(() => {
    console.log("Processed payment for:", order);
    callback("Payment Successful");
  }, 1000);
}
```

```
function sendEmail(status, callback) {
  setTimeout(() => {
    console.log("Email sent:", status);
    callback("Email Sent Successfully");
  }, 1000);
}
// Nested callbacks (Callback Hell)
getUser(1, (user) => {
  getOrders(user.userId, (orders) => {
    processPayment(orders[0], (paymentStatus) => {
      sendEmail(paymentStatus, (emailStatus) => {
        console.log(emailStatus);
      });
    });
  });
});
```

OutPut

```
Fetched user data
Fetched orders for user: 1
Processed payment for: order1
Email sent: Payment Successful
Email Sent Successfully
```

Problems with Callback Hell

1. **Poor Readability**: The code is hard to follow due to deeply nested callbacks.
2. **Difficult to Maintain**: Modifying or debugging the code is challenging.
3. **Error Handling is Complicated**: Adding proper error handling for each callback increases complexity.

How to Avoid Callback Hell?

1. **Using Promises**
   A Promise is an object in JavaScript that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It is used to handle asynchronous operations more effectively than callbacks.

A Promise has three states:

1. **Pending**: The initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation was successful, and the promise returns a resolved value.
3. **Rejected**: The operation failed, and the promise returns a reason for the failure.

Syntax:

```
const promise = new Promise((resolve, reject) => {
  // Perform some asynchronous operation
  if (successCondition) {
    resolve("Operation successful!");
  } else {
    reject("Operation failed!");
  }
});
```

Example :

```
const fetchData = new Promise((resolve, reject) => {
  const data = true; // Simulating a condition
  setTimeout(() => {
    if (data) {
      resolve("Data fetched successfully!");
    } else {
      reject("Failed to fetch data!");
    }
  }, 1000);
});
fetchData
  .then((result) => {
    console.log(result); // Output: Data fetched successfully!
  })
  .catch((error) => {
    console.error(error);
  });
```

What is Promise Chaining?

Promise Chaining is the process of chaining multiple .then() calls together to perform a sequence of asynchronous operations. Each .then() in the chain returns a new promise, allowing operations to be executed in a specific order.

Example

```
function fetchUser() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Fetched user");
      resolve({ userId: 1, name: "Alice" });
    }, 1000);
  });
}
function fetchOrders(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(`Fetched orders for user ${userId}`);
      resolve(["order1", "order2"]);
    }, 1000);
  });
}
function processOrder(order) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(`Processed ${order}`);
      resolve("Order processed successfully!");
    }, 1000);
  });
}
// Promise Chaining
fetchUser()
  .then((user) => {
    return fetchOrders(user.userId);
  })
  .then((orders) => {
    return processOrder(orders[0]);
  })
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

OutPut

```
Fetched user

Fetched orders for user 1

Processed order1

Order processed successfully!
```

Static Method

Promise.all()

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.

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values);
});
// Expected output: Array [3, 42, "foo"]
```

Promise.allSettled()

Promise.allSettled() method in JavaScript is used to handle multiple promises concurrently and return a single promise. This promise is fulfilled with an array of promise state descriptors, each describing the outcome of the corresponding promise in the input array. Unlike Promise.all(), Promise.allSettled() does not short-circuit when one of the promises is rejected; instead, it waits for all promises to settle, providing information about each one.

Syntax:

```
Promise.allSettled(iterable);
```

Example 1: In this example, we will use the Promise **allSettled()** Method

```
// Illustration of Promise.allSettled()
// Method in Javascript with Example

const p1 = Promise.resolve(50);
const p2 = new Promise((resolve, reject) =>
   setTimeout(reject, 100, 'geek'));
const prm = [p1, p2];

Promise.allSettled(prm).
   then((results) => results.forEach((result) =>
      console.log(result.status, result.value)));
```

OutPut

```
fulfilled 50

rejected undefined
```

Promise race()

The Promise.race() method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

We may think of this particular method as in the form of a real-life example where several people are running in a race whosoever wins comes first wins the race, the same scenario is here, which ever promise is successfully fulfills or rejects at early will be executed at first and rest one's results will not be displayed as an output.

Syntax

```
Promise.race(iterable);
```

Example 1: This example shows the basic use of the Promise.race() method in Javascript. As promise2 was faster so it prints its own result which is two.

```
const promise1 = new Promise((resolve, reject) => {
   setTimeout(resolve, 600, "one");
});

const promise2 = new Promise((resolve, reject) => {
   setTimeout(resolve, 200, "two");
});

Promise.race([promise1, promise2]).then((value) => {
   console.log(value);
});
```

OutPut

```
two
```

promise reject()

The Promise.reject() method is used to return a rejected Promise object with a given reason for rejection. It is used for debugging purposes and selective error-catching. The catch() method can be used for logging the output of the reject() method to the console that is catch() method acts as a career that carries the rejected message from the Promise.reject() method and displays that in the user's console.

Syntax

```
Promise.reject(reason);
```

Example : In this example, a Promise is instantly rejected with the reason “I am a reason of error.” The catch() method logs the error to the console, showcasing a concise error-handling mechanism using JavaScript Promises.

```
// Initialize a promise variable and
// use the reject() method with a
// reason as a parameter
let promise = Promise.reject("I am a reason of error");

// Catch the promise and pass the
// function for logging the error in console
promise.catch(function (error) {
    console.log(error);
});
```

Promise.resolve()

The Promise.resolve() method in JavaScript returns a Promise object that is resolved with a given value. If the value is a promise, it returns that promise; otherwise, it resolves the value as a new promise, making it useful for simplifying asynchronous code handling.

Syntax

```
Promise.resolve(value);
```

Example : In this example, we creates a resolved promise with the value 17468. The .then() method handles the resolved value, logging 17468 to the console as the output.

```
let promise = Promise.resolve(17468);

promise.then(function (val) {
    console.log(val);
});
//Output: 17468
```

2. Async/await
   async/await is a syntax introduced in JavaScript to handle asynchronous operations in a cleaner and more readable way. It allows you to write asynchronous code that looks synchronous, avoiding the complexities of promise chaining.

Key Concepts

1. **async Function**:
   - Declaring a function as async ensures it always returns a **Promise**.
   - Inside an async function, you can use await.
2. **await**:
   - Pauses the execution of the async function until the promise is resolved.
   - Returns the resolved value of the promise.

Syntax:

```
async function functionName() {
  const result = await someAsyncFunction();
  console.log(result);
}
```

Example :

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Data fetched successfully!");
    }, 1000);
  });
}
// Using async/await
async function getData() {
  console.log("Fetching data...");
  const result = await fetchData(); // Wait for the promise to resolve
  console.log(result);
}
getData();
```

Error Handling with async/await

To handle errors in async/await, wrap the code in a try...catch block.

```
function fetchDataWithError() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject("Failed to fetch data!");
    }, 1000);
  });
}
async function getData() {
  try {
    const result = await fetchDataWithError();
    console.log(result);
  } catch (error) {
    console.error("Error:", error);
  }
}
getData();
```

What is **XMLHttpRequest** in JavaScript?

XMLHttpRequest (XHR) is an API provided by browsers to interact with servers. It allows you to:

1. Send HTTP requests (GET, POST, etc.).
2. Receive data from a server.
3. Update parts of a web page without reloading (AJAX).

Although modern JavaScript often uses fetch or libraries like axios, XMLHttpRequest is still widely supported.

Steps to Use XMLHttpRequest

1. Create an XMLHttpRequest object.
2. Use .open() to initialize the request.
3. Use .send() to send the request.
4. Listen for changes using the .onreadystatechange event or .onload.

Simple Example: GET Request

The following example fetches data from a placeholder API:

```
// Create an XMLHttpRequest object
const xhr = new XMLHttpRequest();

// Specify the HTTP method, URL, and whether it's asynchronous
xhr.open("GET", "https://jsonplaceholder.typicode.com/posts/1", true);

// Set up an event listener for when the response is ready
xhr.onload = function () {
  if (xhr.status === 200) { // Status 200 means "OK"
    console.log("Response:", xhr.responseText); // Log the response text
  } else {
    console.error("Error:", xhr.status, xhr.statusText);
  }
};

// Send the request
xhr.send();
```

OutPut


```
{
  "userId": 1,
  "id": 1,
  "title": "Sample title",
  "body": "This is a sample body."
}
```


Simple Example: POST Request

To send data to a server, use a POST request

```
// Create an XMLHttpRequest object
const xhr = new XMLHttpRequest();

// Specify the HTTP method, URL, and asynchronous flag
xhr.open("POST", "https://jsonplaceholder.typicode.com/posts", true);

// Set the request header to indicate JSON data
xhr.setRequestHeader("Content-Type", "application/json");

// Set up an event listener for when the response is ready
xhr.onload = function () {
  if (xhr.status === 201) { // Status 201 means "Created"
    console.log("Response:", xhr.responseText);
  } else {
    console.error("Error:", xhr.status, xhr.statusText);
  }
};

// Create JSON data to send
const data = JSON.stringify({
  title: "My Post",
  body: "This is the content of my post.",
  userId: 1,
});

// Send the request with data
xhr.send(data);
```

OutPut


```
{
  "id": 101,
  "title": "My Post",
  "body": "This is the content of my post.",
  "userId": 1
}
```

TASK

Write a function that:

1. Sends a GET request to the following API endpoint: https://jsonplaceholder.typicode.com/posts.
2. Logs the response to the console once the request is successful.
3. Handles any errors by logging an appropriate error message.

Write a function that:

1. Sends a POST request to the API https://jsonplaceholder.typicode.com/posts.
2. Include the following data in the request body (using application/json content type):
    - title: 'New Post'
    - body: 'This is a new post created using XMLHttpRequest.'
    - userId: 1
3. Once the post is successfully created, log the response to the console.
4. Handle errors by logging a message when the request fails.

Write a function that:

1. Sends a PUT request to update an existing post with the following API: https://jsonplaceholder.typicode.com/posts/1.
2. Update the title and body of the post to:
    - title: 'Updated Post Title'
    - body: 'Updated content for the post.'
3. Log the response once the post has been updated.
4. Handle errors by logging an appropriate error message

What is the **fetch()** Method in JavaScript?

The **fetch()** method is a modern way to make HTTP requests in JavaScript. It is used to request resources (like JSON data, text, images, etc.) from a server and returns a **Promise**.

Key Features of fetch()

1. Returns a **Promise** that resolves to the Response object.
2. Handles HTTP requests asynchronously.
3. Works with modern APIs and supports options like GET, POST, PUT, DELETE, etc.
4. Can be used with async/await for better readability.

Basic Syntax

```
fetch(url, options)
  .then((response) => {
    // Process the response
  })
  .catch((error) => {
    // Handle errors
  });
```

Simple Example of fetch()

Fetching Data from a Public API (GET Request)

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then((response) => {
    // Check if the request was successful
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json(); // Convert the response to JSON
  })
  .then((data) => {
    console.log("Data fetched:", data);
  })
  .catch((error) => {
    console.error("Error fetching data:", error);
  });
```

OutPut:

```
Data fetched: { userId: 1, id: 1, title: "...", body: "..." }
```

Using fetch() with async/await

Using async/await makes the code cleaner and easier to read.

```
async function fetchData() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    const data = await response.json(); // Convert the response to JSON
    console.log("Data fetched:", data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}

fetchData();
```

Making a POST Request with fetch()

Sending Data to the Server

```
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST", // HTTP method
  headers: {
    "Content-Type": "application/json", // Specify JSON content
  },
  body: JSON.stringify({
    title: "New Post",
    body: "This is the content of the new post.",
    userId: 1,
  }),
})
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json(); // Convert response to JSON
  })
  .then((data) => {
    console.log("Post created:", data);
  })
  .catch((error) => {
    console.error("Error creating post:", error);
  });
```

OutPut

```
Post created: { id: 101, title: "New Post", body: "This is the content of the new post.", userId: 1 }
```

Tasks

1. Write a function that fetches user details from an API and returns a Promise. Use the API: https://jsonplaceholder.typicode.com/users . (Using promise)

2. You are tasked with creating a function that simulates fetching data from an API using a Promise. The function should:
    a. Return a **Promise** that resolves with user data after a delay of 2 seconds.
    b. If the user ID is not provided or is invalid (e.g., less than 1), the Promise should reject with an error message saying "Invalid User ID".
    c. Once the Promise resolves, log the user data to the console. "https://jsonplaceholder.typicode.com/users"

3. Crate a table when press a button and display data from api("https://jsonplaceholder.typicode.com/todos") . inthis api have datas like user id,title.compete. complte is Boolean if the complete is fasle change the code like "not complted " in red color,complete id tue then show " complte " in green color.
4. You are tasked with creating a login feature that authenticates a user by sending their credentials to an API endpoint. The endpoint is https://dummyjson.com/auth/login.

    Requirements:

    1.Write a fetch request to send a POST request to the given URL.

    2.Include the following credentials in the request body:

    - **username**: 'emilys'
    - **password**: 'emilyspass'
    - **expiresInMins**: 30 (optional, defaults to 60 if not provided).

    3. Add the Content-Type header as 'application/json' to indicate that the request body contains JSON data.

    4.Include the credentials: 'include' option to ensure cookies (e.g., accessToken) are sent with the request.

    5. Log the response in the console as a JSON object.\

5. You need to fetch the authenticated user's details using a Bearer token. The API endpoint is https://dummyjson.com/auth/me.
    a. Write a fetch request to send a GET request to the given URL.
    b. Include an Authorization header with the value 'Bearer /* YOUR_ACCESS_TOKEN_HERE */' to authenticate the request with a JWT token.
    c. Use the credentials: 'include' option to include cookies (such as accessToken) in the request.
    d. Log the response in the console as a JSON object.

6. Write a function that fetches user details from an API and returns a Promise. Use the API: https://jsonplaceholder.typicode.com/users.(Promise.all())
7. Fetch posts from https://jsonplaceholder.typicode.com/posts and, for the first post, fetch the comments related to that post.
8. You are tasked with creating a simple webpage that fetches and displays data from the API https://jsonplaceholder.typicode.com/posts when a button is clicked. Write the JavaScript code to:
    1. Add an event listener to the button with the ID fetch Button so it triggers an API call when clicked.
    2. Fetch the data from the API using the Fetch API and handle any potential errors.
    3. Display the titles and body of the first 5 posts dynamically inside a with the ID output.
    4. Show a "Loading..." message while the data is being fetched.

9. Create whether cast application using weather api.
    a. Based on climate the background will change.
    b. Show all details in single page

LocalStorage and SessionStorage

Both **localStorage** and **sessionStorage** are web storage APIs provided by JavaScript. They allow you to store data directly in the user's browser, which can be retrieved and used later. These are part of the Web Storage API and are much simpler to use than cookies.

Key Differences Between LocalStorage and SessionStorage

Feature	LocalStorage	SessionStorage
Lifetime	Data persists even after the browser is closed and reopened.	Data is cleared when the browser/tab is closed.
Scope	Data is shared across all tabs/windows of the same origin.	Data is specific to the tab where it was created.
Storage Limit	Approximately 5-10 MB per domain.	Approximately 5 MB per domain.
Use Case	Ideal for storing data that needs to persist long-term, like user preferences or tokens.	Ideal for temporary data, like form inputs during a session.

Basic Syntax

Set Data

```
localStorage.setItem('key', 'value'); // For localStorage
sessionStorage.setItem('key', 'value'); // For sessionStorage
```

Get Data

```
localStorage.getItem('key'); // For localStorage
sessionStorage.getItem('key'); // For sessionStorage
```

Remove Data

```
localStorage.removeItem('key'); // For localStorage
sessionStorage.removeItem('key'); // For sessionStorage
```

Clear All Data

```
localStorage.clear(); // For localStorage
sessionStorage.clear(); // For sessionStorage
```

Example: LocalStorage

Storing User Preferences

```
// Save user preferences
localStorage.setItem('theme', 'dark');
localStorage.setItem('fontSize', '16px');

// Retrieve user preferences
const theme = localStorage.getItem('theme');
const fontSize = localStorage.getItem('fontSize');
console.log(`Theme: ${theme}, Font Size: ${fontSize}`);

// Remove a specific preference
localStorage.removeItem('fontSize');

// Clear all stored data
localStorage.clear();
```

Example: SessionStorage

Storing Form Data Temporarily

```
// Save form data
sessionStorage.setItem('username', 'JohnDoe');
sessionStorage.setItem('email', 'john.doe@example.com');

// Retrieve form data
const username = sessionStorage.getItem('username');
const email = sessionStorage.getItem('email');
console.log(`Username: ${username}, Email: ${email}`);

// Clear the form data when the tab is closed
sessionStorage.clear();
```

TASK

1. Create ecommerce application using https://dummyjson.com/products api
    a. display all product in home page
    b. set search input field in navbar and when try to search the prodect will sort based on searching.
    c. Filter the product based on category in same page
    d. When press to a product will shaow all data into another page using api
    e. In product details page have “Add to Cart ”button when we press that butten the product data will stroe in localstorage. And move to cart page
    f. In cart page display all cart product from local storage.
    g. In home page will show cart’s count
2. Create todo using session storage
    a. Add data to session
    b. Delete from session
    c. Update data from session
    d. Display all data from session
3. Create a contact book using local storage
    a. Add first name,last name ,mobile number as an object to local storage
    b. Display all usesr’s first name lastname and contact number
    c. When press delete button It will delete from local storage
    d. When press update button move to another page and update from there and back to home
4. Create a todo and store the task to llocal storage
    a. Add task to local storage
    b. Display all task from local storage
    c. When press delete button then delete its corresponding task
    d. Set a checkbox Infront of tasks and when press the check box set “line through “ property of each task
    e. When press update button the task is move tho same input field (add inout field) and update from there.

IMPORT & EXPORT

import and export are ES6 (ECMAScript 2015) features used to share code between JavaScript files. They allow you to create modular and reusable code by breaking your application into smaller, manageable pieces.

How export Works

You use the export keyword to make functions, objects, or variables available for use in other files.

Types of Exports

1. **Named Export**: Exporting multiple values using their names.
2. **Default Export**: Exporting a single value as the default export.

Named Export

Export

```
// file: mathUtils.js

export const add = (a, b) => a + b;

export const subtract = (a, b) => a - b;
```

Import

```
// file: main.js

import { add, subtract } from './mathUtils.js';


console.log(add(5, 3));       // Output: 8

console.log(subtract(5, 3));   // Output: 2
```

Default Export

Export

```
// file: calculator.js
export default function multiply(a, b) {
  return a * b;
}
```

Import

```
// file: main.js
import multiply from './calculator.js';


console.log(multiply(4, 5));  // Output: 20
```