NAME:RUPASRI A(717823E245)

MERN STACK TRAINING WEEK 3&4

1. Recursion and stack

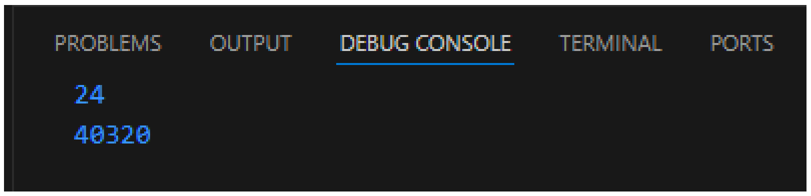
TASK 1.1:

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

CODE:

```
<!DOCTYPE html>

<html>

<title> Task 1.1</title>

<body>

<script>

function fact(a){

if(a==0 || a==1){

return 1;
          }
return a*fact(a-1);
       }
console.log(fact(4));

console.log(fact(8));

</script>

</body>

</html>
```

OUTPUT:



```
24
40320
```

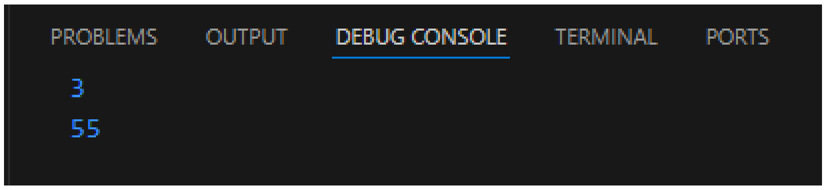
TASK 1.2:

Write a recursive function to find the nth Fibonacci number.

CODE:

```
<!DOCTYPE html>

<html>

<title> Task 1.2</title>

<body>

<script>

functionfibonacci(n){

if(n==0 || n==1){

return n;

          }

returnfibonacci(n-1)+fibonacci(n-2);

       }

console.log(fibonacci(4));

console.log(fibonacci(10));

</script>

</body>

</html>
```

OUTPUT:



```
3
55
```

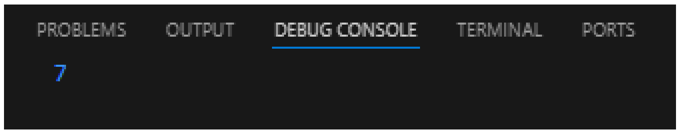
TASK 1.3:

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:

```
<!DOCTYPE html>
<html>
    <title>TASK 1.3</title>
    <body>
        <script>
            function countWays(n){
    if (n ==0)
    return 1;
    if (n < 0) return 0;
       return countWays(n - 1) + countWays(n - 2) + countWays(n - 3);
}
const n = 4;
console.log(countWays(n));
        </script>
    </body>
</html>
```

OUTPUT:



```
7
```

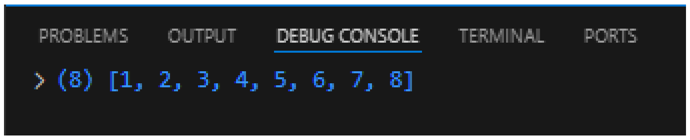
TASK 1.4:

Write a recursive function to flatten a nested array structure.

CODE:

```
<!DOCTYPE html>
<html>
  <head>
    <title>1.4</title>
  </head>
  <body>
    <script>
     function flattenArray(arr) {

      let result = [];

      arr.forEach(element => {
            if (Array.isArray(element)) {

              result = result.concat(flattenArray(element));

            } else {
              result.push(element);
            }
          });
          return result;
          }
        let nestedArray = [1, [2, [3, 4], 5], [6, 7], 8];

        console.log(flattenArray(nestedArray));

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

OUTPUT:



```
> (8) [1, 2, 3, 4, 5, 6, 7, 8]
```

TASK 1.5:

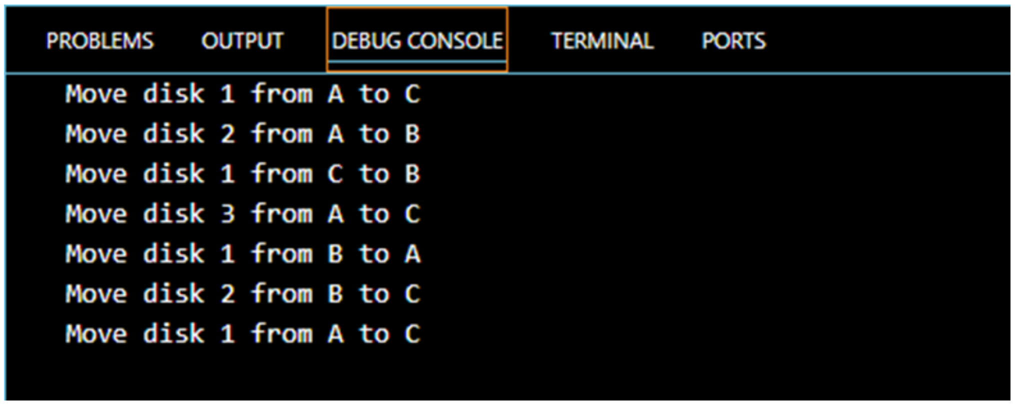
Implement the recursive Tower of Hanoi solution.

CODE:

```
<!DOCTYPE html>
<html>
    <head>
        <title>1.5</title>

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

```
const numberOfDisks = 3;
towerOfHanoi(numberOfDisks, 'A', 'B', 'C');
      </script>
    </body>
</html>
```

OUTPUT:



```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

2. JSON and variable length arguments/spread syntax

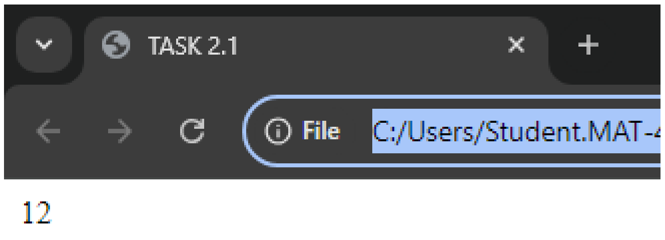
TASK 2.1:

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

CODE:

```
<!DOCTYPE html>
 <html>
    <title>TASK 2.1</title>
    <body>
        <script>
            function add(num1, num2){
                return num1 + num2;
            }
            let a = 4, b = 8
            let res = document.writeln(add(a, b))
        </script>
    </body>
 </html>
```

OUTPUT:



```
12
```

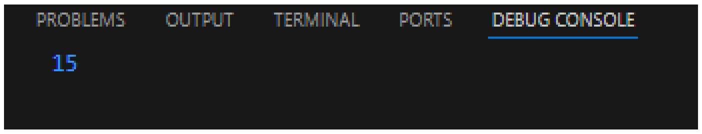
TASK 2.2:

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

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 2.2</title>
  <body>
    <script>
      function sumNumbers(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
const nums = [1, 2, 3, 4, 5];
console.log(sumNumbers(...nums));
      </script>
  </body>
</html>
```

OUTPUT:



```
15
```

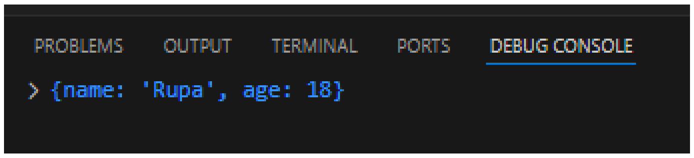
TASK 2.3:

Create a deep clone of an object using JSON methods.

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 2.3</title>
  <body>
    <script>
      function deepClone(obj){
        return JSON.parse(JSON.stringify(obj));
      }
      const originalObj={
        name:'Rupa',
        age:18
      };
      const clonedObject=deepClone(originalObj);
      console.log(clonedObject);
      </script>
  </body>
</html>
```

OUTPUT:



```
> {name: 'Rupa', age: 18}
```

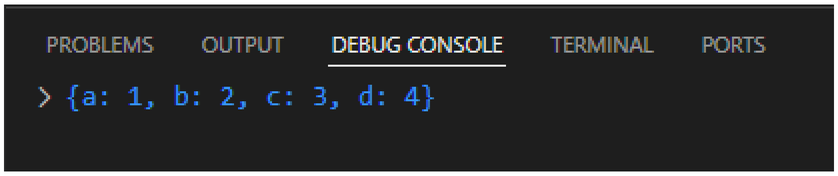
TASK 2.4:

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

CODE:

```
<!DOCTYPE html>
<html>
  <head>
    <title>TASK 2.4</title>
  </head>
  <body>
    <script>
     function mergeObjects(obj1, obj2) {
  return { ...obj1, ...obj2 };
}
const object1 = { a: 1, b: 2 };
const object2 = { b: 3, c: 4 };
const mergedObject = mergeObjects(object1, object2);
console.log(mergedObject);
    </script>
  </body>
</html>
```

OUTPUT:



```
> {a: 1, b: 2, c: 3, d: 4}
```

TASK 2.5:

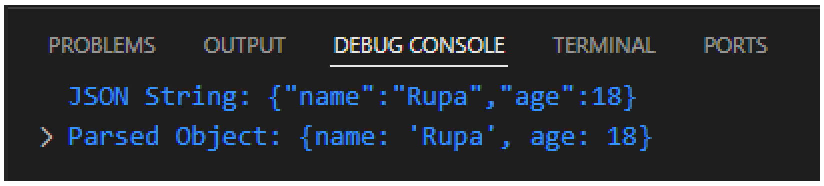
Serialize a JavaScript object into a JSON string and then parse it back into an object.

CODE:

```
<!DOCTYPE html>
<html>
 <title>TASK 2.5</title>
 <body>
  <script>
```

```
      const obj={
        name:"Rupa",
        age:18
      };
      const jsonString=JSON.stringify(obj);
      console.log("JSON String:",jsonString);
const parsedObject=JSON.parse(jsonString);
console.log("Parsed Object:",parsedObject);
      </script>
  </body>
</html>
```

OUTPUT:



```
JSON String: {"name":"Rupa","age":18}
> Parsed Object: {name: 'Rupa', age: 18}
```

3. Closure

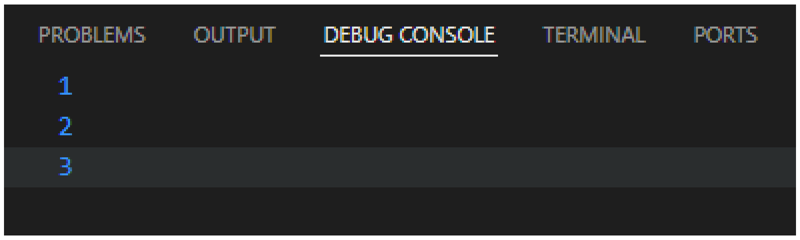
TASK 3.1:

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

CODE:

```
<!DOCTYPE html>

<html>

  <title>TASK 3.1</title>

  <body>

    <script>

      function createCounter() {

  let count = 0;

  return function() {

    count++;

    return count;
  };
}
const counter = createCounter();
console.log(counter());
console.log(counter());
console.log(counter());
     </script>
  </body>
</html>
```

OUTPUT:



```
1
2
3
```

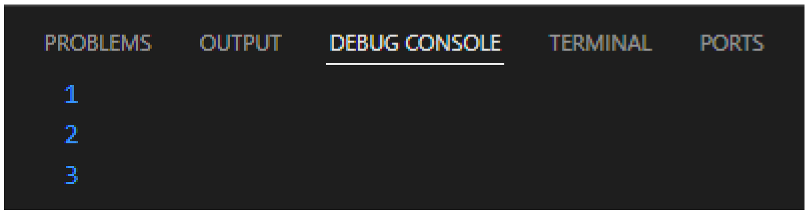
TASK 3.2:

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

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 3.2</title>
  <body>
    <script>
     function createCounter() {
  let count = 0;
  return {
    increment: function() {
      count++;
    },
    getCount: function() {
      return count;
    }
  };
}
const counter = createCounter();
counter.increment();
console.log(counter.getCount());
counter.increment();
console.log(counter.getCount());
counter.increment();
console.log(counter.getCount());
      </script>
  </body>
</html>
```

OUTPUT:



```
1
2
3
```

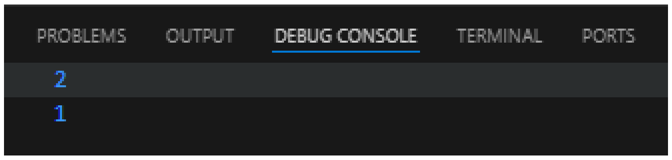
TASK 3.3:

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

CODE:

```
<!DOCTYPE html>
 <html>
   <title>TASK 3.3</title>
   <body>
      <script>
         function createCounter() {
   let count = 0;
   return {
      increment: function() {
         count++;
      },
      getCount: function() {
         return count;
      }
   };
}
const counter1 = createCounter();
const counter2 = createCounter();
counter1.increment();
counter1.increment();
console.log(counter1.getCount());
counter2.increment();
console.log(counter2.getCount());
      </script>
   </body>
 </html>
```

OUTPUT:



```
2
1
```

TASK3.4:

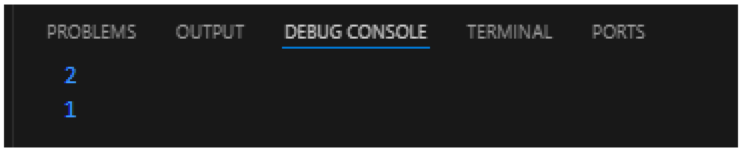
Use closures to create private variables within a function.

CODE:

```
     <!DOCTYPE html>
 <html>
   <title>TASK 3.4</title>
   <body>
      <script>
         function createPrivateCounter() {
   let count = 0;
   return {
      increment: function() {
         count++;
      },
      decrement: function() {
         count--;
      },
      getCount: function() {
         return count;
      }
```

```
    };

}

const privateCounter = createPrivateCounter();

privateCounter.increment();

privateCounter.increment();

console.log(privateCounter.getCount());

privateCounter.decrement();

console.log(privateCounter.getCount());

        </script>

    </body>

 </html>
```

OUTPUT:



```
2
1
```

TASK 3.5:

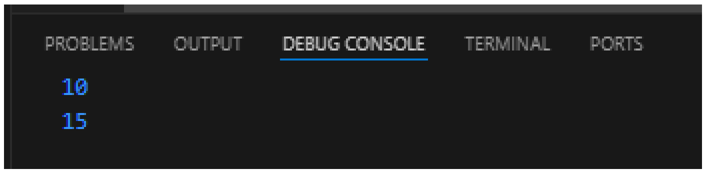
Build a function factory that generates functions based on some input using

closures.

CODE:

```
<!DOCTYPE html>

 <html>

    <title>TASK 3.5</title>

    <body>

        <script>

           function multiplier(factor) {

    return function(number) {
        return number * factor;
    };
}
const double = multiplier(2);
const triple = multiplier(3);
console.log(double(5));
console.log(triple(5));
        </script>
    </body>
 </html>
```

OUTPUT:



```
10
15
```

4. Promise, Promises chaining

TASK 4.1:

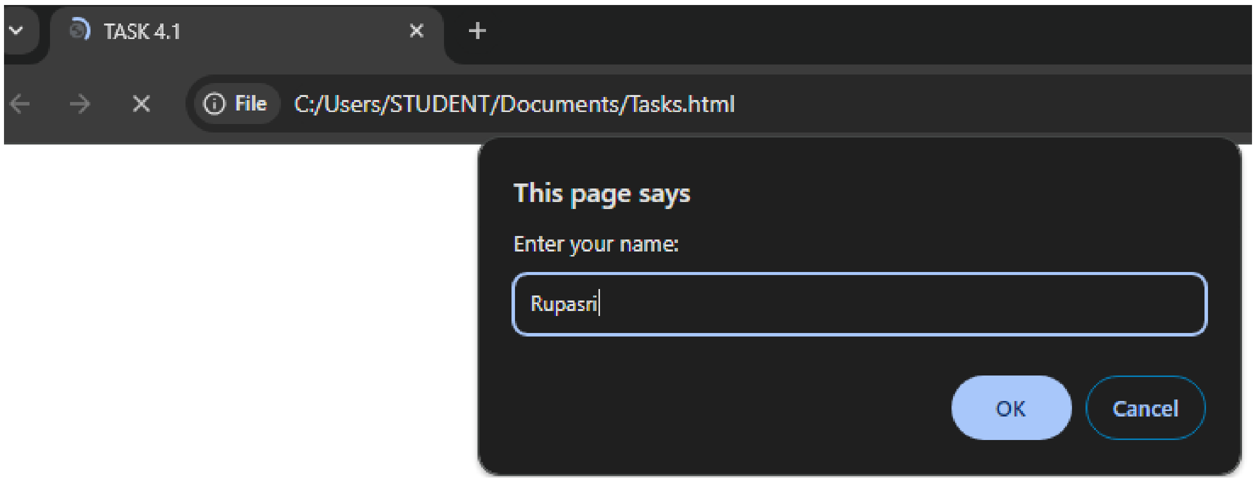
Create a new promise that resolves after a set number of seconds and returns a greeting.

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 4.1</title>
  <body>
    <script>
        let name=prompt("Enter your name:",'user');
        function myPromise(){
```

```
 return new Promise((resolve)=>{
 setTimeout(()=>{
resolve();
console.log("Welcome " + name+"!");
 },2000); })
}
 myPromise();
     </script>
  </body>
</html>
```

OUTPUT:



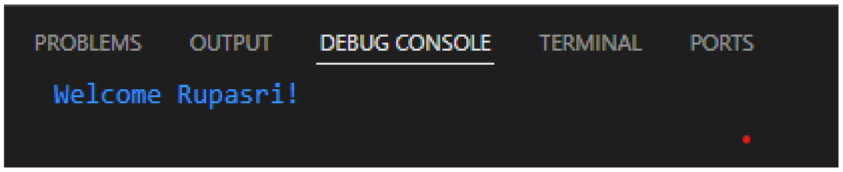
TASK 4.1

C:/Users/STUDENT/Documents/Tasks.html

This page says

Enter your name:

Rupasri



```
Welcome Rupasri!
```

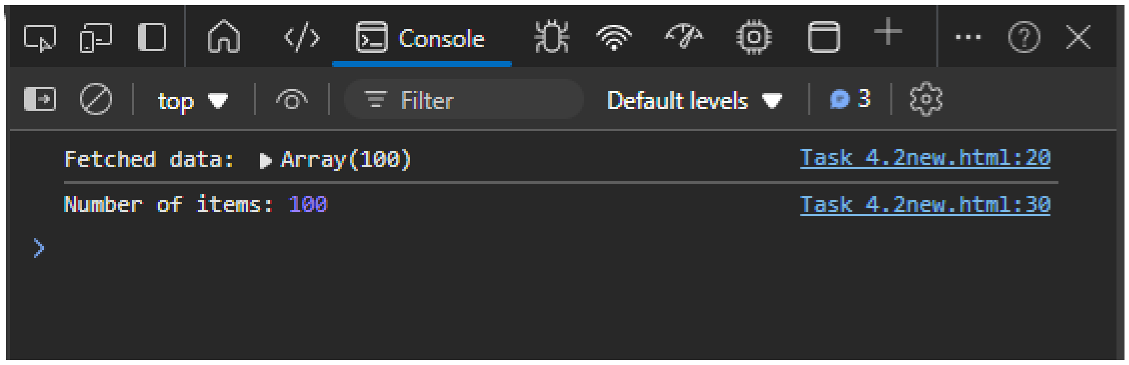
TASK 4.2:

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

CODE:

```
<html>
   <head>
<title>Task 4.2</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:

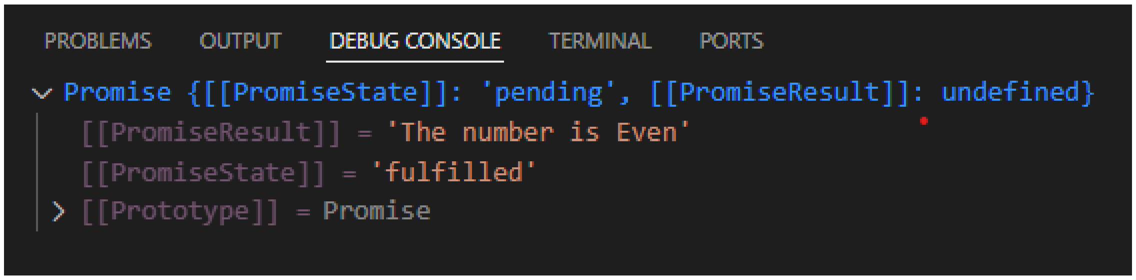


```
Fetched data:  ▶ Array(100)                         Task 4.2new.html:20
Number of items: 100                               Task 4.2new.html:30
>
```

TASK 4.3:

```
<!DOCTYPE html>
<html>
 <title>TASK 4.3</title>
 <body>
  <script>
    var data=new Promise((resolve,reject)=>{
      setTimeout(()=>{
        var name=parseInt(prompt("Enter Number:"));
        if(name%2==0)
         resolve("The number is Even");
        else
          reject("The number is Odd");
      },2000);
    })
    console.log(data);
   </script>
 </body>
</html>
```

OUTPUT:



```
v Promise {[[PromiseState]]: 'pending', [[PromiseResult]]: undefined}
   [[PromiseResult]] = 'The number is Even'
   [[PromiseState]] = 'fulfilled'
 > [[Prototype]] = Promise
```

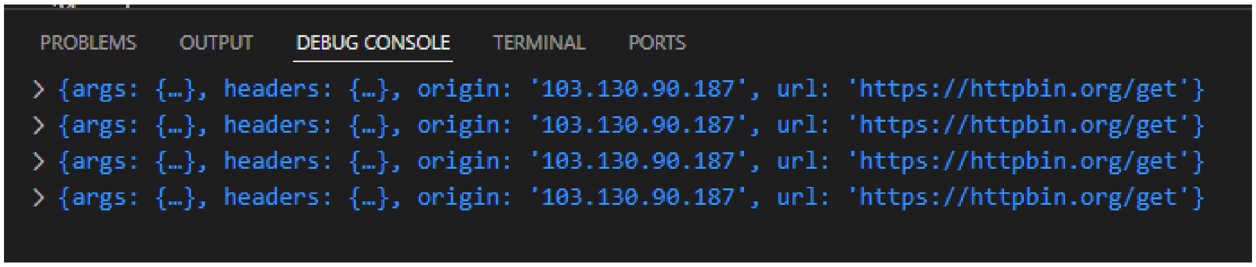
TASK 4.4:

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

CODE:

```
<!DOCTYPE html>
<html>
 <title>TASK 4.4</title>
 <body>
  <script>
        const urls = [

  'https://httpbin.org/get',

  'https://httpbin.org/get',

  'https://httpbin.org/get',

  'https://httpbin.org/get'

];

Promise.all(urls.map((url)=>fetch(url).then((response)=>response.json())))

  .then((jsons)=>{

    jsons.forEach((json)=>console.log(json));

  })

  .catch((error)=>console.error('An error occurred:',error));

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

OUTPUT:



```
> {args: {…}, headers: {…}, origin: '103.130.90.187', url: 'https://httpbin.org/get'}
> {args: {…}, headers: {…}, origin: '103.130.90.187', url: 'https://httpbin.org/get'}
> {args: {…}, headers: {…}, origin: '103.130.90.187', url: 'https://httpbin.org/get'}
> {args: {…}, headers: {…}, origin: '103.130.90.187', url: 'https://httpbin.org/get'}
```

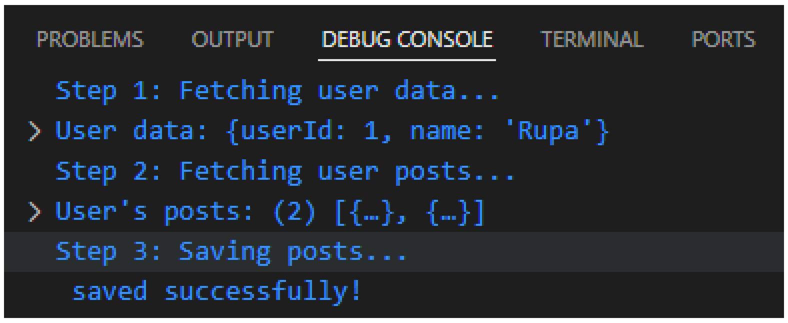
TASK 4.5:

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

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 4.5</title>
  <body>
    <script>
 function step1() {

  return new Promise((resolve) => {

    console.log("Step 1: Fetching user data...");

    setTimeout(() => resolve({ userId: 1, name: "Rupa" }), 1000);

  });

}

function step2(user) {

  return new Promise((resolve) => {

    console.log("Step 2: Fetching user posts...");

    setTimeout(() => resolve([{ id: 1, title: "Post 1" }, { id: 2, title: "Post 2" }]), 1000);

  });

}

function step3(posts) {

  return new Promise((resolve) => {

    console.log("Step 3: Saving posts...");

    setTimeout(() => resolve(" saved successfully!"), 1000);
 });
}
step1()

 .then(user => {

  console.log("User data:", user);

  return step2(user);

 })

 .then(posts => {

  console.log("User's posts:", posts);

  return step3(posts);

 })

 .then(message => {

  console.log(message);

 })

 .catch(error => {

  console.error("Error:", error);

 });
    </script>
 </body>
</html>
```

OUTPUT:



```
  Step 1: Fetching user data...
> User data: {userId: 1, name: 'Rupa'}
  Step 2: Fetching user posts...
> User's posts: (2) [{…}, {…}]
  Step 3: Saving posts...
   saved successfully!
```

5. ASYNC/AWAIT

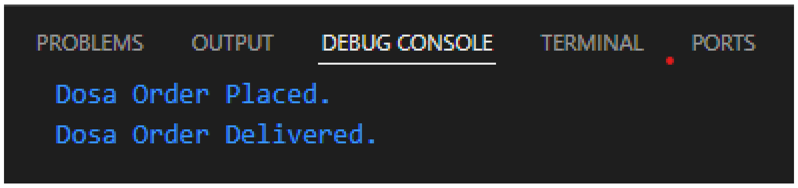
TASK 5.1:

Rewrite a promise-based function using async/await.

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 5.1</title>
  <body>
    <script>
 function PlaceFood(order){

           return new Promise((resolve)=>{

               setTimeout(()=>{

                   console.log(`${order} Order Placed.`);

                   resolve(order);

               },1000);   })  }

        function DeleiverFood(order){

           return new Promise((resolve)=>{

               setTimeout(()=>{

                   console.log(`${order} Order Delivered.`);

                   resolve(`${order} Order Delivered.`);

               },1000); }) }

        async function orders(food){

           const orderss=await PlaceFood(food);

           const deliver=await DeleiverFood(orderss);

           document.write(status); }

        orders("Dosa");

      </script>

  </body>

</html>
```

OUTPUT:



```
Dosa Order Placed.
Dosa Order Delivered.
```

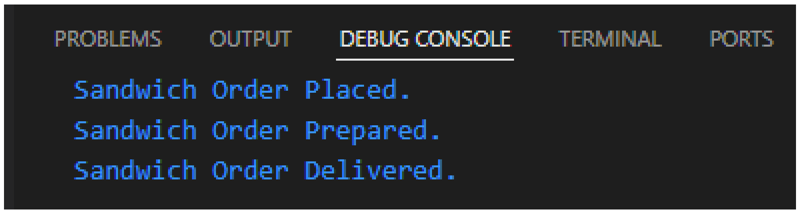
TASK 5.2:

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

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 5.2</title>
  <body>
    <script>
 function PlaceFood(order){

           return new Promise((resolve)=>{

              setTimeout(()=>{

                 console.log(`${order} Order Placed.`);

                 resolve(order);

              },1000);

           })

        }

        function PrepareFood(order){

           return new Promise((resolve)=>{

              setTimeout(()=>{

                 console.log(`${order} Order Prepared.`);

                 resolve(order);

              },1000);

           })

        }

        function DeleiverFood(order){

           return new Promise((resolve)=>{
                setTimeout(()=>{
                    console.log(`${order} Order Delivered.`);
                    resolve(`${order} Order Delivered.`);
                },1000);
            })
        }
        async function orders(food){
            const orderss=await PlaceFood(food);
            const Prepare=await PrepareFood(orderss);
            const deliver=await DeleiverFood(Prepare);
            document.write(status);
        }
        orders("Sandwich");
      </script>
  </body>
</html>
```

OUTPUT:



```
Sandwich Order Placed.
Sandwich Order Prepared.
Sandwich Order Delivered.
```

TASK 5.3:

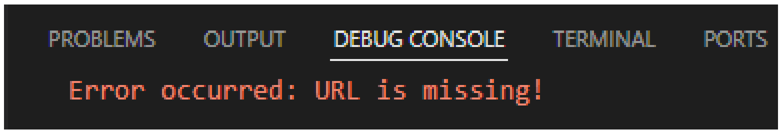
Implement error handling in an async function using try/catch.

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 5.3</title>
  <body>
```

```
    <script>

async function fetchData() {

  throw new Error('URL is missing!');

}

async function main() {

  try {

    const data = await fetchData();

    console.log('Data fetched:', data);

  } catch (error) {

    console.error('Error occurred:', error.message);

  }

}

main();

      </script>

  </body>

</html>
```

OUTPUT:



```
Error occurred: URL is missing!
```

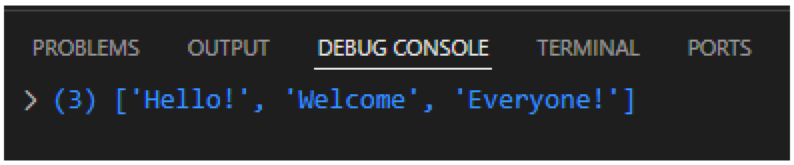
TASK 5.4:

Use async/await in combination with Promise.all.

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 5.4</title>
  <body>
    <script>
function one(){

    return new Promise((resolve,reject)=>{
        resolve("Hello!"); });

};

function two(){

    return new Promise((resolve, reject)=>{

        resolve("Welcome"); });

};

function three(){

    return new Promise((resolve, reject)=>{

        return setTimeout(()=>{

            resolve("Everyone!");

        }, 2000); });

};

async function promiseExecution(){

    let promise = await Promise.all([one(),two(),three()]);

    console.log(promise);

};

promiseExecution();

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

OUTPUT:



```
> (3) ['Hello!', 'Welcome', 'Everyone!']
```

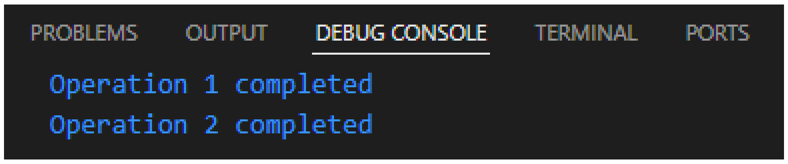
TASK 5.5:
Create an async function that waits for multiple asynchronous operations to complete before proceeding.

CODE:

```
<!DOCTYPE html>

<html>
```

```
  <title>TASK 5.5</title>

  <body>

    <script>

function asyncOperation(name, delay) {

  return new Promise(resolve => {

    setTimeout(() => {

      console.log(`${name} completed`);

      resolve(name);

    }, delay);
  });
}
async function main() {
  try {
    const results = await Promise.all([

      asyncOperation('Operation 1', 1000),

      asyncOperation('Operation 2', 2000)

    ]);
  } catch (error) {
    console.error('Error occurred:', error.message);
  }
}
main();

      </script>

  </body>

</html>
```

OUTPUT:



```
Operation 1 completed
Operation 2 completed
```

6. MODULES INTRODUCTION, EXPORT AND IMPORT

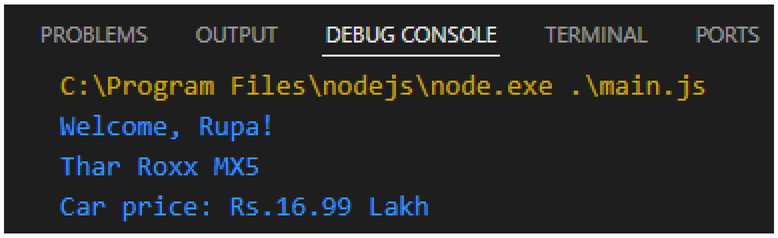
TASK 6.1:

Create a module that exports a function, a class, and a variable.

CODE:

```
function greet(name) {
  return `Welcome, ${name}!`;
}
class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }
  getDetails() {
    return `${this.make} ${this.model}`;
  }
}
const carPrice = "Rs.16.99 Lakh";

export { greet, Car, carPrice };

import { greet, Car, carPrice } from './myModule.js';

console.log(greet('Rupa'));

const myCar = new Car('Thar', 'Roxx MX5');

console.log(myCar.getDetails());

console.log(`Car price: ${carPrice}`);
```

OUTPUT:



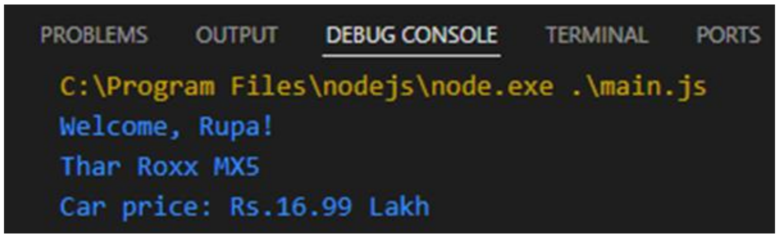
```
C:\Program Files\nodejs\node.exe .\main.js
Welcome, Rupa!
Thar Roxx MX5
Car price: Rs.16.99 Lakh
```

TASK 6.2:

Import the module in another JavaScript file and use the exported entities.
CODE:

```
function greet(name) {

  return `Welcome, ${name}!`;
}

class Car {

   constructor(make, model) {

      this.make = make;

      this.model = model;

   }

   getDetails() {

      return `${this.make} ${this.model}`;

   }

}

const carPrice = "Rs.16.99 Lakh";

export { greet, Car, carPrice };

import { greet, Car, carPrice } from './myModule.js';

console.log(greet('Rupa'));

const myCar = new Car('Thar', 'Roxx MX5');

console.log(myCar.getDetails());

console.log(`Car price: ${carPrice}`);
```

OUTPUT:



```
C:\Program Files\nodejs\node.exe .\main.js
Welcome, Rupa!
Thar Roxx MX5
Car price: Rs.16.99 Lakh
```

TASK 6.3:
Use named exports to export multiple functions from a module.
CODE:

```
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) {
  if (b === 0) {
    return 'Error: Division by zero';

  }
  return a / b;
 }
```

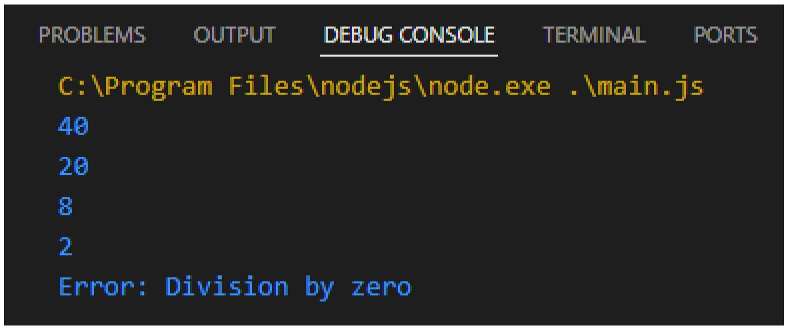
TASK 6.4:

Use named imports to import specific functions from a module.

CODE:

```
import { add, subtract, multiply, divide } from './myModule.js';

console.log(add(20, 20));

console.log(subtract(40,20));

console.log(multiply(2,4));

console.log(divide(10,5));

console.log(divide(20, 0));
```

OUTPUT(3 and 4):

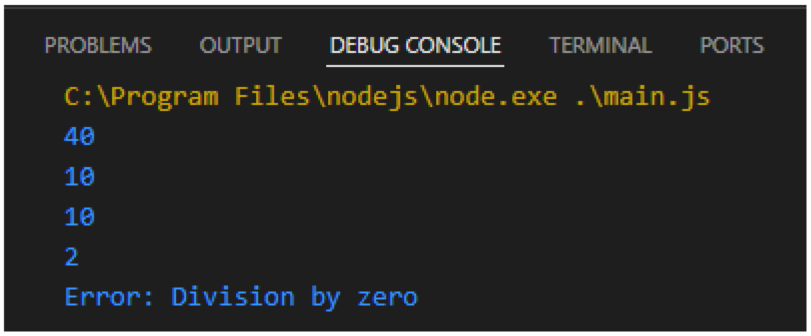


```
C:\Program Files\nodejs\node.exe .\main.js
40
20
8
2
Error: Division by zero
```

TASK 6.5:
Use default export and import for a primary function of a module.
CODE:

```
export default function calculate(a, b, operation) {
  switch (operation) {
   case 'add':
    return a + b;
   case 'subtract':
    return a - b;
   case 'multiply':
    return a * b;
   case 'divide':
    if (b === 0) {
     return 'Error: Division by zero';
    }
    return a / b;
   default:
    return 'Invalid operation';
  }
 }
import calculate from './myModule.js';
console.log(calculate(10, 5, 'add'));
console.log(calculate(10, 5, 'subtract'));
console.log(calculate(10, 5, 'multiply'));
console.log(calculate(10, 5, 'divide'));
console.log(calculate(10, 0, 'divide'));
console.log(calculate(10, 5, 'unknown'));
```

OUTPUT:



```
C:\Program Files\nodejs\node.exe .\main.js
40
10
10
2
Error: Division by zero
```

7. BROWSER: DOM BASICS

TASK 7.1:

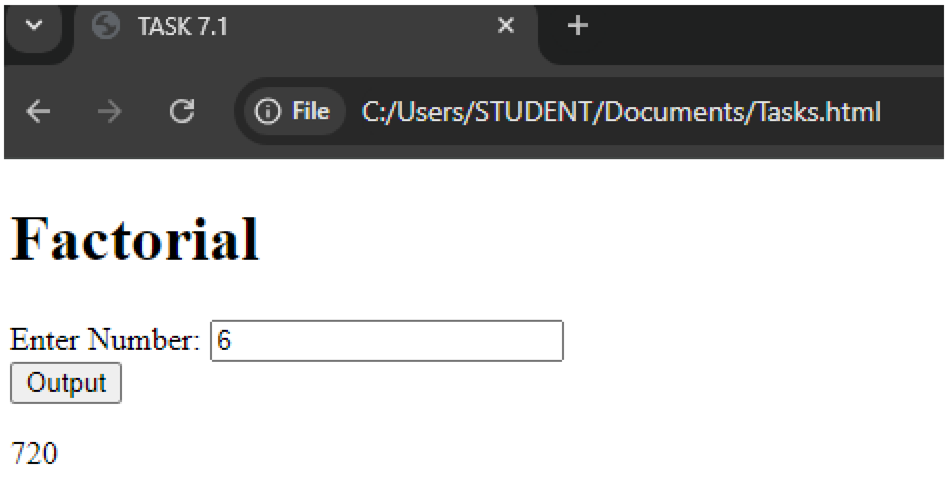
Select an HTML element by its ID and change its content using JavaScript.

CODE:

```
<!DOCTYPE html>
<html>
 <title>TASK 7.1</title>
 <body>
  <h1>Factorial</h1>
  <form>
    <label>Enter Number:</label>
    <input type="number" id="num" name="numm"><br>
    <input type="button" id="cal" value="Output" onclick="fact()">
    <p id="numm"></p>
  </form>
</body>
  <script>
function fact(){
  var num1=parseInt(document.getElementById("num").value);
  var res=factorial(num1);
  document.getElementById("numm").innerHTML=res;
}
function factorial(num){
  if(num==0) return 1;
  else
    return factorial(num-1)*num;
}
```

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

OUTPUT:



Factorial

Enter Number: 6

720

TASK 7.2:

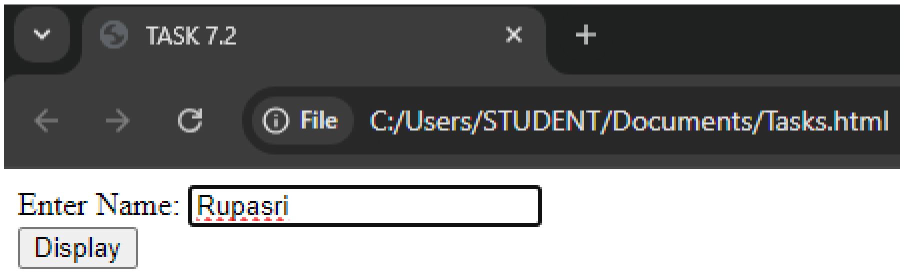
Attach an event listener to a button, making it perform an action when clicked.

CODE:

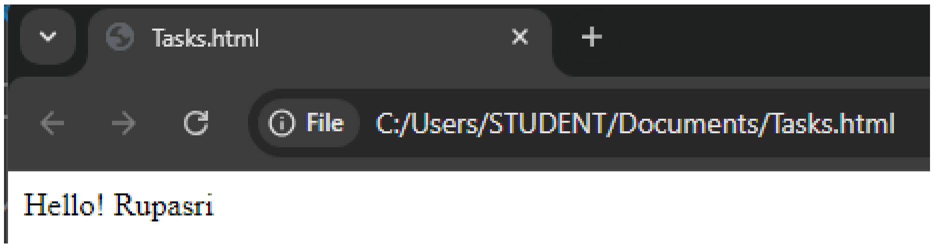
```
<!DOCTYPE html>
<html>
  <title>TASK 7.2</title>
  <body>
    <form>
      <label>Enter Name:</label>
      <input type="text" id="nam" name="namm"><br>
      <input type="button" id="cal" value="Display" onclick="display()">
      <p id="numm"></p>
  </form>
</body>
```

```
    <script>
function display(){
   var name=document.getElementById("nam").value;
   document.getElementById("numm").innerHTML=document.write(`Hello! ${name}`);
}
      </script>
  </body>
</html>
```

OUTPUT:



Enter Name: Rupasri



```
Hello! Rupasri
```

TASK 7.3:

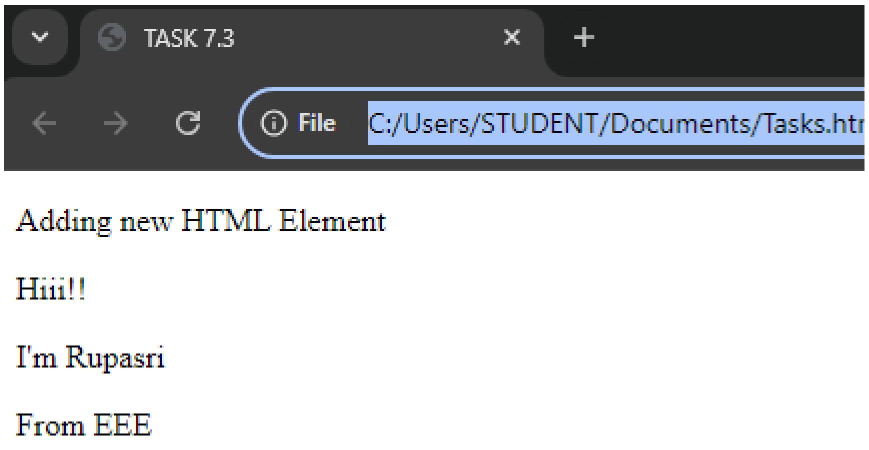
Create a new HTML element and append it to the DOM.

CODE:

```
<!DOCTYPE html>
<html>
  <title>TASK 7.3</title>
  <body>
    <p>Adding new HTML Element</p>
```

```
   <div id="d">
     <p id="p1">Hiii!!</p>
     <p id="p2">I'm Rupasri</p>
   </div>
   <script>
const a=document.createElement("p");
const node=document.createTextNode("From EEE");
a.appendChild(node);
const ele=document.getElementById("d");
ele.appendChild(a);
    </script>
  </body>
</html>
```

OUTPUT:



```
Adding new HTML Element

Hiii!!

I'm Rupasri

From EEE
```

TASK 7.4:

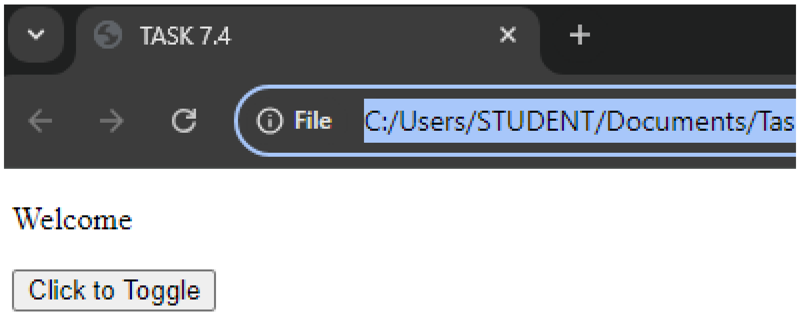
Implement a function to toggle the visibility of an element.

CODE:

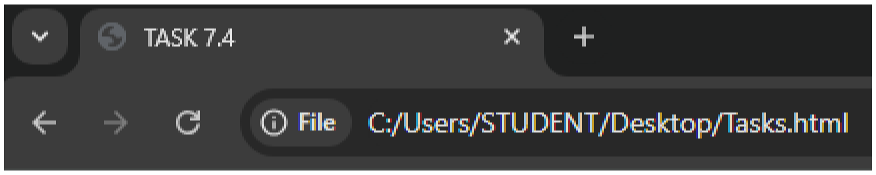
```
<!DOCTYPE html>
<html>
```

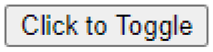
```
 <title>TASK 7.4</title>

 <body>

  <p id="m">Welcome<br></p>

  <button onclick="toggleElement()">

    Click to Toggle

  </button>

  <script>

function toggleElement(){

        const a=document.getElementById('m');

        const vi=window.getComputedStyle(a).visibility;

        if (vi==='hidden')

          a.style.visibility='visible';

         else

          a.style.visibility='hidden';
     }
    </script>
 </body>
</html>
```

OUTPUT:



Welcome





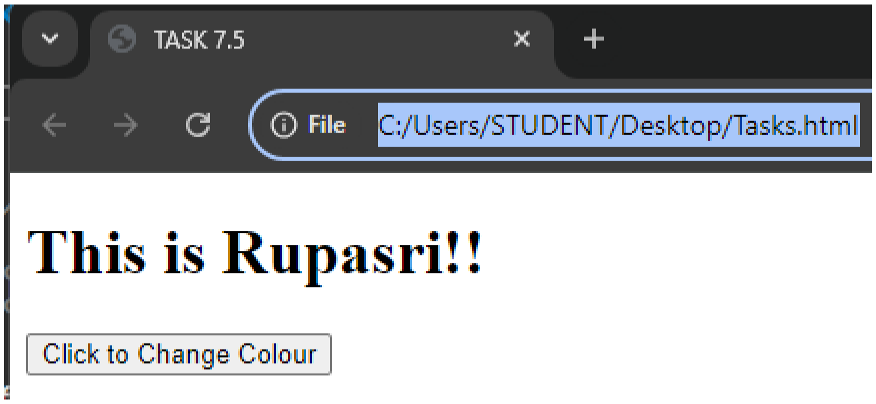
TASK 7.5:

Use the DOM API to retrieve and modify the attributes of an element.

CODE:

```
<!DOCTYPE html>
<html>
 <title>TASK 7.5</title>
 <body>
  <style>
   .attributee{
     color: blue;
   };
 </style>
 <h1 id="Id">This is Rupasri!!</h1>
 <button onclick="addAttribute()">Click to Change Colour</button>
  <script>
function addAttribute(){
      document.getElementById("Id").setAttribute("class","attributee");
    }
   </script>
 </body>
</html>
```

OUTPUT:

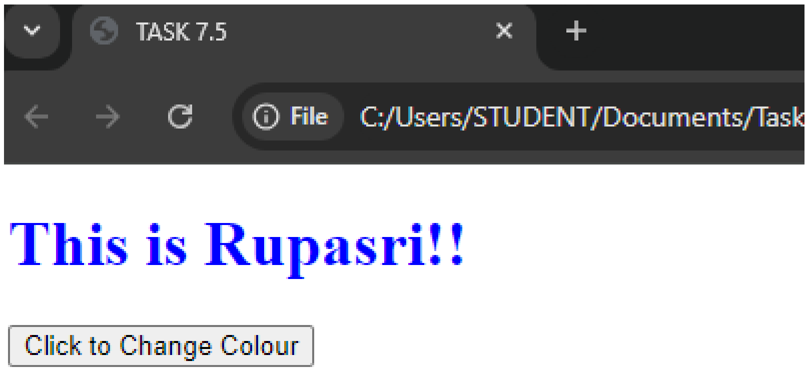


TASK 7.5

File C:/Users/STUDENT/Desktop/Tasks.html

This is Rupasri!!

Click to Change Colour



TASK 7.5

File C:/Users/STUDENT/Documents/Task

This is Rupasri!!

Click to Change Colour