Advanced Javascript

What is JavaScript let statement?

- The JavaScript **let** statement is used to declare a variable. With the *let* statement, we can declare a variable that is **block-scoped**. This mean a variable declared with let is only accessible within the block of code in which it is defined.
- The **let** keyword was introduced in the ES6 (2015) version of JavaScript. It is an alternative to the **var** keyword.

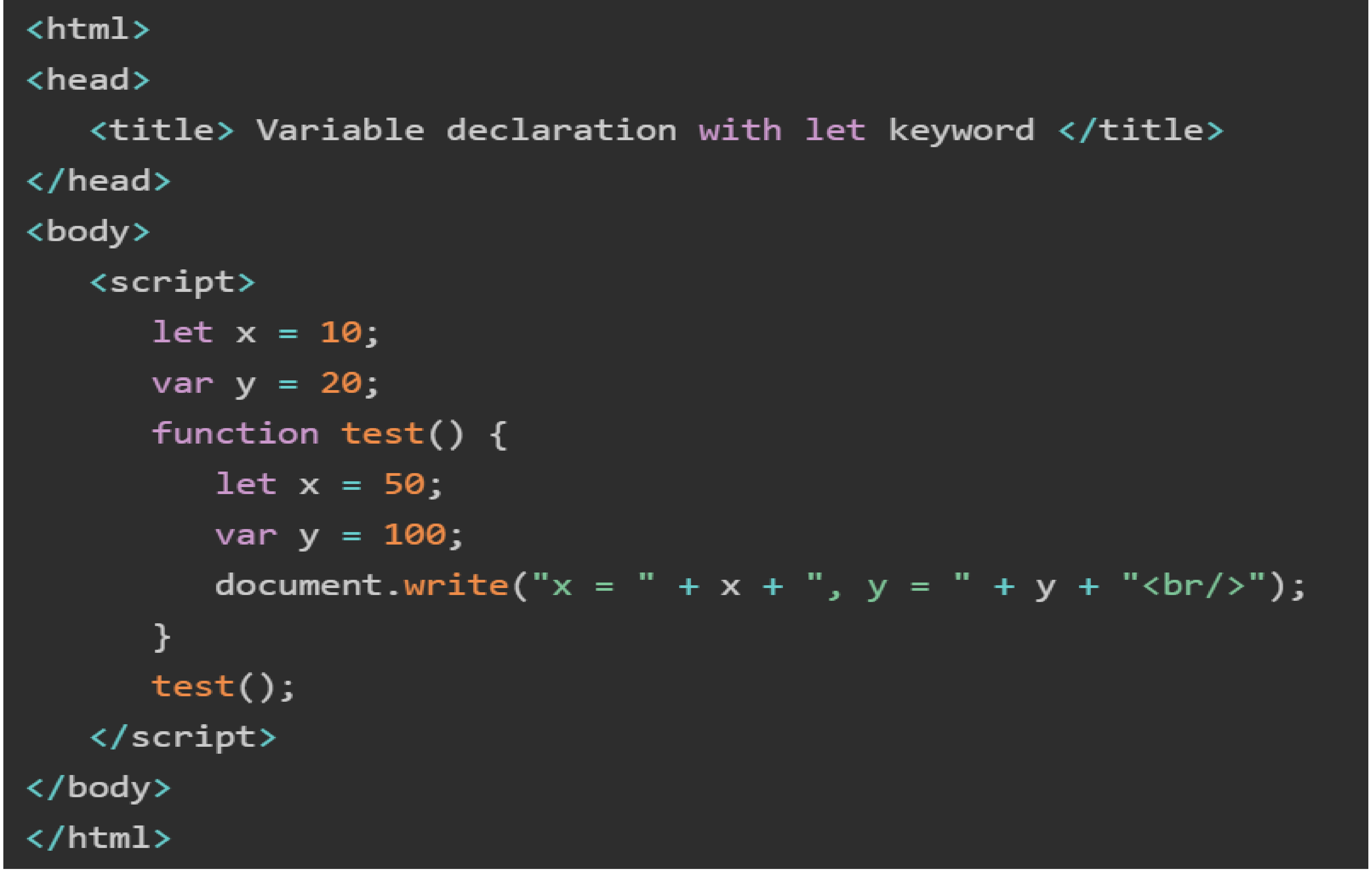
Variable Declaration with let statement

Following is the syntax to declare a variable with let statement –

```
let var_name = value
```

Let's have a look at some examples for variable declaration with let.

```
let name = "John";
let age = 35;
let x = true;
```



```
<html>
<head>
   <title> Variable declaration with let keyword </title>
</head>
<body>
   <script>
      let x = 10;
      var y = 20;
      function test() {
         let x = 50;
         var y = 100;
         document.write("x = " + x + ", y = " + y + "<br/>");
      }
      test();
   </script>
</body>
</html>
```

JavaScript const Keyword

- The **const** keyword is introduced in the ES6 version of JavaScript with the **let** keyword. The const keyword is used to define the variables having **constant reference**.
- A variable defined with const can't be **re-declared**, **reassigned**. The const declaration have block as well as function scope.

Declaring JavaScript Constants

You always need to assign a value at the time of declaration if the variable is declared using the const keyword.

```
const x = 10; // Correct Way
```

In any case, you can't declare the variables with the const keyword without initialization.

```
const y; // Incorrect way
y = 20;
```

Arrow Functions

- The **arrow functions** in JavaScript allow us to create a shorter and **anonymous** function. Arrow functions are written without "function" keyword. The JavaScript *arrow functions* are introduced in ES6.

Let's look at the below syntax to write a function expression –

```
const varName = function(parameters) {
    // function body
};
```

The above function expression can be written as an arrow function –

```
const varName = (parameters) => {
    // function body
};
```

Here the "function" keyword is removed and after parenthesis we added "=>"

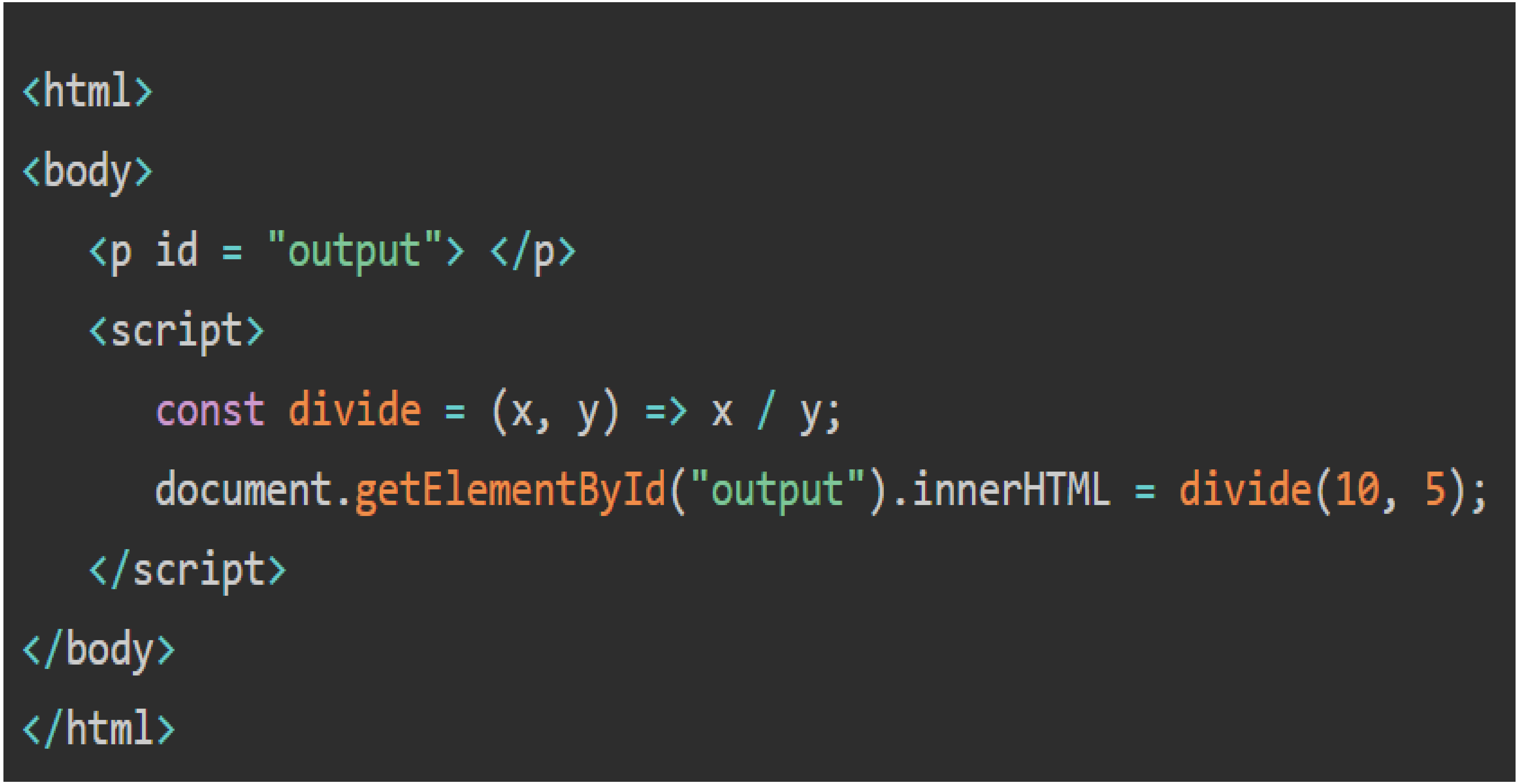
Arrow Function with Single Statement

When the arrow function contains a single statement, we don't need to write the 'return' keyword and braces (curly brackets).

```
const add = (x, y) => x +y;
```

Please note, we can always write an arrow function with return keyword and braces.

```
const add = (x, y) => {return x + y};
```



```html
<html>
<body>
   <p id = "output"> </p>
   <script>
      const divide = (x, y) => x / y;
      document.getElementById("output").innerHTML = divide(10, 5);
   </script>
</body>
</html>
```

Arrow Function with Multiple Statements

When the function body contains multiple statements, we should always use the 'return' statement to return a value. Also we should use the curly brackets.

```html
<html>
<body>
   <p id = "output"> </p>
   <script>
      const divide = (x, y) => {
         let res = x / y;
         return res;
      };
      document.getElementById("output").innerHTML = divide(10, 5);
   </script>
</body>
</html>
```

What is Promise in JavaScript?

- A JavaScript promise is an object that represents the completion or failure of an asynchronous operation. It employs callback functions to manage asynchronous operations, offering a easier syntax for handling such operations more easily.
- A promise object can created using the Promise() constructor. The promise constructor takes a callback function as an argument. The callback function accepts two functions, resolve() and reject(), as arguments. The resolve function is called if the promise returns successfully. The reject function is called when taks fails and returns the reason.

Syntax

Follow the syntax below to create a promise using the Promise() constructor.

```
let promise = new Promise(Callback); // Producing the code
OR
let promise = new Promise(function(resolve, reject){
   // Callback function body
});
```

States of the Promise

There are 4 states of the Promise.

Promise.state	Description	Promise.result
Fulfilled	When a promise is completed with a successful response.	Resultant data
Rejected	When a promise is failed.	An error object
Pending	When a promise is pending to execute.	Undefined
Settled	When a promise is either fulfilled or rejected successfully.	Either result data or an error object

- Example
- In the below code, we have used the Promise() constructor to define an instance of the Promise object.
- In the callback function, we resolve the promise if the value of the num variable is 10. Otherwise, we reject the promise.
- You can observe the promise1 in the output, it prints [Object Promise].

```
<html>
<body>
   <div id = "output">The promise1 object is:  </div>
   <script>
      var num = 10;
      const promise1 = new Promise((resolve, reject) => {
         if (num == 10) {
            resolve('The value of the number is 10 <br>');
         } else {
            reject('The value of the number is not 10 <br>');
         }
      });
      document.getElementById('output').innerHTML += promise1;
   </script>
</body>
</html>
```

JavaScript Objects

- The JavaScript **object** is a non-primitive data type that is used to store data as **key-value** pairs. The key-value pairs are often referred as properties. A key in a key-value pair, also called a "property name", is a string and value can be anything. If a property's value is a function, the property is known as a **method**.
- Objects are created using curly braces and each property is separated by a comma. Each property is written as property name followed by colon (:) followed by property value.

- const student = {
-   name: "Avi",
-   age: 20,
-   course: "IT"
- };

```
<html>
<body>
   <p id = "output"> </p>
   <script>
      const myBook = {
         title: "Perl",
         author: "Mohtashim",
         pages: 355,
      }
      document.getElementById("output").innerHTML =
      "Book name is : " + myBook.title + "<br>"
      +"Book author is : " + myBook.author + "<br>"
      +"Total pages : " + myBook.pages;
   </script>
</body>
</html>
```

Regular Expressions and RegExp Object

- A **regular expression** (RegExp) in JavaScript is an object that describes a pattern of characters. It can contain the alphabetical, numeric, and special characters. Also, the regular expression pattern can have single or multiple characters.
- The JavaScript **RegExp** class represents regular expressions, and both String and **RegExp** define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text.
- The regular expression is used to search for the particular pattern in the string or replace the pattern with a new string.
- There are two ways to construct the regular expression in JavaScript.

1. Using the RegExp() constructor.
2. Using the regular expression literal.

Syntax

A regular expression could be defined with the **RegExp ()** constructor, as follows –

```
var pattern = new RegExp(pattern, attributes);
or simply
var pattern = /pattern/attributes;
```

Parameters

Here is the description of the parameters –

- **pattern** – A string that specifies the pattern of the regular expression or another regular expression.
- **attributes** – An optional string containing any of the "g", "i", and "m" attributes that specify global, case-insensitive, and multi-line matches, respectively.

Modifiers

Several modifiers are available that can simplify the way you work with **regexps,** like case sensitivity, searching in multiple lines, etc.

Sr.No.	Modifier & Description
1	**i** Perform case-insensitive matching.
2	**m** Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary
3	**g** Performs a global matchthat is, find all matches rather than stopping after the first match.

- Brackets
- Brackets ([]) have a special meaning when used in the context of regular expressions. They are used to find a range of characters.

Sr.No.	Expression & Description
1	**[...]** Any one character between the brackets.
2	**[^...]** Any one character not between the brackets.
3	**[0-9]** It matches any decimal digit from 0 through 9.
4	**[a-z]** It matches any character from lowercase **a** through lowercase **z**.
5	**[A-Z]** It matches any character from uppercase **A** through uppercase **Z**.
6	**[a-Z]** It matches any character from lowercase **a** through uppercase **Z**.

- Quantifiers
- The frequency or position of bracketed character sequences and single characters can be denoted by a special character. Each special character has a specific connotation. The +, *, ?, and $ flags all follow a character sequence.

Sr.No.	Expression & Description
1	**p+** It matches any string containing one or more p's.
2	**p*** It matches any string containing zero or more p's.
3	**p?** It matches any string containing at most one p.
4	**p{N}** It matches any string containing a sequence of **N** p's
5	**p{2,3}** It matches any string containing a sequence of two or three p's.
6	**p{2, }** It matches any string containing a sequence of at least two p's.

7	**p$** It matches any string with p at the end of it.
8	**^p** It matches any string with p at the beginning of it.
9	**?!p** It matches any string which is not followed by a string p.

Examples

Following examples explain more about matching characters.

Sr.No.	Expression & Description
1	**[^a-zA-Z]** It matches any string not containing any of the characters ranging from **a** through **z** and **A** through Z.
2	**p.p** It matches any string containing **p,** followed by any character, in turn followed by another **p**.
3	**^.{2}$** It matches any string containing exactly two characters.
4	**<b>(.*)</b>** It matches any string enclosed within <b> and </b>.
5	**p(hp)*** It matches any string containing a **p** followed by zero or more instances of the sequence **hp**.

- Literal characters
- The literal characters can be used with a backslash (\) in the regular expression. They are used to insert special characters, such as tab, null, Unicode, etc., in the regular expression.

1	**Alphanumeric** Itself
2	**\0** The NUL character (\u0000)
3	**\t** Tab (\u0009
4	**\n** Newline (\u000A)
5	**\v** Vertical tab (\u000B)
6	**\f** Form feed (\u000C)
7	**\r** Carriage return (\u000D)

```
let exp = /\d+/
```

- \d It matches 0 to 9 digits.
- + It matches one or more numeric digits.

```
let exp = /^Hi/
```

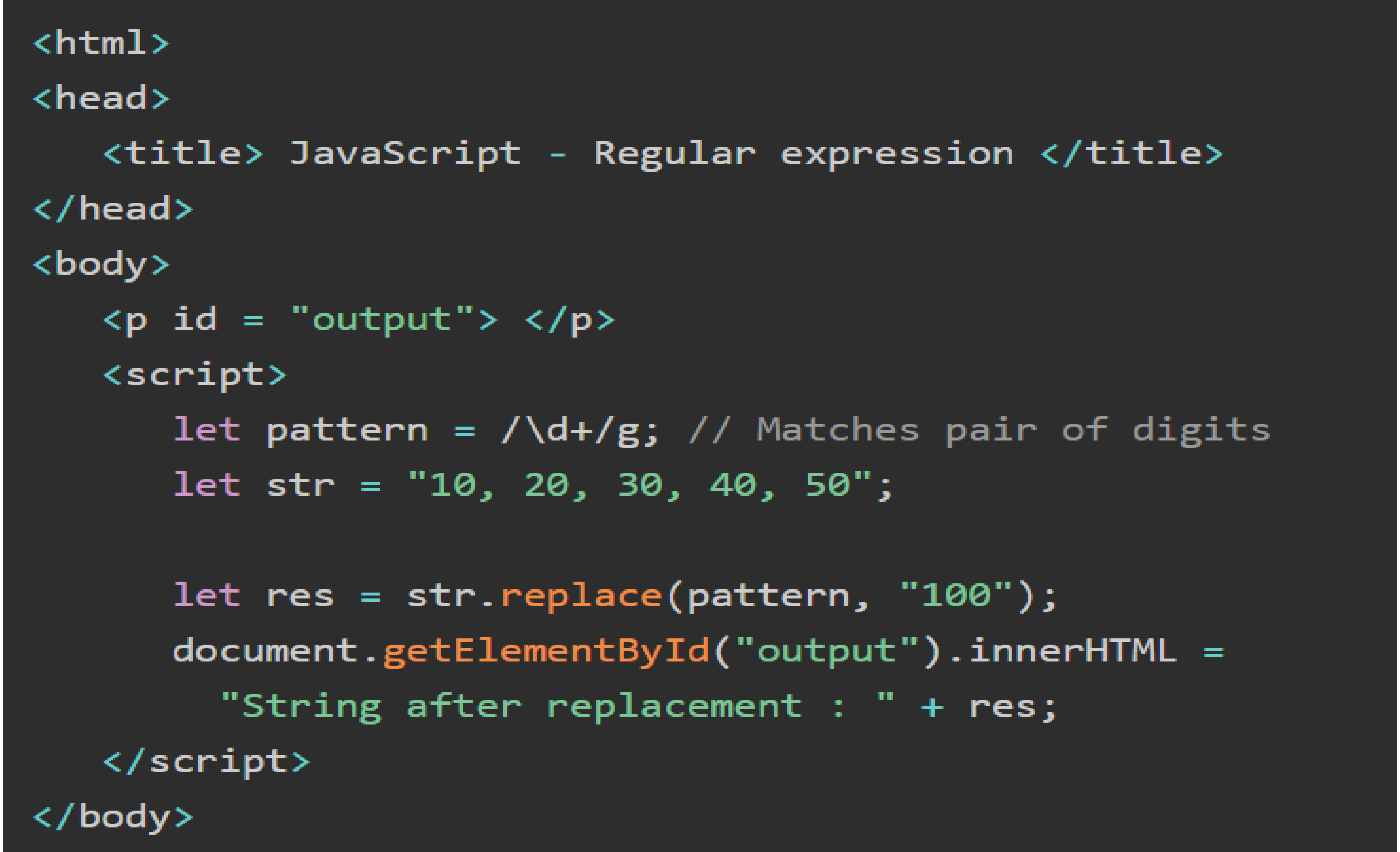
- ^ - It matches the start of the text.
- Hi It checks whether the text contains 'Hi' at the start.

```
Let exp = /^[a-zA-Z0-9]+@[a-zA-Z]+\.[a-zA-Z]{2,3}$/
```

The above regular expression validates the email. It looks complex, but it is very easy to understand.

- ^ - Start of the email address.
- [a-zA-Z0-9] It should contain the alphanumeric characters in the start.
- + - It should contain at least one alphanumeric character.
- @ - It must have the '@' character after the alphanumeric characters.
- [a-zA-Z]+ - After the '@' character, it must contain at least 1 alphanumeric character.
- \. It must contain a dot after that.
- [a-zA-Z] After the dot, the email should contain alphabetical characters.
- {2, 3} After the dot, it should contain only 2 or 3 alphabetical characters. It specifies the length.
- $ - It represents the end of the pattern.

- Example
- In the example below, we used the replace() method to match the pattern and replace it with the '100' string.
- Here, the pattern matches the pair of digits. The output shows that each number is replaced with '100' in the string. You may also add alphabetical characters in the string.

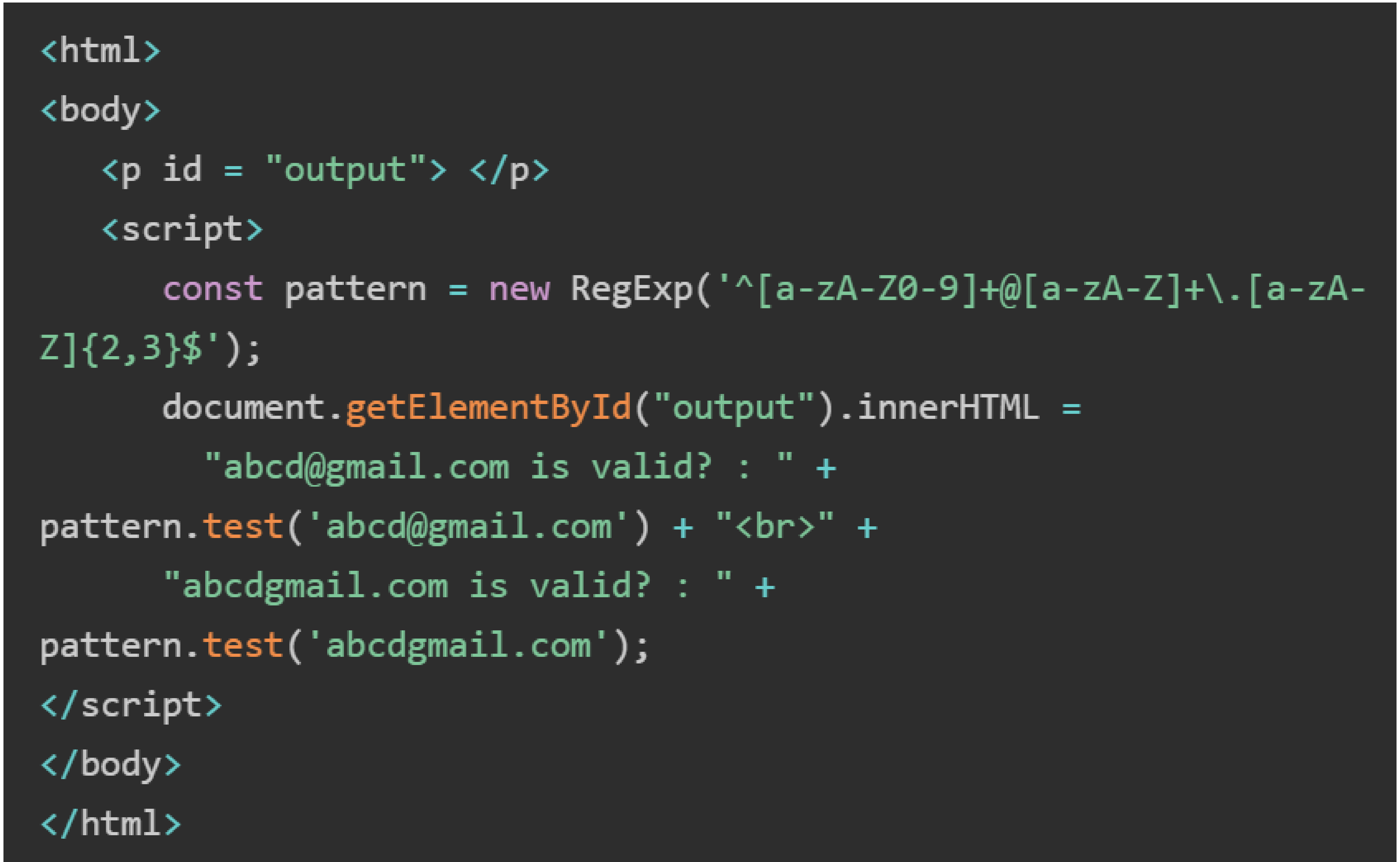


```
<html>
<head>
   <title> JavaScript - Regular expression </title>
</head>
<body>
   <p id = "output"> </p>
   <script>
      let pattern = /\d+/g; // Matches pair of digits
      let str = "10, 20, 30, 40, 50";

      let res = str.replace(pattern, "100");
      document.getElementById("output").innerHTML =
        "String after replacement : " + res;
   </script>
</body>
```

```
String after replacement : 100, 100, 100, 100, 100
```

Example (Email validation)

- In the example below, we used the RegExp() constructor function with a 'new' keyword to create a regular expression. Also, we have passed the pattern in the string format as an argument of the constructor.
- Here, we validate the email using the regular expression. In the first case, email is valid. In the second case, the email doesn't contain the @ character, so the test() method returns false.
-



```
<html>
<body>
   <p id = "output"> </p>
   <script>
      const pattern = new RegExp('^[a-zA-Z0-9]+@[a-zA-Z]+\.[a-zA-
Z]{2,3}$');
      document.getElementById("output").innerHTML =
        "abcd@gmail.com is valid? : " +
pattern.test('abcd@gmail.com') + "<br>" +
      "abcdgmail.com is valid? : " +
pattern.test('abcdgmail.com');
</script>
</body>
</html>
```

- **Common Patterns**
- *// Email validation*
- const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

- *// Phone number (US format)*
- const phoneRegex = /^\(\d{3}\) \d{3}-\d{4}$/;

- *// URL validation*
- const urlRegex = /^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$/;

- *// Password (8+ chars, 1 uppercase, 1 lowercase, 1 number)*
- const passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$/;

Asynchronous JavaScript

- Asynchronous JavaScript is a programming technique that enables your program to start a potentially long-running task and continue to executing other tasks parallelly. JavaScript is a single-threaded programming language. It means you can execute a single script or particular code at a time. JavaScript control flow moves the line by line and executes each line of code.
- We can implement asynchronous operations in our JavaScript programs using callback functions, promises, async/await etc.

What is Synchronous JavaScript?

- The Synchronous JavaScript executes the JavaScript code line-by-line. The control flow moves from top to bottom and runs each statement one by one.
- Lets understand it via the example below.

- Example
- The control flow for the code is given below.
- It calls the test1() function.
- In the test1() function, it prints the start message.
- Next, it calls the test2() function.
- The test2() function prints the start and end messages.
- After that, it prints the end message in the test1() function.
- End of the code execution.

```
<html>
<body>
   <div id = "demo"> </div>
   <script>
      let output = document.getElementById('demo');
      function test2() {
         output.innerHTML += '<br>test2 started!';
         output.innerHTML += '<br>test2 finished!';
      }
      function test1() {
         output.innerHTML += 'test1 started!';
         test2();
         output.innerHTML += '<br>test1 finished!';
      }
      test1();
   </script>
</body>
</html>
```

Output

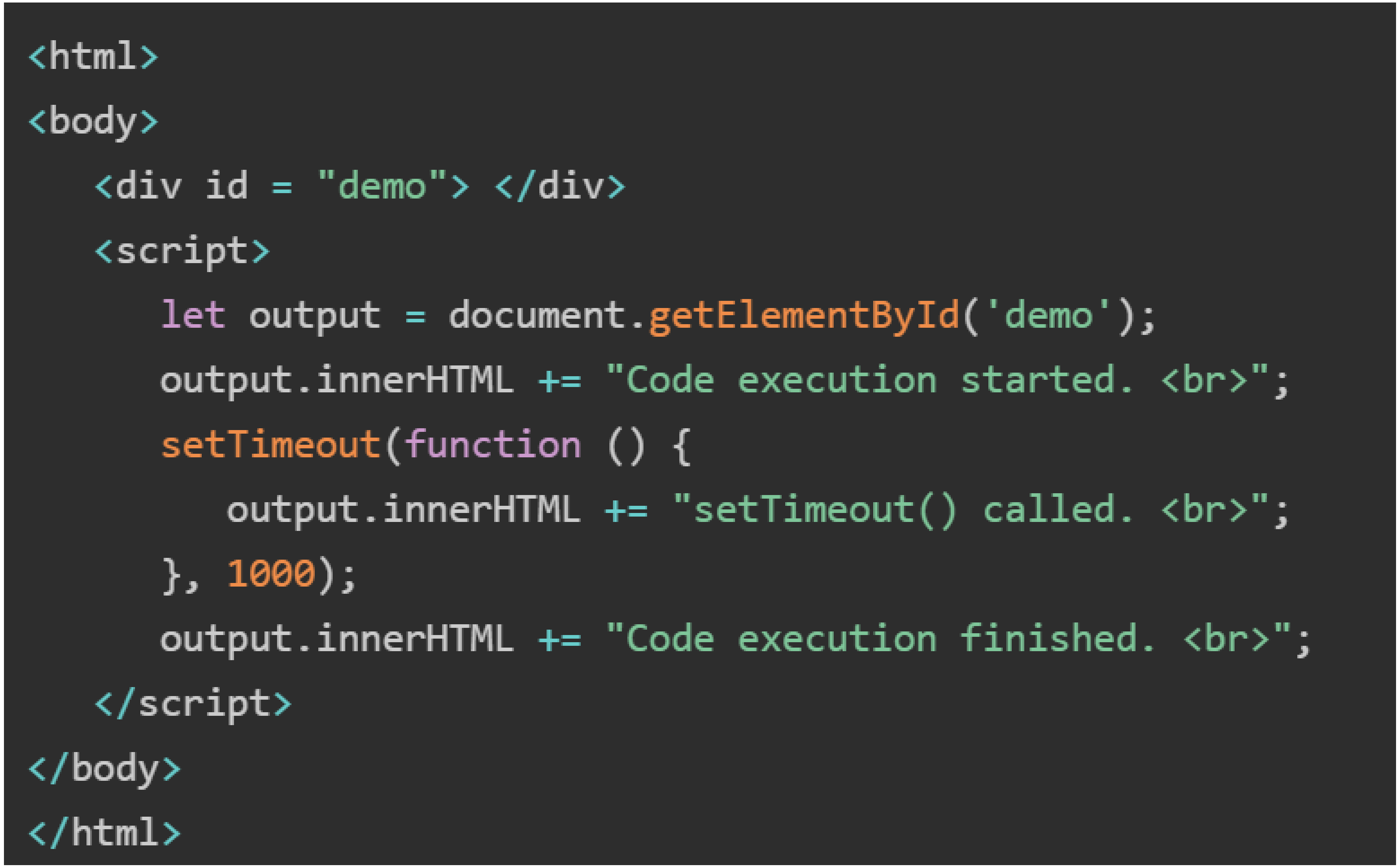
```
test1 started!
test2 started!
test2 finished!
test1 finished!
```

It creates a call stack, adds the code to it, and executes the code in the last in, first out (LIFO) order.

What is Asynchronous JavaScript?

- Asynchronous JavaScript enables simultaneous code execution. You can use asynchronous JavaScript to make your application multi-threaded. It allows you to perform time-consuming or expensive tasks together.
- Lets understand the Asynchronous JavaScript via the example below.

- Example
- The execution flow of the code is explained below.
- It prints the start message.
- After that, it prints the end message without waiting until the execution of the setTimeOut() method is finished.
- At last, it executes the code of the setTimeOut() method.



```
<html>
<body>
   <div id = "demo"> </div>
   <script>
      let output = document.getElementById('demo');
      output.innerHTML += "Code execution started. <br>";
      setTimeout(function () {
         output.innerHTML += "setTimeout() called. <br>";
      }, 1000);
      output.innerHTML += "Code execution finished. <br>";
   </script>
</body>
</html>
```

Output

```
Code execution started.
Code execution finished.
setTimeout() called.
```

By writing the asynchronous JavaScript code, you can execute another JavaScript code without waiting to finish the execution of the particular code.

JavaScript - Async/Await

- The JavaScript functions defined with the **async/await** keyword can perform the same task as promises with fewer lines of code, and it makes the code readable. The promise's syntax is a bit complex, so the async/await syntax is introduced.

- The JavaScript Async Keyword
- A JavaScript function defined with the async keyword is called the asynchronous function. The async function allows you to produce the asynchronous code.
- It always returns the promise. If you don't return the promise manually and return the data, string, number, etc., it creates a new promise and resolves that promise with the returned value.

Syntax

You could use the syntax below to define a function using the async keyword in JavaScript –

```
async function func_name(parameters) {
   // function body
}
```

In the above syntax, we have used the 'async' keyword before the function name.

Look at the below asynchronous function, returning the 'hello world' text. It returns the promise with the 'hello world' success message.

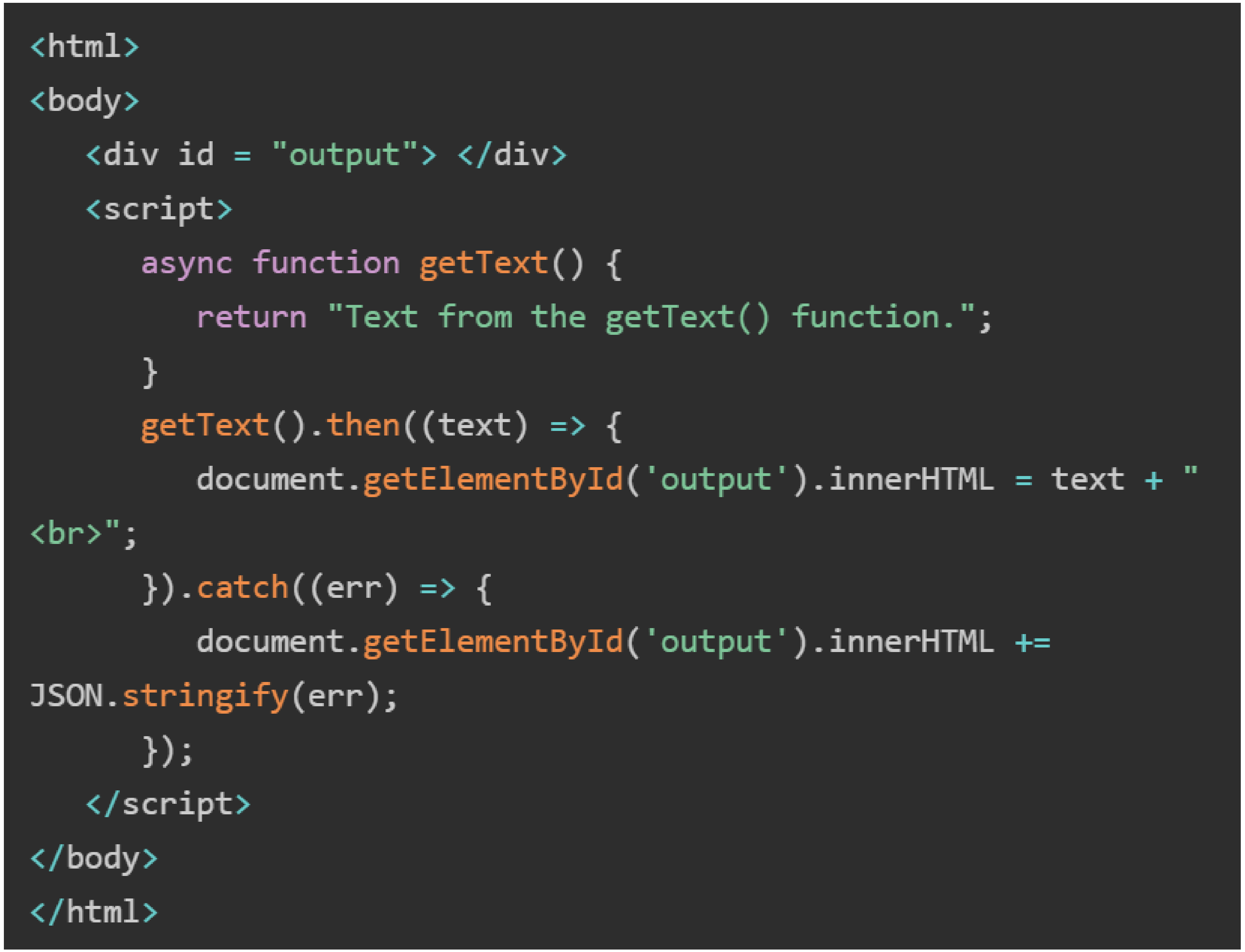
```
async function printMessage() {
   return "Hello World";
}
```

The below code is similar to the above code.

```
function printMessage() {
   return Promise.resolve("Hello World");
}
```

You can use the then() and catch() methods to solve the promise returned from the asynchronous function.

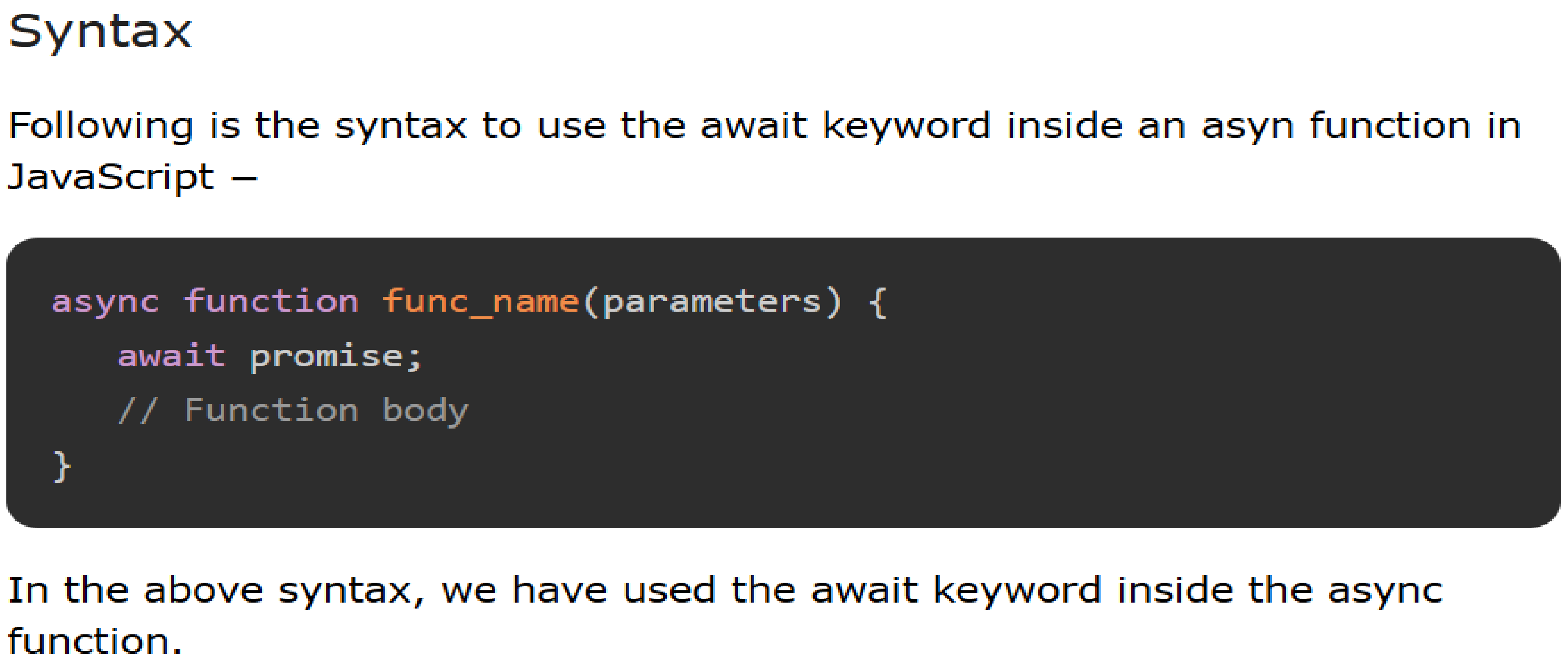
- Example
- We return the text from the getText() function in the code below.
- After that, we use the then() and catch() method with the execution of the getText() method to consume the promise returned by the getText() function.
- Here, you can observe that we have returned text from the asynchronous function, but it is returning the promise.



```html
<html>
<body>
   <div id = "output"> </div>
   <script>
      async function getText() {
         return "Text from the getText() function.";
      }
      getText().then((text) => {
         document.getElementById('output').innerHTML = text + "
<br>";
      }).catch((err) => {
         document.getElementById('output').innerHTML +=
JSON.stringify(err);
      });
   </script>
</body>
</html>
```

JavaScript Await Keyword

- You can use the await keyword inside a JavaScript asynchronous function only. It pauses the execution of the function until the promise gets settled, which means it is either rejected or fulfilled.



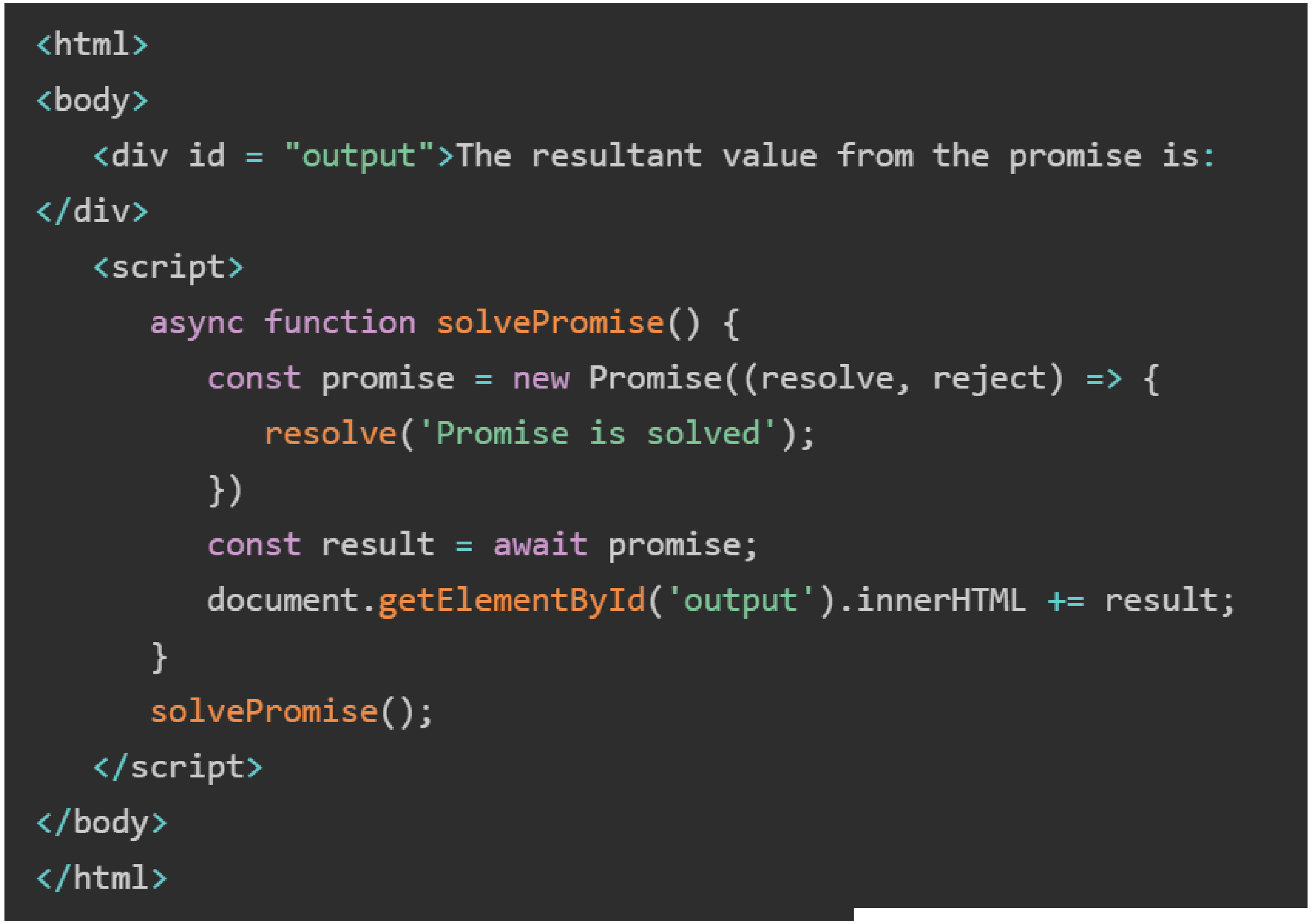
Syntax

Following is the syntax to use the await keyword inside an asyn function in JavaScript –

```
async function func_name(parameters) {
   await promise;
   // Function body
}
```

In the above syntax, we have used the await keyword inside the async function.

- Example
- We have defined the solvePromise() async function in the code below. We have created the new promise using the Promise() constructor in the function. After that, we used the await keyword with the promise to resolve it rather than using the then() or catch() method.
- In the output, you can observe that it prints the fulfillment message.



```html
<html>
<body>
   <div id = "output">The resultant value from the promise is:
</div>
   <script>
      async function solvePromise() {
         const promise = new Promise((resolve, reject) => {
            resolve('Promise is solved');
         })
         const result = await promise;
         document.getElementById('output').innerHTML += result;
      }
      solvePromise();
   </script>
</body>
</html>
```

Output

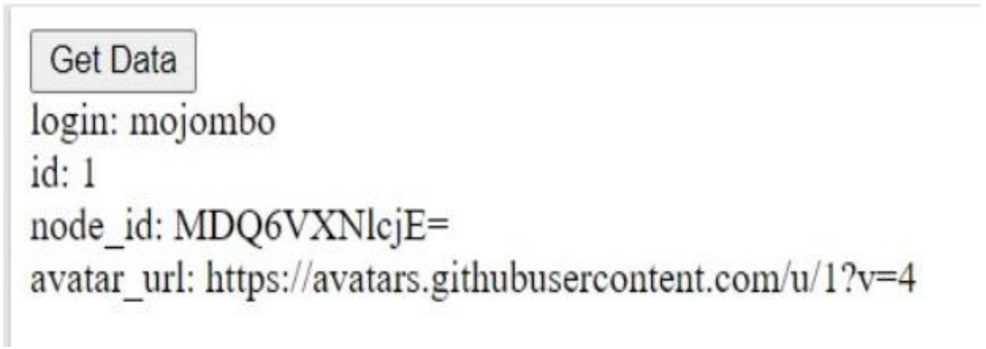
```
The resultant value from the promise is: Promise is solved
```

- Real-time Example of JavaScript Async/Await
- The above examples are basic examples to demonstrate the use of the async/await keywords in JavaScript.
- Lets understand how to use the async/await in real-time development.

- Example
- In the code below, when the user clicks the button, it calls the getData() function.
- The getData() function is an asynchronous function. We used the trycatch block inside the function to handle errors.
- In the try block, we used the fetch() API to fetch the data from the API and used the await keyword.
- After that, we used the json() method with the response to convert into the JSON and used the await keyword with that.
- Next, we print the data.
- Also, we print the error message in the catch() method.

```
<html>
<body>
  <button onclick = "getData()">Get Data</button>
  <div id = "demo"> </div>
  <script>
    let output = document.getElementById('demo');
    async function getData() {
      try {
        let response = await
fetch('https://api.github.com/users');
        // Pauses the execution until it gets the data
        let data = await response.json();
        // Pauses the execution until it converts the data
into json
        output.innerHTML += "login: " + data[0].login + "
<br>";
        output.innerHTML += "id: " + data[0].id + "<br>";
        output.innerHTML += "node_id: " + data[0].node_id +
"<br>";
        output.innerHTML += "avatar_url: " +
data[0].avatar_url + "<br>";
      } catch (err) {
        output.innerHTML += "The error is: " +
json.stringify(err);
      }
    }
  </script>
</body>
</html>
```

Output



Get Data

```
login: mojombo
id: 1
node_id: MDQ6VXNlcjE=
avatar_url: https://avatars.githubusercontent.com/u/1?v=4
```

Benefits of Using JavaScript Async Function

Here are some benefits of using the asynchronous function.

- It increases the readability of code.
- It reduces the complexity of the code.
- It can handle multiple promises easily.
- It makes debugging easier.
- You can replace the callback function and promises with the asynchronous function.

What is a Fetch API?

- The JavaScript **Fetch API** is a web API that allows a web browser to make HTTP request to the web server. In JavaScript, the fetch API is introduced in the ES6 version. It is an alternative of the XMLHttpRequest (XHR) object, used to make a 'GET', 'POST', 'PUT', or 'DELETE' request to the server.
- The window object of the browser contains the **Fetch API** by default.

Syntax

You can follow the syntax below to use the fetch() API in JavaScript

```
window.fetch(URL, [options]);
OR
fetch(URL, [options]);
```

Parameters

The fetch() method takes two parameters.

- **URL** – It is an API endpoint where you need to make a request.
- **[options]** – It is an optional parameter. It is an object containing the method, headers, etc., as a key.

Handling Fetch() API Response with 'then...catch' Block

The JavaScript fetch() API returns the promise, and you can handle it using the 'then...catch' block.

Follow the syntax below to use the fetch() API with the 'then...catch' block.

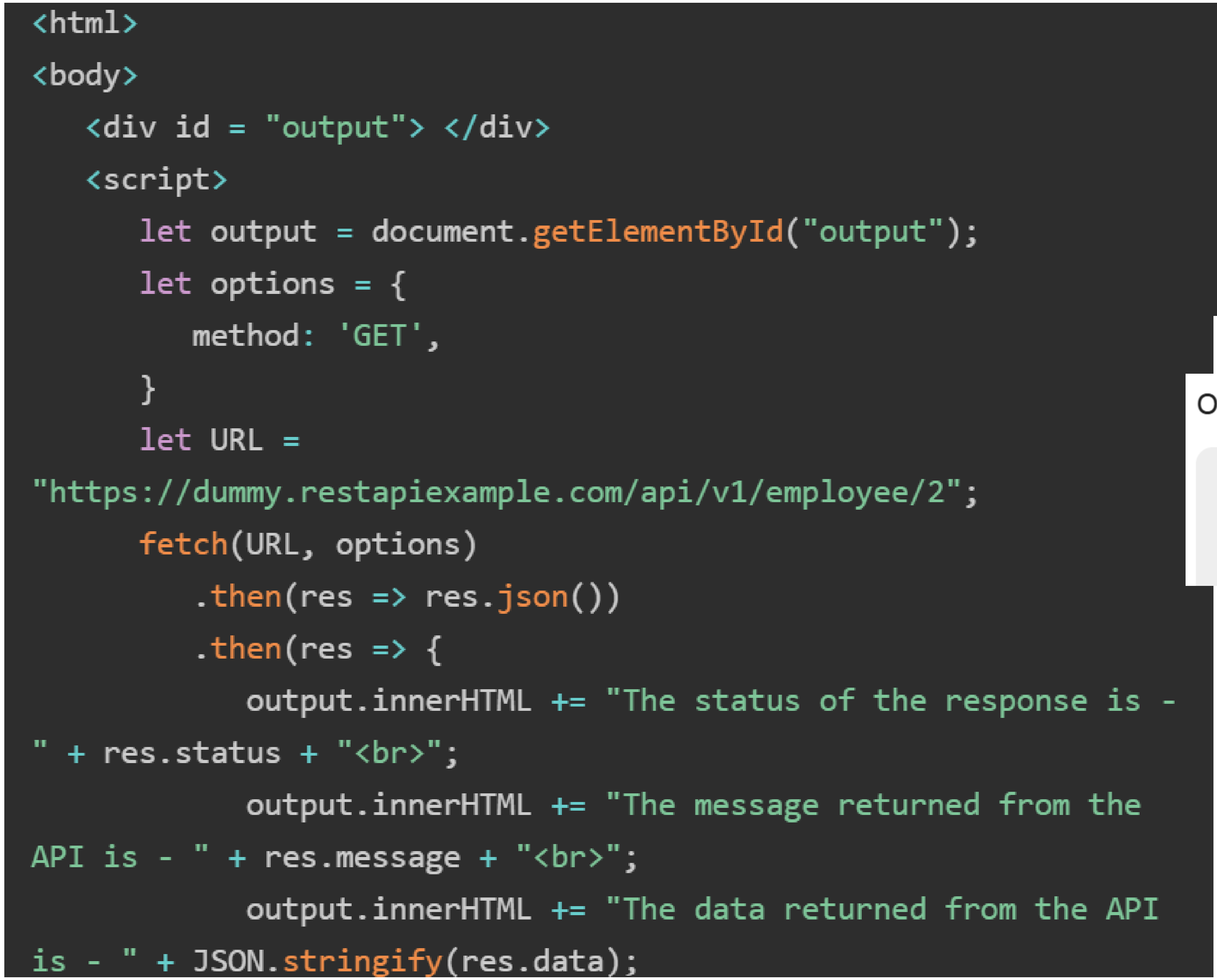
```
fetch(URL)
.then(data => {
   // Handle data
})
.catch(err=> {
   // Handle error
})
```

In the above syntax, you can handle the 'data' in the 'then' block and the error in the 'catch' block.

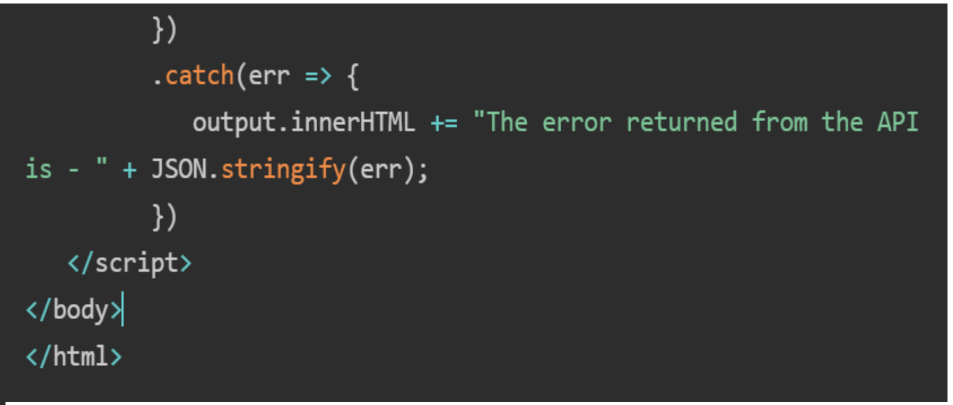
- Example
- In the below code, we fetch the data from the given URL using the fetch() API. It returns the promise we handle using the 'then' block.
- First, we convert the data into the JSON format. After that, we convert the data into the string and print it on the web page.

```
<html>
<body>
   <div id = "output"> </div>
   <script>
      const output = document.getElementById('output');
      const URL = 'https://jsonplaceholder.typicode.com/todos/5';
      fetch(URL)
      .then(res => res.json())
      .then(data => {
         output.innerHTML += "The data from the API is: " + "
<br>";
         output.innerHTML += JSON.stringify(data);
      });
   </script>
</body>
</html>
```

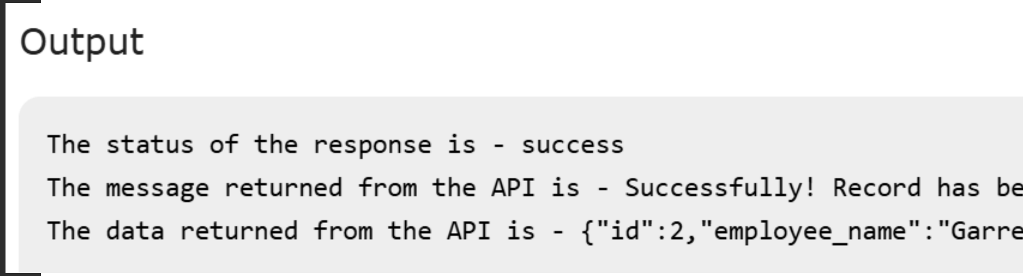
- Example: Making a GET Request
- In the below code, we have passed the "GET" method as an option to the fetch() API.
- Fetch () API fetches the data from the given API endpoint.



```
<html>
<body>
   <div id = "output"> </div>
   <script>
      let output = document.getElementById("output");
      let options = {
         method: 'GET',
      }
      let URL =
"https://dummy.restapiexample.com/api/v1/employee/2";
      fetch(URL, options)
         .then(res => res.json())
         .then(res => {
            output.innerHTML += "The status of the response is -
" + res.status + "<br>";
            output.innerHTML += "The message returned from the
API is - " + res.message + "<br>";
            output.innerHTML += "The data returned from the API
is - " + JSON.stringify(res.data);
```



```
            })
            .catch(err => {
               output.innerHTML += "The error returned from the API
is - " + JSON.stringify(err);
            })
   </script>
</body>
</html>
```



Output

```
The status of the response is - success
The message returned from the API is - Successfully! Record has be
The data returned from the API is - {"id":2,"employee_name":"Garre
```

Advantages of Using the Fetch() API

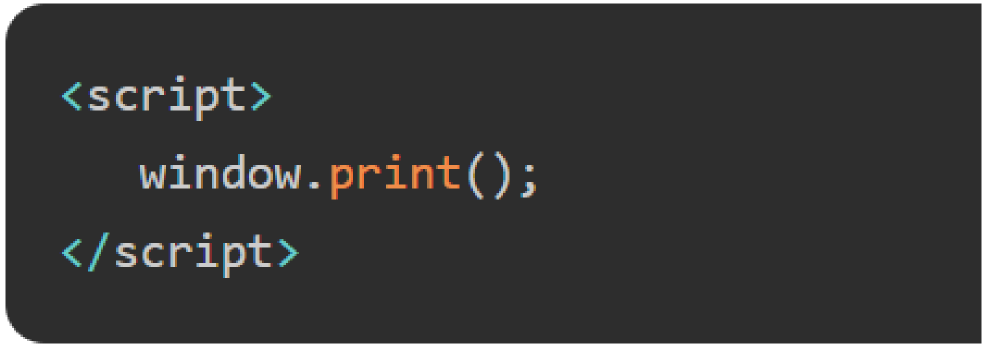
Here are some benefits of using the fetch() API to interact with third-party software or web servers.

- **Easy Syntax** – It provides a straightforward syntax to make an API request to the servers.
- **Promise-based** – It returns the promise, which you can solve asynchronously using the 'then...catch' block or 'async/await' keywords.
- **JSON handling** – It has built-in functionality to convert the string response data into JSON data.
- **Options** – You can pass multiple options to the request using the fetch() API.

Error Handling

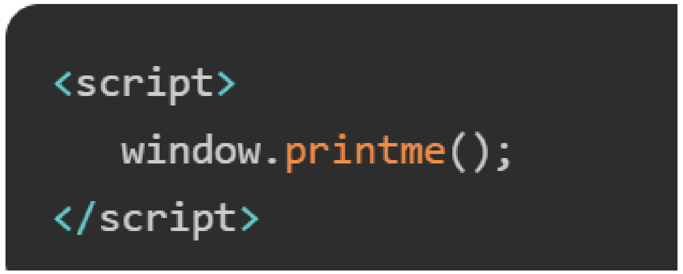
- What is an Error?
- An error is an event that occurs during the execution of a program that prevents it from continuing normally. Errors can be caused by a variety of factors, such as **syntax** errors, **runtime** errors, and **logical** errors.

- Syntax Errors
- Syntax errors, also called **parsing errors**, occur at compile time in traditional programming languages and at interpret time in JavaScript.
- For example, the following line causes a syntax error because it is missing a closing parenthesis.



```
<script>
   window.print();
</script>
```

- Runtime Errors (Exceptions)
- Runtime errors, also called **exceptions**, occur during execution (after compilation/interpretation).
- For example, the following line causes a runtime error because the syntax is correct here, but at runtime, it is trying to call a method that does not exist.



```
<script>
   window.printme();
</script>
```

There are many JavaScript runtime errors (exceptions), some are as follows –

- **ReferenceError** – Trying to access an undefined variable/ method.
- **TypeError** – Attempting an operation on incompatible data types.
- **RangeError** – A value exceeds the allowed range.

Logical Errors

Logic errors can be the most difficult type of errors to track down. These errors are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and do not get the expected result.

For example, when you divide any numeric value with 10, it returns undefined.

```
<script>
   let num = 10/0;
</script>
```

What is Error Handling?

Whenever any error occurs in the JavaScript code, the JavaScript engine stops the execution of the whole code. If you handle such errors in the proper way, you can skip the code with errors and continue to execute the other JavaScript code.

You can use the following mechanisms to handle the error.

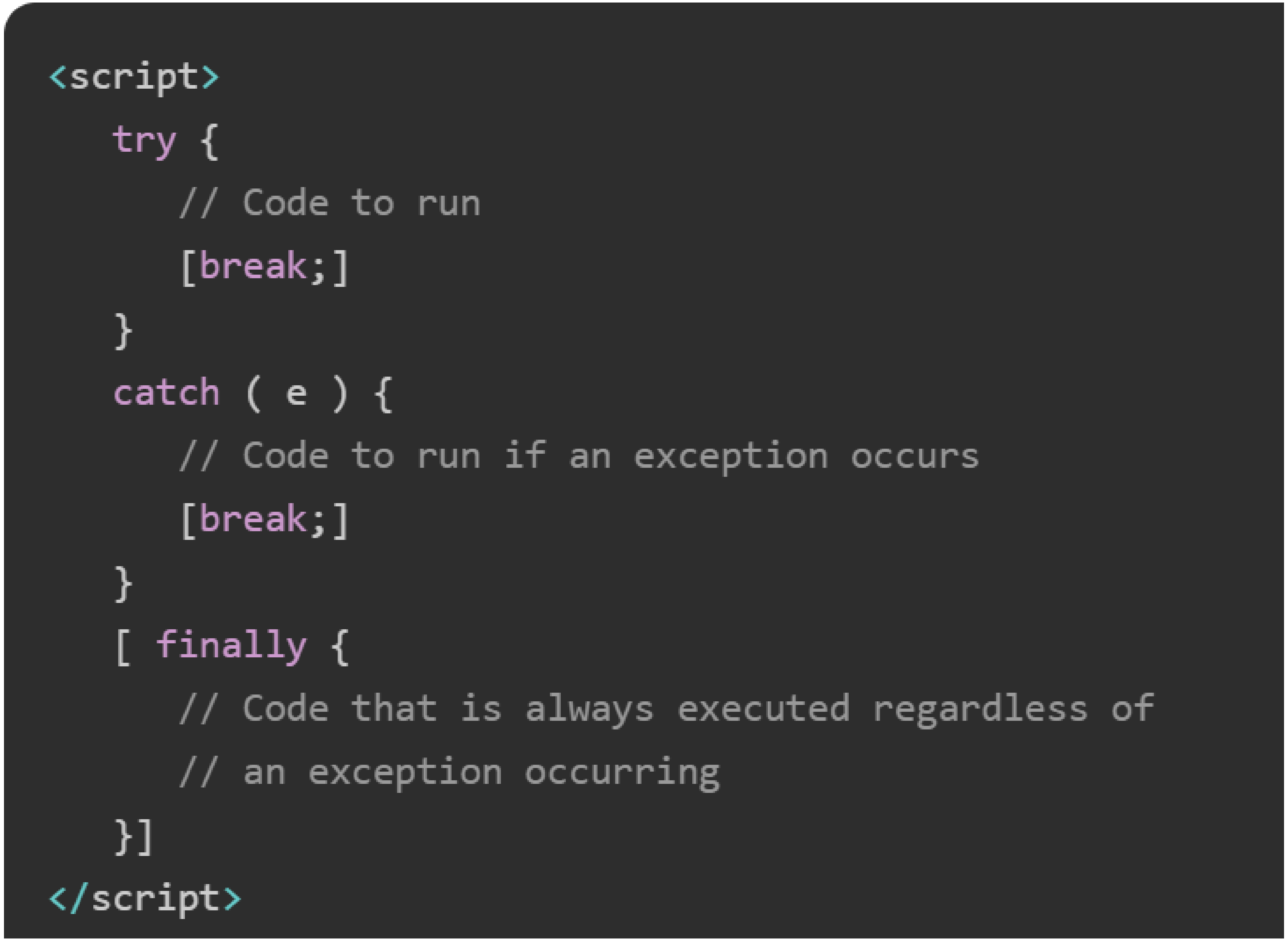
- try...catch...finally statements
- throw statements
- the onerror() event handler property
- Custom Errors

The try...catch...finally Statement

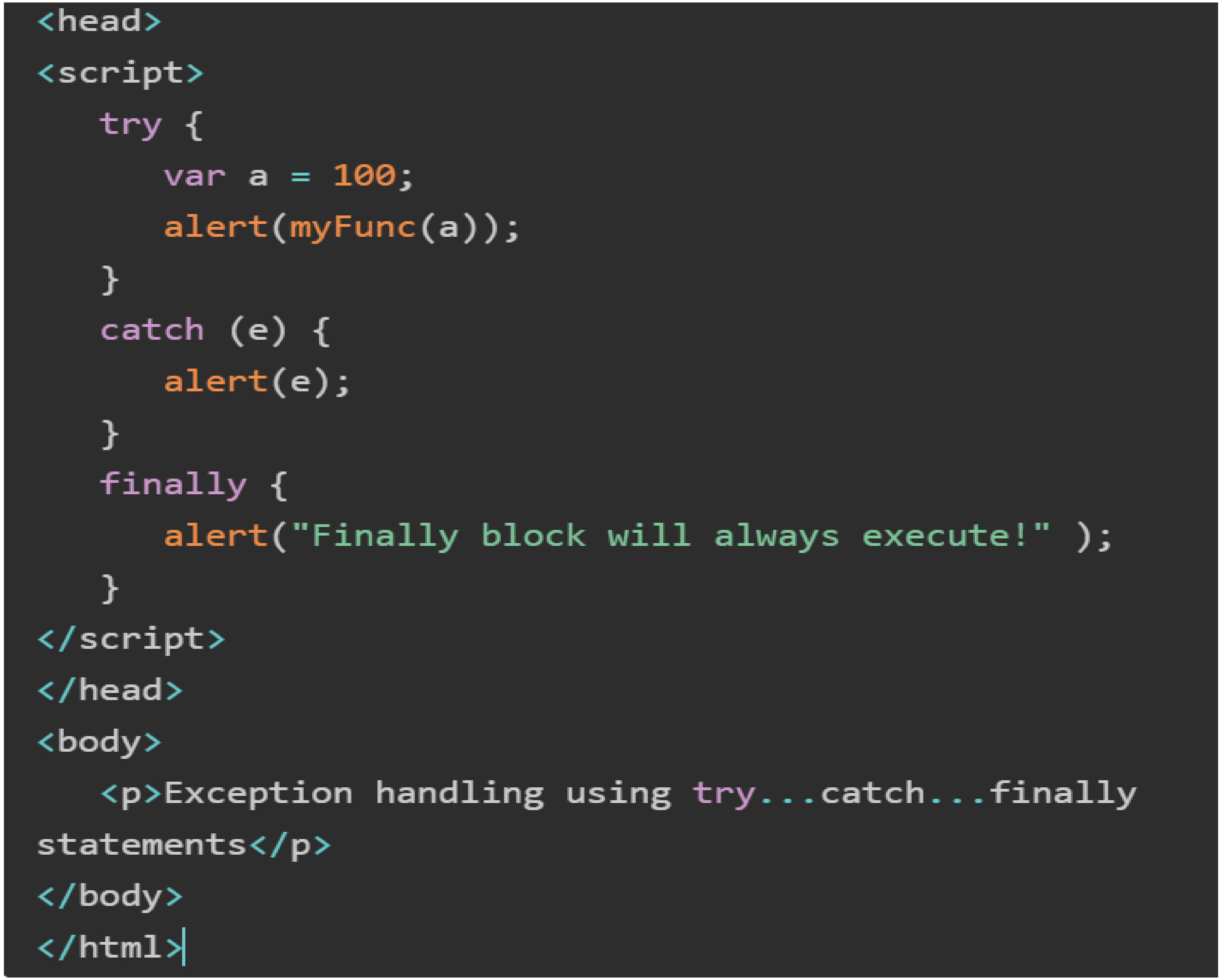
The latest versions of JavaScript added exception handling capabilities. JavaScript implements the try...catch...finally construct as well as the throw operator to handle exceptions.

You can catch programmer-generated and runtime exceptions, but you cannot catch JavaScript syntax errors.

Here is the try...catch...finally block syntax −



```
<script>
   try {
      // Code to run
      [break;]
   }
   catch ( e ) {
      // Code to run if an exception occurs
      [break;]
   }
   [ finally {
      // Code that is always executed regardless of
      // an exception occurring
   }]
</script>
```



```
<head>
<script>
   try {
      var a = 100;
      alert(myFunc(a));
   }
   catch (e) {
      alert(e);
   }
   finally {
      alert("Finally block will always execute!" );
   }
</script>
</head>
<body>
   <p>Exception handling using try...catch...finally
statements</p>
</body>
</html>
```

Output

```
Exception handling using try...catch...finaly statements
```

- The throw Statement
- You can use throw statement to raise your built-in exceptions or your customized exceptions. Later these exceptions can be captured and you can take an appropriate action.
- Example
- The following example demonstrates how to use a throw statement.

```html
<html>
<head>
<script>
   function myFunc() {
      var a = 100;
      var b = 0;

      try {
         if ( b == 0 ) {
            throw( "Divide by zero error." );
         } else {
            var c = a / b;
         }
      }
      catch ( e ) {
         alert("Error: " + e );
      }
   }
</script>
</head>
<body>
   <p>Click the following to see the result:</p>
   <form>
      <input type = "button" value = "Click Me" onclick =
"myFunc();" />
   </form>
</body>
</html>
```

Output

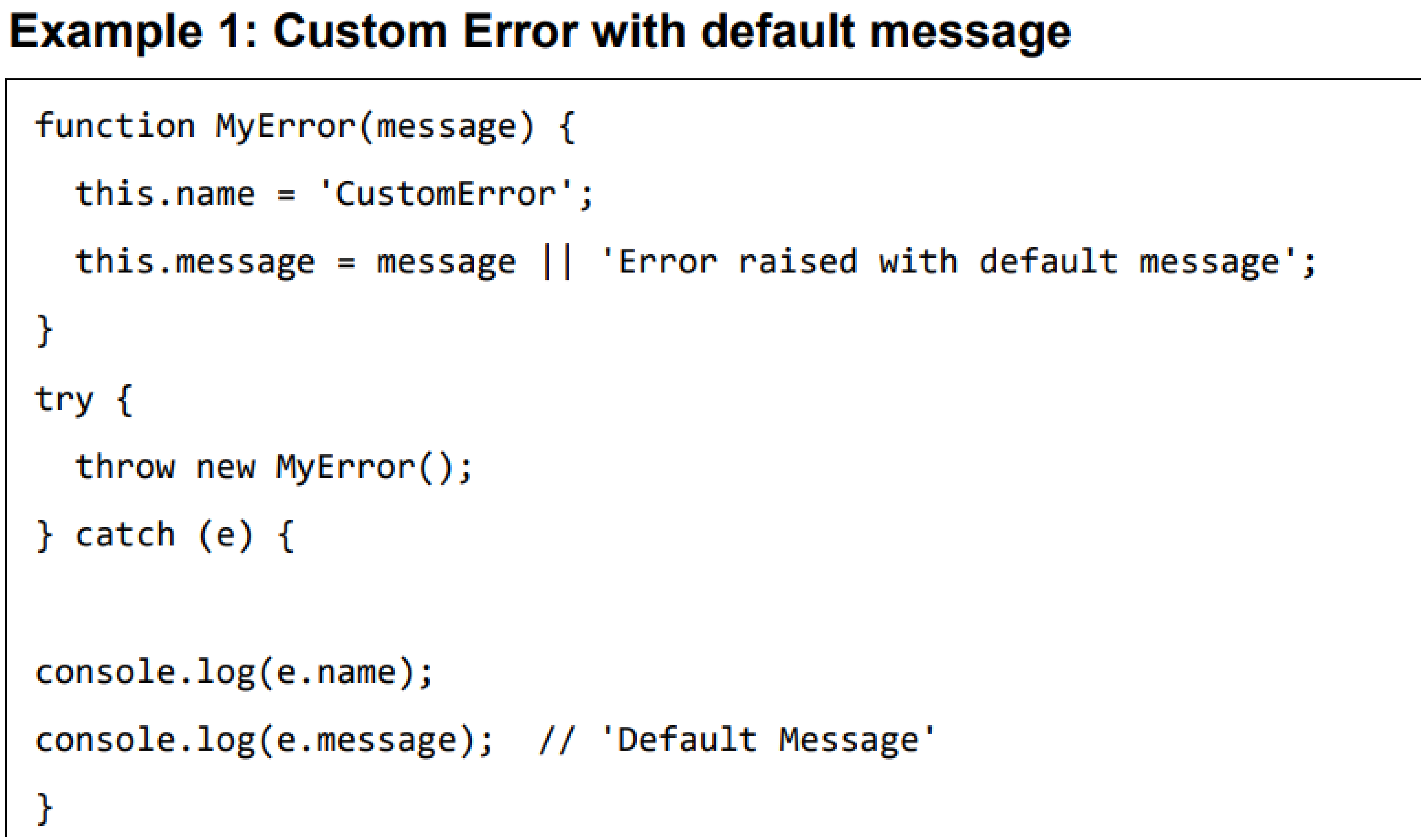
```
Click the following to see the result:
Click Me
```

- The onerror Event Handler Property
- The onerror event handler was the first feature to facilitate error handling in JavaScript. The onerror is an event handler property of the 'window' object, which automatically triggers when any error occurs on any element of the web page. You can call the callback function when any error occurs to handle the error.
- The **onerror** event handler provides three pieces of information to identify the exact nature of the error −
- **Error message** − The same message that the browser would display for the given error
- **URL** − The file in which the error occurred
- **Line number** − The line number in the given URL that caused the error

- Example
- In the code below, we added the onclick event on the <input> element, and we called the myFunc() function when users click the input element. The myFunc() function is not defined. So, it will throw an error.
- We used the 'onerror' event handler to catch the error. In the callback function, we print the error message, file URL, and line number in the file where the error occurs.

```
<html>
<body>
   <p> Click the following button to see the result:</p>
   <form>
      <input type = "button" value = "Click Me" onclick =
"myFunc();" />
   </form>
   <div id = "demo"> </div>
   <script>
      const output = document.getElementById("demo");
      window.onerror = function (msg, url, line) {
         output.innerHTML = "Error: " + msg + "<br>";
         output.innerHTML += "URL: " + url + "<br>";
         output.innerHTML += "Line: " + line + "<br>";
      }
   </script>
</body>
</html>
```

- Custom Errors
- JavaScript supports the concept of custom errors. The following example explains the same.



Example 1: Custom Error with default message

```
function MyError(message) {
  this.name = 'CustomError';
  this.message = message || 'Error raised with default message';
}
try {
  throw new MyError();
} catch (e) {

console.log(e.name);
console.log(e.message);  // 'Default Message'
}
```

The following output is displayed on successful execution of the above code.

```
CustomError

Error raised with default message
```

Example 2: Custom Error with user-defined error message

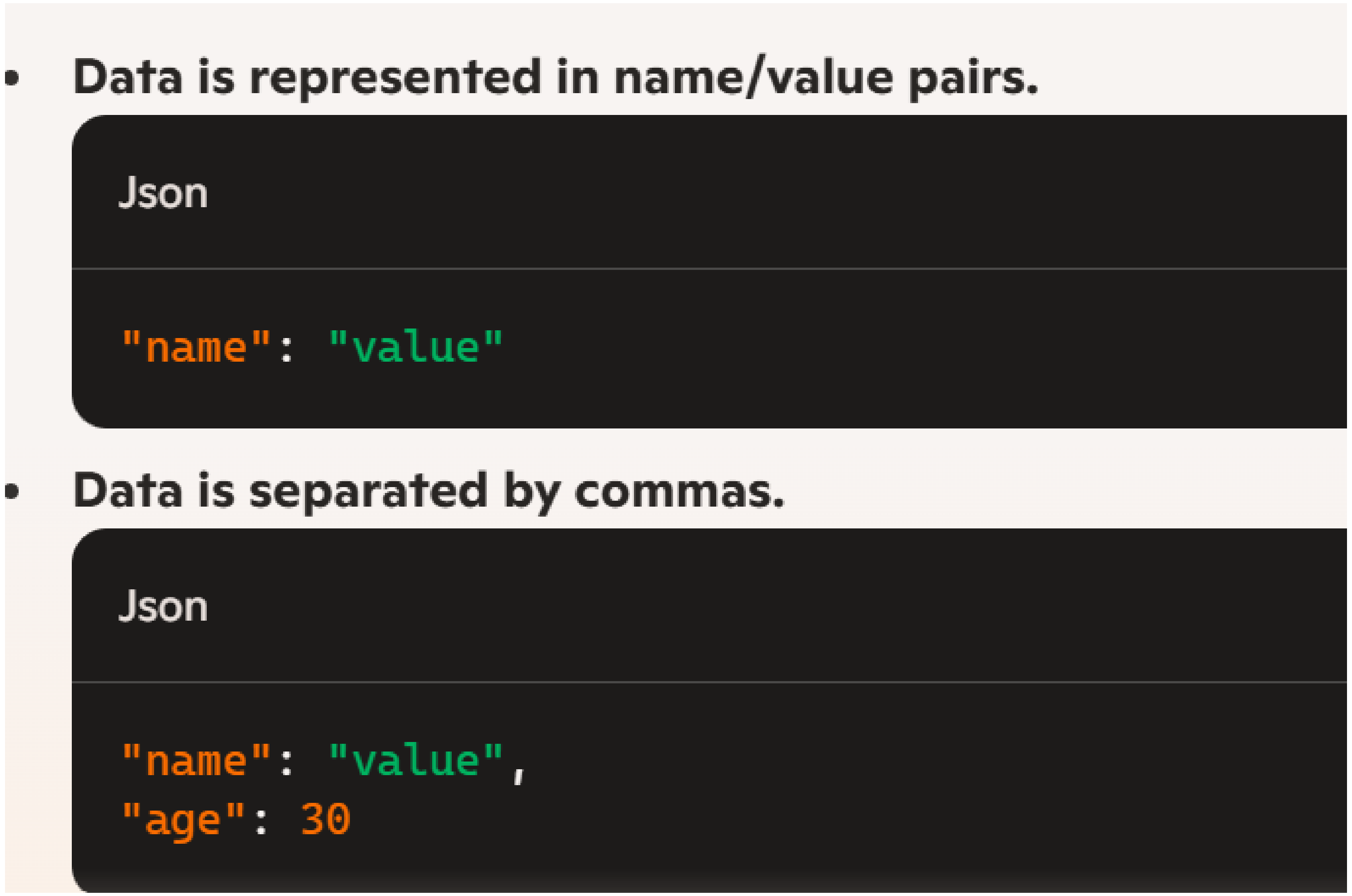
```
function MyError(message) {
  this.name = 'CustomError';
  this.message = message || 'Default Error Message';


}


try {
  throw new MyError('Printing Custom Error message');
} catch (e) {
  console.log(e.name);
  console.log(e.message);
}
```

The following output is displayed on successful execution of the above code.

```
CustomError
Printing Custom Error message
```

Introduction to JSON

- JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON is often used to transmit data between a server and a web application as text. It is language-independent and can be used with many programming languages, including JavaScript, Python, Java, and more.
- **JSON Syntax and Structure**
- JSON syntax is a subset of JavaScript object notation syntax. It follows these rules:
- **Data is represented in name/value pairs.**



- Data is represented in name/value pairs.

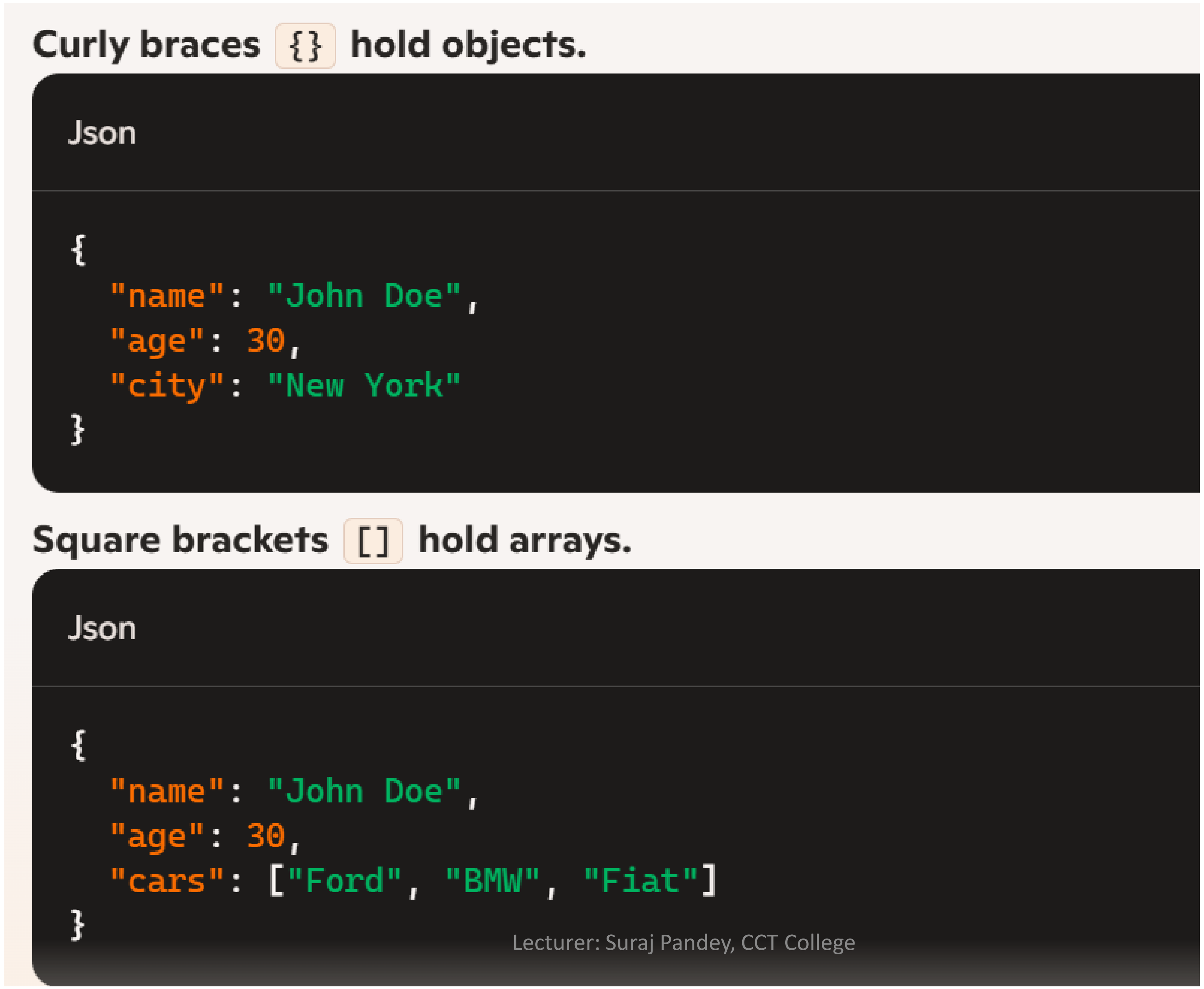
Json

```json
"name": "value"
```

- Data is separated by commas.

Json

```json
"name": "value",
"age": 30
```



Curly braces `{}` hold objects.

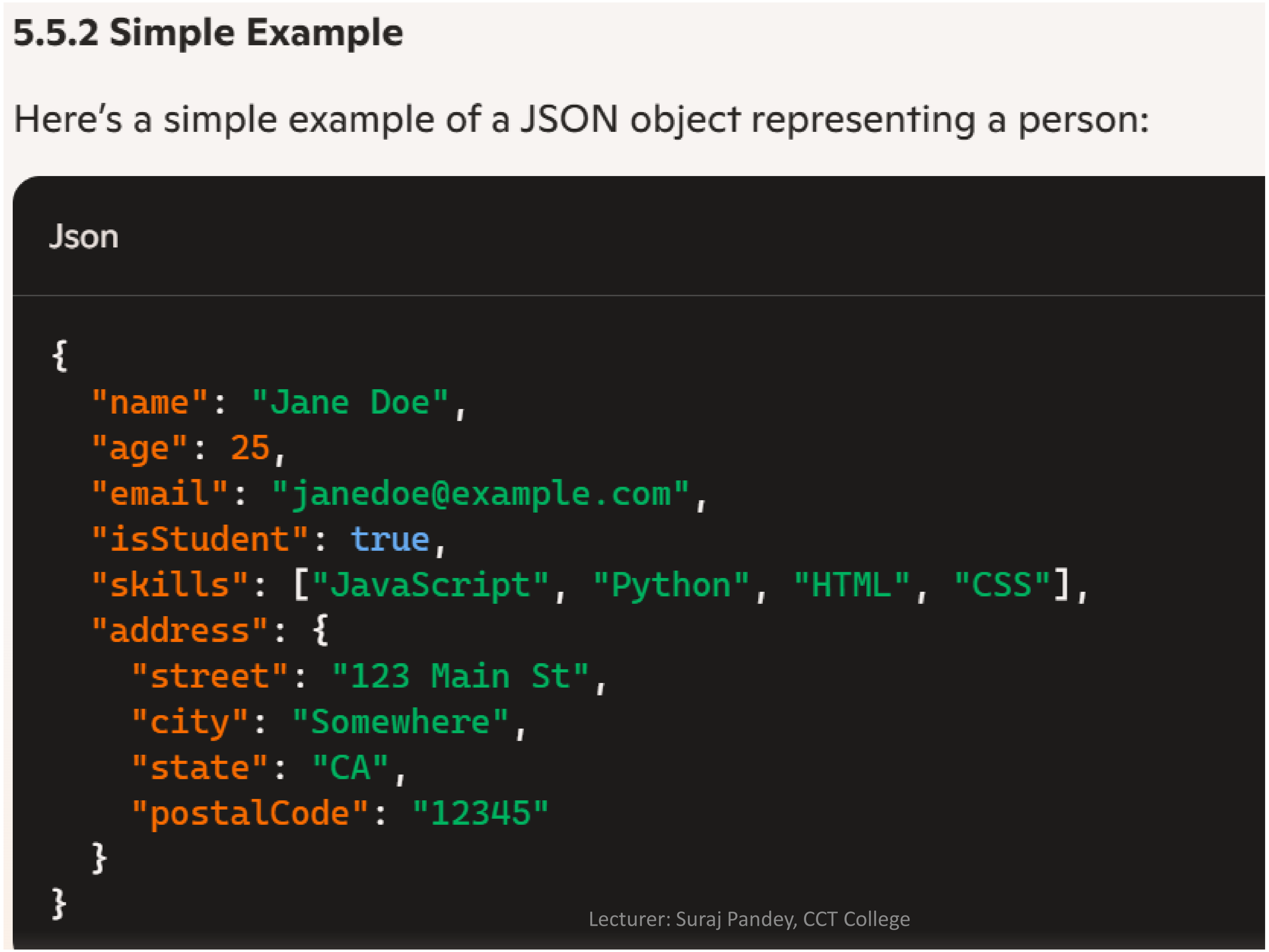
Json

```json
{
  "name": "John Doe",
  "age": 30,
  "city": "New York"
}
```

Square brackets `[]` hold arrays.

Json

```json
{
  "name": "John Doe",
  "age": 30,
  "cars": ["Ford", "BMW", "Fiat"]
}
```



5.5.2 Simple Example

Here's a simple example of a JSON object representing a person:

Json

```json
{
  "name": "Jane Doe",
  "age": 25,
  "email": "janedoe@example.com",
  "isStudent": true,
  "skills": ["JavaScript", "Python", "HTML", "CSS"],
  "address": {
    "street": "123 Main St",
    "city": "Somewhere",
    "state": "CA",
    "postalCode": "12345"
  }
}
```

- **Common Uses of JSON**
- **Data Interchange**:
  - JSON is commonly used to exchange data between a server and a client. It's often used in web applications to send and receive data in a structured format.
  - **Example**: An API might use JSON to send data from a server to a web browser, such as user information or product details.
- **Configuration Files**:
  - JSON is used for configuration files in various programming environments. It's a simple format that allows for easy parsing and modification.
  - **Example**: Application settings, environment configurations, and build scripts may use JSON files to store configuration data.

- **Storing Data**:
- JSON can be used to store data in a structured format. It's often used in databases, particularly NoSQL databases like MongoDB.
- **Example**: A document in a MongoDB collection might be stored in JSON format to represent complex data structures.
- **Serialization and Deserialization**:
- JSON is used for serializing and deserializing data. Serialization converts data structures into a JSON string, while deserialization converts a JSON string back into data structures.
- **Example**: A JavaScript object can be serialized into JSON for storage or transmission and deserialized back into a JavaScript object for use in an application.

- **APIs and Web Services**:
- JSON is widely used in APIs and web services to enable communication between different systems. It provides a standardized way to structure data for APIs.
- **Example**: RESTful APIs often use JSON to format the request and response data, making it easy for clients and servers to understand the exchanged information.

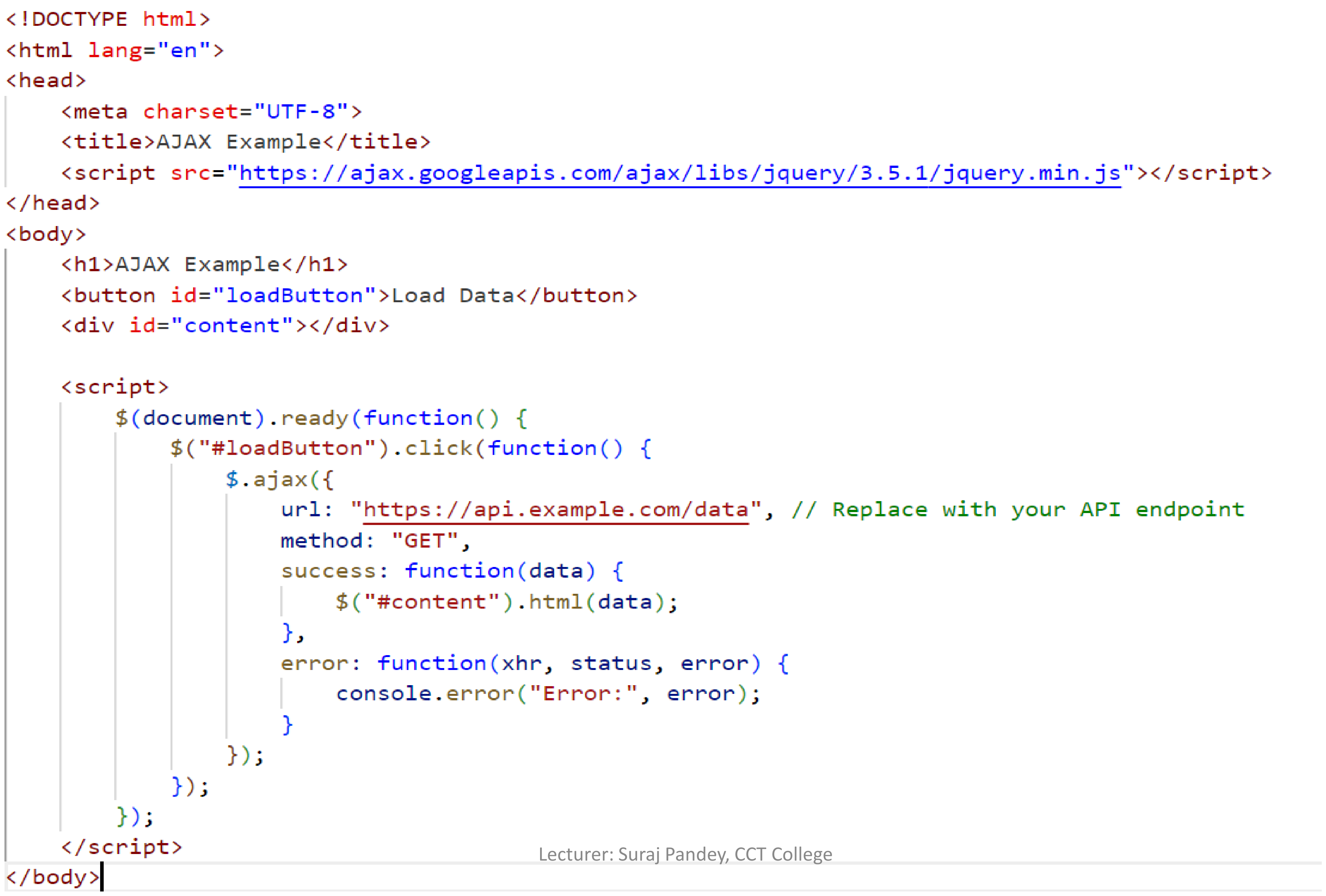
- **Benefits of JSON**
- **Lightweight**: JSON is a minimal and lightweight format, making it efficient for data transfer over the internet.
- **Readable**: JSON is easy to read and write for humans, with a simple syntax that is straightforward to understand.
- **Language-Independent**: JSON is language-agnostic, meaning it can be used with virtually any programming language. Many languages have built-in support for parsing and generating JSON.
- **Structured**: JSON provides a structured way to represent complex data, including nested objects and arrays, which is useful for representing hierarchical data.

- **Introduction to AJAX**
- **AJAX** (Asynchronous JavaScript and XML) is a technique used in web development to create dynamic and interactive web pages. It allows web pages to be updated asynchronously by exchanging data with a web server in the background. This means that parts of a web page can be updated without having to reload the entire page.
- **Key Concepts of AJAX**
- **Asynchronous**:
  - **Description**: AJAX operates asynchronously, meaning that the browser does not have to wait for a server response before continuing to execute other scripts. This allows for a smoother and more responsive user experience.
  - **Example**: Loading new content or submitting form data without refreshing

- **JavaScript and XML**:
- **JavaScript**: AJAX relies on JavaScript to make asynchronous HTTP requests to the server and handle the server's response.
- **XML**: Originally, AJAX used XML to format the data sent and received. However, JSON (JavaScript Object Notation) has become more popular due to its simplicity and ease of use.

How AJAX Works

- **Create an XMLHttpRequest Object**: This object is used to interact with the server and make HTTP requests.
- **Configure the Request**: Set up the request by specifying the HTTP method (e.g., GET, POST) and the URL to which the request is sent.
- **Send the Request**: Send the HTTP request to the server.
- **Handle the Response**: Define a callback function to handle the server's response. This function updates the web page content based on the data received from the server.
- **Basic AJAX Example**
- Here's a simple example of using AJAX to fetch data from a server and update the web page content without refreshing the page:



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>AJAX Example</title>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
</head>
<body>
    <h1>AJAX Example</h1>
    <button id="loadButton">Load Data</button>
    <div id="content"></div>

    <script>
        $(document).ready(function() {
            $("#loadButton").click(function() {
                $.ajax({
                    url: "https://api.example.com/data", // Replace with your API endpoint
                    method: "GET",
                    success: function(data) {
                        $("#content").html(data);
                    },
                    error: function(xhr, status, error) {
                        console.error("Error:", error);
                    }
                });
            });
        });
    </script>
</body>
```

- **Benefits of Using AJAX**
- **Improved User Experience**: AJAX allows web pages to be more responsive and interactive by updating content without reloading the entire page.
- **Reduced Server Load**: By fetching only the necessary data, AJAX reduces the amount of data transferred between the client and server, leading to faster load times and reduced server load.
- **Enhanced Performance**: AJAX enables faster and more efficient data retrieval and display, improving the overall performance of web applications.