Introduction to JSON

What is JSON?

JSON, or 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 well as to store and exchange data.

JSON Syntax

JSON data is represented as key-value pairs, similar to how objects are represented in JavaScript. The basic syntax includes:

- Data is in name/value pairs.
- Data is separated by commas.
- Curly braces `{}` hold objects.
- Square brackets `[]` hold arrays.

Example:

```
{
  "name": "John Doe",
  "age": 30,
  "city": "New York",
  "isStudent": false,
  "grades": [95, 87, 92]
}
```

In this example, we have an object with various properties such as name, age, city, isStudent, and grades. The "grades" property contains an array of numbers

JSON Data Types

JSON supports several data types:

1. **String**: A sequence of characters, represented in double quotes.

   ```
   "name": "John Doe"
   ```

2. **Number:** An integer or a floating-point number.

   ```
   "age": 30
   ```

3. **Boolean**: Either true or false.

   ```
   "isStudent": false
   ```

4. **Array**: An ordered list of values.

   ```
   "grades": [95, 87, 92]
   ```

5. Object: An unordered collection of key-value pairs.

   ```
   {
          "name": "John Doe",
          "age": 30
   }
   ```

6. Null: Represents an empty value or absence of value.

   ```
   "middleName": null
   ```

JSON Example Use Cases

1. **Web APIs:** JSON is commonly used for data exchange between a web server and a web page. APIs often return JSON data.
2. **Configuration Files:** JSON is used in configuration files due to its simple and human-readable format.
3. **Storage of Data:** JSON is often used to store data in databases or files.

Working with JSON in Programming Languages

Most programming languages provide libraries or built-in functions to work with JSON. For example:

- In JavaScript: `JSON.parse()` to convert a JSON string to an object and `JSON.stringify()` to convert an object to a JSON string.
- **JSON Object**
  - A JSON object is a data structure that consists of key-value pairs, where each key is a string and each value can be a string, number, boolean, array, another JSON object, or `null`. JSON objects are enclosed in curly braces `{}` and are unordered collections of data.
  - Example:

```
{
  "name": "John Doe",
  "age": 30,
  "city": "New York",
  "isStudent": false,
  "grades": [95, 87, 92],
  "address": {
    "street": "123 Main St",
    "zipCode": "10001"
  },
  "contact": null
}
```

  - The curly braces `{}` denote the beginning and end of the JSON object.
  - Inside the braces, you have key-value pairs, separated by commas.
    - `"name": "John Doe"`: The key is `"name"` with the value `"John Doe"`.
    - `"age": 30`: The key is `"age"` with the value `30`.
    - `"city": "New York"`: The key is `"city"` with the value `"New York"`.
    - `"isStudent": false`: The key is `"isStudent"` with the boolean value `false`.
    - `"grades": [95, 87, 92]`: The key is `"grades"` with an array value `[95, 87, 92]`.

        - `"address": {...}`: The key is `"address"` with the value being another nested JSON object.
        - `"contact": null`: The key is `"contact"` with the value `null`
- JSON String
    - Introduction:
        - A sequence of characters following the JSON (JavaScript Object Notation) format.
        - Enclosed in double quotation marks (`"`).
        - Represents structured data in a human-readable and easily parsable way.
        - Used for data interchange between different systems and programming languages.
        - Can represent various data types: text, numbers, booleans, arrays, objects, and null values.
    - Usage:
        - Commonly employed in web development for transmitting and exchanging data.
        - Provides a standardized format for communication between systems and platforms.
    - Conversion:
        - `JSON.stringify()`: Converts JavaScript objects into JSON strings.
        - `JSON.parse()`: Converts JSON strings back into JavaScript objects.
    - Interchangeability:
        - JSON strings play a crucial role in data representation and communication.
        - Enables effective and widely adopted data interchange in web development.
- **`JSON.stringify()`**
    - **Purpose:** Converts a JavaScript object into a JSON-formatted string.
    - **Syntax:** `JSON.stringify(obj, replacer, space)`
        - `obj`: The JavaScript object to be converted.
        - `replacer`: Optional. A function or array that alters the behavior of the stringification.
        - `space`: Optional. Adds indentation, making the resulting string more readable.
    - Example:
      ```
      const userObject = { name: "John", age: 25, city: "New York"
      };
      const jsonString = JSON.stringify(userObject, null, 2);
      ```

```
console.log(jsonString);
```

Output:

```
{
  "name": "John",
  "age": 25,
  "city": "New York"
}
```

- **`JSON.Parse()`**
  - **Purpose:** Parses a JSON-formatted string and converts it into a JavaScript object.
  - **Syntax:** `JSON.parse(jsonString, reviver)`
    - `jsonString`: The JSON-formatted string to be parsed.
    - `reviver`: Optional. A function that transforms the parsed object.
  - **Example:**

    ```
    const jsonString = '{"name": "Alice",
    "age": 30, "city": "London"}';
    const userObject =
    JSON.parse(jsonString);
    console.log(userObject.name);
    ```

    Output: `Alice`

Both `JSON.stringify()` and `JSON.parse()` are essential for data interchange, allowing developers to convert JavaScript objects to JSON strings for transmission and convert received JSON strings back into JavaScript objects for manipulation in the code.

A program demonstrating the use of `JSON.stringify()` to convert a JavaScript object to a JSON string and `JSON.parse()` to convert a JSON string back to a JavaScript object:

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>JSON Stringify and Parse Example</title>
</head>
```

```
<body>

<script>
 // JavaScript object
 const userObject = {
 name: "John Doe",
 age: 30,
 city: "New York",
 isStudent: false,
 grades: [95, 87, 92],
 address: {
 street: "123 Main St",
 zipCode: "10001"
 },
 contact: null
 };

 // Convert JavaScript object to JSON string
 const jsonString = JSON.stringify(userObject, null, 2);

 // Output the JSON string
 console.log("JSON String:");
 console.log(jsonString);

 // Convert JSON string back to JavaScript object
 const parsedObject = JSON.parse(jsonString);

 // Output the parsed JavaScript object
 console.log("\nParsed JavaScript Object:");
 console.log(parsedObject);
</script>

</body>
</html>
```

Explanation:

- The JavaScript object `userObject` represents some user data.
- `JSON.stringify(userObject, null, 2)` converts the JavaScript object to a JSON string with an indentation of 2 spaces for better readability.
- The JSON string is logged to the console.
- `JSON.parse(jsonString)` converts the JSON string back to a JavaScript object.

- The parsed JavaScript object is logged to the console.

You can run this HTML file in a web browser, open the browser's developer tools (usually by pressing F12 or right-clicking on the page and selecting "Inspect"), and view the console to see the output.

jQuery

Definition:
jQuery is a fast, small, and feature-rich JavaScript library. It simplifies tasks such as HTML document traversal and manipulation, event handling, animation, and Ajax (asynchronous JavaScript and XML) interactions for web development. jQuery is designed to be lightweight and easy to use, providing a simple and concise syntax for common tasks that can be cumbersome and verbose with plain JavaScript.

Key features of jQuery include:

1. **DOM Manipulation:** jQuery simplifies DOM (Document Object Model) manipulation, allowing developers to easily select, traverse, and modify HTML elements.
2. **Event Handling:** jQuery provides a streamlined way to manage events such as clicks, keypresses, and mouse interactions. Event handling is simplified with functions like `on()`.
3. **Ajax Interaction:** jQuery makes it easier to perform asynchronous HTTP requests, commonly known as Ajax, allowing web pages to update content without requiring a full page reload.
4. **Animation and Effects:** jQuery includes built-in methods for creating animations and effects, making it straightforward to add dynamic elements and transitions to a webpage.
5. **Cross-browser Compatibility:** jQuery abstracts away many of the inconsistencies between different web browsers, providing a consistent API that works across various browsers.

For studying JQuery further, refer to material of w3schools.com

https://www.w3schools.com/jquery/default.asp

In JavaScript, variables are used to store and manage data. The scope, as well as the keywords `let`, `const`, and `var`, play crucial roles in how variables behave within the language.

JavaScript Variables

Variables are Containers for Storing Data

JavaScript Variables can be declared in 4 ways:

- Automatically
- Using `var`
- Using `let`
- Using `const`

Example

```
x = 5;
y = 6;
z = x + y;
```

In this first example, `x`, `y`, and `z` are undeclared variables.

They are automatically declared when first used:

From the examples you can guess:

- x stores the value 5
- y stores the value 6
- z stores the value 11

JavaScript Identifiers

All JavaScript variables must be identified with unique names.

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter.
- Names can also begin with $ and _ (but we will not use it in this tutorial).
- Names are case sensitive (y and Y are different variables).
- Reserved words (like JavaScript keywords) cannot be used as names.

JavaScript Let

The `let` keyword was introduced in ES6 (2015)

Variables declared with `let` have Block Scope

Variables declared with `let` must be Declared before use

Variables declared with `let` cannot be Redeclared in the same scope

Block Scope

Before ES6 (2015), JavaScript did not have Block Scope.

JavaScript had Global Scope and Function Scope.

ES6 introduced the two new JavaScript keywords: `let` and `const`.

These two keywords provided Block Scope in JavaScript:

Example

Variables declared inside a { } block cannot be accessed from outside the block:

```
{
  let x = 2;
}
// x can NOT be used here
```

Global Scope

Variables declared with the `var` always have Global Scope.

Variables declared with the `var` keyword can NOT have block scope:

Example

Variables declared with `var` inside a { } block can be accessed from outside the block:

```
{
  var x = 2;
}
// x CAN be used here
```

Cannot be Redeclared

Variables defined with `let` can not be redeclared.

You can not accidentally redeclare a variable declared with `let`.

With `let` you can not do this:

```
let x = "John Doe";

let x = 0;
```

Variables defined with `var` can be redeclared.
With `var` you can do this:

```
var x = "John Doe";

var x = 0;
```

Redeclaring Variables

Redeclaring a variable using the `var` keyword can impose problems.

Redeclaring a variable inside a block will also redeclare the variable outside the block:

Example

```
var x = 10;

// Here x is 10


{

var x = 2;

// Here x is 2

}


// Here x is 2
```

Redeclaring a variable using the `let` keyword can solve this problem.

Redeclaring a variable inside a block will not redeclare the variable outside the block:

Example

```
let x = 10;

// Here x is 10


{

let x = 2;

// Here x is 2
}


// Here x is 10
```

Redeclaring

Redeclaring a JavaScript variable with `var` is allowed anywhere in a program:

Example

```
var x = 2;

// Now x is 2


var x = 3;

// Now x is 3
```

With `let`, redeclaring a variable in the same block is NOT allowed:

Example

```
var x = 2;    // Allowed

let x = 3;    // Not allowed


{

let x = 2;    // Allowed

let x = 3;    // Not allowed

}


{
```

```
let x = 2;    // Allowed

var x = 3;    // Not allowed

}
```

Redeclaring a variable with `let`, in another block, IS allowed:

Example

```
let x = 2;    // Allowed

{
let x = 3;    // Allowed
}

{
let x = 4;     // Allowed
}
```

Hoisted - var:

```
console.log(x); // undefined

var x = 10;

console.log(x); // 10
```

In this example, the `var x` declaration is hoisted to the top of its scope (in this case, the global scope). The first `console.log(x)` outputs `undefined` because the variable is hoisted but not yet initialized. The second `console.log(x)` after the assignment outputs `10`.

Hoisted - let:

```
// console.log(y); // ReferenceError: Cannot access 'y' before initialization
let y = 20;
console.log(y); // 20
```

With `let`, the variable is hoisted to the top of the block but is not initialized until the declaration statement. Trying to access the variable before the declaration results in a `ReferenceError`. The `console.log(y)` after the declaration outputs `20`.

Hoisted - const:

```
// console.log(z); // ReferenceError: Cannot access 'z' before
initialization
const z = 30;
console.log(z); // 30
```

Similar to `let`, `const` is hoisted to the top of the block but not initialized until the declaration statement. Attempting to access the variable before the declaration results in a `ReferenceError`. The `console.log(z)` after the declaration outputs `30`.

Difference Between var, let and const

Characteristics	**var**	**let**	**const**
Scope	Global scoped	Block-Scoped	Block Scoped
Redeclaration	Allows within same scope	Allows within block scope	Does not allow redeclaration
Reassignment	Allows	Allows	Does not allow
Hoisted	Hoisted to the top with ‘undefined’ during hoisting	Hoisted to the top but not initialized until declaration statement	Hoisted to the top but not initialized until declaration statement

Explanation:

- Scope:
    - Function Scope: `var` has function-level scope, while `let` and `const` have block-level scope.
    - Block Scope: Both `let` and `const` have block-level scope.
- Redeclare:
    - `var` allows redeclaration within the same scope, while `let` and `const` do not allow redeclaration in the same scope.
- Reassign:
    - All three (`var`, `let`, and `const`) allow reassignment. However, `const` variables cannot be reassigned after their initial assignment.
- Hoisted:
    - All three are hoisted to the top of their respective scopes during the compilation phase. However, `var` is initialized with `undefined` during hoisting, while `let` and `const` are not initialized until the declaration statement is encountered.

JavaScript Loop

Loops can execute a block of code a number of times.

Different Kinds of Loops

JavaScript supports different kinds of loops:

- `for` - loops through a block of code a number of times
- `while` - loops through a block of code while a specified condition is true
- `do-while` - also loops through a block of code while a specified condition is true
- `for-in` - loops through the properties of an object
- `for-of` - loops through the values of an iterable object
- `for-await-of` - loops through the values of an iterable object (especially useful for asynchronous operations)
- `for-each` - method available for arrays, iterates through the values of an array

1. JavaScript For Loop
2. JavaScript While and Do While Loop
3. JavaScript For In

The JavaScript `for in` statement loops through the properties of an Object:

Syntax

```
for (key in object) {
  // code block to be executed
}
```

Example

```
const person = {fname:"John", lname:"Doe", age:25};

let text = "";
for (let x in person) {
  text += person[x];
}
```

Example Explained

- The for in loop iterates over a person object
- Each iteration returns a key (x)
- The key is used to access the value of the key
- The value of the key is person[x]

For In Over Arrays

The JavaScript `for in` statement can also loop over the properties of an Array:

Syntax

```
for (variable in array) {
  code
}
```

Example

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

4. Array.forEach()

The `forEach()` method calls a function (a callback function) once for each array element.

Example

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";

numbers.forEach(myFunction);

function myFunction(value, index, array) {

  txt += value;

}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

The example above uses only the value parameter.

5. The For Of Loop

The JavaScript `for of` statement loops through the values of an iterable object.

It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more:

Syntax

```
for (variable of iterable) {

  // code block to be executed

}
```

variable - For every iteration the value of the next property is assigned to the variable. *Variable* can be declared with `const`, `let`, or `var`.

iterable - An object that has iterable properties.

Looping over an Array

Example

```
const cars = ["BMW", "Volvo", "Mini"];



let text = "";

for (let x of cars) {

  text += x;

}
```

Looping over a String

Example

```
let language = "JavaScript";


let text = "";

for (let x of language) {

text += x;

}
```

JavaScript Objects

JavaScript Accessors (Getters and Setters)

ECMAScript 5 (ES5 2009) introduced Getter and Setters.

Getters and setters allow you to define Object Accessors (Computed Properties).

JavaScript Getter (The get Keyword)

This example uses a `lang` property to `get` the value of the `language` property.

Example

```
// Create an object:
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "en",
  get lang() {
    return this.language;
  }
};

// Display data from the object using a getter:
```

```
document.getElementById("demo").innerHTML = person.lang;
```

JavaScript Setter (The set Keyword)

This example uses a `lang` property to `set` the value of the `language` property.

Example

```
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "",
  set lang(lang) {
    this.language = lang;
  }
};

// Set an object property using a setter:
person.lang = "en";

// Display data from the object:
document.getElementById("demo").innerHTML = person.language;
```

JavaScript Function or Getter?

What is the differences between these two examples?

Example 1

```
const person = {
  firstName: "John",
  lastName: "Doe",

  fullName: function() {

    return this.firstName + " " + this.lastName;

  }

};



// Display data from the object using a method:

document.getElementById("demo").innerHTML = person.fullName();
```

Example 2

```
const person = {
  firstName: "John",
  lastName: "Doe",
  get fullName() {
    return this.firstName + " " + this.lastName;
  }
};

// Display data from the object using a getter:
document.getElementById("demo").innerHTML = person.fullName;
```

Example 1 access fullName as a function: person.fullName().

Example 2 access fullName as a property: person.fullName.

The second example provides a simpler syntax.

Why Using Getters and Setters?

- It gives simpler syntax
- It allows equal syntax for properties and methods
- It can secure better data quality
- It is useful for doing things behind-the-scenes

JavaScript Object Constructors

The JavaScript this Keyword

JavaScript Classes

JS Constructors

JavaScript Class Inheritance

Javascript Async Await

JavaScript's `async` and `await` are language features introduced in ECMAScript 2017 (ES8) that make it easier to work with asynchronous code. They provide a more synchronous way to write asynchronous code, making it easier to understand and maintain.

Here's an explanation of `async` and `await`:

1. **async Function Declaration:**
   The `async` keyword is used to declare a function as asynchronous. When a

function is declared with `async`, it means that the function will always return a promise, and the value of that promise will be whatever the function returns.

```
async function fetchData() {

    // Asynchronous code here

}
```

2. await Operator:

   The `await` keyword is used to pause the execution of an `async` function until a promise is resolved. It can only be used inside an `async` function. When you `await` a promise, the function is paused at that line until the promise resolves, and then it continues execution.

```
async function fetchData() {

    let result = await someAsyncFunction();

    // Code here will execute after the promise is resolved

}
```

3. Handling Promises:
   When using `async` and `await`, you can handle promises in a more synchronous-looking manner, using `try...catch` blocks to handle errors.

```
async function fetchData() {
    try {
        let result = await someAsyncFunction();
        // Code here executes if the promise is resolved
    } catch (error) {
        // Code here executes if the promise is rejected
    }
}
```

4. Error Handling:
   Errors in `async` functions can be handled using traditional `try...catch` blocks. If an error occurs inside an `async` function and it is not caught, it will be propagated as a rejected promise.

```
async function fetchData() {
    try {
        let result = await someAsyncFunction();
        // Code here executes if the promise is resolved
    } catch (error) {
        // Code here executes if the promise is rejected
        console.error('Error:', error);
    }
}
```

Complete Example of Async Await

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-
scale=1.0">
<title>Async/Await Example</title>
</head>
<body>
<h1>Async/Await Example</h1>
<p id="result"></p>

<script>
// Simulating an asynchronous function that fetches data
function fetchData() {
    return new Promise((resolve, reject) => {
        // Simulating an asynchronous operation with setTimeout
        setTimeout(() => {
            const success = true; // Simulating success or
failure
            if (success) {
                resolve("Data fetched successfully!");
            } else {
                reject("Error fetching data!");
            }
        }, 2000); // Simulating a delay of 2 seconds
    });
}

// An async function that uses await to handle the asynchronous
operation
async function getData() {
    try {

        const data = await fetchData(); // Waiting for the
promise to resolve
        return data; // Returning data to the caller
    } catch (error) {
        throw error; // Propagating the error to the caller
    }
}

// Function to handle button click event
async function fetchDataAndDisplay() {
    try {
        document.getElementById("result").innerText = "Fetching
Data...........";
        const result = await getData(); // Waiting for getData()
to complete
        document.getElementById("result").innerText = result;
    } catch (error) {
        document.getElementById("result").innerText = "Error: " +
error;
    }
}
</script>

<button onclick="fetchDataAndDisplay()">Fetch Data</button>

</body>
</html>
```

Arrow Functions in Javascript

Arrow functions are a concise way to write anonymous functions in JavaScript. They provide a shorter syntax compared to traditional function expressions and offer implicit binding of `this`. Here's a breakdown of arrow functions:

Syntax:

- The basic syntax of an arrow function is:

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

- If there's only one parameter, you can omit the parentheses around it:

```
const singleParamFunction = parameter => {
    // function body
};
```

- If the function body consists of a single expression, you can omit the curly braces and the `return` keyword, and the expression will be implicitly returned:

```
const singleExpressionFunction = () => expression;
```

Examples:

1. Basic arrow function:

```
const greet = () => {

    console.log("Hello!");

};
```

2. Arrow function with parameters:

```
const add = (a, b) => {

    return a + b;

};
```

3. Arrow function with implicit return:

```
const square = x => x * x;
```

4. Arrow function in array method

   ```
   const numbers = [1, 2, 3, 4, 5];

   const squaredNumbers = numbers.map(number => number * number);
   ```

Implicit Binding of `this`:

In arrow functions, `this` is lexically scoped. This means that the value of `this` inside an arrow function is determined by the surrounding lexical context (i.e., the scope in which the arrow function is defined), and it does not have its own `this` value.

```
const obj = {

    name: "John",

    greet: function() {

        setTimeout(() => {

            console.log(`Hello, ${this.name}!`); // 'this' refers to 'obj'

        }, 1000);

    }

};

obj.greet(); // Output: Hello, John!
```

Advantages:

- Concise syntax leads to cleaner and more readable code.

- Lexical scoping of `this` eliminates the need for using `bind()`, `call()`, or `apply()` to maintain the context of `this`.

Limitations:

- Cannot be used as constructor functions (`new ArrowFunction()`).
- Do not have their own `arguments` object.

Fetch API in javascript

The Fetch API is a modern JavaScript interface for fetching resources (such as data or files) across the network. It provides a more powerful and flexible way to make HTTP requests compared to older techniques like XMLHttpRequest.

Here are some key points about the Fetch API:

1. **Asynchronous and Promise-Based: F**etch API is designed to work asynchronously, meaning that it doesn't block the execution of JavaScript while waiting for the response. It returns a Promise that resolves to the response to that request.
2. **Simpler Syntax:** The Fetch API provides a simpler and cleaner syntax compared to XMLHttpRequest, making it easier to read and write code for making HTTP requests.
3. **Supports a Wide Range of Data Formats**: Fetch allows you to easily fetch various types of data, including JSON, text, HTML, and even binary data.
4. **Modern Replacement for XMLHttpRequest:** While XMLHttpRequest is still widely used, the Fetch API is considered the modern replacement for it due to its improved features and ease of use.

```
<html>
<body>

<h1>JavaScript Fetch API</h1>
<h2>The fetch() Method</h2>
<p id="demo">Fetch a file to change this text.</p>

<script>
// Replace the URL with the published link of your Google Docs document
let data= "https://jsonplaceholder.typicode.com/posts/1";
```

```
fetch(data)
    .then(function(response) {
        if (response.ok) {
            return response.text();
        }
        throw new Error('Network response was not ok.');
    })
    .then(function(data) {
        document.getElementById("demo").innerHTML = data;
    })
    .catch(function(error) {
        console.error('Fetch error:', error);
        document.getElementById("demo").innerHTML = 'Error fetching data:
' + error.message;
    });
</script>

</body>
</html>
```

- It sets up an HTML structure with a heading (`<h1>` and `<h2>`) and a paragraph (`<p>`) element where the fetched data will be displayed.
- In the `<script>` tag, it defines a variable `data` containing the URL to fetch data from. In this case, it's the JSONPlaceholder API endpoint for retrieving a single post (`https://jsonplaceholder.typicode.com/posts/1`).
- It then uses the Fetch API to make an HTTP GET request to the specified URL.
- Upon receiving the response, it checks if the response is successful (status code between 200 and 299). If successful, it extracts the response body as text using the `response.text()` method.
- If the response is not successful, it throws an error.
- It then handles the fetched data in the second `.then()` block, where it updates the content of the HTML element with the ID 'demo' to display the fetched data.
- If any error occurs during the fetch operation, it is caught in the `.catch()` block, where it logs the error to the console and updates the content of the HTML element with an error message.