JAVASCRIPT INTRODUCTION

1. What is JavaScript? Explain the role of JavaScript in web development.

-> JavaScript is a high-level, interpreted programming language primarily used for adding interactivity and dynamic behavior to web pages.

- **Role of JavaScript in Web Development:**

=>Client-Side Interactivity – Enables interactive elements like dropdowns, sliders, and modals.

=>DOM Manipulation – Allows developers to dynamically update HTML and CSS.

=>Form Validation – Checks user input before submitting data to the server.

=>Asynchronous Operations – Fetches data from a server without reloading the page (AJAX, Fetch API).

=>Event Handling – Listens for user actions like clicks, keystrokes, and scrolls.

=>Animations & Effects – Enhances UI/UX with smooth animations.

=>Backend Development – Used with Node.js to build server-side applications.

2. How is JavaScript different from other programming languages like Python or Java?

->

Feature	JavaScript	Python	Java
Paradigm	Multi-paradigm (OOP, Functional, Event-driven)	Multi-paradigm (OOP, Procedural, Functional)	Object-Oriented
Execution	Runs in browsers and on servers (Node.js)	Interpreted, runs on various platforms	Compiled to bytecode, runs on JVM
Typing	Dynamically typed	Dynamically typed	Statically typed
Use Case	Web development, interactive UI, backend (Node.js)	Data science, automation, backend, AI	Enterprise applications, Android apps
Syntax	Uses {} for blocks, ; at the end of statements (optional)	Uses indentation for blocks	Uses {} for blocks, ; required
Performance	Fast due to JIT compilation	Slower than JS and Java	Faster than Python, slower than C++

3. Discuss the use of tag in HTML. How can you link an external JavaScript file to an HTML document? <!--EndFragment--> </body> </html>.

->

=> The <script> tag is used in HTML to include JavaScript code, either inline or as an external file.

1. Inline JavaScript Example:

```
<!DOCTYPE html>
<html>
<head>
<title>Inline JS Example</title>
</head>
<body>
<button onclick="alert('Hello, World!')">Click Me</button>
</body>
</html>
```

2. External JavaScript File:

=> To link an external JavaScript file, use the <script> tag with the src attribute.

```
<!DOCTYPE html>
<html>
<head>
<title>External JS Example</title>
<script src="script.js"></script>
</head>
<body>
<button id="myButton">Click Me</button>
</body>
</html>
```

Variables and Data Types

1. What are variables in JavaScript? How do you declare a variable using var, let,and const?

-> Variables in JavaScript are used to store data that can be manipulated and referenced throughout the program. There are three ways to declare variables:

1.var :–

```
var name = "Alice";
var name = "Bob";
name = "Charlie";
```

2.let :–

```
let age = 25;
age = 30; // Allowed
```

3.const :–

```
const PI = 3.14;
// PI = 3.1415; // Error: Assignment to a constant variable
```

2. Explain the different data types in JavaScript. Provide examples for each

-> JavaScript has primitive and non-primitive data types:

- P**rimitive Data Types :-**

Number–

```
let num = 42;
let price = 99.99;
```

String–

```
let message = "Hello, World!";
```

Boolean –

```
let isRaining = false;
```

Undefined–

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

Null–

```
let emptyValue = null;
```

Symbol–

```
let sym = Symbol("id");
```

BigInt–

```
let bigNumber = 9007199254740991n;
```

- N**on-Primitive Data Types :-**

Object–

```
let person = { name: "Alice", age: 25 };
```

Array–

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

3. What is the difference between undefined and null in JavaScript?

->

Feature	undefined	null
Definition	A variable that has been declared but not assigned a value	An intentional absence of value
Type	undefined (primitive)	object (primitive)
Usage	Default value for uninitialized variables	Explicitly assigned to represent "no value"
Example	let x; console.log(x); // undefined	let y = null; console.log(y); // null

JavaScript Operators

1. What are the different types of operators in JavaScript? Explain with examples.

- **Arithmetic operators**
- **Assignment operators**
- **Comparison operators**
- **Logical operators**

->

1. Arithmetic Operators

Operator	Description	Example
+	Addition	5 + 3
-	Subtraction	10 - 4
*	Multiplication	2 * 6
/	Division	8 / 2
%	Modulus (Remainder)	10 % 3
++	Increment	let a = 5; a++
--	Decrement	let b = 10; b--

Example:

```
let x = 10;

let y = 3;

console.log(x + y);

console.log(x % y);
```

---

2. Assignment Operators

Operator	Description	Example
=	Assign	x = 10
+=	Add and assign	x += 5
-=	Subtract and assign	x -= 3
*=	Multiply and assign	x *= 2
/=	Divide and assign	x /= 4
%=	Modulus and assign	x %= 3

Example:-

```
let a = 10;

a = a + 5;

console.log(a);
```

3. Comparison Operators

Operator	Description	Example	Result
==	Equal to (loose equality)	5 == '5'	true
===	Strict equal (checks value & type)	5 === '5'	false
!=	Not equal to	10 != 5	true
!==	Strict not equal	10 !== '10'	true
>	Greater than	10 > 5	true
<	Less than	5 < 10	true
>=	Greater than or equal	10 >= 10	true
<=	Less than or equal	5 <= 10	true

Example:

```
console.log(5 == '5'); // true

console.log(5 === '5'); // false

console.log(10 > 5); // true
```

4. Logical Operators

Operator	Description	Example	Result
&&	Logical AND	(5 > 2) && (10 > 8)	true
`		`	Logical OR
!	Logical NOT	!(5 > 2)	false

Example:

```
let a = true;
let b = false;

console.log(a && b);
console.log(a || b);
console.log(!a);
```

---

2. What is the difference between == and === in JavaScript?

->

- **== (Equality Operator):-**

Compares only values (type conversion happens automatically).

Example:

```
console.log(5 == '5');
console.log(0 == false);
```

- **=== (Strict Equality Operator):-**

Compares both value and data type.

Example:

```
console.log(5 === '5');
console.log(0 === false);
```

Control Flow (If-Else, Switch)

Question 1: What is control flow in JavaScript? Explain how if-else statements work with an example.

Control Flow: Control flow in JavaScript refers to the order in which individual statements, instructions, or function calls are executed or evaluated. By default, JavaScript code runs from top to bottom, but control structures like conditionals, loops, and function calls can change this flow.

If-Else Statements: An if-else statement is a control structure that allows you to execute certain code blocks based on whether a specified condition is true or false.

Syntax:

```
if (condition) {
   // code to execute if condition is true
} else {
   // code to execute if condition is false
}
```

Example:

```
let age = 18;
if (age >= 18) {
   console.log('You are an adult.');
} else {
   console.log('You are a minor.');
}
```

Output:

You are an adult.

In this example, since the condition age >= 18 is true, the code inside the if block runs.

Question 2: Describe how switch statements work in JavaScript. When should you use a switch statement instead of if-else?

Switch Statements: A switch statement evaluates an expression and matches its value against multiple possible cases. Each case has a block of code that executes if there is a match. You can also include a default case to handle situations where none of the specified cases match.

Syntax:

```
switch (expression) {
   case value1:
      // code to execute if expression === value1
      break;
   case value2:
      // code to execute if expression === value2
      break;
   default:
      // code to execute if none of the cases match
}
```

Example:

```
let day = 'Monday';


switch (day) {
   case 'Monday':
      console.log('Start of the work week.');
      break;
   case 'Friday':
```

```
        console.log('Almost the weekend!');
        break;
    default:
        console.log('Just another day.');
}
```

Output:

Start of the work week.

Loops (For, While, Do-While)

Question 1: Types of Loops in JavaScript

In JavaScript, loops are used to execute a block of code multiple times, depending on a condition. There are three main types of loops:

1. For Loop
   A for loop is typically used when the number of iterations is known beforehand. It has three main components: initialization, condition, and iteration.

Example:

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

Explanation:

- let i = 0: Initialization (variable i starts at 0).
- i < 5: Condition (loop continues as long as i is less than 5).

- i++: Iteration (increment i by 1 each time).

2 While Loop
A while loop runs as long as the given condition is true. The condition is evaluated before each iteration.

Example:

```
let i = 0;

while (i < 5) {

   console.log(i);

   i++;

}
```

Explanation:

- The loop continues as long as i < 5. The condition is checked before executing the loop body.
- i++ increments the counter after each iteration.

3 Do-While Loop
A do-while loop is similar to the while loop, but it guarantees that the loop will run at least once, because the condition is checked after the loop body.

Example:

```
let i = 0;

do {

   console.log(i);

   i++;

} while (i < 5);
```

Question 2: Difference Between a While Loop and a Do-While Loop

- **While Loop**:
  In a while loop, the condition is checked **before** the code block is executed. If the condition is false from the start, the loop will not execute at all.

Example (does not run if i >= 5 initially):

```
let i = 5;

while (i < 5) {

  console.log(i);

  i++;

}
```

- **Do-While Loop**:
  In a do-while loop, the code block is executed **once** before the condition is checked. Even if the condition is false, the loop will run at least once.
- **Example** (runs at least once even if i >= 5 initially):

```
let i = 5;

do {

  console.log(i);

  i++;

} while (i < 5);
```

Functions

Question 1: What are Functions in JavaScript? Explain the Syntax for Declaring and Calling a Function.

In JavaScript, a **function** is a block of reusable code that can be executed when called. Functions are used to perform specific tasks or calculate values. They can accept input (parameters) and return a result (return value).

Syntax for Declaring a Function:

```
function functionName(parameters) {
   // code to be executed
}
```

- function: Keyword used to define a function.
- functionName: The name of the function.
- parameters: Optional values passed to the function (also called arguments).
- {}: Code block that defines what the function does.

Example of Declaring a Function:

```
function greet(name) {
   console.log("Hello, " + name);
}
```

Syntax for Calling a Function:

```
functionName(arguments);
```

Example of Calling a Function:

```
greet("Alice");  // Output: Hello, Alice
```

Question 2: What is the Difference Between a Function Declaration and a Function Expression?

Function Declaration: A function declaration defines a function with the function keyword. It is hoisted, meaning the function can be called before it is defined in the code.

Example:

```
function greet() {
  console.log("Hello!");
}
greet(); // Output: Hello!
```

- **Hoisting**: Function declarations are hoisted, meaning they are available for use before they are defined in the code.

Function Expression: A function expression creates a function and assigns it to a variable. Function expressions are **not hoisted**, so they must be defined before being called.

Example:

```
const greet = function() {
  console.log("Hello!");
};
greet(); // Output: Hello!
```

Question 3: Discuss the Concept of Parameters and Return Values in Functions.

- **Parameters**:
  Parameters are the named variables passed to a function to provide input. These values are used inside the function to perform actions or calculations.
- **Example:**

```
function add(a, b) {
   return a + b;
}
```

- **Return Values**:
  A return value is the result a function provides after execution. The return statement is used to return a value from a function.
- **Example:**

```
function multiply(x, y) {
   return x * y;
}
let result = multiply(2, 3);  // result = 6
```

Arrays

Question 1: What is an Array in JavaScript? How Do You Declare and Initialize an Array?

An **array** in JavaScript is a special variable that can hold more than one value at a time. Arrays are used to store multiple values in a single variable. These values can be of any data type, such as numbers, strings, or even other arrays.

Declaring and Initializing an Array:

- You can declare an array using square brackets [].
- Arrays are zero-indexed, meaning the first element is at index 0, the second at index 1, and so on.

Syntax to Declare and Initialize an Array:

```
let arrayName = [element1, element2, element3];
```

Example:

```
let fruits = ["Apple", "Banana", "Orange"];

console.log(fruits[0]);  // Output: Apple
```

You can also create an empty array and add elements later:

```
let numbers = [];

numbers.push(10);

numbers.push(20);

console.log(numbers);  // Output: [10, 20]
```

Question 2: Explain the Methods push(), pop(), shift(), and unshift() Used in Arrays

1. **push()**
   The push() method adds one or more elements to the **end** of an array and returns the new length of the array.

Example:

```
let fruits = ["Apple", "Banana"];

fruits.push("Orange");

console.log(fruits);  // Output: ["Apple", "Banana", "Orange"]
```

2 pop()

The pop() method removes the **last** element from an array and returns that element. It changes the length of the array.

Example:

```
let fruits = ["Apple", "Banana", "Orange"];
```

```
let lastFruit = fruits.pop();

console.log(lastFruit);  // Output: Orange

console.log(fruits);     // Output: ["Apple", "Banana"]
```

3 shift()

The shift() method removes the **first** element from an array and returns that element. It also shifts all other elements to a lower index.

Example:

```
let fruits = ["Apple", "Banana", "Orange"];

let firstFruit = fruits.shift();

console.log(firstFruit);  // Output: Apple

console.log(fruits);      // Output: ["Banana", "Orange"]
```

4 unshift()

The unshift() method adds one or more elements to the **beginning** of an array and returns the new length of the array.

Example:

```
let fruits = ["Banana", "Orange"];

fruits.unshift("Apple");

console.log(fruits);  // Output: ["Apple", "Banana", "Orange"]
```

Objects

Question 1: What is an Object in JavaScript? How Are Objects Different from Arrays?

An **object** in JavaScript is a collection of key-value pairs where each key (also called a property) is associated with a value. The values can be of any data type, including other objects or arrays. Objects are typically used to represent and store structured data, such as information about a person or a car.

Example of an Object:

```
let person = {
   name: "John",
   age: 30,
   city: "New York"
};
```

Key Differences Between Objects and Arrays:

- **Arrays** are ordered collections of values, where each value is accessed by its **index** (numeric position) starting from 0.
- **Objects** are unordered collections of key-value pairs, where each value is accessed by a **key** (string or symbol).

Example of an Array:

```
let fruits = ["Apple", "Banana", "Orange"];
console.log(fruits[0]);
```

Question 2: Explain How to Access and Update Object Properties Using Dot Notation and Bracket Notation

1. **Dot Notation**:

Dot notation is the most common way to access or update object properties. You access the property directly by writing the object name followed by a dot (.) and the property name.

Example of Accessing Properties Using Dot Notation:

```
let person = {
   name: "John",
   age: 30,
   city: "New York"
};
console.log(person.name);  // Output: John
console.log(person.age);   // Output: 30
```

Example of Updating Properties Using Dot Notation:

```
person.age = 31;  // Updating the age
console.log(person.age);  // Output: 31
```

2. **Bracket Notation**:

Bracket notation is useful when the property name contains special characters, spaces, or when the property name is stored in a variable.

Example of Accessing Properties Using Bracket Notation:

```
let person = {
   name: "John",
   age: 30,
   "favorite color": "blue"
};
console.log(person["favorite color"]);  // Output: blue
```

Example of Updating Properties Using Bracket Notation:

```
person["age"] = 32;  // Updating age using bracket notation
```

```
console.log(person.age);  // Output: 32
```

Bracket notation also allows access to properties using variables:

```
let property = "city";

console.log(person[property]);
```

JavaScript Events

Question 1: What are JavaScript Events? Explain the Role of Event Listeners.

JavaScript events are actions or occurrences that happen in the system you are interacting with, often triggered by user actions, such as clicking a button, submitting a form, or pressing a key. Events allow us to respond to these actions by executing JavaScript code. Common types of events include clicks, key presses, mouse movements, form submissions, etc.

Example of Events:

- click: Triggered when an element is clicked.
- keydown: Triggered when a key is pressed.
- submit: Triggered when a form is submitted.

Event Listeners:

An **event listener** is a JavaScript function that waits for a specific event to occur. When the event occurs, the event listener runs the associated code (callback function). Event listeners allow us to respond to user interactions with the web page.

Role of Event Listeners:

- They "listen" for specific events (like a click or keypress).

- They allow you to define code to be executed when the event occurs.
- You can attach multiple event listeners to a single element or event type.

Question 2: How Does the addEventListener() Method Work in JavaScript? Provide an Example.

The addEventListener() method is used to attach an event listener to a DOM element. It listens for an event to occur and then calls a specified function (callback) when the event happens.

Syntax:

element.addEventListener(event, function, useCapture);

- **event**: The type of event to listen for (e.g., "click", "keydown", etc.).
- **function**: The function to run when the event occurs.
- **useCapture** (optional): A boolean value that determines whether the event should be captured or bubbled. Default is false (bubbling phase).

Example:

Let's add a click event listener to a button.

HTML:

<button id="myButton">Click Me!</button>

JavaScript:

let button = document.getElementById("myButton");

// Add an event listener to the button

button.addEventListener("click", function() {

```
    alert("Button was clicked!");
});
```

In this example:

1. The addEventListener method is used to attach a click event listener to the button.
2. When the button is clicked, the anonymous function is called, which triggers an alert displaying "Button was clicked!".

More Examples of addEventListener():

1. **Click Event:**

```
let button = document.getElementById("myButton");
button.addEventListener("click", function() {
    console.log("Button was clicked!");
});
```

1. **Keydown Event:**

```
document.addEventListener("keydown", function(event) {
    console.log("Key pressed: " + event.key);
});
```

1. **Mouseover Event:**

```
let div = document.getElementById("myDiv");
div.addEventListener("mouseover", function() {
    div.style.backgroundColor = "yellow";
});
```

DOM Manipulation

Question 1: What is the DOM (Document Object Model) in JavaScript? How Does JavaScript Interact with the DOM?

The **DOM (Document Object Model)** is a programming interface for web documents. It represents the structure of an HTML document as a tree of objects, where each object corresponds to a part of the web page (such as elements, attributes, and text). The DOM allows JavaScript to interact with the HTML structure, enabling dynamic changes to the page content, style, and structure.

How JavaScript Interacts with the DOM:

- **Accessing Elements**: JavaScript can access HTML elements by using methods like getElementById() or querySelector(). Once an element is accessed, JavaScript can manipulate it, such as changing its text, style, or attributes.
- **Modifying Content**: JavaScript can change the content of an element using properties like .innerHTML or .textContent.
- **Handling Events**: JavaScript can listen for user interactions (like clicks or keystrokes) by adding event listeners to DOM elements.
- **Changing Styles**: JavaScript can modify the style of HTML elements using the .style property.
- **Adding/Removing Elements**: JavaScript can dynamically add or remove HTML elements using methods like .appendChild(), .removeChild(), or .createElement().

Example of interacting with the DOM:

```
// Accessing the DOM

let heading = document.getElementById("myHeading");

// Changing the text content
```

```
heading.textContent = "New Heading";

// Changing the style

heading.style.color = "blue";
```

Question 2: Explain the Methods getElementById(), getElementsByClassName(), and querySelector() Used to Select Elements from the DOM

1. **getElementById()**:

The getElementById() method is used to select an element by its unique id attribute. Since id values are unique within a page, this method will return a single element.

Syntax:

```
let element = document.getElementById("id");
```

Example:

```
let myElement = document.getElementById("myHeading");

console.log(myElement.textContent);  // Output: The text inside the element with id "myHeading"
```

2. **getElementsByClassName()**:

The getElementsByClassName() method returns a **live HTMLCollection** of all elements that have a specific class name. Since the returned object is a collection, you can loop through it to access individual elements.

Syntax:

```
let elements = document.getElementsByClassName("className");
```

Example:

```
let items = document.getElementsByClassName("list-item");
```

```
for (let i = 0; i < items.length; i++) {
    console.log(items[i].textContent);
}
```

This will log the text content of all elements with the class name "list-item".

3. **querySelector()**:

The querySelector() method allows you to select the first element that matches a CSS selector. It is more flexible than getElementById() and getElementsByClassName() because it supports any valid CSS selector.

Syntax:

```
let element = document.querySelector("selector");
```

Example:

```
let heading = document.querySelector("#myHeading");
console.log(heading.textContent);  // Output: The text inside the element with id "myHeading"
```

You can use CSS selectors for selecting based on id, class, attributes, or more complex combinations.

Example of using querySelector() with a class selector:

```
let item = document.querySelector(".list-item");
console.log(item.textContent);  // Output: The text of the first element with class "list-item.
```

JavaScript Timing Events (setTimeout, setInterval)

Question 1: Explain the setTimeout() and setInterval() Functions in JavaScript. How Are They Used for Timing Events?

In JavaScript, both setTimeout() and setInterval() are functions that allow you to control the timing of actions. These functions are often used for events that need to be delayed or repeated after a certain time.

1. **setTimeout()**:

The setTimeout() function is used to delay the execution of a single function or piece of code by a specified number of milliseconds. It runs the code once after the time delay has passed.

Syntax:

setTimeout(function, delayInMilliseconds);

- **function**: The function to be executed after the delay.
- **delayInMilliseconds**: The delay before the function is executed (in milliseconds).

Example:

```
setTimeout(function() {
    console.log("This message is delayed by 3 seconds!");
}, 3000);  // 3000 milliseconds = 3 seconds
```

This will print the message to the console after 3 seconds.

2. **setInterval()**:

The setInterval() function is used to repeatedly execute a function or code snippet at specified intervals (in milliseconds). The function will keep running every given interval until you stop it using clearInterval().

Syntax:

setInterval(function, intervalInMilliseconds);

- **function**: The function to be executed repeatedly.
- **intervalInMilliseconds**: The interval time between each execution (in milliseconds).

Example:

```
let counter = 0;

let intervalId = setInterval(function() {
    counter++;
    console.log("Counter:", counter);
    if (counter === 5) {
        clearInterval(intervalId);  // Stops the interval after it runs 5 times
    }
}, 1000);  // 1000 milliseconds = 1 second
```

Question 2: Provide an Example of How to Use setTimeout() to Delay an Action by 2 Seconds.

Here's an example of using setTimeout() to delay an action by 2 seconds (2000 milliseconds):

```
console.log("Action will be delayed by 2 seconds...");

setTimeout(function() {
    console.log("This message is displayed after a 2-second delay.");
}, 2000);  // 2000 milliseconds = 2 seconds
```

In this example:

1. The message "Action will be delayed by 2 seconds..." will be displayed immediately.
2. After 2 seconds, the message "This message is displayed after a 2-second delay." will be printed to the console.

JavaScript Error Handling

Question 1: What is Error Handling in JavaScript? Explain the try, catch, and finally Blocks with an Example.

Error handling in JavaScript is the process of catching and managing errors that may occur during the execution of a program. It allows developers to anticipate potential issues, handle them gracefully, and provide meaningful feedback to users or developers.

1. **try Block**:

The try block contains the code that might throw an error. If an error occurs within this block, it is caught by the catch block.

2. **catch Block**:

The catch block is used to catch and handle errors that occur in the try block. If an error is thrown, execution jumps to the catch block, where you can log the error or handle it appropriately.

3. **finally Block**:

The finally block is optional and, if provided, will always execute regardless of whether an error occurred or not. It is typically used for cleanup tasks, such as closing file handlers or releasing resources.

Syntax:

try {

```
    // Code that may throw an error
} catch (error) {
    // Code to handle the error
} finally {
    // Code that will run regardless of whether there was an error or not
}
```

Example:

```
try {
    let x = 10;
    let y = 0;
    let result = x / y;  // This will cause a division by zero error
    console.log("Result: " + result);
} catch (error) {
    console.log("An error occurred: " + error.message);  // Catches and logs the error
} finally {
    console.log("This will always run, regardless of error.");
}
```

In this example:

- The try block attempts to divide x by y, which results in Infinity (a special value in JavaScript).
- The catch block catches the error and logs it.
- The finally block runs regardless of whether the error was caught or not.

Output:

An error occurred: Infinity

This will always run, regardless of error.

Question 2: Why is Error Handling Important in JavaScript Applications?

Error handling is crucial for the following reasons:

1. **Prevents Application Crashes**: Without error handling, a small mistake or unexpected input can cause the entire application to crash. By handling errors properly, you can prevent such crashes and ensure the program continues to run smoothly.
2. **Improves User Experience**: If something goes wrong, error handling allows you to provide helpful messages to the user, rather than the application failing silently or showing an unhelpful error message. This enhances the user experience and guides the user toward resolving issues.
3. **Provides Debugging Information**: When an error occurs, the catch block allows developers to log specific information about the error. This helps in debugging and understanding why the error occurred, especially in production environments.
4. **Ensures Clean-Up and Resource Management**: The finally block ensures that resources (like database connections, files, or timers) are always cleaned up, regardless of whether an error occurred or not. This avoids memory leaks or other resource-related issues.
5. **Maintain Application Flow**: Proper error handling allows the application to continue running even if one part of it encounters an issue. For instance, if a network request fails, the user can still interact with other parts of the app, which would be impossible without error handling.
6. **Handles External Failures Gracefully**: In real-world applications, many operations rely on external resources, such as APIs or

databases. If these external resources fail, error handling allows the application to handle those failures and take appropriate action, such as retrying the request or informing the user about the issue.

Example of Real-World Error Handling in an API Call:

```
try {
    fetch("https://api.example.com/data")
        .then(response => response.json())
        .then(data => console.log(data))
        .catch(error => {
            console.log("API request failed: " + error.message);
        });
} catch (error) {
    console.log("An unexpected error occurred: " + error.message);
}
```