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

JavaScript is used to create client-side dynamic pages. - JavaScript is a lightweight, cross-platform, and interpreted compiled programming language which is also known as the scripting language for webpages.

Role in Web Development

1. Interactivity and User Experience

JavaScript is primarily responsible for adding interactivity to web pages.

2. DOM Manipulation

The Document Object Model (DOM) is a programming interface that represents the structure of an HTML document as a tree of objects. JavaScript enables developers to manipulate the DOM, which means they can programmatically modify the content, structure, and style of a web page.

3. Asynchronous Programming

JavaScript supports asynchronous programming, allowing developers to execute tasks without blocking the main thread. This is crucial for tasks like fetching data from servers, handling user inputs, and performing animations

4. Web APIs

Web browsers expose a wide range of APIs (Application Programming Interfaces) through JavaScript. These APIs provide access to various functionalities, such as manipulating browser history, accessing device hardware (e.g., camera and microphone), and performing geolocation services.

5. Frameworks and Libraries

JavaScript has a thriving ecosystem of frameworks and libraries that streamline web development. Frameworks like React, Angular, and Vue.js provide structured ways to build complex user interfaces, while libraries like jQuery simplify common tasks like DOM manipulation and AJAX requests.

6. Server-Side Development

While JavaScript is primarily known for its client-side capabilities, it has also expanded into server-side development.

2: How is JavaScript different from other programming languages like Python orJava?

JavaScript

- Purpose: Primarily used for web development to make web pages interactive.
- Syntax: Uses curly braces for blocks of code { } and semicolons ; to end statements. It's dynamically typed.
- Main Use Cases: Client-side scripting, interactive web applications, front-end and back-end development.

Python

- Purpose: Known for its simplicity and readability, used for a wide range of applications including web development, data analysis, AI, and more.
- Syntax: Uses indentation to define code blocks and does not require semicolons. It's dynamically typed.
- Main Use Cases: Data science, machine learning, web development, automation scripts, scientific computing.

Java

- Purpose: Designed to be platform-independent at both the source and binary levels, often used for enterprise-level applications.
- Syntax: Similar to C and C++, uses curly braces for code blocks { } and semicolons ; to end statements. It's statically typed.
- Main Use Cases: Enterprise applications, Android app development, large-scale systems.

3: Discuss the use of <script> tag in HTML. How can you link an external JavaScript file to an HTML document?

The <script> tag in HTML is used to embed or reference JavaScript code within an HTML document. JavaScript can be included in two main ways: directly within the HTML file or by linking to an external JavaScript file.

1. **Link the External JavaScript File in Your HTML:** In HTML file, use the <script> tag with the src attribute to link to the external JavaScript file html

```
<!DOCTYPE html>

<html>

<head>

   <title>External JavaScript File</title>

</head>

<body>

   <h1 id="message">Hello World</h1>

  <script src="script.js"></script>

</body>

</html>
```

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

variables are used to store and manipulate data. Variables act as containers for storing data values.

var:

- The var keyword was traditionally used to declare variables in JavaScript.
- Variables declared with var are function-scoped or globally scoped if not declared within a function.

- **Example:**
- <script>
- var a = 10;
- document.writeln(a)
- var a = 14;
- document.writeln(a)
- </script>

let:

- The let keyword is used to declare block-scoped variables.
- Variables declared with let are only accessible within the block (e.g., {}) they are defined in.
- let allows for reassignment of variables.
- **Example:**
- <script>
- let b = 13;
- document.writeln(b)
- b = 55;
- document.writeln(b)
- </script>
- **`const`:**
- The `const` keyword is used to declare block-scoped, read-only constants.
- Variables declared with `const` must be initialized at the time of declaration and cannot be reassigned.

- **Example:**
- <script>
- const c = 12;
- document.writeln(c)
- c = 14;
- document.write(c)
- </script>

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

There is two type of datatype in javascript.

(1) Primitive datatype

(2) Non-primitive(reference) datatype

Primitive Data Types

1. **Number** Used for numeric values, both integers and floating-point numbers.

   let age = 25;   / Integer

   let price = 99.99; / Floating-point

2. **String** Represents textual data, enclosed in single or double quotes.

   let name = "Riya";

   let greeting = 'Hello, world!';

3. **Boolean** Represents a logical entity, with only two values: true or false.

   let is Active = true;

   let is LoggedIn = false;

4. **Undefined** A variable that has been declared but not assigned a value.

   let city;

   console.log(city);

   / Output: undefined

5. **Null** Represents an intentional absence of value.

   ```
   let result = null;
   ```

   Non-Primitive (Reference) Data Types

   These are mutable and can store collections of values.

1. **Object** A collection of key-value pairs.

   ```
   let user = {
      name: "Riya",
      age: 25
   };
   ```

2. **Array** Represent group of Similar values.

   ```
   let fruits = ["apple", "banana", "cherry"];
   ```

6: What is the difference between undefined and null in JavaScript?

Undefined	**Null**
Indicates a variable that has been declared but not yet assigned a value.	Represents the intentional absence of any object value.
Primitive data type in JavaScript.	Primitive data type in JavaScript.
Automatically assigned to variables that are declared but not initialized.	Must be explicitly assigned to a variable.
undefined == null // true (loose equality considers them equal).	undefined == null // true (loose equality considers them equal).

Undefined	Null
undefined === null // false (they are not strictly equal in type).	null === undefined // false (they are not strictly equal in type).

7: What are the different types of operators in JavaScript? Explain with examples. • Arithmetic operators • Assignment operators • Comparison operators • Logical operators

- Operator means which is operate data.

Arithmetic operators

These are used for mathematical calculations. Common arithmetic operators include +, -, *, /, % (modulus), ++ (increment), and -- (decrement).

Example:

```
let a = 10;

let b = 3;

console.log(a + b);  / Addition: 13

console.log(a - b);  / Subtraction: 7

console.log(a * b);  / Multiplication: 30

console.log(a / b);  / Division: 3.3333...

console.log(a % b);  / Modulus (remainder): 1


a++;  / Increment a: becomes 11

b--;  / decrement b:becomes 2
```

2. Assignment Operators

These are used to assign or update the value of a variable. Common ones include =, +=, -=, *=, /=, and %=.

Examples:

```
let x = 5;

x += 3;  /Equivalent to: x = x + 3; x becomes 8

x -= 2;  / Equivalent to: x = x - 2; x becomes 6

x *= 4; / Equivalent to: x = x * 4; x becomes 24

x /= 3;  / Equivalent to: x = x / 3; x becomes 8

x %= 5;  / Equivalent to: x = x % 5; x becomes 3
```

3. Comparison Operators

These compare two values and return a Boolean (true or false). Examples include == (equal to), === (strict equal to), != (not equal to), !== (strict not equal to), >, <, >=, and <=.

Examples:

```
let p = 10, q = '10';

console.log(p == q);   // true (values are equal, type is ignored)

console.log(p === q);  // false (strict equality: value and type must match)

console.log(p != q);   // false (values are equal)

console.log(p !== q);  // true (strict inequality: type doesn't match)

console.log(p > 5);    // true

console.log(p <= 10);  // true
```

4. Logical Operators

Used to combine or invert Boolean values. The main logical operators are && (AND), || (OR), and ! (NOT).

Examples:

```
let r = true;

let s = false;

console.log(r && s);  // false (both must be true for AND)

console.log(r || s);  // true (only one needs to be true for OR)

console.log(!r);      // false (NOT inverts the value)
```

8: What is the difference between == and === in JavaScript?

Aspect	**== (Equality)**	**=== (Strict Equality)**
Comparison	Checks if the values are equal, ignoring their types.	Checks if the values are equal and of the same type.
Examples (True)	5 == '5' (true, string is converted to number)	5 === 5 (true, both value and type match)
Examples (False)	5 == '5abc' (false, string cannot be converted to number)	5 === '5' (false, type mismatch: number vs string)
Use Case	Use when type conversion is acceptable, but be cautious.	Use when strict comparison is required—generally safer.

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

Control flow in JavaScript determines the order in which code is executed., JavaScript executes code line by line, from top to bottom.

How if-else Statements Work

The if-else statement is a control structure used to execute code blocks conditionally. The if block executes a code section only if a specific condition evaluates to true.

1. The else block runs when the if condition evaluates to false.
2. You can also add multiple conditions using else if to handle various cases.

Syntax

```
if (condition) {
    Code to run if the condition is true
} else {
    Code to run if the condition is false
}
```

Example

Here's an example of using an if-else statement to check if a number is positive, negative, or zero:

```
let number = 5;
if (number > 0) {
   console.log("The number is positive.");
} else if (number < 0) {
   console.log("The number is negative.");
} else {
    console.log("The number is zero.");
}
```

Output (for number = 5):

```
The number is positive.
```

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

A switch statement allows you to execute different blocks of code based on the value of an expression. It's especially useful when dealing with multiple possible conditions for a single variable or expression.

Syntax

```
switch (expression) {
    case value1:
        // Code to run if expression === value1
        break;
    case value2:
        // Code to run if expression === value2
        break;
    default:
        // Code to run if no cases match
}
```

When to Use a switch Statement Instead of if-else

1. **Multiple Values for a Single Variable**: Use switch when comparing a single variable or expression against multiple values. It makes the code more readable and concise.

Example:

```
 let fruit = "apple";
switch (fruit) {
  case "apple":
  case "banana":
      console.log("This is a popular fruit!");
      break;
  default:
      console.log("This fruit is less common.");
}
```

2. **Avoiding Long if-else Chains**: When there are many conditions for a single variable, switch statements are cleaner and easier to understand than a long chain of if-else statements.

```
if (day === "Monday") {
   console.log("Start of the work week!");
} else if (day === "Tuesday" || day === "Wednesday") {
   console.log("It's midweek hustle time!");
} else if (day === "Thursday") {
   console.log("Almost Friday!");
} else if (day === "Friday") {
   console.log("Weekend is almost here!");
} else {
   console.log("It's the weekend! Relax and enjoy!");
}
```

3. **Code Clarity**: If you have multiple conditions for unrelated variables or complex logic, it's better to stick with if-else instead of switch.

11: Explain the different types of loops in JavaScript (for, while, do-while). Provide a basic example of each.

loops are used to repeatedly execute a block of code as long as a specified condition is true. Here are the three main types of loops with basic examples:

1. for Loop

The for loop is used when the number of iterations is known in advance. It consists of three parts: initialization, condition, and increment/decrement.

Syntax:

```
for (initialization; condition; update) {
   // Code to execute
}
```

Example:

```
for (let i = 1; i <= 5; i++) {
   console.log("Iteration number: " + i);
}
```

Output:

```
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
```

2. while Loop

The while loop is used when the number of iterations is not known before hand, and the loop continues as long as the condition is true.

Syntax:

```
while (condition) {
   // Code to execute
}
```

Example:

```
let count = 1;
while (count <= 3) {
   console.log("Count is: " + count);
   count++;
}
```

Output:

```
Count is: 1
Count is: 2
Count is: 3
```

3. do-while Loop

The do-while loop is similar to the while loop, but it executes the code block at least once, regardless of the condition, because the condition is checked *after* the execution.

Syntax:

```
   do {
   // Code to execute
```

```
} while (condition);
```

Example:

```
let num = 0;
do {
  console.log("The number is: " + num);
  num++;
} while (num < 3);
```

Output:

```
The number is: 0
The number is: 1
The number is: 2
```

12: What is the difference between a while loop and a do-while loop?

Feature	**while Loop**	**do-while Loop**
Condition Check	Checked before the loop body runs	Checked after the loop body runs
Execution Guarantee	May not execute if the condition is false	Executes the loop body at least once
Use Case	Used when you want to run the loop only if the condition is true at the start	Used when the loop body must run at least once, regardless of the initial condition

13: What are functions in JavaScript? Explain the syntax for declaring and calling a function.

a function is a block of reusable code that performs a specific task. Functions help in modularizing code, making it easier to read, debug, and reuse.

Declaring a Function

A function can be declared using the function keyword, followed by:

1. The function name (optional for anonymous functions).
2. Parentheses () which may include parameters.
3. Curly braces {} enclosing the code block (function body).

Syntax:

```
function functionName(parameters) {
  // Code block (function body)
  // Perform specific tasks
}
```

Example:

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

In this example:

- greet is the function name.
- name is a parameter that the function expects when called.

Calling a Function

You invoke a function by using its name, followed by parentheses (). You can also pass values (arguments) inside the parentheses if the function expects parameters.

Syntax:

functionName(arguments);

Example:

greet("Riya");

// Output: Hello, Riya!

14: What is the difference between a function declaration and a function expression?

Feature	**Function Declaration**	**Function Expression**
Definition	Defines a function with a specific name.	Assigns a function (can be named or anonymous) to a variable.
Syntax	function myFunction() { ... }	const myFunction = function() { ... };
Hoisting	Hoisted to the top of the scope, so it can be called before it's defined.	Not hoisted; must be defined before it's called.
Name Requirement	Must have a name.	Can be anonymous or named.
Example	javascript function greet() { console.log("Hello!"); }	javascript const greet = function() { console.log("Hello!"); };

15: Discuss the concept of parameters and return values in functions?.

Parameters

Parameters act as placeholders for the values (called **arguments**) that a function takes when it is called. They enable functions to work with dynamic input.

Defining Parameters: Parameters are declared inside the parentheses () of the function definition

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

1. **Passing Arguments**: When calling a function, you pass arguments (actual values) corresponding to the parameters.

```
greet("Riya");
// Output: Hello, Riya!
```

2. **Multiple Parameters**: Functions can take multiple parameters, separated by commas.

```
function add(a, b) {
  console.log(a + b);
}
add(5, 3);
// Output: 8
```

3. **Default Parameters**: JavaScript allows setting **default values** for parameters, used when no argument is provide.

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

```
// Output: Hello, Guest!
```

Return Values

The **return value** is what a function sends back as its output when it is called. This allows the result of a function to be used elsewhere in the code.

1. **Returning Values**: A function uses the return keyword to send back a value.

```
function square(num) {
   return num * num;
}
let result = square(4);
console.log(result);
// Output: 16
```

2. **Without return**: If return is not used, the function returns undefined by default.

```
function sayHi() {
   console.log("Hi!");
}
let message = sayHi();
console.log(message);
// Output: Hi! followed by undefined
```

3. **Chaining**: Functions with return values can be used in expressions or as arguments in other functions.

```
function double(num) {
   return num * 2;
}
console.log(double(square(3)));
// Output: 18
```

16: What is an array in JavaScript? How do you declare and initialize an array?

An array in JavaScript is a data structure that allows you to store multiple values in a single variable. Arrays can hold a collection of values, which can be of any data type—numbers, strings, objects, or even other arrays.

Declaring and Initializing an Array

1. Using Array Literal Syntax

This is the most common and straightforward way to create an array. You use square brackets [] and include the elements (values) separated by commas.

```
let fruits = ["apple", "banana", "mango"];
```

- Here, fruits is an array containing three string elements.

2. Using the new Array() Constructor

You can also create an array using the Array constructor. However, this is less commonly used.

```
let numbers = new Array(1, 2, 3, 4, 5);
```

- This creates an array numbers containing five elements.

3. Empty Arrays

You can declare an array and initialize it as empty. You can then add elements later.

```
let emptyArray = [];

emptyArray[0] = "firstElement";
```

17: Explain the methods push(), pop(), shift(), and unshift() used in arrays.

These four methods—push(), pop(), shift(), and unshift()—are used to manipulate arrays in JavaScript.

1. push()

- Purpose: Adds one or more elements to the end of an array.
- Returns: The new length of the array.
- Example:

```
let fruits = ["apple", "banana"];
fruits.push("mango");
console.log(fruits);
// Output: ["apple", "banana", "mango"]
```

2. pop()

- Purpose: Removes the last element of an array.
- Returns: The element that was removed.
- Example:

```
let fruits = ["apple", "banana", "mango"];
let lastFruit = fruits.pop();
console.log(fruits);
// Output: ["apple", "banana"]
console.log(lastFruit);
// Output: "mango"
```

3. shift()

- Purpose: Removes the first element of an array.
- Returns: The element that was removed.
- Example:

```
let fruits = ["apple", "banana", "mango"];
let firstFruit = fruits.shift();

console.log(fruits);

// Output: ["banana", "mango"]

console.log(firstFruit);

// Output: "apple"
```

4. unshift()

- Purpose: Adds one or more elements to the beginning of an array.
- Returns: The new length of the array.
- Example:

```
let fruits = ["banana", "mango"];

fruits.unshift("apple");

console.log(fruits);

// Output: ["apple", "banana", "mango"]
```

18: What is an object in JavaScript? How are objects different from arrays?

In JavaScript, an object is a collection of key-value pairs used to store and manage data.

Creating an Object

You can create an object using object literal syntax {} or the new Object() constructor.

Syntax:

```
const objectName = {

   key1: value1,

   key2: value2,

   key3: value3,
};
```

Example:

```
const person = {
   name: "Riya",
   age: 25,
   greet: function() {
      console.log("Hello, " + this.name);
   },
};
person.greet();
// Output: Hello, Riya
```

How are Objects Different from Arrays?

Feature	**Objects**	**Arrays**
Definition	A collection of key-value pairs.	A collection of ordered values.
Structure	Organized as properties and values.	Organized as a list of elements (indexed).
Access	Accessed via keys: object.key or object["key"].	Accessed via indices: array[0].
Data Association	Best for representing entities with properties.	Best for managing ordered, sequential data.
Iteration	Use for...in or Object.keys() for iteration.	Use loops like for, for...of, or array methods like forEach().

19: Explain how to access and update object properties using dot notation and bracket notation.

Dot Notation

- How it Works: You use a period (.) followed by the property name.
- When to Use: When the property name is a valid JavaScript identifier (e.g., no spaces, special characters, or numbers as the first character).

Example:

Accessing a property:

```
const person = { name: "Riya", age: 25 };

console.log(person.name);

// Output: Riya
```

Updating a property:

```
person.age = 26;

console.log(person.age);

// Output: 26
```

Bracket Notation

- How it Works: You use square brackets [] and put the property name as a string or a variable inside the brackets.
- When to Use: When the property name contains special characters, spaces, or when you want to dynamically access a property.

Example:

Accessing a property:

```
const person = { "first name": "Riya", age: 25 };

console.log(person["first name"]);
```

```
// Output: Riya
```

Updating a property:

```
person["age"] = 26;

console.log(person["age"]);

// Output: 26
```

Example with Dynamic Access

```
let property = "name";

console.log(person[property]);

// Output: Riya
```

20: What are JavaScript events? Explain the role of event listeners

The change in the state of an object is known as an Event. - Js react over these events and allow the execution.

An event listener is a method in JavaScript that waits for a specific event to happen on an element, and then executes a callback function (a function designed to respond to the event).

Role of Event Listeners:

1. **Listening for Events**: It allows JavaScript to "listen" for specific events (like click, keydown, submit, etc.).
2. **Handling Events**: It connects the event with a function to specify what should happen when the event occurs.
3. **Dynamic Interaction**: Event listeners enable dynamic behavior on a webpage, such as responding to user interactions.

Syntax of addEventListener():

```
element.addEventListener(event, callbackFunction);
```

- event: The name of the event (e.g., "click", "keydown").

- callbackFunction: The function to execute when the event occurs.

21: How does the addEventListener() method work in JavaScript? Provide an example.

The addEventListener() method in JavaScript is used to attach an event handler to an HTML element. This allows you to specify a function that will execute whenever a specific event (like a click or keypress) occurs on that element.

1. **Attaching to an Element**: Use addEventListener() on an element to listen for an event.
2. **Event Type**: Specify the type of event (e.g., "click", "keydown", "mouseover").
3. **Callback Function**: Provide a function (either named or anonymous) that will execute when the event happens.
4. **Optional Parameters**: Additional options, such as setting the listener to execute only once ({ once: true }).

Syntax:

element.addEventListener(event, callbackFunction, options);

22: What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?

The Document Object Model (DOM) is a programming interface that represents the structure of an HTML or XML document. It allows developers to manipulate the content, structure, and style of a webpage dynamically using JavaScript.

How JavaScript Interacts with the DOM

JavaScript can access and manipulate the DOM using various methods and properties provided by the document object.

1. Accessing Elements

JavaScript allows you to select and access specific elements on a webpage.

Examples:

- **By ID:**

```
let title = document.getElementById("mainTitle");

console.log(title.innerText);
```

- **By Class Name:**

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

2. Modifying Content

JavaScript can change the content of elements dynamically.

Examples:

- **Updating text:**

```
title.innerText = "Welcome to the DOM!";
```

- **Adding HTML content:**

```
title.innerHTML = "<strong>DOM Manipulation</strong>";
```

3. Adding or Removing Elements

JavaScript can create, add, or remove elements.

Examples:

- **Creating a new element:**

```
let newElement = document.createElement("p");

newElement.innerText = "Hello, World!";

document.body.appendChild(newElement);
```

- **Removing an element:**

```
let elementToRemove = document.getElementById("oldElement");

elementToRemove.remove();

});
```

23: Explain the methods getElementById(), getElementsByClassName(),and querySelector() used to select elements from the DOM.

1. getElementById()

- Purpose: Selects a single element based on its unique ID.
- Returns: A reference to the first element with the specified id. If no element is found, it returns null.
- **Example:**

```
// HTML: <div id="header">Welcome</div>

let header = document.getElementById("header");

console.log(header.innerText);

// Output: "Welcome"
```

2**. getElementsByClassName()**

- Purpose: Selects all elements with a specific class name.
- Returns: A live HTMLCollection (similar to an array, but not exactly) of elements. If no matching elements are found, it returns an empty collection.
- **Example:**

```
// HTML: <div class="item">Item 1</div><div class="item">Item 2</div>

let items = document.getElementsByClassName("item");

console.log(items[0].innerText);
```

```
// Output: "Item 1"
```

3. **querySelector()**

- Purpose: Selects the first element that matches a CSS selector.
- Returns: A single element (the first match). If no matching element is found, it returns null.
- **Example:**

```
// HTML: <div class="content" id="main">Hello</div>

let main = document.querySelector("#main");

console.log(main.innerText);

// Output: "Hello"
```

24: Explain the setTimeout() and setInterval() functions in JavaScript. Howare they used for timing events?

setTimeout() Use to execute function after waiting for the specified time interval. - setTimeout() return numeric value that represents the ID value of the timer. - setTimeout() executes the function only once

Example:

```
console.log("Timer starts...");

setTimeout(() => {

   console.log("This message appears after 2 seconds!");

}, 2000);
```

Set Interval

setInverval() used to repeat a specified function at every given time-interval. - setInverval() evaluates an expression or calls a function at given intervals. - setInverval() invokes the function multiple times .

setInterval() continues the calling of function until the window is closed or the clearInterval() method is called.

Example:

```
setInterval(() => {
   console.log("This message appears every 1 second!");
}, 1000);
```

25: Provide an example of how to use setTimeout() to delay an action by 2 seconds.

Example:

```
console.log("Action will happen in 2 seconds...");
setTimeout(() => {
   console.log("This message is displayed after a 2-second delay.");
}, 2000);
```

Explanation:

1. Before Delay: The first console.log message, "Action will happen in 2 seconds...", runs immediately.
2. setTimeout:
   - The setTimeout() function schedules the callback function (arrow function in this case) to execute after a 2-second delay (2000 milliseconds).
   - When the 2 seconds are up, the second console.log message is displayed: "This message is displayed after a 2-second delay."

26: What is error handling in JavaScript? Explain the try, catch, and finally blocks with an example

Error handling in JavaScript is the process of managing and responding to errors that occur during the execution of code.

try Block:

- Contains the code that might throw an error.
- If an error occurs, execution is transferred to the catch block.

catch Block:

- Executes if an error is thrown in the try block.
- You can access the error object, which contains details about the error.

finally Block (Optional):

- Always executes, regardless of whether an error occurred or not.
- Typically used for cleanup activities like closing resources or resetting states.
- Example:

```
const jsonString = '{"name": "Riya", "age": 25}';

try {
    // Trying to parse JSON
    const user = JSON.parse(jsonString);
    console.log("User's name is:", user.name);

    // Simulate an error
    console.log(user.address.toString()); // 'address' is undefined
} catch (error) {
    // Catch block handles the error
    console.error("An error occurred:", error.message);
} finally {
    // Finally block runs regardless of the error
    console.log("Parsing attempt completed.");
```

- }

27: Why is error handling important in JavaScript applications?

1**. Prevents Application Crashes**

- Without proper error handling, unhandled errors can crash an application, leading to a poor user experience.

2**. Improves User Experience**

- Users prefer applications that handle errors transparently and provide meaningful feedback, such as error messages or fallback behavior.

3**. Facilitates Debugging**

- Capturing and logging errors helps developers identify bugs, understand their root cause, and fix them more efficiently.

4**. Enhances Code Reliability**

- Applications are less likely to break down in unexpected scenarios (e.g., invalid user input, unavailable API resources).

5**. Ensures Proper Cleanup**

- Errors can leave resources like files, memory, or database connections in an improper state.

6. **Allows Fallback Behavior**

- Error handling allows the implementation of fallback solutions when an operation fails.

  .