

Javascript Essentials And Advanced

1. Javascript Introduction

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

ANS: **JavaScript Basics**

JavaScript is a programming language that adds interactivity to websites. While HTML creates structure and CSS handles styling, JavaScript makes web pages dynamic and responsive.

Key roles of JavaScript in web development:

- Makes websites interactive (buttons, forms, animations)
- Updates content without reloading the page
- Responds to user actions (clicks, typing)
- Communicates with servers to fetch and send data
- Powers modern web applications like social media and online stores

JavaScript works in all web browsers and has expanded to server-side development through Node.js, allowing developers to use one language for both front-end and back-end programming.

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

Ans: **JavaScript vs Python and Java**

JavaScript:

- Made for websites and runs in browsers
- Flexible with variable types
- Uses curly braces {} for code blocks
- Handles one task at a time but switches quickly
- Great for making websites interactive

Python:

- Used for data, AI, and general programming
- Clean, readable code with spaces instead of braces
- Easy to learn with simple syntax
- Popular in science and education
- Strong for data analysis and automation

Java:

- Built for large business applications
- Very strict about variable types
- Needs more code to do simple tasks
- Runs on many different devices
- Good for Android apps and enterprise software

The biggest difference: JavaScript is built into web browsers, while Python and Java need separate programs to run their code.

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

Ans: **The Script Tag in HTML**

The <script> tag is how we add JavaScript to web pages. It tells the browser "this is code, not regular content."

Using the script tag:

Internal JavaScript:

html

<script>

 alert("Hello World!");

</script>

This puts JavaScript directly in your HTML file.

Linking External JavaScript:

To connect a separate JavaScript file:

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

This loads the file named "myscript.js" from the same folder as your HTML file.

Common Placement:

- Many developers put script tags just before the closing `</body>` tag so the page content loads first
- You can also place them in the `<head>` section

2. Variables and Data Types.

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

Ans: **Variables in JavaScript**

Variables are containers that store values in your code. They let you save and reuse information like numbers, text, or more complex data.

Declaring Variables

Using var (older way)

```
var age = 25;
```

```
var name = "Alex";
```

Using let (modern way)

```
let score = 100;
```

```
let isActive = true;
```

Using const (for constants)

```
javascript
```

```
const PI = 3.14159;
```

```
const USERNAME = "admin";
```

Simple Rules

1. Use const by default (for values that won't change)
2. Use let when you need to change the value later
3. Avoid var in modern code (it's older and can cause unexpected issues)

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

Ans: **JavaScript Data Types**

JavaScript has several basic data types to store different kinds of information:

Primitive Types

Number

For both integers and decimals:

```
let age = 25;
```

```
let price = 19.99;
```

String

For text of any length:

```
let name = "Sarah";
```

```
let message = 'Hello, world!';
```

```
let address = `123 Main St`; // Template string
```

Boolean

For true/false values:

```
let isLoggedIn = true;
```

```
let hasPermission = false;
```

Undefined

For variables declared but not assigned:

```
let userName; // Value is undefined
```

Null

For intentionally empty values:

```
let selectedItem = null;
```

Symbol

For unique identifiers:

```
let id = Symbol('id');
```

BigInt

For very large integers:

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

Reference Types

Object

For storing collections of data:

```
let person = {  
  name: "John",  
  age: 30,  
  isStudent: false  
};
```

Array

For ordered lists (technically objects):

```
let colors = ["red", "green", "blue"];  
let mixed = [1, "hello", true, null];
```

Function

Functions are also a type:

```
let greet = function() {  
  return "Hello!";  
};
```

You can check a value's type using the `typeof` operator:

```
typeof 42; // "number"
```

```
typeof "hello"; // "string"
```

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

Ans : **undefined vs null in JavaScript**

Both represent "empty" values, but they're used differently:

undefined

- **What it means:** "This variable has been declared but has no value assigned yet"
- **When it happens automatically:**
 - Variables declared without assignment
 - Function parameters that weren't provided
 - Function returns with no explicit return value
 - Trying to access object properties that don't exist

javascript

```
let user; // automatically undefined
```

```
function test() { } // returns undefined
```

null

- **What it means:** "This variable intentionally has no value"
- **Never happens automatically:** You must explicitly assign it
- **Used to reset or clear variables:** Shows a deliberate absence of value

javascript

```
let selectedItem = null;
```

3. Javascript Operators

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

o Arithmetic operators

- o Assignment operators
- o Comparison operators
- o Logical operators

Ans: **JavaScript Operators**

Arithmetic Operators

These do math calculations:

- + adds numbers together ($5 + 3 = 8$)
- - subtracts numbers ($10 - 4 = 6$)
- * multiplies numbers ($6 \times 7 = 42$)
- / divides numbers ($20 \div 5 = 4$)
- % gives the remainder after division ($10 \div 3 = 3$ with 1 leftover)
- ++ adds 1 to a number
- -- subtracts 1 from a number

Assignment Operators

These put values into variables:

- = puts a value in a variable
- += adds then stores ($x += 5$ means "add 5 to x")
- -= subtracts then stores
- *= multiplies then stores
- /= divides then stores

Comparison Operators

These check if things are equal, bigger, or smaller:

- == checks if values are equal
- === checks if values AND types are equal
- != checks if values are not equal

- `!==` checks if values OR types are not equal
- `>` checks if left side is bigger
- `<` checks if left side is smaller
- `>=` checks if left side is bigger or equal
- `<=` checks if left side is smaller or equal

Logical Operators

These combine true/false values:

- `&&` (AND) - true only when BOTH sides are true
- `||` (OR) - true when EITHER side is true
- `!` (NOT) - flips true to false or false to true

Que 2) What is the difference between `==` and `===` in JavaScript?

Ans: **`==` vs `===` in JavaScript**

Simple Explanation

The double equals (`==`) and triple equals (`===`) are both comparison operators, but they work differently:

Double Equals (`==`)

- Checks if values are equal
- Converts types before comparing
- More flexible but less precise
- Called "loose equality" or "abstract equality"

Triple Equals (`===`)

- Checks if values AND types are equal
- No type conversion happens
- More strict and precise
- Called "strict equality" or "identity operator"

Examples

javascript

5 == "5" *// true (number 5 equals string "5" after conversion)*

5 === "5" *// false (number 5 is not the same as string "5")*

0 == false *// true (converts to same value)*

0 === false *// false (different types: number vs boolean)*

null == undefined *// true (considered equal in loose comparison)*

null === undefined *// false (different types)*

4. Control Flow (If-Else, Switch)

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

Ans: **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 flow statements like if-else, loops, switch, and function calls allow you to change that order based on conditions.

if-else Statement in JavaScript

An if-else statement is used to run a block of code based on whether a condition is true or false.

Syntax:

```
if (condition) {  
    // block of code if condition is true  
} else {  
    // block of code if condition is false
```

```
}
```

Example:

```
let age = 20;  
if (age >= 18) {  
  console.log("You are an adult.");  
} else {  
  console.log("You are a minor.");  
}
```

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

Ans: A switch statement is used to perform different actions based on different possible values of a single variable or expression. It's a cleaner alternative to using many if-else if statements when checking for equality against multiple values.

Syntax:

```
switch (expression) {  
  case value1:  
    // code block  
    break;  
  case value2:  
    // code block  
    break;  
  default:  
    // code block  
}
```

How It Works:

1. JavaScript evaluates the expression once.
2. It then compares the result to each case value.
3. If a match is found, it runs the corresponding code block.
4. The break statement prevents the code from "falling through" to the next case.
5. If no match is found, the default block is executed (if provided).

When to Use switch vs. if-else:

Use switch When:

You're checking one variable against many values

Each condition is a simple equality check

Readability is improved with multiple branches

Use if-else When:

You have complex conditions (e.g., >, <, &&),

Each condition may involve ranges or expressions

You only have a few conditions or need flexibility

5. Loops (for, While, Do-While)

Que 1) Explain the different types of loops in JavaScript (for, while, do-while). Provide a basic example of each.

Ans: **1. for Loop**

Used when you know in advance how many times to loop.

Syntax:

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

Example:

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

Output:

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

2. while Loop

Used when the number of iterations is not known in advance. It checks the condition **before** each iteration.

Syntax:

```
while (condition) {  
  // code block 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

Like the while loop, but it checks the condition **after** executing the code block. It **always runs at least once**.

Syntax:

```
do {  
    // code block to execute  
} while (condition);
```

Example:

```
let x = 1;  
do {  
    console.log("Value: " + x);  
    x++;  
} while (x <= 2);
```

Output:

Value: 1
Value: 2

Summary Table:

Loop Type	Condition Checked	Best For
-----------	-------------------	----------

for	Before loop	Known number of iterations
while	Before loop	Unknown iterations, might not run
do-while	After loop	Always run at least once

Que 2) What is the difference between a while loop and a do-while loop?

Ans: The **main difference** between a while loop and a do-while loop in JavaScript is **when the condition is checked**:

while Loop

- **Checks the condition first**, before executing the loop body.
- If the condition is false at the start, the loop body may **not run at all**.

Example:

```
let count = 0;
while (count > 0) {
  console.log("This won't run");
}
```

Output: *(Nothing)*

do-while Loop

- **Executes the loop body first**, then checks the condition.
- The loop body will **run at least once**, even if the condition is false.

Example:

```
let count = 0;
do {
  console.log("This will run once");
} while (count > 0);
```

Output:

This will run once

Summary Table:

Feature	while Loop	do-while Loop
Condition checked	Before loop starts	After first loop execution
Executes at least once	No	Yes
Use case	When the loop might not run	When it should run at least once

6. Loops (for, While, Do-While)

Que 1) What are functions in JavaScript? Explain the syntax for declaring and calling a function.

Ans: A **function** in JavaScript is a reusable block of code that performs a specific task. Functions help you organize and reuse code efficiently.

Why Use Functions?

- Avoid repeating code (DRY principle).
- Make code more readable and maintainable.
- Separate logic into smaller, manageable parts.

Declaring a Function

Syntax:

```
function functionName(parameters) {
    // code to execute
}
```

Que 2) What is the difference between a function declaration and a function expression?

Ans: In JavaScript, **both function declarations and function expressions** define functions, but they differ in **syntax, hoisting behavior, and how they are used**.

✓ 1. Function Declaration

Syntax:

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

✓ Key Features:

- **Hoisted:** You can call the function **before** it's defined in the code.
- Defined with the function keyword and a name.

Example:

```
sayHi(); // ✓ Works  
  
function sayHi() {  
  console.log("Hi!");  
}
```

2. Function Expression

Syntax:

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

Key Features:

- **Not hoisted:** You **must define** the function before calling it.
- Often stored in a variable.
- Can be **anonymous** (no name).

Example:

```
sayHi(); // Error: Cannot access 'sayHi' before initialization
```



```
const sayHi = function() {  
  console.log("Hi!");  
};
```

Comparison Table:

Feature	Function Declaration	Function Expression
Syntax	function name() {}	const name = function() {}
Hoisting	✔ Yes	✗ No
Can be anonymous	✗ No	✔ Yes
Call before definition	✔ Allowed	✗ Not allowed

Que 3) Discuss the concept of parameters and return values in functions.

Ans: Functions in JavaScript can take **parameters** (inputs) and optionally return **values** (outputs). This makes functions flexible and reusable.

1. Parameters

- Parameters are **placeholders** for values passed into a function.
- They are defined in the **function declaration**.

Example:

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

- Here, name is a **parameter**.
- When the function is called, an **argument** (like "Alice") is passed.

Calling the function:

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

2. Return Values

- A function can **return a value** using the return keyword.
- The value can be used or stored later.

Example:

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

```
let result = add(3, 5); // result = 8  
console.log(result); // Output: 8
```

- a and b are **parameters**.
- return a + b; sends back the result to wherever the function is called.

7. Arrays

Que 1) What is an array in JavaScript? How do you declare and initialize an array?

Ans : An **array** in JavaScript is a special variable used to store **multiple values** in a single variable. It can hold elements of **any data type**, including numbers, strings, objects, or even other arrays.

Declaring and Initializing an Array

1. Using square brackets ([]) — Most common:

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

Que 2) Explain the methods push(), pop(), shift(), and unshift() used in arrays.

Ans: 1. **push()** – Add to the End

It adds a new item to the **end** of the array.

Example:

```
let fruits = ["apple", "banana"];  
fruits.push("orange");  
// Now: ["apple", "banana", "orange"]
```

2. pop() – Remove from the End

It removes the **last** item from the array.

Example:

```
let fruits = ["apple", "banana"];  
fruits.pop();  
// Now: ["apple"]
```

3. unshift() – Add to the Start

It adds a new item to the **beginning** of the array.

Example:

```
let fruits = ["banana", "cherry"];  
fruits.unshift("apple");  
// Now: ["apple", "banana", "cherry"]
```

4. shift() – Remove from the Start

It removes the **first** item from the array.

Example:

```
let fruits = ["apple", "banana"];  
fruits.shift();  
// Now: ["banana"]
```

Simple Summary:

Method What it does

push() Add at the **end**

pop() Remove from the **end**

unshift() Add at the **start**

shift() Remove from the **start**

8. Object

Que 1) What is an object in JavaScript? How are objects different from arrays?

Ans: An **object** in JavaScript is a collection of **key-value pairs**. Each key (called a **property**) is a string (or symbol), and each value can be anything: a number, string, array, function, or even another object.

How to Create an Object

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

- name, age, and isStudent are **keys (or properties)**.
- "Alice", 25, and true are their **values**.

Accessing Object Properties

```
console.log(person.name);    // Output: Alice
```

```
console.log(person["age"]); // Output: 25
```

How Objects Are Different from Arrays

Feature	Object	Array
Structure	Key-value pairs	Ordered list of values
Keys	Strings or symbols	Numeric indexes (0, 1, 2...)
Use Case	Store related data with labels	Store lists of items
Example	{name: "Alice", age: 25}	["Alice", 25]

In Short:

- Use an **object** when you want to label and organize data.
- Use an **array** when you have a list of items in order.

Que 2) Explain how to access and update object properties using dot notation and bracket notation.

Ans: JavaScript objects store data in **key-value pairs**, and there are two main ways to **access and update** these properties:

1. Dot Notation (object.key)

Access:

```
let person = { name: "Alice", age: 25 };  
console.log(person.name); // Output: Alice
```

Update:

```
person.age = 26;  
console.log(person.age); // Output: 26
```

Use dot notation when the key is a valid variable name (no spaces or special characters).

2. Bracket Notation (object["key"])

Access:

```
let person = { name: "Alice", age: 25 };  
console.log(person["name"]); // Output: Alice
```

Update:





```
person["age"] = 30;  
console.log(person["age"]); // Output: 30
```

9. JavaScript Events

Que 1) What are JavaScript events? Explain the role of event listeners.

Ans: **JavaScript events** are actions or occurrences that happen in the browser — often triggered by the user — that your code can respond to.

Examples of events:

- A user clicks a button  click event
- A page finishes loading  load event
- A user presses a key  keydown event
- A form is submitted  submit event

What Is an Event Listener?

An **event listener** is a function that waits for a specific event to happen and then runs some code in response.

Syntax:

```
element.addEventListener("eventType", function);
```

Example: Button Click

HTML:

```
<button id="myBtn">Click Me</button>
```

JavaScript:

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

Explanation:

- "click" is the event type.
- The function inside runs **only when** the button is clicked.

Why Use Event Listeners?

- They **separate HTML and JavaScript**.
- You can add multiple listeners to one element.
- You can control when and how code reacts to user actions.

Que 2) How does the addEventListener() method work in JavaScript? Provide an example.

Ans: The addEventListener() method in JavaScript is used to attach an **event listener** to an element. This method allows you to specify which **event type** you want to listen for (e.g., click, keydown, etc.) and the **function** that should be executed when that event occurs.

Syntax:

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

- eventType: The type of event to listen for (e.g., "click", "keydown", etc.).
- callbackFunction: The function to execute when the event is triggered.

Example:

Let's say you have a button, and you want to show an alert when the button is clicked.

HTML:

```
<button id="myBtn">Click Me</button>
```

JavaScript:

```
// Get the button element
```

```
let button = document.getElementById("myBtn");
```

```
// Attach an event listener to the button
```

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

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

```
});
```

Explanation:

1. We get the button element using `document.getElementById("myBtn")`.
2. We use `addEventListener()` to listen for the "click" event.
3. When the button is clicked, the provided **callback function** (which shows an alert) is executed.

10. DOM Manipulation

Que 1) What is the DOM (Document Object Model) in JavaScript? How does JavaScript interact with the DOM?

Ans: The **DOM (Document Object Model)** is a programming interface for web documents. It represents the **structure** of an HTML (or XML) document as a **tree of objects**, where each element, attribute, and text is a **node** in that tree.

- It allows you to **access** and **manipulate** the content, structure, and style of a web page using JavaScript.
 - It provides a **dynamic view** of the document, meaning you can change the page after it's loaded (e.g., add/remove elements, change text, styles, etc.).
-

How Does JavaScript Interact with the DOM?

JavaScript interacts with the DOM through various methods, such as **accessing elements**, **modifying content**, and **changing styles**. These interactions allow you to **dynamically** change the page without reloading it.

Common Interactions:

1. Accessing Elements:

- You can use JavaScript to **select elements** (like `<div>`, `<p>`, etc.) from the DOM.

Example:

```
let heading = document.getElementById("myHeading");  
let paragraphs = document.getElementsByTagName("p");
```

2. Manipulating Content:

- You can **change the text** or **HTML** of an element.

Example:

```
heading.innerHTML = "New Heading Text"; // Change heading text
```

3. Modifying Styles:

- JavaScript allows you to **change the styles** of HTML elements dynamically.

Example:

```
heading.style.color = "blue"; // Change text color to blue
```

4. Adding/Removing Elements:

- You can **add** new elements or **remove** existing ones.

Example:

```
let newParagraph = document.createElement("p");  
newParagraph.textContent = "This is a new paragraph!";  
document.body.appendChild(newParagraph); // Adds the new paragraph to  
the body
```

5. Handling Events:

- JavaScript can **attach event listeners** to DOM elements (like buttons, inputs) to **respond to user actions** (e.g., clicks, key presses).

Example:

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

Que 2) Explain the methods `getElementById()`, `getElementsByClassName()`, and `querySelector()` used to select elements from the DOM.

Ans: JavaScript provides several methods to select elements from the **DOM (Document Object Model)**. These methods are used to find elements on the web page that you can **manipulate, style, or add event listeners** to.

Here are three common methods: `getElementById()`, `getElementsByClassName()`, and `querySelector()`.

1. `getElementById()`

- Purpose:** This method selects an element by its **unique ID**.
- Returns:** A **single element** (the first element with the given ID).
- Usage:** Useful when you want to select a single element with a specific id attribute.

Syntax:

```
document.getElementById("idName");
```

Example:

HTML:

```
<div id="mainContent">This is the main content</div>
```

JavaScript:

```
let content = document.getElementById("mainContent");  
content.style.color = "blue"; // Change text color to blue
```

Note: IDs are **unique** on a page, so only **one element** will be selected.

2. `getElementsByClassName()`

- **Purpose:** This method selects elements by their **class name**.
- **Returns:** A **live HTMLCollection** of all elements with the given class name.
- **Usage:** Useful when you want to select multiple elements that share the same class.

Syntax:

```
document.getElementsByClassName("className");
```

Example:

HTML:

```
<div class="card">Card 1</div>
```

```
<div class="card">Card 2</div>
```

JavaScript:

```
let cards = document.getElementsByClassName("card");
```

```
cards[0].style.backgroundColor = "yellow"; // Change the first card's  
background color
```

Note: `getElementsByClassName()` returns a **live collection**, meaning if the DOM changes, the collection will automatically update.

3. `querySelector()`

- **Purpose:** This method allows you to select **any element** using **CSS selectors** (IDs, classes, attributes, etc.).
- **Returns:** The **first matching element** that matches the CSS selector.

- **Usage:** It's very versatile, as you can use any valid CSS selector to target elements.

Syntax:

```
document.querySelector("selector");
```

Example:

HTML:

```
<div id="mainContent" class="content">Main Content</div>
```

```
<p class="content">Another content</p>
```

JavaScript:

```
let mainDiv = document.querySelector("#mainContent");
```

```
mainDiv.style.color = "green"; // Change color of the first matched element
```

You can also use more complex CSS selectors:

```
let content = document.querySelector(".content");
```

11. JavaScript Timing Events (setTimeout, setInterval)

Que 1) Explain the setTimeout() and setInterval() functions in JavaScript. How are they used for timing events?

Ans: **setTimeout() in JavaScript:**

- setTimeout() is used to **execute a function once** after a specified delay (in milliseconds).
- It is commonly used to delay the execution of code.
- Syntax: setTimeout(function, delay)
- The function runs only one time after the delay has passed.
- The delay time is not guaranteed to be exact; it's the minimum delay before execution.

setInterval() in JavaScript:

- `setInterval()` is used to **repeatedly execute a function** at specified time intervals (in milliseconds).
 - It continues to execute the function until it is stopped using `clearInterval()`.
 - Syntax: `setInterval(function, interval)`
 - Useful for creating timers, clocks, or repeating tasks.
-

Use for Timing Events:

- Both functions allow JavaScript to perform **asynchronous tasks**.
- `setTimeout()` is ideal for **one-time delayed actions**.
- `setInterval()` is used for **continuous repeated actions** at regular time intervals.

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

Ans: `setTimeout(function() {
 console.log("This message is displayed after 2 seconds");
}, 2000);`

Explanation:

- The anonymous function inside `setTimeout()` will execute after a delay of 2000 milliseconds (i.e., 2 seconds).
- This is useful for creating pauses or delaying certain operations in a program.

12. JavaScript Error Handling

Que 1) What is error handling in JavaScript? Explain the try, catch, and finally blocks with an example.

Ans: **Error Handling in JavaScript:**

Error handling in JavaScript is the process of **managing runtime errors** so that the program doesn't crash unexpectedly and can handle issues gracefully.

JavaScript provides a mechanism using the **try, catch, and finally** blocks to detect and respond to errors.

1. try Block

- Contains code that **may throw an error**.
- If an error occurs, control immediately moves to the catch block.

2. catch Block

- Executes if an error occurs in the try block.
- Receives the error object and allows handling or displaying the error.

3. finally Block

- Always executes **after** the try and catch blocks, whether an error occurred or not.
 - Used for **cleanup operations** (like closing files or clearing timers).
-

Syntax:

```
try {  
    // Code that may cause an error  
} catch (error) {  
    // Code to handle the error  
} finally {  
    // Code that runs no matter what  
}
```

Example:

```
try {
```

```
let result = 10 / 0;

console.log("Result:", result);


// Simulating an error

let x = y + 1; // y is not defined, this will throw an error
} catch (error) {
    console.log("An error occurred:", error.message);
} finally {
    console.log("This block runs regardless of the error.");
}
```

Que 2) Why is error handling important in JavaScript applications?

Ans: **Importance of Error Handling in JavaScript Applications:**

Error handling is crucial in JavaScript applications for several key reasons:

1. Prevents Application Crashes

- Without proper error handling, unhandled errors can stop script execution.
- Catching errors ensures the application continues running or fails gracefully.

2. Improves User Experience

- Instead of showing confusing or broken pages, users see meaningful messages (e.g., "Something went wrong, please try again").

3. Debugging and Logging

- Capturing errors helps developers log them for later analysis.

- Makes it easier to identify and fix bugs during development or after deployment.

4. Ensures Application Stability

- Protects critical operations like API calls, file access, or user input processing from breaking the entire application.

5. Allows Fallbacks and Recovery

- Enables implementation of alternative logic when errors occur (e.g., loading cached data if API fails).

6. Security

- Prevents exposure of sensitive information by controlling what error messages are shown to users.

