JavaScript

JavaScript Fundamentals (Core Concepts)

Syntax & Basics

- Variables (var, let, const)
- Data Types (String, Number, Boolean, Null, Undefined, Symbol)
- Operators (Arithmetic, Comparison, Logical)

1. JavaScript Syntax

JavaScript syntax is the set of rules that define how JavaScript programs are written and interpreted. Some key points about JavaScript syntax are:

✅ **Case-sensitive** — Keywords like let, const, and function must follow exact casing.
✅ **Statements end with a semicolon ;** (optional but recommended for clarity).
✅ **Curly Braces {}** are used to define code blocks.
✅ **Comments** can be added using:

- // for single-line comments
- /* */ for multi-line comments

2. Variables in JavaScript

JavaScript provides three ways to declare variables:

✅ **var (Old method - Avoid using this)**

- Function-scoped (not block-scoped).
- Can be redeclared and reassigned.
- May cause unexpected behavior due to its scope.

✅ **let (Preferred for mutable values)**

- Block-scoped (only accessible within the block where declared).
- Can be reassigned but **cannot be redeclared** in the same scope.

✅ **const (Preferred for immutable values)**

- Block-scoped like let.
- **Cannot be reassigned** or **redeclared**.

Example:

```
var x = 10;      // Global scope
let y = 20;      // Block scope
const PI = 3.14;  // Constant value
```

output::

```
x = 15;  // ☑ Allowed
y = 25;  // ☑ Allowed
// PI = 3.14159; // ✗ Error: Assignment to constant variable
console.log(x, y, PI); // Output: 15 25 3.14
```

3. Data Types in JavaScript

JavaScript has the following primary data types:

✅ **Primitive Data Types (Immutable)**

- **String** → Text data (e.g., "Hello" or 'World')
- **Number** → Any numeric value (e.g., 10, 3.14)
- **Boolean** → True or false values
- **Null** → Represents an empty or non-existent value
- **Undefined** → A declared variable that has not been assigned a value
- **Symbol** → Unique and immutable values (mostly used for object properties)

✅ **Non-Primitive Data Types (Reference Types)**

- **Object** → Used to store collections of data
- **Array** → Special type of object for lists

Example:

```
let name = "Trupti";   // String
let age = 25;        // Number
let isDeveloper = true; // Boolean
let address = null;    // Null
let salary;          // Undefined (no value assigned)

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

```
console.log(typeof name);     // Output: string
console.log(typeof age);      // Output: number
console.log(typeof isDeveloper); // Output: boolean
console.log(typeof address);  // Output: object (special case for null)
console.log(typeof salary);   // Output: undefined
console.log(typeof uniqueId); // Output: symbol
```

---

4. Operators in JavaScript

JavaScript operators are used to perform operations on variables and values. Common types include:

✅ **Arithmetic Operators**

Operator	Description	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y (Remainder)
**	Exponentiation	x ** y (Power)

✅ **Comparison Operators**

Operator	Description	Example
==	Equal to	x == y
===	Strict equal	x === y
!=	Not equal	x != y
!==	Strict not equal	x !== y
>	Greater than	x > y
<	Less than	x < y
>=	Greater or equal	x >= y

Operator	Description	Example
<=	Less or equal	x <= y

✅ **Logical Operators**

Operator	Description	Example
&&	Logical AND	x && y (Both true)
`	`	`
!	Logical NOT	!x (Negates condition)

---

5. Example Code Combining Everything

```
// Variables
let firstName = "Trupti";  // String
const PI = 3.14;         // Constant Number
var isWorking = true;     // Boolean
let salary;             // Undefined
let address = null;       // Null

// Arithmetic Operations
let num1 = 10;
let num2 = 5;
let result = num1 + num2;  // Addition

// Comparison
let isEqual = num1 === num2;  // Strict equality check (false)

// Logical Operation
let hasAccess = isWorking && result > 10; // true

// Display Results
console.log("Name:", firstName);     // Output: Name: Trupti
```

```
console.log("PI Value:", PI);        // Output: PI Value: 3.14
console.log("Salary:", salary);      // Output: Salary: undefined
console.log("Result:", result);      // Output: Result: 15
console.log("Is Equal:", isEqual);   // Output: Is Equal: false
console.log("Has Access:", hasAccess); // Output: Has Access: true
```

Key Takeaways

- ✅ Use let for variables that may change and const for constants.
- ✅ Avoid using var unless necessary for legacy code.
- ✅ Always use === for strict comparisons to avoid unexpected behavior.
- ✅ Use typeof to check the data type of a value.
- ✅ Practice combining variables, data types, and operators to build logic efficiently.

Control Flow in JavaScript

Control flow determines the order in which statements are executed in a program. JavaScript provides several control flow structures like if, else, switch, and ternary operators to make decisions and execute code based on conditions.

---

1. if Statement

The if statement executes a block of code **only if the condition is true**.

Syntax

```
if (condition) {
  // Code to execute if condition is true
}
```

Example

```
javascript
CopyEdit
let age = 18;

if (age >= 18) {
  console.log("You are eligible to vote."); // Output: You are eligible to vote.
}
```

---

2. if...else Statement

The if...else statement executes one block of code if the condition is true and another block if the condition is false.

Syntax

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

Example

javascript

CopyEdit

```
let score = 45;

if (score >= 50) {
  console.log("Pass"); // This block won't run
} else {
  console.log("Fail"); // Output: Fail
}
```

---

3. if...else if...else Statement

This structure checks multiple conditions and executes the code block for the first true condition.

Syntax

```
if (condition1) {
  // Code to execute if condition1 is true
} else if (condition2) {
  // Code to execute if condition2 is true
} else {
  // Code to execute if all conditions are false
```

```
}
```

Example

```
javascript

CopyEdit

let marks = 75;


if (marks >= 90) {
   console.log("Grade: A");
} else if (marks >= 70) {
   console.log("Grade: B");  // Output: Grade: B
} else if (marks >= 50) {
   console.log("Grade: C");
} else {
   console.log("Fail");
}
```

---

4. switch Statement

The switch statement is used to perform different actions based on different conditions. It's useful when you have multiple possible conditions.

Syntax

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

break *is necessary to stop the code from continuing to the next case.*

Example

javascript

CopyEdit

```
let day = "Monday";

switch (day) {
  case "Monday":
    console.log("Start of the work week!");
    break;
  case "Friday":
    console.log("Weekend is near!");
    break;
  case "Saturday":
  case "Sunday":
    console.log("It's the weekend!");
    break;
  default:
    console.log("It's a regular day.");
}
// Output: Start of the work week!
```

---

5. Ternary (Conditional) Operator

The **ternary operator** provides a concise way to write if...else conditions. It is written using ? and :.

Syntax

```
condition ? trueResult : falseResult;
```

Example

javascript

CopyEdit

```
let age = 20;

let access = (age >= 18) ? "Access granted" : "Access denied";
```

```
console.log(access); // Output: Access granted
```

---

6. Example Combining All Concepts

Here's a comprehensive example that combines if...else, switch, and the ternary operator.

Example

```
let userRole = "admin";
let isLoggedIn = true;

if (isLoggedIn) {
  switch (userRole) {
    case "admin":
      console.log("Welcome, Admin! You have full access.");
      break;
    case "editor":
      console.log("Welcome, Editor! You can edit content.");
      break;
    case "viewer":
      console.log("Welcome, Viewer! You can view content.");
      break;
    default:
      console.log("Unknown role. Access denied.");
  }
} else {
  console.log("Please log in to continue.");
}

// Ternary Operator Example
let message = isLoggedIn ? "You're logged in!" : "Please log in.";
console.log(message);
```

Output

pgsql

CopyEdit

Welcome, Admin! You have full access.

You're logged in!

Key Takeaways

✅ Use if...else for simple conditions.
✅ Use switch for multiple fixed conditions (great for menu options or roles).
✅ Use the **ternary operator** for concise conditions (ideal for simple checks).
✅ Always remember to use break inside switch statements to avoid fall-through.

Loops in JavaScript

Loops are used to repeatedly execute a block of code as long as a specified condition is true. JavaScript provides several types of loops:

✅ **for loop** – Iterates a fixed number of times.
✅ **while loop** – Iterates as long as the condition is true.
✅ **do...while loop** – Executes at least once, then repeats if the condition is true.
✅ **.forEach() method** – Iterates over each element in an array.

---

1. for Loop

The for loop is ideal when you know **how many times** you want to iterate.

Syntax

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

Explanation

- **Initialization:** Initializes the loop counter (e.g., let i = 0).
- **Condition:** The loop runs **while this condition is true**.
- **Increment/Decrement:** Updates the loop counter at the end of each iteration.

Example

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

Output:

makefile

CopyEdit

Iteration: 1

Iteration: 2

Iteration: 3

Iteration: 4

Iteration: 5

---

2. while Loop

The while loop is ideal when **you don't know in advance** how many times the loop should run.

Syntax

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

Example

javascript

CopyEdit

```
let count = 1;

while (count <= 5) {
   console.log("Count:", count);
   count++;
}
```

Output:

makefile

CopyEdit

Count: 1

Count: 2

Count: 3

Count: 4

Count: 5

---

3. do...while Loop

The do...while loop is similar to the while loop, but it guarantees that the code block **runs at least once** before checking the condition.

Syntax

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

Example

```
javascript
CopyEdit
let number = 5;

do {
  console.log("Number is:", number);
  number++;
} while (number <= 3);
```

Output:

```
csharp
CopyEdit
Number is: 5
```

*Even though the condition number <= 3 is false initially, the loop executes **once**.*

---

4. .forEach() Method

The .forEach() method is used to iterate through an **array** and execute a callback function on each element.

Syntax

```
array.forEach(function(element, index, array) {
  // Code to execute for each element
});
```

- **element** → Current element in the array.
- **index** → Index of the current element (optional).
- **array** → The array itself (optional).

Example

```
const fruits = ["Apple", "Banana", "Cherry"];
fruits.forEach(function(fruit, index) {
  console.log(` Fruit ${index + 1}: ${fruit}` );
});
```

Output:

yaml

CopyEdit

Fruit 1: Apple

Fruit 2: Banana

Fruit 3: Cherry

---

5. Example Combining All Loops

Here's a practical example that combines different loop types to show their behavior:

```
// Using 'for' loop
console.log("Using for loop:");
for (let i = 1; i <= 3; i++) {
  console.log(` Step ${i}` );
}

// Using 'while' loop
console.log("\nUsing while loop:");
let count = 1;
while (count <= 3) {
  console.log(` Count ${count}` );
  count++;
}
```

```
// Using 'do...while' loop
console.log("\nUsing do...while loop:");
let number = 3;
do {
  console.log(` Number: ${number}` );
  number--;
} while (number > 0);

// Using '.forEach()' loop
console.log("\nUsing forEach loop:");
const colors = ["Red", "Green", "Blue"];
colors.forEach(color => console.log(color));
```

Output:

```
vbnet
CopyEdit
Using for loop:
Step 1
Step 2
Step 3

Using while loop:
Count 1
Count 2
Count 3

Using do...while loop:
Number: 3
Number: 2
Number: 1
```

Using forEach loop:

Red

Green

Blue

---

6. Key Differences

Feature	for Loop	while Loop	do...while Loop	.forEach()
Best For	Known number of iterations	Unknown number of iterations	Ensuring code runs at least once	Iterating over arrays
Initial Check	Before first iteration	Before first iteration	After first iteration	Automatically handles iteration
Requires Counter	Yes	Yes	Yes	No (uses array elements directly)
Ideal Use Case	Counting from 0 to N	Waiting for a condition to be true	Executing code at least once	Processing elements in an array

---

7. Key Takeaways

✅ Use for when you know how many times the loop should run.
✅ Use while when the condition is unpredictable.
✅ Use do...while when the code must run **at least once**.
✅ Use .forEach() for clean and efficient array iteration.

Loops in JavaScript

Loops are used to repeatedly execute a block of code as long as a specified condition is true. JavaScript provides several types of loops:

✅ **for loop** – Iterates a fixed number of times.
✅ **while loop** – Iterates as long as the condition is true.
✅ **do...while loop** – Executes at least once, then repeats if the condition is true.
✅ **.forEach() method** – Iterates over each element in an array.

---

1. for Loop

The for loop is ideal when you know **how many times** you want to iterate.

Syntax

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

Explanation

- **Initialization:** Initializes the loop counter (e.g., let i = 0).
- **Condition:** The loop runs **while this condition is true**.
- **Increment/Decrement:** Updates the loop counter at the end of each iteration.

Example

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

Output:

```
makefile
CopyEdit
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

---

2. while Loop

The while loop is ideal when **you don't know in advance** how many times the loop should run.

Syntax

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

Example

```
javascript
CopyEdit
let count = 1;
```

```
while (count <= 5) {
  console.log("Count:", count);
  count++;
}
```

Output:

```
makefile
CopyEdit
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

---

3. do...while Loop

The do...while loop is similar to the while loop, but it guarantees that the code block **runs at least once** before checking the condition.

Syntax

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

Example

```
let number = 5;

do {
  console.log("Number is:", number);
  number++;
} while (number <= 3);
```

Output:

```
Number is: 5
```

*Even though the condition number <= 3 is false initially, the loop executes **once**.*

---

4. .forEach() Method

The .forEach() method is used to iterate through an **array** and execute a callback function on each element.

Syntax

```
array.forEach(function(element, index, array) {
  // Code to execute for each element
});
```

- **element** → Current element in the array.
- **index** → Index of the current element (optional).
- **array** → The array itself (optional).

Example

```
const fruits = ["Apple", "Banana", "Cherry"];
fruits.forEach(function(fruit, index) {
  console.log(` Fruit ${index + 1}: ${fruit}` );
});
```

Output:

```
Fruit 1: Apple
Fruit 2: Banana
Fruit 3: Cherry
```

---

5. Example Combining All Loops

Here's a practical example that combines different loop types to show their behavior:

```
// Using 'for' loop
console.log("Using for loop:");
for (let i = 1; i <= 3; i++) {
  console.log(` Step ${i}` );
}

// Using 'while' loop
console.log("\nUsing while loop:");
let count = 1;
```

```
while (count <= 3) {
   console.log(` Count ${count}` );
   count++;
}

// Using 'do...while' loop
console.log("\nUsing do...while loop:");
let number = 3;
do {
   console.log(` Number: ${number}` );
   number--;
} while (number > 0);

// Using '.forEach()' loop
console.log("\nUsing forEach loop:");
const colors = ["Red", "Green", "Blue"];
colors.forEach(color => console.log(color));
```

Output:

```
Using for loop:
Step 1
Step 2
Step 3

Using while loop:
Count 1
Count 2
Count 3

Using do...while loop:
Number: 3
Number: 2
```

Number: 1

Using forEach loop:

Red

Green

Blue

---

6. Key Differences

Feature	for Loop	while Loop	do...while Loop	.forEach()
Best For	Known number of iterations	Unknown number of iterations	Ensuring code runs at least once	Iterating over arrays
Initial Check	Before first iteration	Before first iteration	After first iteration	Automatically handles iteration
Requires Counter	Yes	Yes	Yes	No (uses array elements directly)
Ideal Use Case	Counting from 0 to N	Waiting for a condition to be true	Executing code at least once	Processing elements in an array

---

7. Key Takeaways

- ✅ Use for when you know how many times the loop should run.
- ✅ Use while when the condition is unpredictable.
- ✅ Use do...while when the code must run **at least once**.
- ✅ Use .forEach() for clean and efficient array iteration.

Functions in JavaScript

Functions are reusable blocks of code that perform a specific task. They allow you to write clean, modular, and maintainable code.

---

1. Types of Functions in JavaScript

JavaScript supports several types of functions:

A. Function Declaration

A traditional way of defining a function using the function keyword.

Syntax:

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

Example:

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

✅ **Hoisted:** Function declarations are hoisted, meaning they can be called before being defined.

B. Function Expression

A function assigned to a variable.

Syntax:

```
const functionName = function(parameters) {
  // Code to execute
};
```

Example:

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

✅ **Not Hoisted:** Function expressions are NOT hoisted; they must be defined before use.

C. Arrow Function (ES6+)

A modern and concise way to write functions using =>.

Syntax:

```
const functionName = (parameters) => {
  // Code to execute
```

```
};
```

Example: *(With multiple parameters)*

```
const multiply = (a, b) => a * b;
console.log(multiply(4, 3)); // Output: 12
```

Example: *(With a single parameter – parentheses are optional)*

```
const greet = name => console.log(`Hello, ${name}!`);
greet("Trupti"); // Output: Hello, Trupti!
```

Example: *(Without parameters – requires empty ())*

```
const sayHello = () => console.log("Hello World!");
sayHello(); // Output: Hello World!
```

✅ **Arrow functions** are great for short, simple logic and callbacks.
❗ **this behaves differently** in arrow functions (more on this later if needed).

D. Immediately Invoked Function Expression (IIFE)

A function that executes **immediately** after its definition.

Syntax:

```
(function() {
  // Code executes immediately
})();
```

Example:

```
(function() {
  console.log("IIFE executed!");
})(); // Output: IIFE executed!
```

✅ Useful for **isolating code** to avoid polluting the global scope.

E. Anonymous Functions

Functions without a name (commonly used in callbacks).

Example:

```
setTimeout(function() {
   console.log("Delayed message!");
}, 2000); // Output after 2 seconds: Delayed message!
```

F. Callback Functions

A callback function is a **function passed as an argument** to another function.

Example:

```
function greet(name, callback) {
   console.log(` Hello, ${name}!` );
   callback();  // Calling the callback function
}

function displayMessage() {
   console.log("Welcome to JavaScript!");
}

greet("Trupti", displayMessage);

// Output:
// Hello, Trupti!
// Welcome to JavaScript!
```

☑ *Commonly used in asynchronous code (e.g., API calls, setTimeout, etc.).*

G. Higher-Order Functions

A function that **takes another function as a parameter** or **returns a function**.

Example:

```
function calculate(operation, a, b) {
   return operation(a, b);
}

function add(x, y) {
```

```
  return x + y;
}

console.log(calculate(add, 5, 3)); // Output: 8
```

✅ These are essential for functional programming concepts like .map(), .filter(), and .reduce().

H. Recursive Functions

A function that **calls itself** until a base condition is met.

Example:

```
function factorial(n) {
  if (n === 1) return 1; // Base condition
  return n * factorial(n - 1); // Recursive call
}
console.log(factorial(5)); // Output: 120
```

✅ Ideal for problems like calculating factorial, Fibonacci sequence, etc.

I. Generator Functions (ES6+)

A special function defined using function* that can **pause** and **resume** execution using yield.

Syntax:

```
function* generatorFunction() {
  yield "Step 1";
  yield "Step 2";
  yield "Step 3";
}
const gen = generatorFunction();
console.log(gen.next().value); // Output: Step 1
```

☑ *Useful for handling asynchronous data streams.*

---

2. Key Differences Between Function Types

Feature	Function Declaration	Function Expression	Arrow Function
Syntax	function name() {}	const name = function() {}	const name = () => {}
Hoisting	✅ Hoisted	❌ Not Hoisted	❌ Not Hoisted
this behavior	Refers to the calling object	Refers to the calling object	Inherits this from parent scope
Best Use Case	Reusable functions	Assigning functions to variables	Short, concise callbacks or inline functions

---

3. Example Combining All Concepts

```
// Function Declaration
function greet(name) {
  console.log(` Hello, ${name}!` );
}

// Function Expression
const add = function(a, b) {
  return a + b;
};

// Arrow Function
const square = num => num * num;

// Callback Function
function processUserInput(name, callback) {
  console.log(` Processing data for ${name}... ` );
  callback();
}
```

```
// IIFE
(function() {
   console.log("IIFE executed immediately!");
})();

// Calling functions
greet("Trupti");
console.log(` Addition: ${add(5, 3)}` );
console.log(` Square of 4: ${square(4)}` );
processUserInput("Trupti", () => console.log("Callback executed!"));
```

Output:

```
IIFE executed immediately!
Hello, Trupti!
Addition: 8
Square of 4: 16
Processing data for Trupti...
Callback executed!
```

---

4. Key Takeaways

✅ Use **Function Declarations** for reusable, named functions.
✅ Use **Function Expressions** for dynamic logic assigned to variables.
✅ Use **Arrow Functions** for concise syntax, especially in callbacks.
✅ Use **Callbacks** to handle asynchronous operations.
✅ Use **IIFE** to isolate code and avoid global scope pollution.

Scope & Hoisting in JavaScript

Scope defines the **accessibility** (or visibility) of variables, functions, and objects in your code. JavaScript has three main types of scope:

1. **Global Scope**
2. **Function Scope**
3. **Block Scope**

---

1. Global Scope

- Variables declared **outside any function or block** are in the **global scope**.
- They can be accessed from **anywhere** in the code.
- In browsers, global variables become properties of the window object.

Example:

```
let globalVar = "I'm Global!";

function showGlobal() {
  console.log(globalVar); // Accessible inside the function
}

showGlobal(); // Output: I'm Global!
console.log(globalVar);  // Output: I'm Global!
```

---

2. Function Scope

- Variables declared **inside a function** are accessible **only within that function**.
- These variables are **not accessible outside** the function.

Example:

```
function greet() {
  let message = "Hello from Function Scope!";
  console.log(message);  // Output: Hello from Function Scope!
}

greet();
console.log(message); // Error: message is not defined
```

✅ **var** has **function scope**.
✅ **let** and **const** have **block scope** (explained next).

---

3. Block Scope

- Variables declared using **let** or **const** inside a block ({}) are **only accessible within that block**.

- A block can be an if statement, for loop, etc.

Example:

```
{
  let blockScoped = "I'm inside a block!";
  console.log(blockScoped); // Output: I'm inside a block!
}

console.log(blockScoped); // Error: blockScoped is not defined
```

✅ **let** and **const** respect block scope.
❗ **var** ignores block scope.

---

4. Key Difference Between var, let, and const Scope

Feature	var	let	const
Scope Type	Function Scope	Block Scope	Block Scope
Hoisting	✅ Hoisted with undefined	✅ Hoisted without value	✅ Hoisted without value
Reassignable	✅ Yes	✅ Yes	❌ No
Redeclarable	✅ Yes	❌ No	❌ No

---

5. Hoisting in JavaScript

Hoisting is JavaScript's default behavior of moving **variable** and **function declarations** to the top of their scope during compilation.

- **var** is hoisted with **undefined** as its initial value.
- **let** and **const** are hoisted but **not initialized**.
- **Function Declarations** are fully hoisted, meaning they can be called before they are defined.
- **Function Expressions** and **Arrow Functions** are **not hoisted**.

Example 1: Hoisting with var (Hoisted with undefined)

```
console.log(x); // Output: undefined
var x = 5;
console.log(x); // Output: 5
```

Behind the scenes:

```
var x;       // Declaration hoisted
console.log(x); // undefined
x = 5;       // Assignment stays in place
```

Example 2: Hoisting with let and const (Hoisted but Uninitialized)

```
console.log(y); // ReferenceError: Cannot access 'y' before initialization
let y = 10;
```

Example 3: Hoisting with Functions

✅ **Function Declaration (Hoisted)**

```
sayHello(); // Output: Hello World!

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

❗ Function Expression (Not Hoisted)

```
greet(); // Error: greet is not a function

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

---

6. Practical Example - Scope + Hoisting Combined

```
var a = 10;
function example() {
  if (true) {
    var a = 20;  // `var` is function-scoped
    let b = 30;  // `let` is block-scoped
    const c = 40; // `const` is block-scoped
```

```
    console.log(a); // Output: 20
    console.log(b); // Output: 30
    console.log(c); // Output: 40
  }

  console.log(a); // Output: 20 (Since `var` is function-scoped)
  console.log(b); // Error: b is not defined
  console.log(c); // Error: c is not defined
}

example();
```

---

7. Key Takeaways

✅ Use **let** and **const** instead of var for safer, predictable scoping.
✅ Remember that **var** ignores block scope but respects function scope.
✅ Functions declared with function keyword are **hoisted**, while **function expressions** are **not hoisted**.
✅ Always **declare variables before using them** to avoid unexpected behavior.

Closures in JavaScript

Closures are one of the most powerful and essential concepts in JavaScript. Understanding closures requires knowing about **lexical scope** first.

---

1. What is Lexical Scope?

Lexical Scope (also called **static scope**) refers to the ability of a function to access variables from its **parent scope** where it was defined, even if called outside that scope.

Example of Lexical Scope:

```
function outer() {
  let outerVar = "I'm from outer scope";

  function inner() {
    console.log(outerVar); // Accessible due to lexical scope
  }
```

```
  inner();
}

outer(); // Output: I'm from outer scope
```

☑ *Here, inner() can access outerVar because of **lexical scope**.*
☑ *JavaScript remembers the **scope chain** where the function was defined.*

---

2. What is a Closure?

A **closure** is created when a **function retains access to its parent scope**, even after the parent function has finished executing.

In simple terms:

- A closure is formed when an **inner function** remembers and accesses variables from its **outer function** even after the outer function has completed execution.

---

3. Example of Closure

```
function outer() {
  let count = 0;  // Outer scope variable

  return function inner() {  // Inner function (closure)
    count++;
    console.log(` Count is: ${count}` );
  };
}

const counter = outer(); // outer() is called, returns inner()
counter(); // Output: Count is: 1
counter(); // Output: Count is: 2
counter(); // Output: Count is: 3
```

✅ The inner() function **remembers** the count variable even though outer() has already finished executing.
✅ This happens because inner() forms a **closure** — it "closes over" the count variable.

4. Why Do Closures Work?

- When outer() finishes executing, its **execution context** is removed from the call stack.
- However, since inner() still references count, JavaScript keeps count in memory — this is what makes closures possible.

5. Real-World Example of Closures

Creating a Private Variable (Encapsulation)

Closures are often used to create **private variables** — variables that can't be accessed directly from outside the function.

Example:

```
function createCounter() {
  let count = 0;  // Private variable

  return {
    increment: function() {
      count++;
      console.log(` Count: ${count}` );
    },
    decrement: function() {
      count--;
      console.log(` Count: ${count}` );
    },
    getCount: function() {
      return count;
    }
  };
}

const counter = createCounter();
counter.increment(); // Output: Count: 1
```

```
counter.increment(); // Output: Count: 2

counter.decrement(); // Output: Count: 1


console.log(counter.getCount()); // Output: 1

console.log(counter.count); // Undefined (count is private)
```

☑ *The count variable is private because it's inside the closure and cannot be accessed directly.*

---

6. Common Use Cases of Closures

✅ **Data Privacy** (e.g., creating private variables)
✅ **Event Handlers**
✅ **Callback Functions**
✅ **Functional Programming Concepts** (e.g., .map(), .filter(), .reduce())
✅ **Currying Functions**

---

7. Example of Closure in setTimeout()

Closures are crucial in asynchronous code like setTimeout().

```
function delayedMessage(message, delay) {
  setTimeout(function() {
    console.log(message);
  }, delay);
}


delayedMessage("Hello after 2 seconds", 2000); // Output after 2 seconds: Hello after 2 seconds
```

✅ The callback function **retains access** to the message variable because of the closure.

---

8. Closure Pitfall Example

Closures can sometimes cause unexpected behavior if not handled correctly.

Example: Unexpected Output

```
for (var i = 1; i <= 3; i++) {
  setTimeout(function() {
```

```
    console.log(` Count: ${i}` );
  }, 1000);
}
```

Output:

```
makefile
CopyEdit
Count: 4
Count: 4
Count: 4
```

Why?

- var is **function-scoped**, so the loop increments i to 4 before the setTimeout() callback executes.
- The callback references the **same i variable**, not its value at the time of iteration.

Solution Using let (Block Scope Fix)

```
for (let i = 1; i <= 3; i++) {
  setTimeout(function() {
    console.log(` Count: ${i}` );
  }, 1000);
}
```

Correct Output:

```
makefile
CopyEdit
Count: 1
Count: 2
Count: 3
```

✅ Using let creates a **new scope** for each loop iteration, fixing the issue.

---

9. Key Takeaways

✅ A **closure** is formed when an inner function retains access to its outer function's variables even after the outer function has finished executing.
✅ Closures are essential for **data privacy**, **callback functions**, and **async programming**.
✅ Use let in loops to avoid unexpected behavior caused by closure references.

Error Handling in JavaScript

Error handling is crucial in JavaScript to ensure your application runs smoothly and gracefully handles unexpected issues. The primary mechanism for error handling in JavaScript is the try...catch...finally block.

---

1. Syntax of try...catch...finally

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

---

2. Explanation

- **try block**:
  - Contains the code that may throw an error.
  - If no error occurs, the catch block is **skipped**.
- **catch block**:
  - Executes only if an error occurs inside the try block.
  - The error object contains details about the error.
- **finally block** (optional):
  - Executes **regardless of whether an error occurred or not**.
  - Used for cleanup tasks like closing files, clearing timers, etc.

---

3. Example: Basic try...catch

```
try {
  let result = 10 / 0;
  console.log(` Result: ${result}` );  // Infinity (No error)
} catch (error) {
  console.log(` Error: ${error.message}` );
```

```
} finally {
  console.log("This will run no matter what!");
}
```

Output:

```
yaml
CopyEdit
Result: Infinity
```

This will run no matter what!

✅ Since dividing by zero doesn't throw an error in JavaScript, the catch block is skipped.

---

4. Example: Handling Runtime Errors

```
try {
  let user = undefined;
  console.log(user.name);  // Error: Cannot read properties of undefined
} catch (error) {
  console.log(` Caught an error: ${error.message}` );
} finally {
  console.log("Execution completed.");
}
```

Output:

```
Caught an error: Cannot read properties of undefined (reading 'name')
Execution completed.
```

✅ The error is caught in the catch block, preventing the program from crashing.

---

5. Example: Using finally for Cleanup

The finally block is useful for cleanup tasks, like closing database connections or releasing resources.

```
function processData() {
  try {
    console.log("Processing data...");
    throw new Error("Unexpected error during processing!");
```

```
  } catch (error) {
    console.error(` Error: ${error.message}` );
  } finally {
    console.log("Cleaning up resources...");
  }
}

processData();
```

Output:

```
Processing data...
Error: Unexpected error during processing!
Cleaning up resources...
```

☑ The finally block runs even after an error occurs.

---

6. Example: Custom Error Handling

You can throw custom errors using the throw keyword.

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Division by zero is not allowed.");
  }
  return a / b;
}

try {
  console.log(divide(10, 2)); // Output: 5
  console.log(divide(10, 0)); // Throws custom error
} catch (error) {
  console.error(` Caught Error: ${error.message}` );
} finally {
  console.log("End of calculation.");
}
```

Output:

5

Caught Error: Division by zero is not allowed.

End of calculation.

☑ *Custom errors make debugging easier by providing clear, meaningful messages*.

---

7. Example: Using catch Without error Parameter

In modern JavaScript (ES10+), you can use catch without the error parameter if you don't need to reference the error.

```
try {
  throw new Error("Oops! Something went wrong.");
} catch {
  console.log("An error occurred, but no error details needed.");
}
```

---

8. Best Practices for Error Handling

✅ **Use specific error types** for better clarity:

- TypeError, ReferenceError, SyntaxError, etc.
  - ✅ **Avoid generic error messages**; provide meaningful feedback.
  - ✅ **Use finally** for cleanup logic that must run regardless of success or failure.
  - ✅ **Don't rely solely on try...catch**; ensure data validation and checks are in place.

---

9. Real-World Scenario: API Request with Error Handling

```
async function fetchData() {
  try {
    let response = await fetch("https://jsonplaceholder.typicode.com/users");
    if (!response.ok) {
      throw new Error(` HTTP Error! Status: ${response.status}` );
    }
    let data = await response.json();
    console.log("Data fetched successfully:", data);
  } catch (error) {
```

```
    console.error(` Error fetching data: ${error.message}` );
  } finally {
    console.log("Fetch attempt complete.");
  }
}

fetchData();
```

☑ *Ensures network issues, failed responses, or unexpected data structures are handled gracefully.*

---

10. Key Takeaways

- ✅ Use try...catch for code blocks that might throw errors.
- ✅ The catch block handles errors gracefully.
- ✅ The finally block **always executes**, making it ideal for cleanup tasks.
- ✅ Use meaningful error messages for better debugging.

Types of Errors in JavaScript

Error Type	Cause	Example
SyntaxError	Invalid syntax	console.log("Hello"
ReferenceError	Using undefined variables	console.log(x);
TypeError	Invalid data type operation	"text".push()
RangeError	Out of range values	new Array(-1)
URIError	Incorrect URI encoding/decoding	decodeURI("%")
AggregateError	Multiple errors in Promises	Promise.any([...])
InternalError	Excessive recursion or resource limit	function recurse() { recurse(); }