## Advanced JavaScript & TypeScript

- The most distinguishing aspect of advanced TypeScript is its powerful type system, which enables developers to write safer and more maintainable code for large projects:
- Object-Oriented Programming (OOP) in JavaScript: In-depth understanding of inheritance, the prototype chain, function constructors, and the distinctions between primitives and objects.
- Asynchronous Patterns: Mastery of callbacks, Promises, Async/Await syntax, the Event Loop, and using the Fetch API for robust web requests.
- Modern JavaScript (ES6+): Proficiency with let, const, arrow functions, template literals, destructuring, modules, and classes.

- TypeScript is a superset of JavaScript that adds extra features like static typing, interfaces, enums, and more. Essentially, TypeScript is JavaScript with additional syntax for defining types, making it a powerful tool for building scalable and maintainable applications.
- Static typing allows you to define variable types and helps catch errors before running the code.
- TypeScript is compiled into JavaScript, ensuring it works in all JavaScript environments.
- It's widely used for web development on both the client and server sides.

## ES6+ mastery, functional programming, design patterns

- **ES6+ mastery**
  ES6 (ECMAScript) powerful features that make JavaScript more modern, readable, and powerful for real projects.

- **Features:**
- **Let and Const:** let (Modern JavaScript)

  ➢ Features

  Block scoped

  Cannot be redeclared

  Can be reassigned

  Hoisted but not initialized

Safe & recommended **Example**
let b = 10; b = 20; //
allowed console.log(b); //
20

- **const (Constant Variable)**
- **Features**

Block scoped
Must be initialized
Cannot be redeclared or reassigned
Hoisted but in TDZ
**Example** const PI = 3.14;
console.log(PI);

- **Hoisting (Very Important Concept)**

JavaScript moves **declaration** to the top of scope before execution.

- **var Hoisting** console.log(a); var a = 10; **Output:** undefined Because:

var a; console.log(a);
a = 10;

- **Arrow Functions (=>)**
- **What is an Arrow Function?**

Arrow function is a **shorter syntax** to write functions in JavaScript. Introduced in **ES6**, it makes code **cleaner and readable**.

**Normal Function** function add(a,
b) { return a + b;
}
console.log(add(10, 20)); // 30 **Arrow**
**Function** const add = (a, b) => a + b;
console.log(add(10, 20)); // 30 **Template**
**Literals (`)**

Template literals allow **string interpolation** and **multi-line strings** using backticks ( ` ).

**Old Way vs Template Literal Old Way**

```
let name = "Shivam"; let age = 22;

console.log("My name is " + name + " and age is " + age);
```

**New Way (Template Literal)**

```
let name
= "Shivam"; let age = 22;

console.log(`My name is ${name} and age is ${age}`);
```

**Destructuring**

Destructuring allows you to **extract values from arrays or objects** into variables easily.

**Array Destructuring Basic**

**Example**

```
let arr = [10, 20, 30];

let [a, b, c] = arr;

console.log(a); // 10 console.log(b); // 20
console.log(c); // 30
```

**Spread Operator (...)**

Spread **expands** an array or object into individual elements.

**Spread with Arrays Copy Array**

```
let a = [1, 2, 3]; let b =
[...a];

console.log(b); // [1,2,3]
```

**Rest Operator (...)**

📌 **What is Rest?**

Rest **collects multiple values** into a single array.

◇ **Rest in Function Parameters**

```
function total(...numbers) {
return numbers.reduce((sum, n) => sum + n, 0);
}
```

```
console.log(total(10, 20, 30)); // 60
```

- **Functional Programming (FP)**

Functional Programming is a **Declarative** paradigm where you treat programs as a sequence of function evaluations rather than a list of commands.

**features of FP:**

- **Pure Functions:** * Given the same input, they always return the same output.
    - They have **zero side effects** (don't change global variables, don't log to console, don't modify the DOM).
- **Immutability:** Instead of changing an existing array or object (mutation), you create a new one with the updated values. This is the heart of frameworks like React.
- **Example**

```
let arr = [1, 2, 3]; let newArr
= [...arr, 4];

console.log(arr);    // [1,2,3] console.log(newArr); // [1,2,3,4]
```

- **Higher-Order Functions (HOF):** Functions that take other functions as arguments or return them.
    - Map: Transforms every element.
    - Filter: Selects elements based on a condition. o Reduce: Aggregates an array into a single value (the most powerful HOF).

- Example

```
function greet(name) {     return `Hello ${name}`;
}
```

```
function processUser(fn, value) {     return fn(value);}

console.log(processUser(greet, "Shivam"));

// Hello Shivam
```

- **Function Composition:** Combining small, simple functions to build complex logic.

f(g(x))

- **Currying:** Breaking down a function that takes multiple arguments into a series of functions that take one argument at a time.

- **<u>Design Patterns</u>**

  Design patterns are reusable solutions to common software problems. In JavaScript, they help in managing complexity and memory efficiency.

- **Category 1: Creational Patterns**
- **Singleton:** Ensures a class has only one instance (e.g., a Global State Store or a Database Connection).

- **Factory Pattern:** Uses a function to create objects without specifying the exact class. Useful when the object creation logic is complex.
- **Category 2: Structural Patterns**
- **Module Pattern:** Encapsulating private variables and exposing only a public API. Modern ESM has largely replaced the old "IIFE" module pattern.
- **Proxy Pattern:** Intercepting and redefining fundamental operations for an object (e.g., logging every time a property is accessed).
- **Category 3: Behavioral Patterns**
- **Observer Pattern:** A "subject" maintains a list of "observers" and notifies them of state changes. This is how **Event Listeners** and **RxJS** work.