**DESIGN PATTERNS AND PRINCIPLES**

**EXICERCISE 1: Implementing the Singleton Pattern**

### Objective:

Ensure only one instance of a class exists throughout the application and provide a global point of access to it.

**Steps:**

* Define a class with a private static field of its own type.
* Make the constructor private.
* Provide a static method GetInstance() that returns the instance.

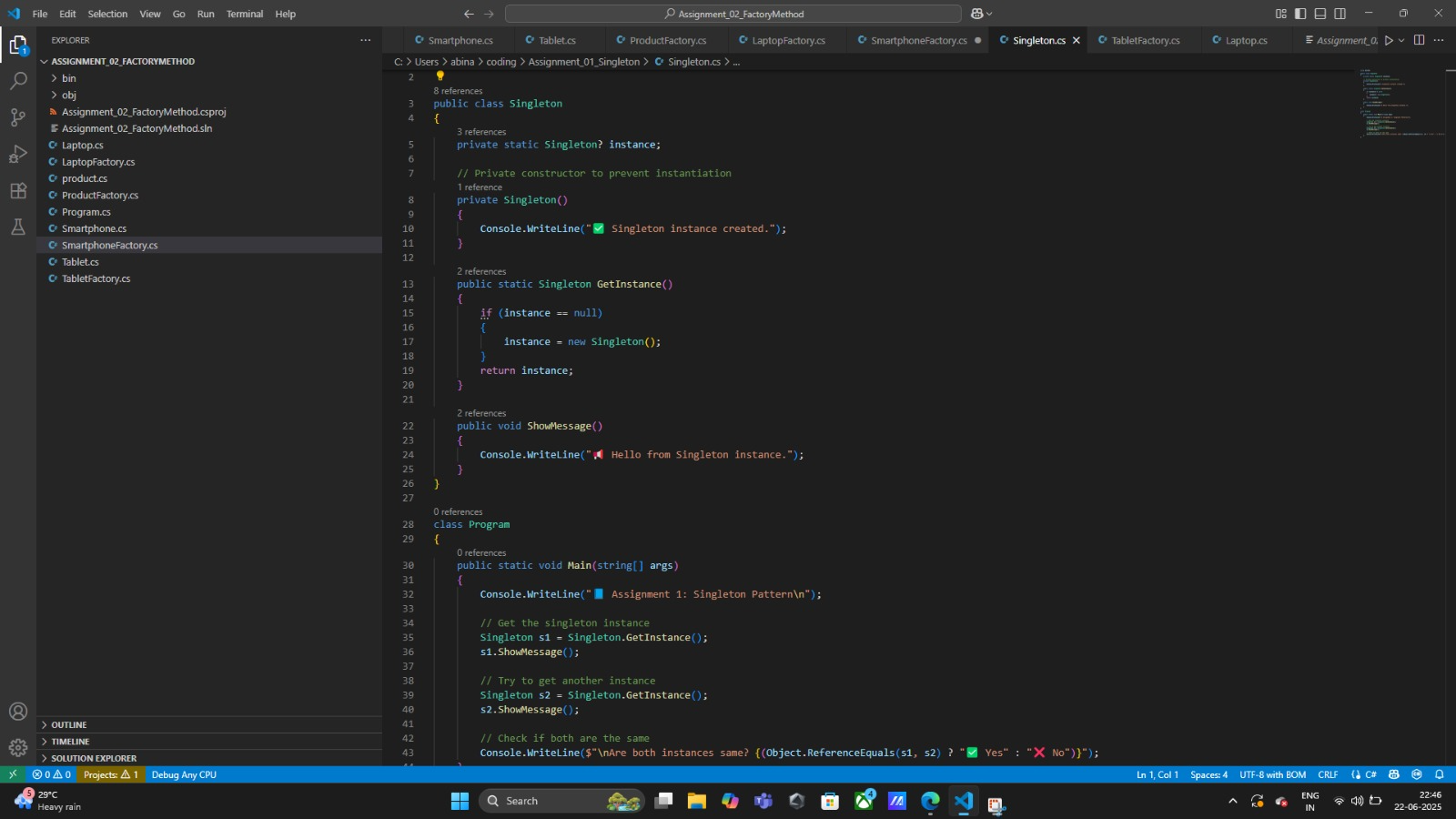
### Concept:

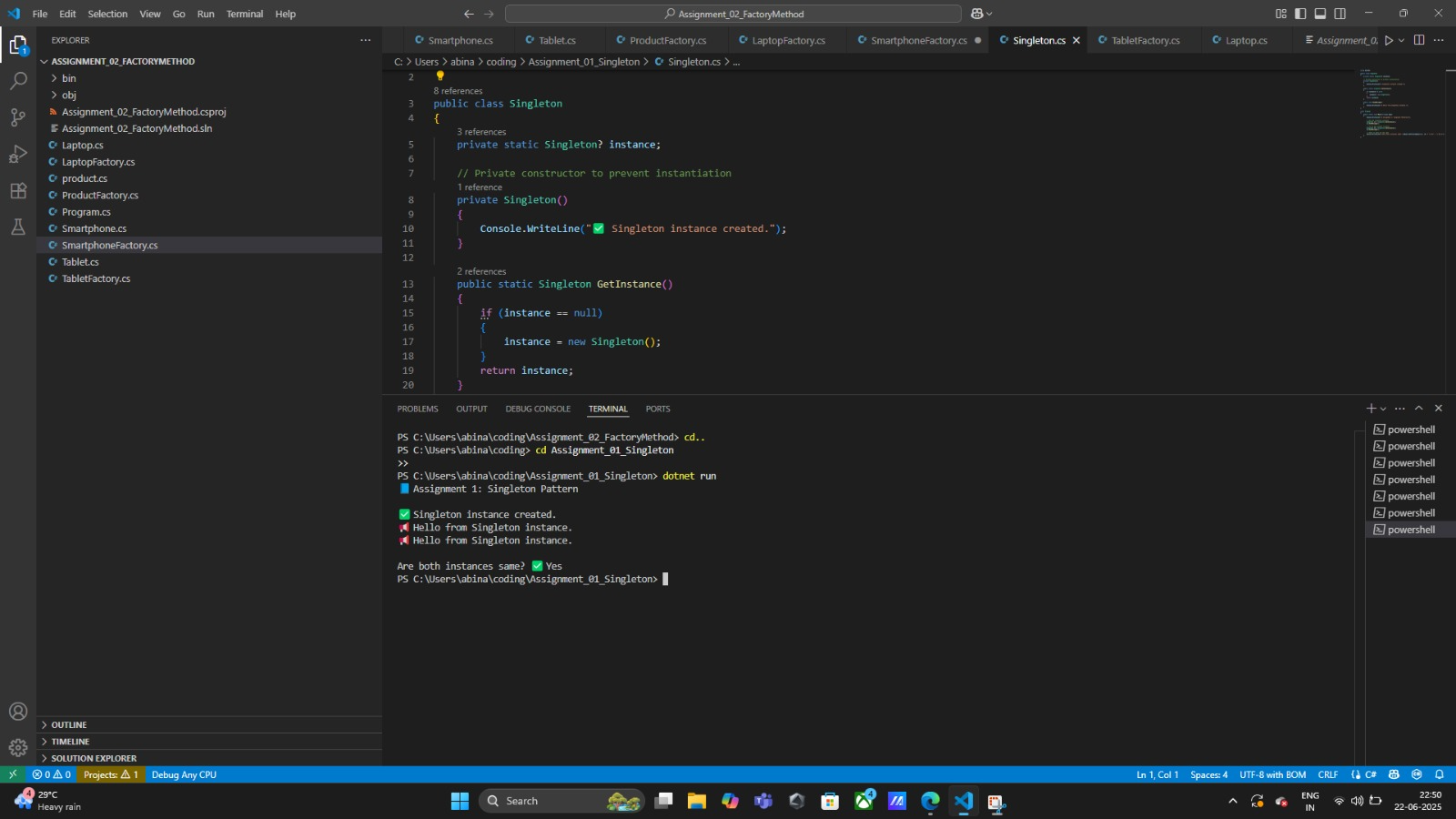
Used for logging, config managers, DB connection pools where multiple instances are unnecessary or dangerous.

**Final Output:**

Prints only one creation message regardless of multiple instance requests.

**Implementation:**





## ****EXERCISE 2: Implementing the Factory Method Pattern****

**Objective:**  
Create objects without exposing the exact class that needs to be instantiated. Delegate object creation to subclasses or factories.

**Steps:**

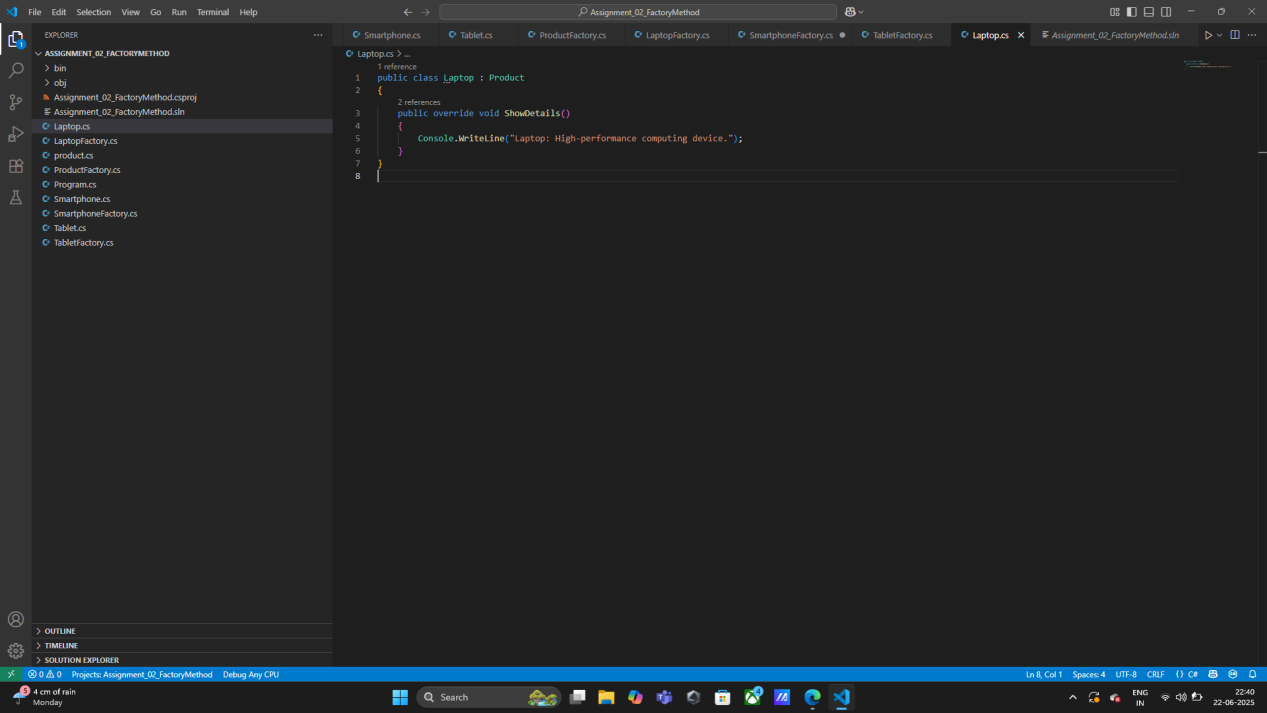
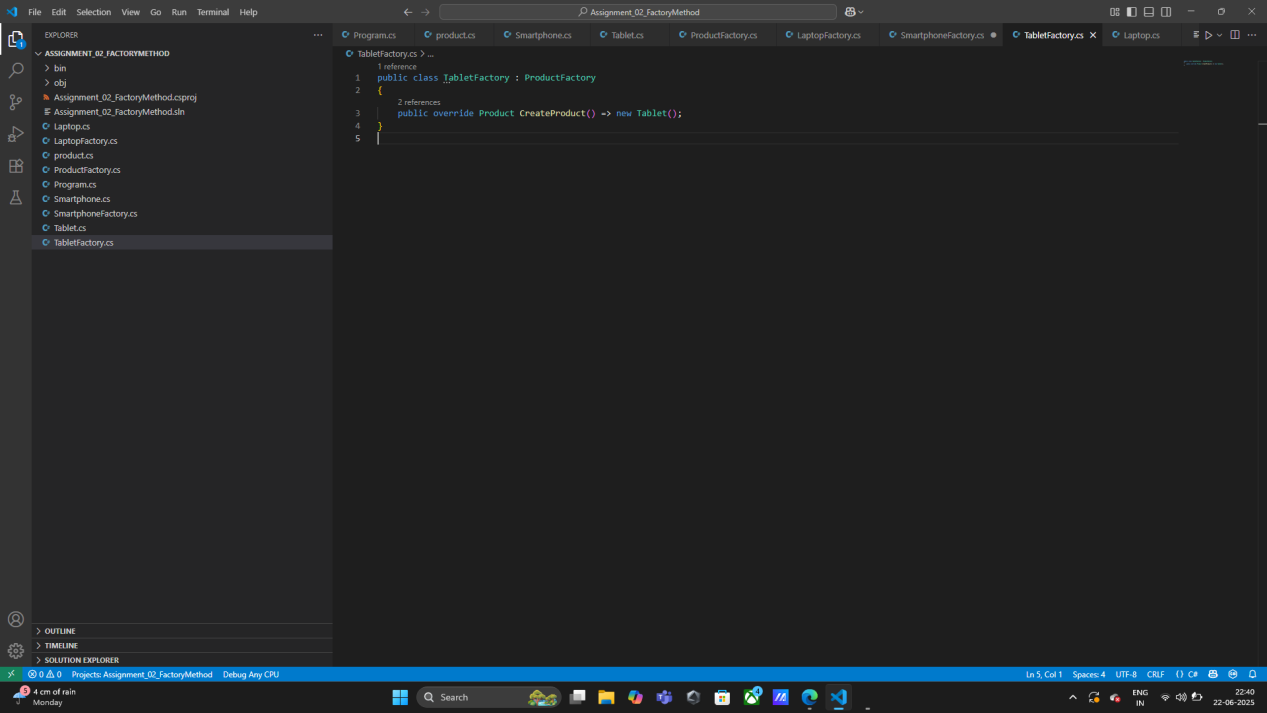
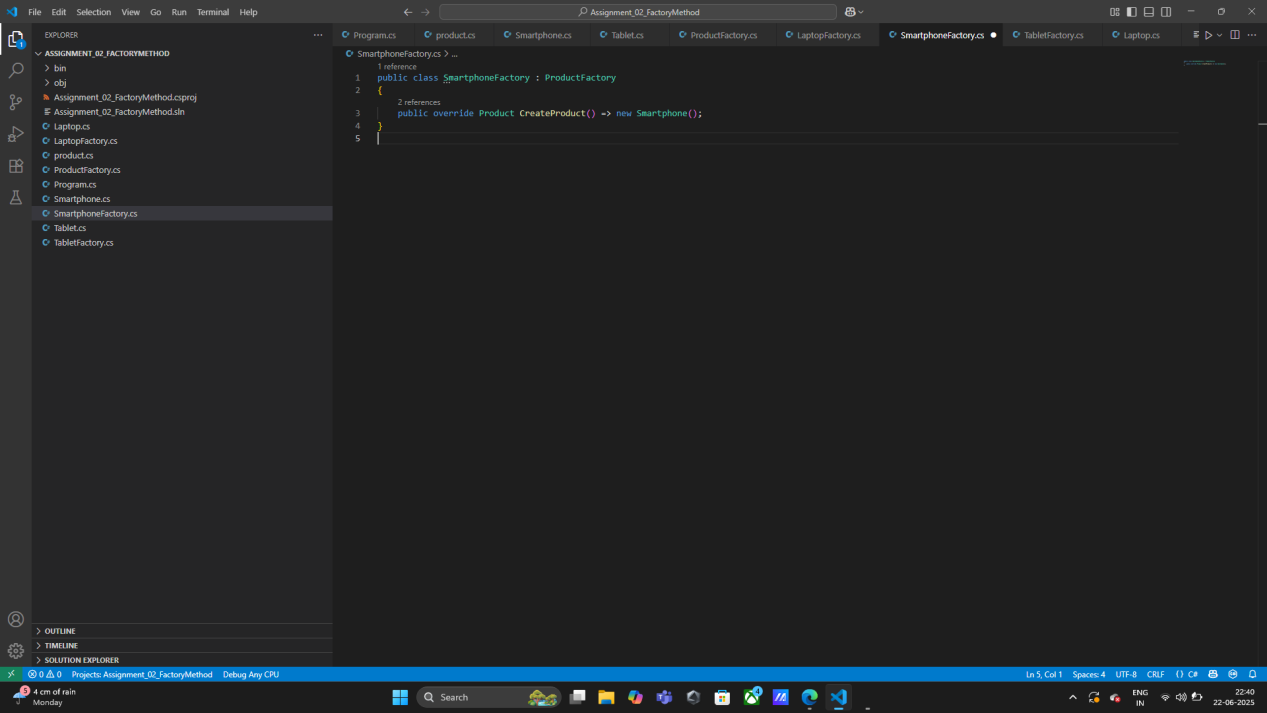
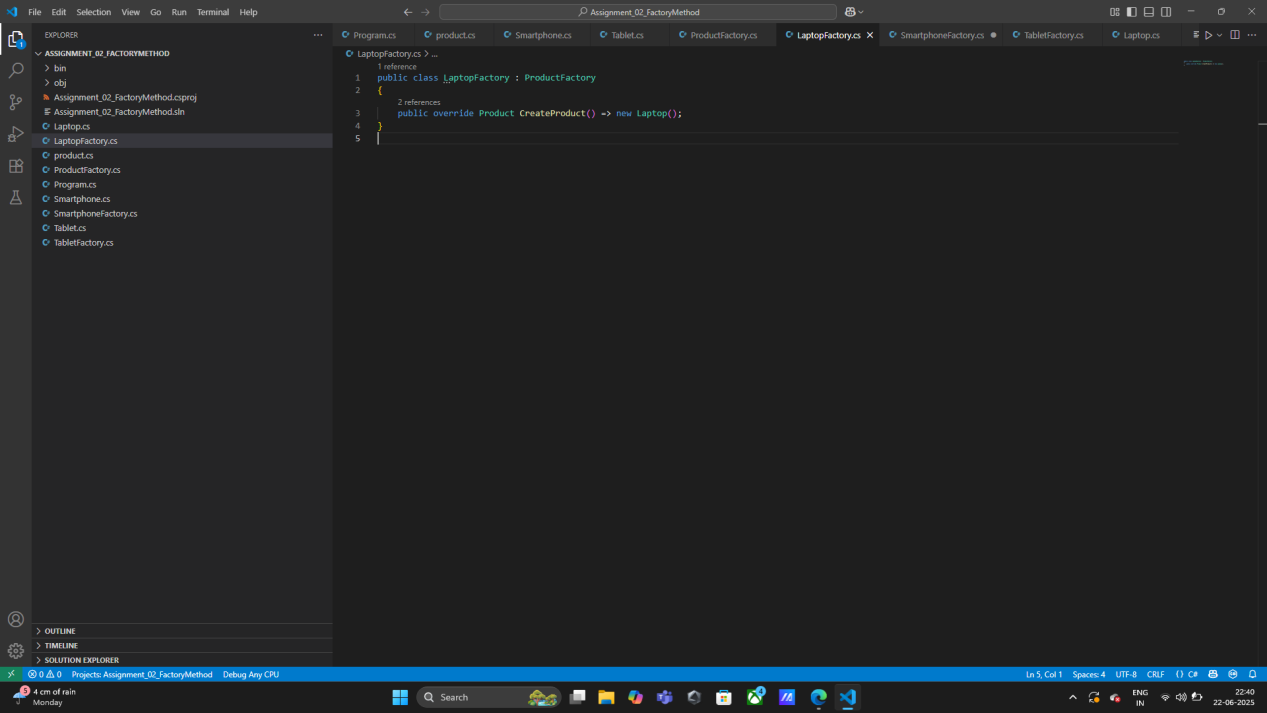
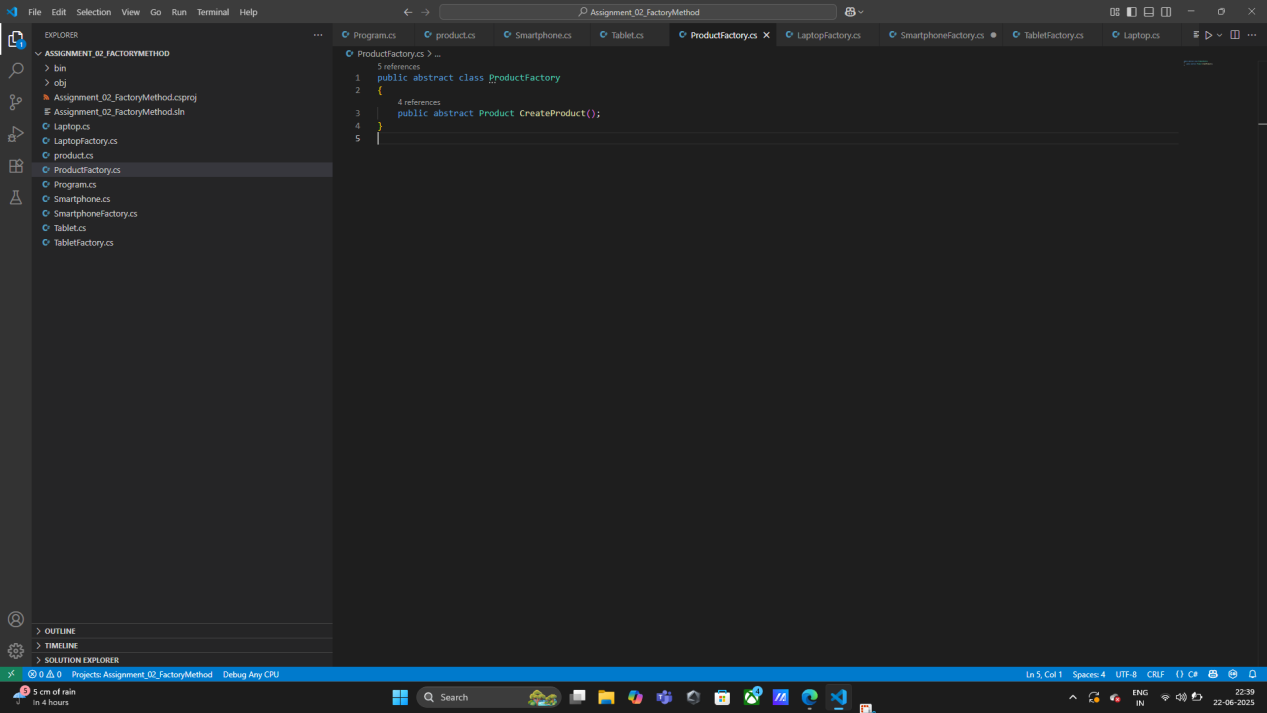
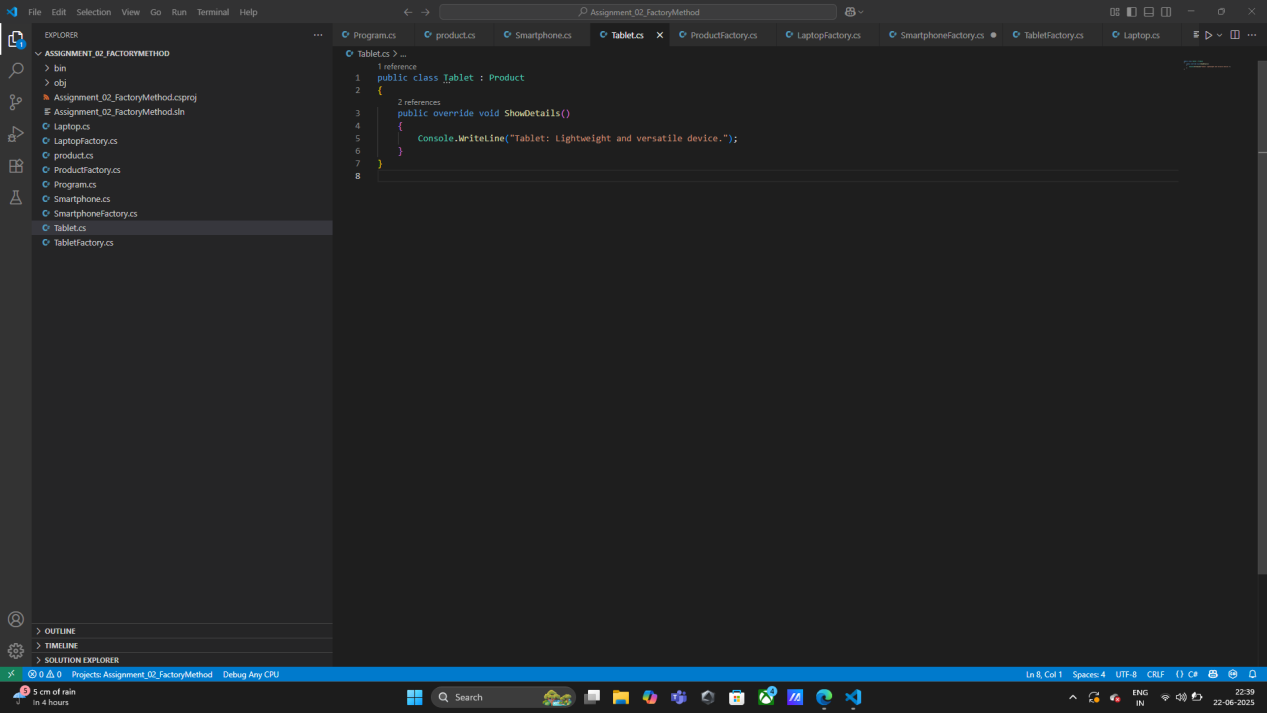
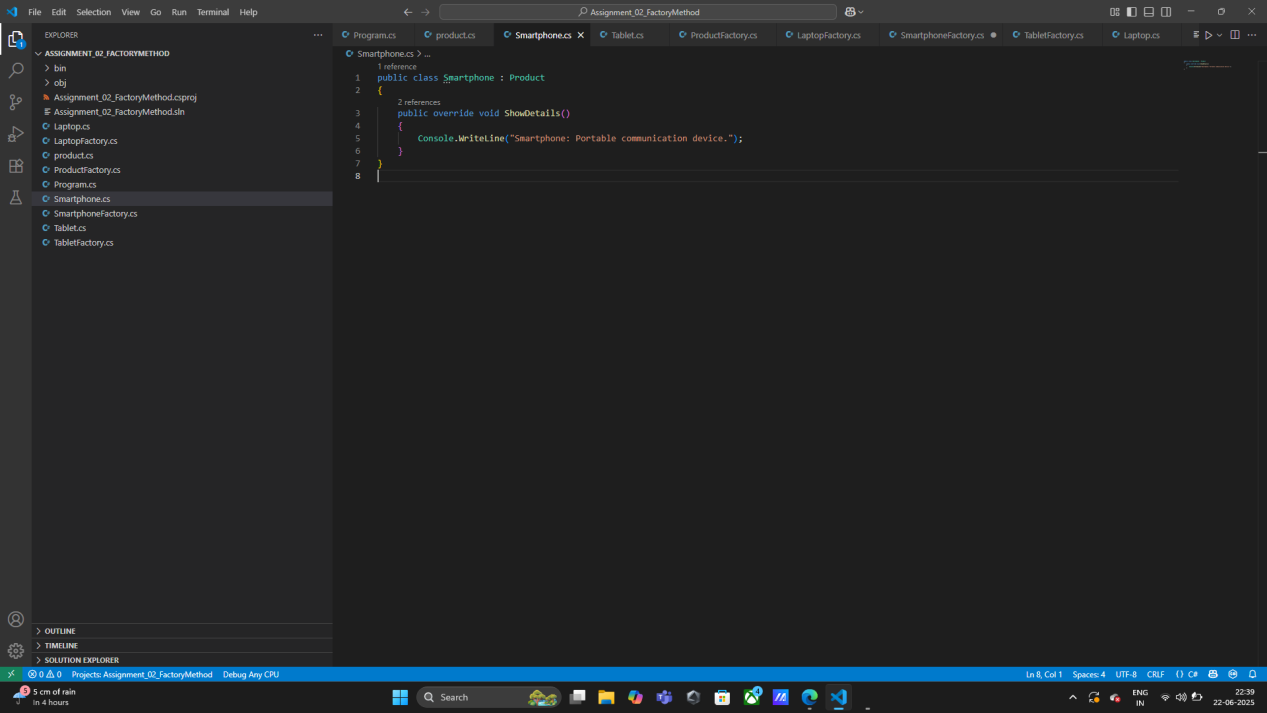
* Define an abstract product class (e.g., Product).
* Create concrete product classes (e.g., Laptop, Smartphone, Tablet).
* Define an abstract factory class with a method to create products.
* Implement concrete factory classes for each product type.
* In the main program, accept user input and use the appropriate factory to create and return the correct object.

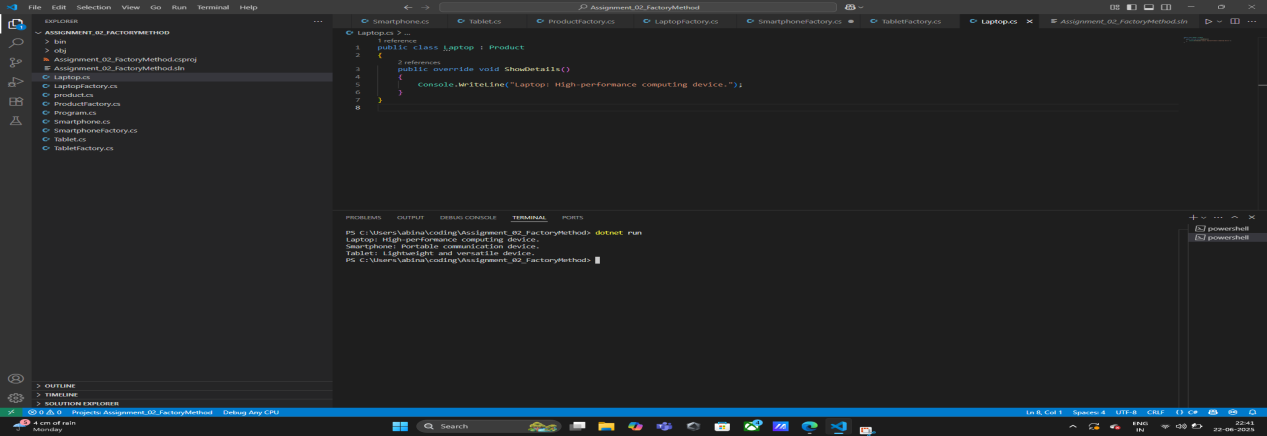
**Concept:**  
Factory Method Pattern supports the **Open/Closed Principle** — it allows adding new product types without modifying the existing code. This pattern is ideal when the exact type of object to be created is determined at runtime.

**Final Output:**  
Displays the selected product’s creation message using polymorphism. Demonstrates that the program can create different product objects based on user input while remaining decoupled from concrete classes.

**Implementation:**

* Abstract Class: Product
* Concrete Products: Laptop, Smartphone, Tablet
* Abstract Factory: ProductFactory
* Concrete Factories: LaptopFactory, SmartphoneFactory, TabletFactory
* Entry Point: Program.cs with dynamic object creation





## DATA STRUCTURES & ALGORITHM

## Exercise 3: E-Commerce Search

### Objective:

Build a search function for an e-commerce product list using C# collections.

**Steps:**

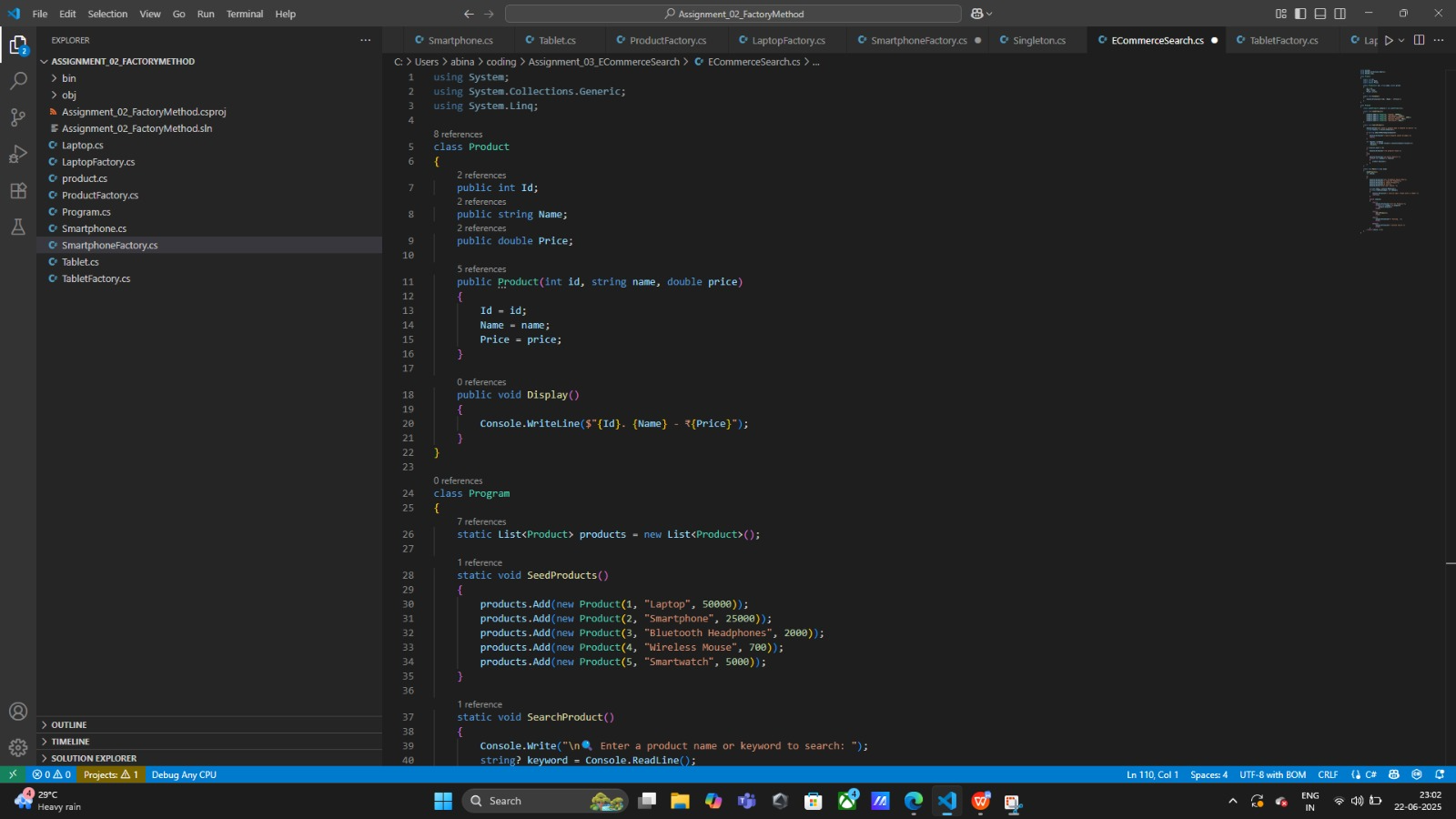
* Create a Product class with Id, Name, and Price.
* Use List<Product> to hold a catalog.
* Accept user input, use .Where() with Contains() to filter search results.

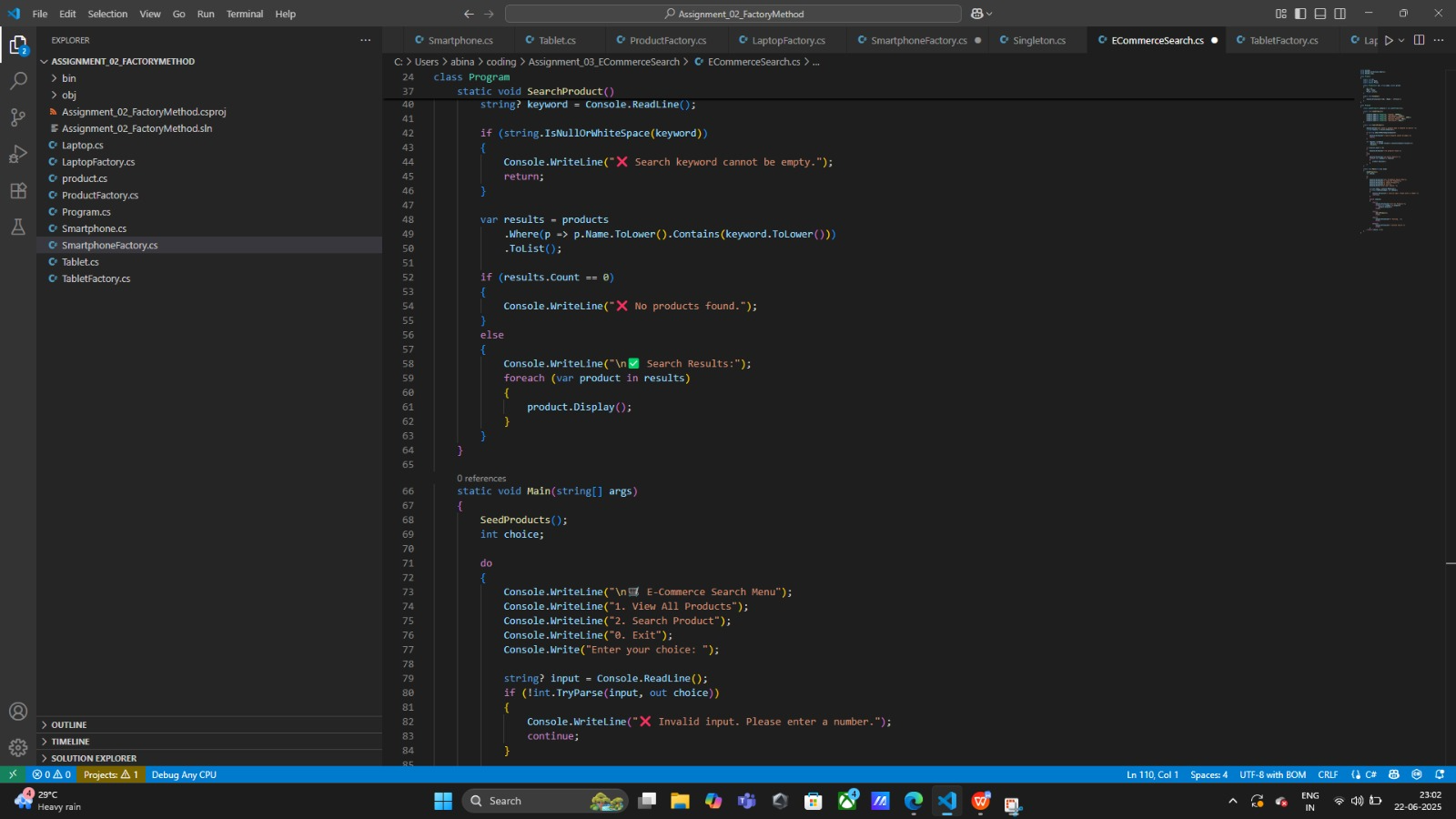
### Concept:

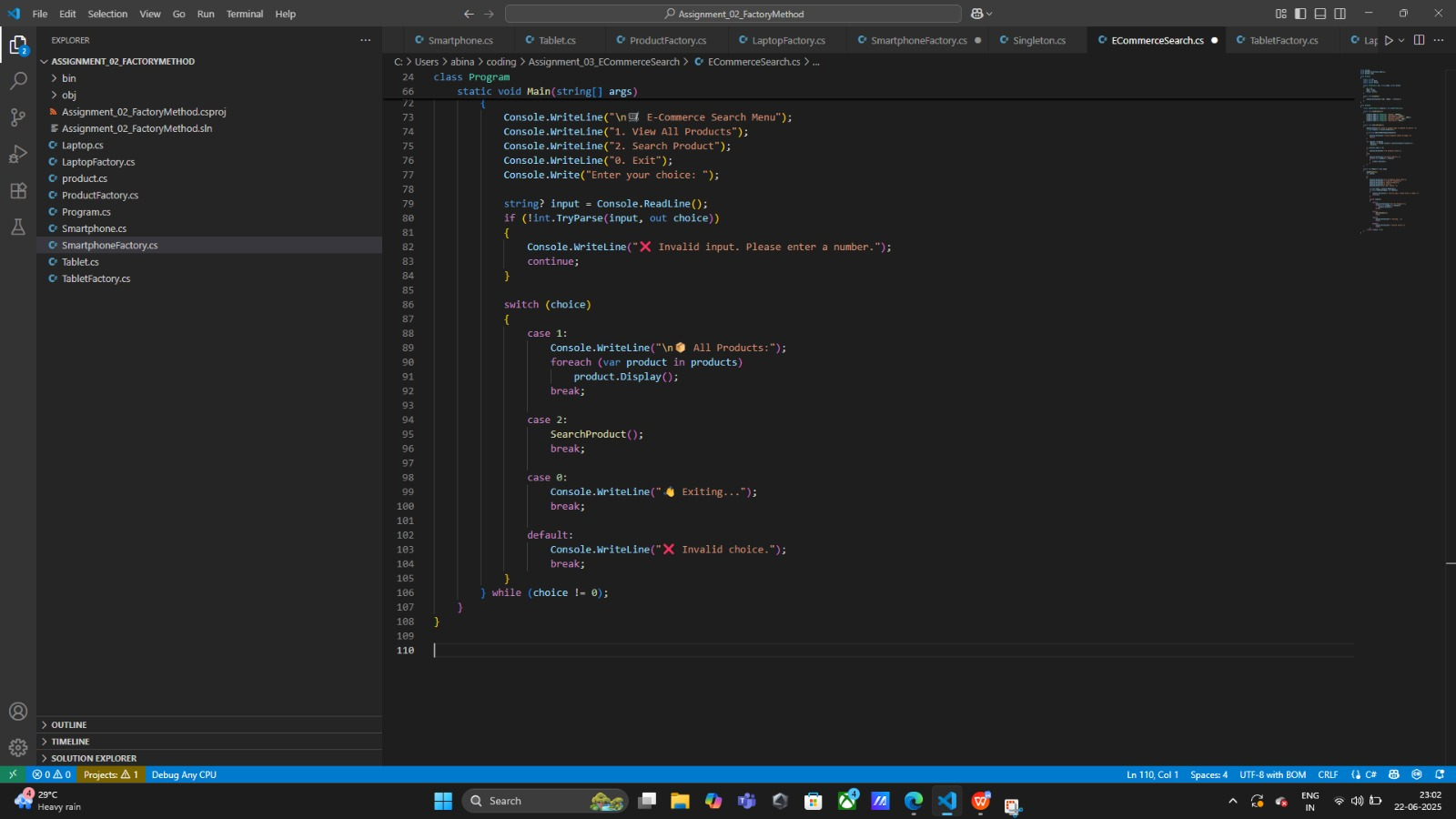
Demonstrates List, user input validation, filtering using LINQ.

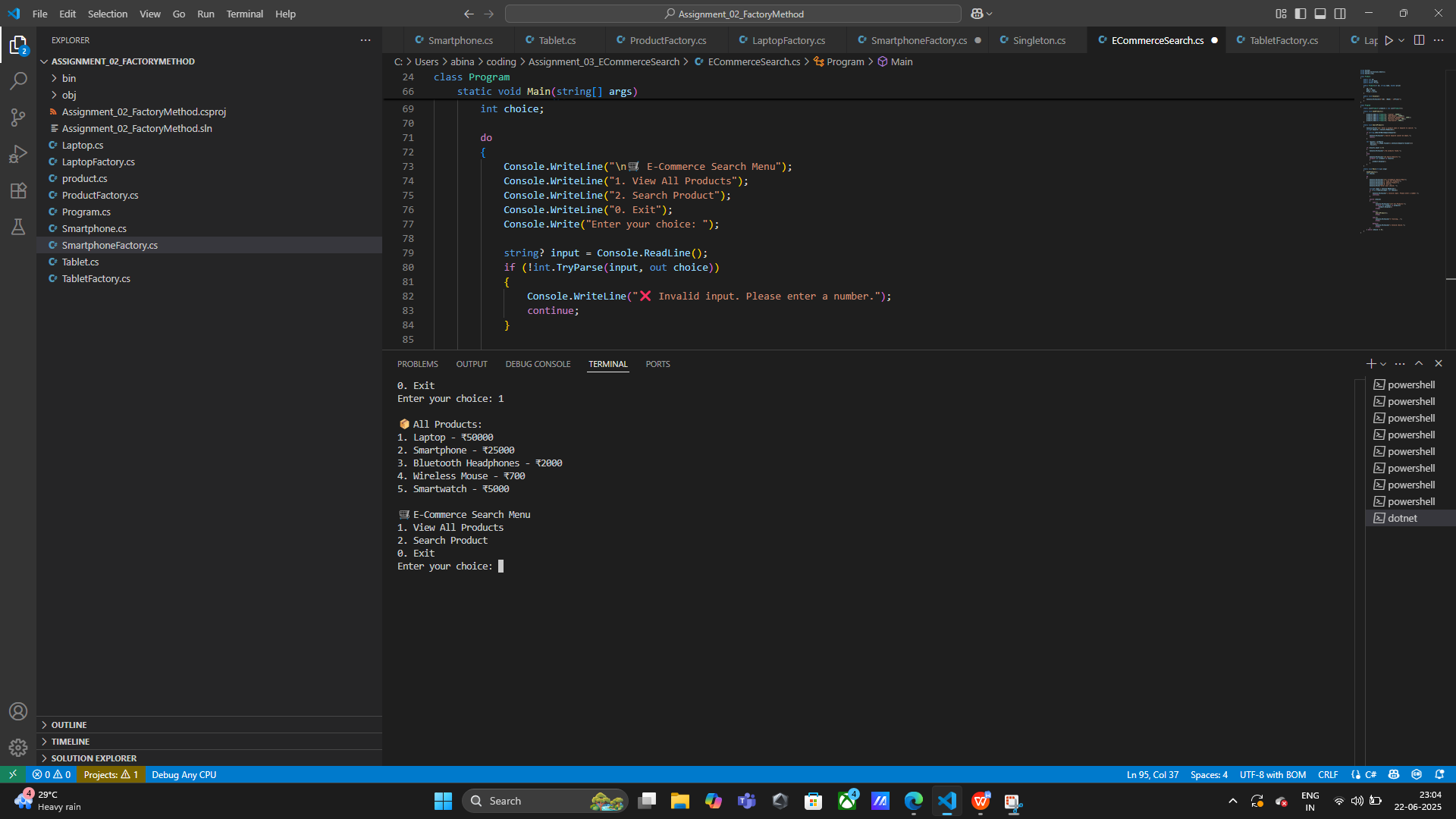
**Final Output:**

Searches by keyword and prints matching product names and prices.









## Algorithm Analysis

### Linear Search

**Used In:** E-Commerce keyword match  
**Steps:**

Loop through each product

Compare name with input string

| **Case** | **Time Complexity** |
| --- | --- |
| Best Case | O(1) |
| Worst Case | O(n) |

### Binary Search (if list is sorted)

**Example (not in this project but extendable):**

csharp

CopyEdit

int BinarySearch(int[] arr, int target) {

int left = 0, right = arr.Length - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (arr[mid] == target) return mid;

else if (arr[mid] < target) left = mid + 1;

else right = mid - 1;

}

return -1;

}

| **Case** | **Time Complexity** |
| --- | --- |
| Best Case | O(1) |
| Worst Case | O(log n) |

## EXERCISE4: Financial Forecasting (DSA & Math Logic)

### Objective:

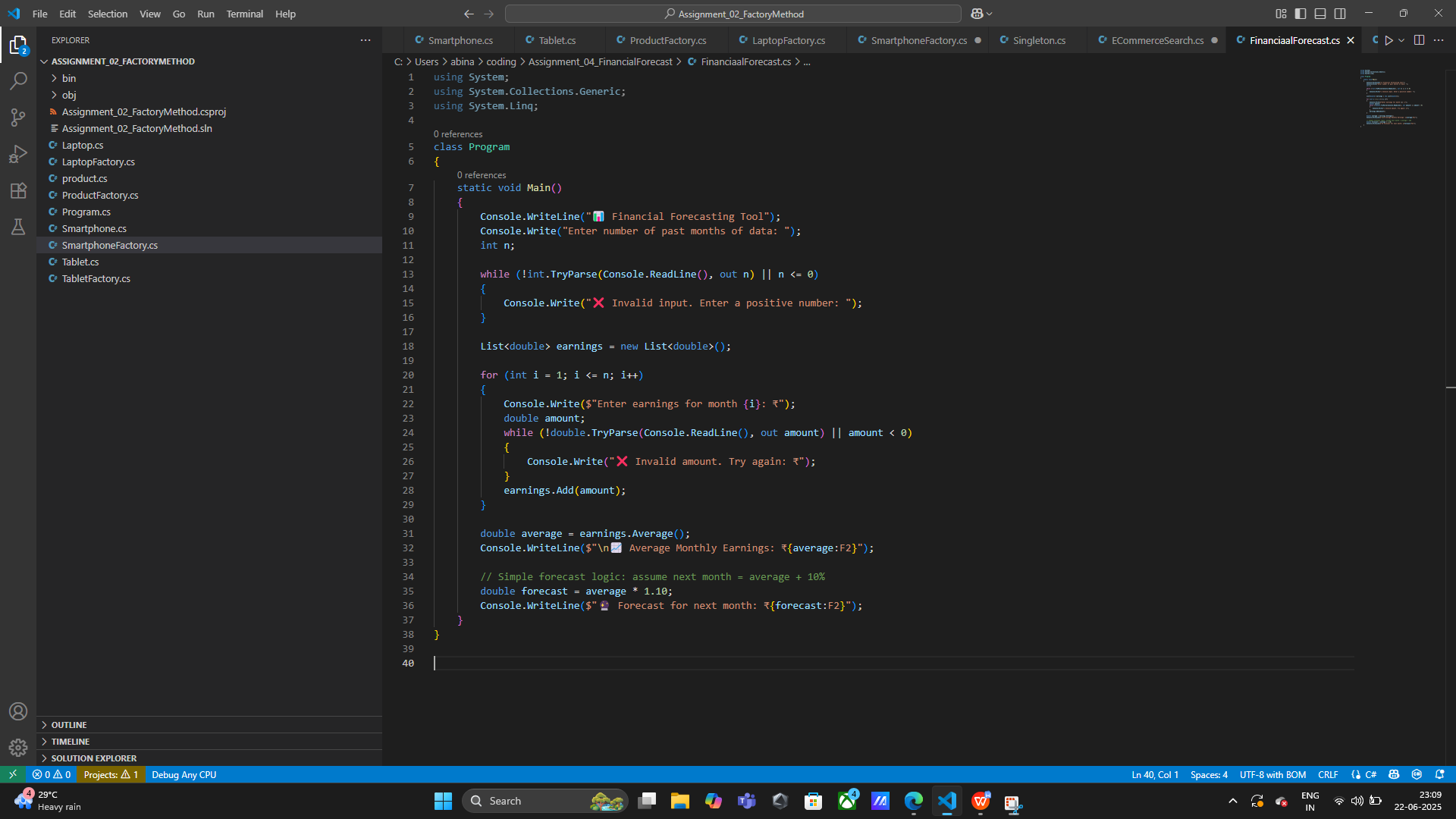
Build a C# console app that forecasts future monthly earnings based on past N months' input.

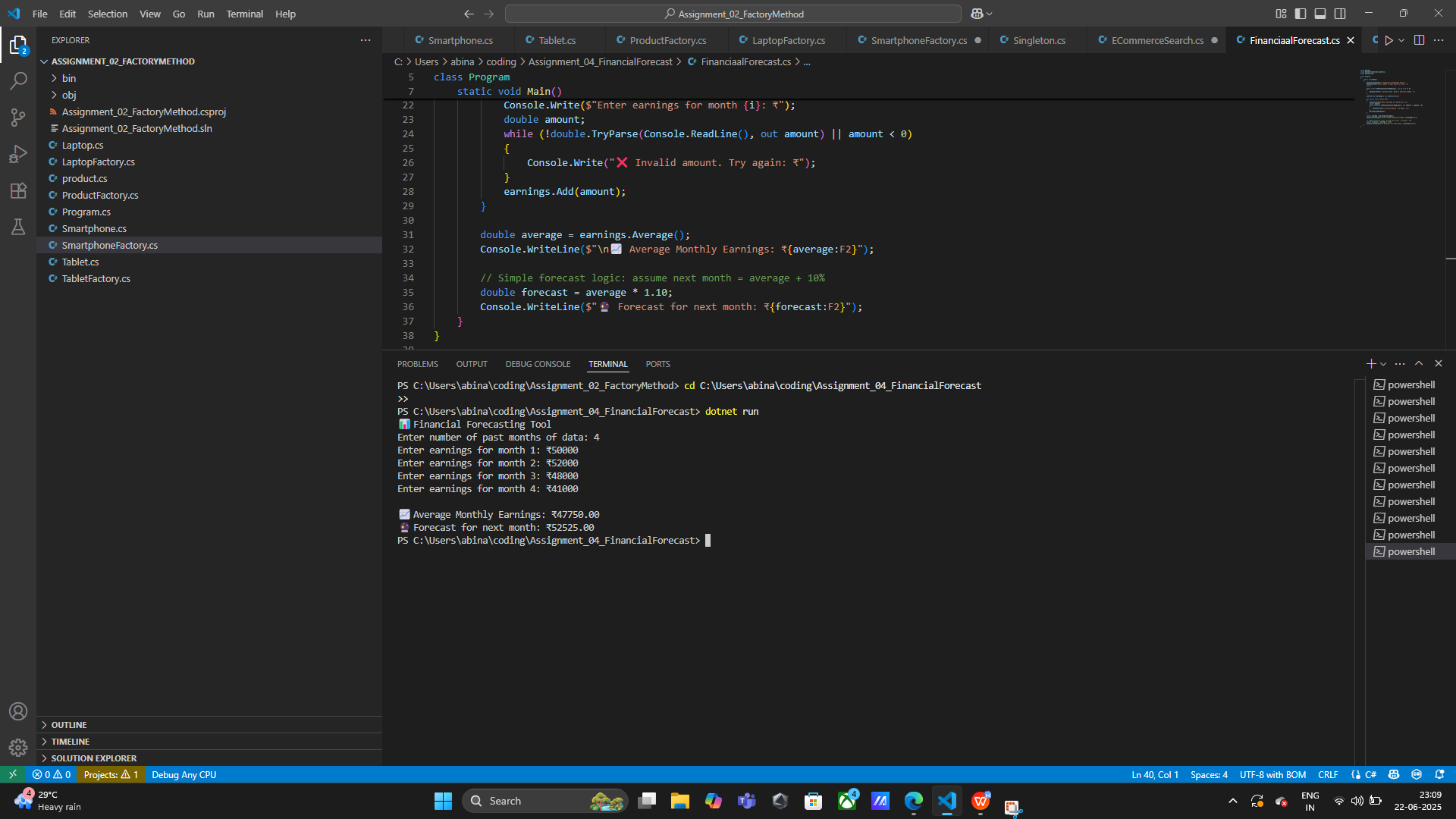
### Steps:

* Accept integer input for the number of months.
* Accept earnings for each month using a loops
* Compute average, then project next month's earnings using +10%.

### Concept:

Practices lists, math functions, input validation with TryParse, Average().





|  |
| --- |
| ) |

**OPTIMIZED RECURSIVE USE**

**Used in Financial Forecasting / Algorithms:**

#### Optimization Strategies:

* Use memoization to store intermediate results (in recursive search/forecast logic).
* Use loops when recursion isn’t needed (to avoid stack overflow).