**MODULE 1: Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Solution:**

**Code 1: Logger.cs:**

using System;

public class Logger

{

private static Logger instance = null;

private static readonly object padlock = new object();

// Private constructor

private Logger()

{

Console.WriteLine("Logger Initialized.");

}

// Public static method to get the instance

public static Logger GetInstance()

{

// Thread-safe Singleton implementation

if (instance == null)

{

lock (padlock)

{

if (instance == null)

{

instance = new Logger();

}

}

}

return instance;

}

public void Log(string message)

{

Console.WriteLine($"[LOG]: {message}");

}

}

**Code 2: Program.cs:**

using System;

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Testing Singleton Logger...\n");

Logger logger1 = Logger.GetInstance();

logger1.Log("First log message");

Logger logger2 = Logger.GetInstance();

logger2.Log("Second log message");

// Check if both loggers are the same instance

if (logger1 == logger2)

{

Console.WriteLine("\n Both logger instances are the same.");

}

else

{

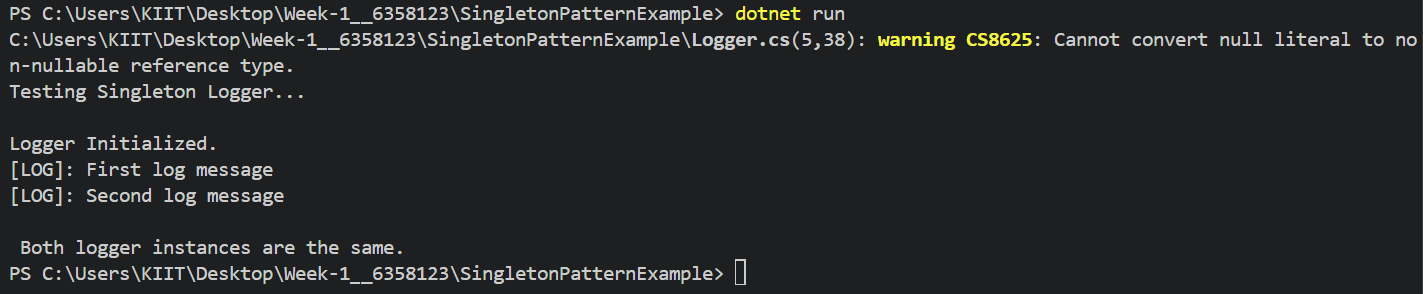
Console.WriteLine("\n Logger instances are different!");

}

}

}

**Output:**



**Exercise 2: Implementing the Factory Method Pattern**

**Solution:**

**Code 1: IDocument.cs**

public interface IDocument

{

void Open();

}

**Code 2: WordDocument.cs**

using System;

public class WordDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening a Word document...");

}

}

**Code 3: PdfDocument.cs**

using System;

public class PdfDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening a PDF document...");

}

}

**Code 4: ExcelDocument.cs**

using System;

public class ExcelDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening an Excel document...");

}

}

**Code 5: DocumentFactory.cs**

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

**Code 6: WordDocumentFactory.cs**

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

**Code 7: PdfDocumentFactory.cs**

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

**Code 8: ExcelDocumentFactory.cs**

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

**Code 9: Program.cs**

using System;

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Factory Method Pattern Demo:\n");

DocumentFactory wordFactory = new WordDocumentFactory();

IDocument wordDoc = wordFactory.CreateDocument();

wordDoc.Open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

IDocument pdfDoc = pdfFactory.CreateDocument();

pdfDoc.Open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

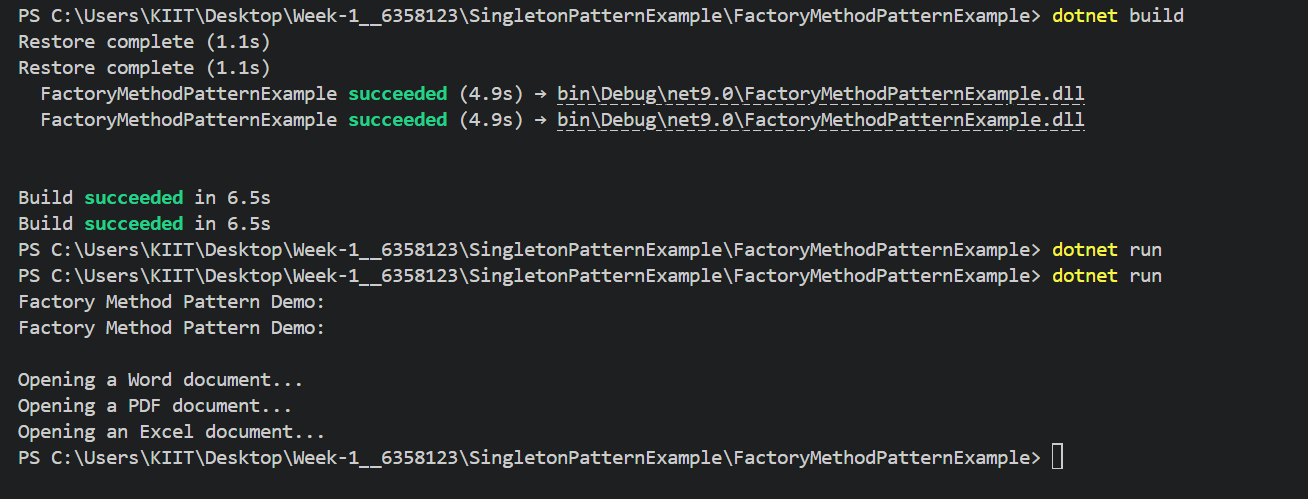
IDocument excelDoc = excelFactory.CreateDocument();

excelDoc.Open();

}

}

**Output:**



**MODULE 2: Data Structures and Algorithms**

**Exercise 1: Inventory Management System**

**Solution:**

**Code 1: Product.cs**

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public int Quantity { get; set; }

public double Price { get; set; }

public Product(int productId, string productName, int quantity, double price)

{

ProductId = productId;

ProductName = productName;

Quantity = quantity;

Price = price;

}

public override string ToString()

{

return $"ID: {ProductId}, Name: {ProductName}, Qty: {Quantity}, Price: {Price:C}";

}

}

**Code 2: InventoryManager.cs**

using System;

using System.Collections.Generic;

public class InventoryManager

{

private Dictionary<int, Product> inventory = new Dictionary<int, Product>();

public void AddProduct(Product product)

{

if (!inventory.ContainsKey(product.ProductId))

{

inventory[product.ProductId] = product;

Console.WriteLine("Product added.");

}

else

{

Console.WriteLine("Product with the same ID already exists.");

}

}

public void UpdateProduct(int productId, int quantity, double price)

{

if (inventory.ContainsKey(productId))

{

inventory[productId].Quantity = quantity;

inventory[productId].Price = price;

Console.WriteLine("Product updated.");

}

else

{

Console.WriteLine("Product not found.");

}

}

public void DeleteProduct(int productId)

{

if (inventory.Remove(productId))

{

Console.WriteLine("Product deleted.");

}

else

{

Console.WriteLine("Product not found.");

}

}

public void DisplayInventory()

{

Console.WriteLine("\nCurrent Inventory:");

foreach (var product in inventory.Values)

{

Console.WriteLine(product);

}

Console.WriteLine();

}

}

**Code 3: Program.cs**

using System;

class Program

{

static void Main(string[] args)

{

InventoryManager manager = new InventoryManager();

Product p1 = new Product(101, "Mouse", 50, 499.99);

Product p2 = new Product(102, "Keyboard", 30, 899.50);

Product p3 = new Product(103, "Monitor", 20, 6999.00);

manager.AddProduct(p1);

manager.AddProduct(p2);

manager.AddProduct(p3);

manager.DisplayInventory();

manager.UpdateProduct(101, 45, 479.00);

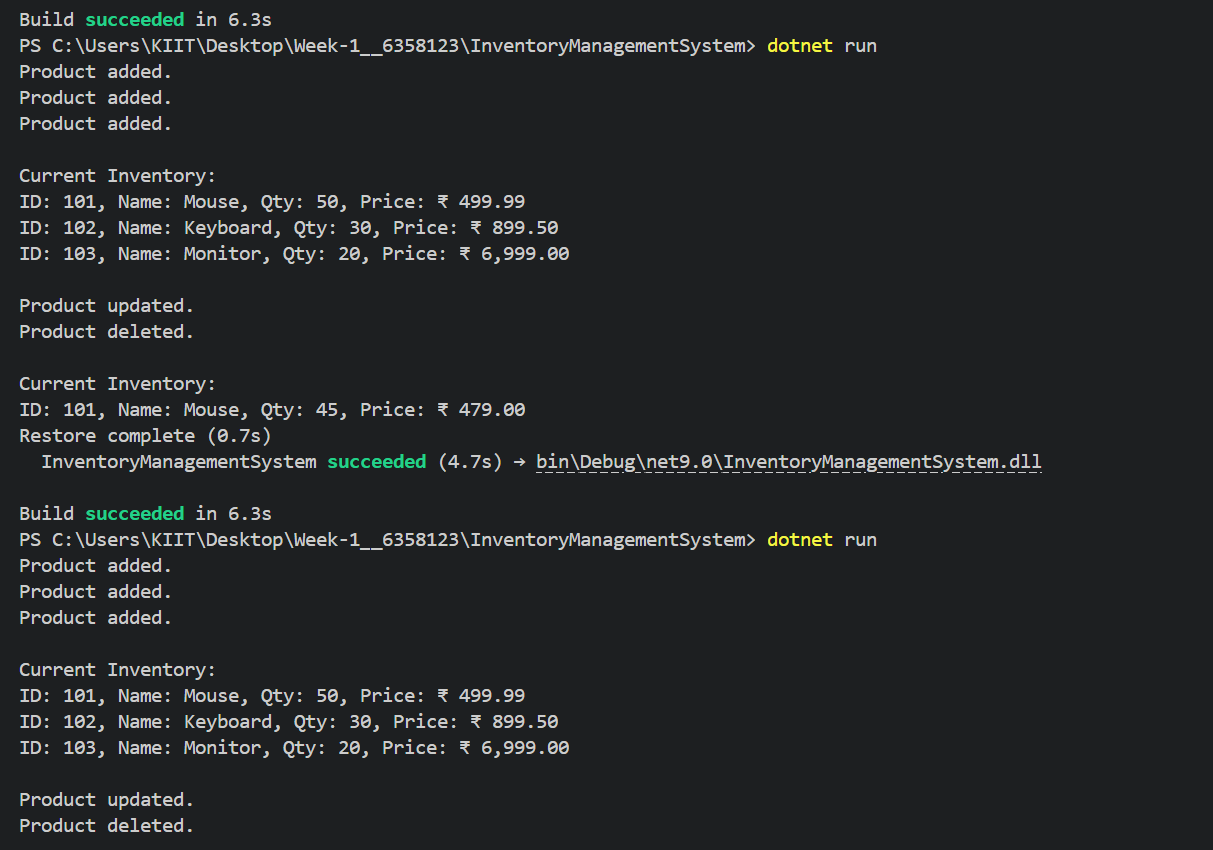
manager.DeleteProduct(102);

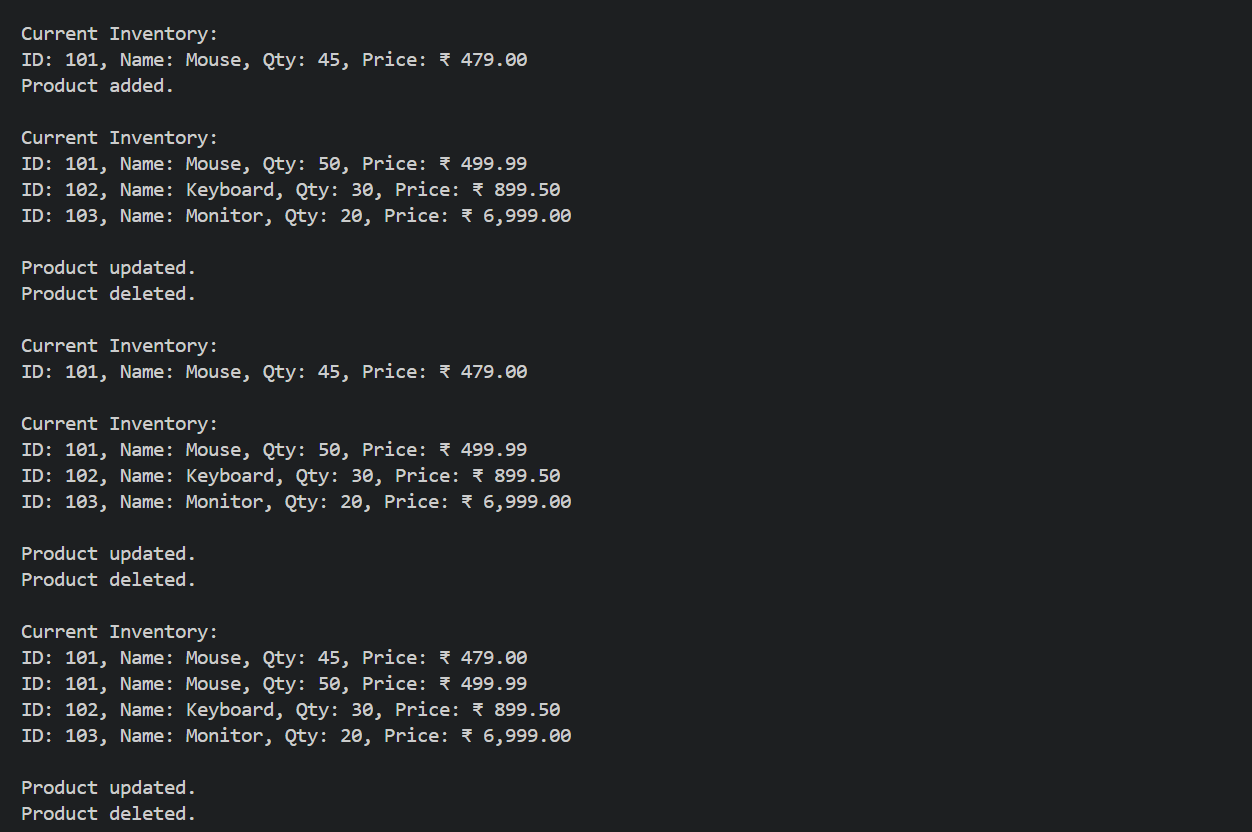
manager.DisplayInventory();

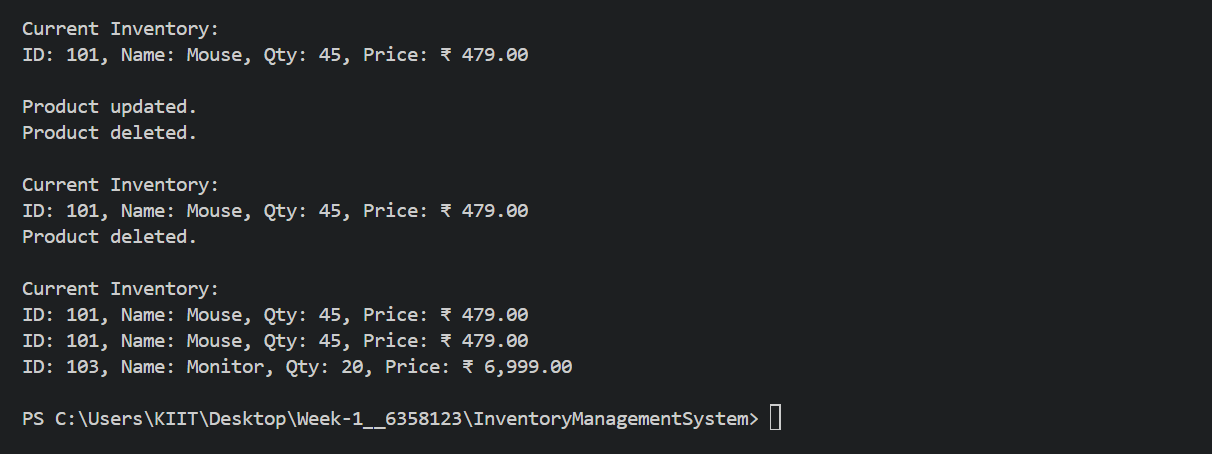
}

}

**Output:**







| **Operation** | **Method Used** | **Time Complexity** | **Explanation** |
| --- | --- | --- | --- |
| **Add** | inventory[productId] = p | O(1) average, O(n) worst | Inserts into a hash table by computing a hash of the key. Worst case occurs when many items hash to the same slot (rare due to rehashing). |
| **Update** | inventory[productId] = ... | O(1) average | Lookup and overwrite value by key; hash lookup is fast. |
| **Delete** | inventory.Remove(productId) | O(1) average | Directly removes a key-value pair based on hash key. |
| **Display All** | foreach (var item in inventory.Values) | O(n) | Must iterate through all products to print or process them. |

We May Optimize These Operations by:

1. Use Dictionary Over List or Array

List<Product> would require linear searches (O(n)) for each operation. Dictionary gives constant-time lookup (O(1)) on average.

2. Use Proper Hash Keys

Ensure productId is unique and used as the key.This prevents collisions and avoids accidental overwrites.

3. Minimize Hash Collisions

Use simple, non-repeating integers for productId (e.g., auto-incrementing IDs). Avoid using strings or complex objects as keys unless properly hashed.

4. Lazy Loading or Pagination for Display

If inventory is huge, consider displaying data in pages (batches of 10, 50, etc.) instead of printing all products at once. This improves memory usage and user experience.

5. Consider Using a Database for Scalability

For enterprise-level applications, move from in-memory dictionaries to persistent storage (SQL, NoSQL). Then cache frequently accessed items in memory for performance (using Dictionary, ConcurrentDictionary, etc.).