**Week1-HandsOn**

**Submitted By:**

**Piyus Kaushik Bhattacharyya**

**Superset-ID: 6361243**

**Q1. Implement a Singleton Pattern**

**Design Principle:**

**Ensure a class has only one instance and provide a global point of access to it.**

This example demonstrates the **Singleton Design Pattern**, which is a **creational pattern**. The goal is to restrict instantiation of a class to a single object and provide a global point of access to it.

**Key Characteristics in the Code:**

1. **Private Constructor**
   * Prevents external classes from creating new instances of Singleton.
2. **Static Instance Field**
   * The static field public static Singleton instance = new Singleton(); ensures that only one instance is created and shared across the application.
3. **Global Access Point**
   * Singleton.instance gives direct global access to the sole instance of the class.

**Note:** This implementation is known as **eager initialization**, where the instance is created when the class is loaded. It is thread-safe but might lead to resource wastage if the instance is never used.

**Console Output:**

Singleton Created.

Hello World!

**Explanation of Output:**

* Singleton Created.: This message is printed when the singleton instance is initialized.
* Hello World!: Printed when the HelloWorld() method is called from Main().

**Q2. Implement a Factory Pattern**

**Design Principle:**

**Define an interface for creating an object, but let subclasses or methods decide which class to instantiate.**

This example demonstrates the **Factory Method Design Pattern**, a **creational pattern** that abstracts the process of object creation, allowing the code to remain independent of the concrete class that is instantiated.

**Key Characteristics in the Code:**

1. **Common Interface (Animal)**
   * Declares a method MakeSound() that all animal types must implement.
2. **Concrete Implementations (Dog, Cat)**
   * Each subclass implements MakeSound() with its unique behavior.
3. **Factory Class (AnimalFactory)**
   * Contains a static method CreateAnimal(string animalType) that encapsulates the logic for instantiating the correct object based on input.
4. **Decoupling Object Creation from Usage**
   * The client code (in Main) interacts with the Animal interface and does not need to know the specific class name or instantiation logic.

**NOTE:** This promotes loose coupling and adheres to the Open/Closed Principle — the code is open for extension but closed for modification.

**Console Output:**

Requesting Dog:

Dog Object created.

Woof!

Requesting Cat:

Cat Object created.

Meow!

Requesting Horse:

Invalid animal type.

**Explanation of Output:**

* The factory creates a Dog when requested and prints "Dog Object created." followed by "Woof!".
* Similarly, a Cat results in "Cat Object created." followed by "Meow!".
* When an unsupported type like "horse" is requested, the factory prints "Invalid animal type." and returns null. The null-conditional operator (?.) ensures no exception occurs.

**Q3. E-Commerce Search Functionality**

**Design Principle:**

Use data structures to organize and retrieve objects efficiently.

This program demonstrates the application of encapsulation, modularization, and data-driven design to simulate a simple e-commerce product search engine. It follows Single Responsibility and Open/Closed Principles, where each function serves a clear purpose and is easy to extend.

**Core Components:**

**Class: Product**

* Represents an individual product with:
  + Name, Category, and Price (immutable properties).
* Converts Name and Category to lowercase for **case-insensitive search**.
* Display() method prints product details.

**Data Structures Used:**

1. List<Product> products:
   * Stores all product entries sequentially for **name-based search**.
2. Dictionary<string, HashSet<Product>> catMap:
   * Maps each category to a set of products for **category-based lookup** in **O(1)** time.

**Functions:**

* AddProduct(Product p): Adds a product to both products list and catMap.
* SearchByName(string name):
  + Iterates through all products.
  + Prints those whose names contain the search keyword (case-insensitive).
  + Time Complexity: **O(n)**
* SearchByCategory(string category):
  + Retrieves products for a category in constant time using the dictionary.
  + Time Complexity: **O(1)** + **O(k)** where *k = number of products in category*

**Sample Output:**

E-Commerce Product Search

1. Search by Product Name

2. Search by Category

0. Exit

Enter your choice: 1

Enter Product Name (P1 to P5): P1

Search Products by Product Name: P1

Product Name: p1, Category: c1, Price: 79999

E-Commerce Product Search

Enter your choice: 2

Enter Category (C1 to C3): C2

Search Products for Category: C2

Product Name: p3, Category: c2, Price: 4999

Product Name: p4, Category: c2, Price: 2999

Enter your choice: 2

Enter Category (C1 to C3): c4

Search Products for Category: c4

No products found in this category.

Enter your choice: 0

Exiting the program. Thank you!

**Q4. Financial Forecasting**

**Design Principle:**

Use historical trends and average growth to predict future values.

This example demonstrates a data-driven financial forecasting tool based on average monthly growth rate. It applies fundamental programming concepts such as looping, mathematical modeling, and input/output operations to simulate a forecasting system used in business analytics.

**Key Components and Logic:**

1. **CalculateAvgGrowthRate(double[] monthlyRevenue, int size)**

* Computes average month-over-month growth rate using:
* Averages these growths across all months.

1. **Forecast(double[] hist, int histSize, int n)**

* Creates a forecast for the next n months based on:
  + The last known revenue
  + The average growth rate calculated from history
* Uses a **compounded growth model**:

1. **Main()**

* Accepts user input for historical revenue data.
* Calls the forecasting function.
* Outputs both historical and predicted revenue in a readable format.

**Sample Output:**

**If the user enters the following values:**

Enter number of past months data: 3

Enter revenue for month 1: 10000

Enter revenue for month 2: 11000

Enter revenue for month 3: 12100

Enter number of months to forecast: 2

**The output will be:**

Historical Revenue Data

Month 1: Rs.10000.00

Month 2: Rs.11000.00

Month 3: Rs.12100.00

Forecasted Revenue

Month 4: Rs.13310.00

Month 5: Rs.14641.00