**Initial Zoo Creation Task:**

### Step 1: Define the Animal Class Hierarchy

1.1. Create a new C# file or add to the existing one.

1.2. Define a base class **Animal** with properties like **Name** and methods like **MakeSound**.

1.3. Create derived classes for specific types of animals (e.g., **Lion** and **Elephant**) that inherit from the **Animal** class. Override the **MakeSound** method in each derived class.

### Step 2: Implement the Zoo Class

2.1. Add a new class called **Zoo** to represent the zoo.

2.2. In the **Zoo** class, maintain a collection of animals, such as a list.

2.3. Implement methods in the **Zoo** class to add animals, remove animals, and display information about the animals in the zoo.

### Step 3: Implement Main Program

3.1. In the **Program** class, create an instance of the **Zoo**.

3.2. Create instances of different animals (e.g., **Lion** and **Elephant**) and add them to the zoo.

3.3. Display information about the animals in the zoo.

### Step 4: Compile and Run

4.1. Compile the C# code using a C# compiler or an integrated development environment (IDE).

4.2. Run the compiled program.

4.3. Observe the output in the console, which should display information about the animals in the zoo.

**Zoo Task 1:**

**Step 1: Define the Observer Interface and Concrete Observer**

1.1. Create an interface **IZooObserver** with a method **Update(string action, Animal animal)**.

1.2. Implement a concrete observer class **ZooVisitor** that implements **IZooObserver**. Add a **Name** property and implement the **Update** method.

**Step 2: Define the Subject (Observable) Interface and Concrete Subject**

2.1. Create an interface **IZooObservable** with methods **AddObserver**, **RemoveObserver**, and **NotifyObservers**.

2.2. Implement a concrete subject class **Zoo** that implements **IZooObservable**. Maintain a list of observers and implement the three methods.

**Step 3: Define the Product Interface and Concrete Products**

3.1. Create an interface **Animal** with a method **MakeSound**.

3.2. Implement concrete product classes such as **Lion** and **Elephant** that implement the **Animal** interface. In each class, implement the **MakeSound** method.

**Step 4: Implement Zookeeper Class**

4.1. Create a class **Zookeeper** with a method **PerformTask** that takes an **Animal** parameter and prints a message, e.g., "Zookeeper performs a task with Lion: Roar!"

**Step 5: Define the Factory Interface and Concrete Factories**

5.1. Create an interface **IAnimalFactory** with a method **CreateAnimal** returning an **Animal**.

5.2. Implement concrete factory classes such as **LionFactory** and **ElephantFactory** that implement **IAnimalFactory**. In each class, implement the **CreateAnimal** method.

**Step 6: Implement Main Program**

6.1. In the **Main** method:

* Create an instance of **Zoo**.
* Create instances of **ZooVisitor** (observers) and add them to the zoo's observer list.
* Create instances of concrete factories (**LionFactory** and **ElephantFactory**).
* Use the factories to create concrete products (animals).
* Notify observers about events in the zoo using the **NotifyObservers** method.
* Create an instance of **Zookeeper** and demonstrate the **PerformTask** method.

**Step 7: Compile and Run**

7.1. Compile the C# code using a C# compiler.

7.2. Run the compiled program.

7.3. Observe the output in the console, which should demonstrate the zoo interactions, observer notifications, and zookeeper tasks.

**Step 8: Experiment and Modify**

8.1. Experiment with the code:

* Add more types of animals or events in the zoo.
* Create additional observers.
* Modify the zookeeper's tasks.

8.2. Explore how the Observer Pattern allows you to add new observers without modifying the subject.

8.3. Observe how the Factory Pattern facilitates the creation of different types of animals without exposing their instantiation logic.

Factory pattern HINTS:

<https://www.dotnettricks.com/learn/designpatterns/factory-method-design-pattern-dotnet>

<https://www.dofactory.com/net/factory-method-design-pattern>

Obeserver pattern sample:

<https://www.csharptutorial.net/csharp-design-patterns/c-observer-pattern/>

**Extension of above task:**

**Zoo Task 2**

**Step 1: Define the Strategy Interface and Concrete Strategies**

1.1. Create a new C# file or add to the existing one.

1.2. Define an interface **IFeedingStrategy** with a method **Feed**.

1.3. Implement concrete strategy classes, such as **HerbivoreFeedingStrategy** and **CarnivoreFeedingStrategy**, that implement **IFeedingStrategy** and provide their own implementation of the **Feed** method.

**Step 2: Implement the Context Class**

2.1. Add a new class called **Animal** to represent animals in the zoo.

2.2. Add properties to the **Animal** class: **Name** (string) and **FeedingStrategy** (type **IFeedingStrategy**).

2.3. Implement a method **FeedAnimal** in the **Animal** class that delegates the feeding behavior to the **FeedingStrategy.Feed** method.

**Step 3: Define the Singleton Pattern for Zoo Management**

3.1. Add a new class called **ZooManager** to manage zoo-related tasks.

3.2. In the **ZooManager** class, create a private static instance and a private constructor.

3.3. Implement a public static method **GetInstance** in **ZooManager** that returns the singleton instance.

3.4. Add any additional management-related functionality to the **ZooManager**.

**Step 4: Implement Main Program**

4.1. In the **Program** class, create instances of concrete strategies (**HerbivoreFeedingStrategy** and **CarnivoreFeedingStrategy**).

4.2. Create instances of the **Animal** class, setting different feeding strategies for different animals.

4.3. Demonstrate feeding the animals using the **FeedAnimal** method.

4.4. Use the **ZooManager** singleton to manage zoo-related tasks.

**Step 5: Compile and Run**

5.1. Compile the C# code using a C# compiler or an integrated development environment (IDE).

5.2. Run the compiled program.

5.3. Observe the output in the console, which should demonstrate different feeding strategies for herbivores and carnivores and any additional zoo management tasks.

**Step 6: Experiment and Modify**

6.1. Experiment with the code:

* Add more types of animals or feeding strategies.
* Create additional zoo-related tasks in the **ZooManager**.

6.2. Explore how the Strategy Pattern allows you to change the feeding behavior of animals without modifying the **Animal** class.

6.3. Observe how the Singleton Pattern ensures there is only one instance of the **ZooManager**.

Strategy pattern explanation: <https://www.dofactory.com/net/strategy-design-pattern>

Singelton explanation from simple to advanced:

<https://www.tutorialsteacher.com/csharp/singleton>

<https://www.c-sharpcorner.com/UploadFile/8911c4/singleton-design-pattern-in-C-Sharp/>