Singleton Pattern

LCSCI5202: Object Oriented Design Week 4

Polymorphism

- A key concept in object-oriented programming that means "many forms."
- Allows objects of different types to be treated as objects of a common base type.
- Enables a single method, property, or operator to have different implementations based on the object that is invoking it.

Types of Polymorphism

- Compile-time Polymorphism (Static Polymorphism):
  - Achieved using method overloading or operator overloading.
  - Method behavior is determined at compile time.
- Runtime Polymorphism (Dynamic Polymorphism):
  - Achieved using method overriding with inheritance.
  - The exact method that gets called is determined at runtime.

Static Polymorphism- Method Overloading

- Allows you to define multiple methods with the same name in a class but with different parameter lists.
- Enables methods to perform similar but distinct tasks.
- Methods must differ in parameter count, parameter type, or parameter order.

Runtime Polymorphism – Abstract and Virtual

- Allows a method to behave differently based on the object that is invoking it.
- Achieved through method overriding in an inheritance hierarchy
- The **virtual** or **abstract** keyword is used to mark a method in the base class that can be overridden.
- The **override** keyword is used in the derived class to provide the new implementation.

Calling Abstract Methods

- Abstract methods are declared without implementation in an abstract base class.
- Derived classes must override these methods, providing their own implementation.
- Base class can call the overridden method, demonstrating runtime polymorphism.

Lists and polymorphism

- Polymorphism allows different derived class objects to be treated as instances of a common base class.
- Enables storing multiple derived class objects in a collection like List<BaseClass>.

```
List<Animal> animals = new List<Animal>();
animals.Add(new Dog());
animals.Add(new Cat());

foreach (Animal animal in animals) {
    animal.MakeSound();
}
```

What is the Singleton Pattern

- Singleton pattern ensures that a class has only one instance throughout the lifetime of an application and provides a global point of access to that instance.
- It’s commonly used when you need centralized control of a resource, like managing a single connection to a database or a centralized configuration manager.

Key Characteristics

- Single Instance: Only one instance of the class is ever created.
- Global Access: Provides a global point of access to that instance through a static property or method.
- Private Constructor: Prevents external instantiation of the class.

Structure in C#

```
public class AnimalManager {
    static AnimalManager _Instance;
    public static AnimalManager Instance {
        get {
            if (_Instance == null) {
                _Instance = new AnimalManager();
            }
            return _Instance;
        }
    }
    private AnimalManager() {

    }

    public void AddAnimal() {
        // Add animal logic
    }
}
```

Example usage

- `AnimalManager.Instance.AddAnimal()`
- `AnimalManager obj1 = AnimalManager.Instance;`

Key Components

- Private Static Field (_Instance): Stores the single instance of the class.
- Public Static Property (Instance): Provides global access to the single instance.
- Private Constructor: Ensures that the class cannot be instantiated from outside.

Instance Property

- The static Instance property checks if the _Instance is null
- If no instance exists (_Instance == null), it creates a new AnimalManager object.
- If an instance already exists, it returns the existing instance, ensuring only one object is ever created.

Lazy Initialization

- The instance is created on-demand, i.e., it’s only created when the Instance property is accessed for the first time.

- This is called lazy initialization and helps save resources by not creating the object until it's needed.

```
static AnimalManager _Instance;
public static AnimalManager Instance { get {
        if (_Instance == null) {
            _Instance = new AnimalManager();
        }
        return _Instance;
    }
}
AnimalManager obj1 = AnimalManager.Instance;
```

Eager Initialization

- The instance is created pre-emptively, i.e., it's created when the Instance variable is defined.
- This is called Eager Initialization and helps save pre-defining the behaviour of application.

```
static AnimalManager _Instance = new AnimalManager() ;
public static AnimalManager Instance
{ get {
        return _Instance;
    }
}
AnimalManager obj1 = AnimalManager.Instance;
```

Generics

Introduction to T in Generics

- In C#, T is a placeholder for a type used in generics.
- Generics allow classes, methods, or interfaces to be type-agnostic.
- T represents a generic type parameter, which is decided when the code is instantiated.

Structure in C#

```
public class Box<T> {
    private T _value;

    public void SetValue(T value) {
        _value = value;
    }

    public T GetValue() {
        return _value;
    }
}

Box<int> intBox = new Box<int>();
intBox.SetValue(123);

Box<string> strBox = new Box<string>();
strBox.SetValue("Hello");
```

The where Keyword in Generics

- The where keyword in C# is used to constrain the type that can be passed into a generic class, method, or interface.
- It ensures that T has certain capabilities or characteristics.
  - where T : class — T must be a reference type.
  - where T : new() — T must have a parameterless constructor.

where keyword in Generics (where T : new())

```
using System;

public class Factory<T> where T : new()
{
public T Create()
   {
      return new T();
   }
}

public class Animal
{
   public string Name { get; set; }

   public Animal()
   {
      Name = "Default Animal";
   }
}
```

```
class Program
{
   static void Main()
   {
Factory<Animal> animalFactory = new Factory<Animal>();

Animal a1 = animalFactory.Create();
      Console.WriteLine(a1.Name);

Animal a2 = animalFactory.Create();
      a2.Name = "Lion";
      Console.WriteLine(a2.Name);
}
}
```

Summary

- **Polymorphism** – Enables objects to take many forms for flexible code reuse.
- **Singleton Pattern** – Guarantees only one instance with global access.
- **Generics** – Provide type-safe, reusable classes and methods.