OOPS & DESIGN PATTERNS

1. [The Four Principles of Object-Oriented-Programming (OOP)](#_The_Four_Principles)
2. [Static class vs. Singleton class](#_Static_class_vs._1)
3. Design Patterns

# The Four Principles of Object-Oriented-Programming (OOP):

## ****Encapsulation****

Encapsulation is accomplished when each object maintains a private state, inside a class. Other objects cannot access this state directly; instead, they can only invoke a list of public functions. The object manages its own state via these functions and no other class can alter it unless explicitly allowed. In order to communicate with the object, you will need to utilize the methods provided. One way I like to think of encapsulation is by using the example of people and their dogs. If we want to apply encapsulation, we do so by encapsulating all “dog” logic into a Dog class. The “state” of the dog is in the private variables playful, hungry and energy and each of these variables has their respective fields.

      There is also a private method: bark (). The dog class can call this whenever it wants, and the other classes cannot tell the dog when to bark. There are also public methods such as sleep (), play () and eat () that are available to other classes. Each of these functions modifies the internal state of the Dog class and may invoke bark(), when this happens the private state and public methods are bonded.

## ****Abstraction****

      Abstraction is an extension of encapsulation. It is the process of selecting data from a larger pool to show only the relevant details to the object. Suppose you want to create a dating application and you are asked to collect all the information about your users. You might receive the following data from your user: Full name, address, phone number, favorite food, favorite movie, hobbies, tax information, social security number, and credit score. This amount of data is great however not all of it is required to create a dating profile. You only need to select the information that is pertinent to your dating application from that pool. Data like Full name, favorite food, favorite movie, and hobbies make sense for a dating application. The process of fetching/removing/selecting the user information from a larger pool is referred to as Abstraction. One of the advantages of Abstraction is being able to apply the same information you used for the dating application to other applications with little or no modification.

## ****Inheritance****

      Inheritance is the ability of one object to acquire some/all properties of another object. For example, a child inherits the traits of his/her parents. With inheritance, reusability is a major advantage. You can reuse the fields and methods of the existing class. In Java, there are various types of inheritances: single, multiple, multilevel, hierarchical, and hybrid. For example, Apple is a fruit so assume that we have a class Fruit and a subclass of it named Apple. Our Apple acquires the properties of the Fruit class. Other classifications could be grape, pear, and mango, etc. Fruit defines a class of foods that are mature ovaries of a plant, fleshy, contains a large seed within or numerous tiny seeds. Apple the sub-class acquires these properties from Fruit and has some unique properties, which are different from other sub-classes of Fruit such as red, round, depression at the top.

## ****Polymorphism****

       Polymorphism gives us a way to use a class exactly like its parent so there is no confusion with mixing types. This being said, each child sub-class keeps its own functions/methods as they are. If we had a super class called Mammal that has a method called mammal Sound (). The sub-classes of Mammals could be Dogs, whales, elephants, and horses. Each of these would have their own iteration of a mammal sound (dog-barks, whale-clicks).

# Static class vs. Singleton class

* A static class cannot be a top level class and cannot implement interfaces where a singleton class can.
* All members of a static class are static but for a Singleton class it is not a requirement.
* A static class gets initialized when it is loaded so it cannot be lazily loaded where a singleton class can be lazily loaded.
* A static class object is stored in stack whereas singleton class object is stored in heap memory space.

# Design Patterns

[**https://refactoring.guru/design-patterns/csharp**](https://refactoring.guru/design-patterns/csharp)

*“Design patterns are solutions to software design problems you find again and again in real-world application development. Patterns are about reusable designs and interactions of objects.”*

## Creational Patterns

1. **Abstract Factory:** The Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

**abstract class AbstractFactory**

**{**

**public abstract AbstractProductA CreateProductA();**

**public abstract AbstractProductB CreateProductB();**

**}**

1. **Builder:** Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

// The Builder interface specifies methods for creating the different parts

// of the Product objects.

**public interface IBuilder**

**{**

**void BuildPartA();**

**void BuildPartB();**

**void BuildPartC();**

**}**

1. **Factory Method:** Factory method is a creational design pattern which solves the problem of creating product objects without specifying their concrete classes.

Factory Method defines a method, which should be used for creating objects instead of direct constructor call (new operator). Subclasses can override this method to change the class of objects that will be created.

1. **abstract class Product**
2. **{ }**
3. **class ConcreteProductA : Product**
4. **{ }**
5. **class ConcreteProductB : Product**
6. **{ }**
7. **abstract class Creator**
8. **{**
9. **public abstract Product FactoryMethod();**
10. **}**
11. **class ConcreteCreatorA : Creator**
12. **{**
13. **public override Product FactoryMethod()**
14. **{**
15. **return new ConcreteProductA();**
16. **}**
17. **}**
18. **class ConcreteCreatorB : Creator**
19. **{**
20. **public override Product FactoryMethod()**
21. **{**
22. **return new ConcreteProductB();**
23. **}**
24. **}**
25. **class MainApp**
26. **{**
27. **static void Main()**
28. **{**
29. **// An array of creators**
30. **Creator[] creators = new Creator[2];**
31. **creators[0] = new ConcreteCreatorA();**
32. **creators[1] = new ConcreteCreatorB();**
33. **// Iterate over creators and create products**
34. **foreach (Creator creator in creators)**
35. **{**
36. **Product product = creator.FactoryMethod();**
37. **Console.WriteLine("Created {0}",**
38. **product.GetType().Name);**
39. **}**
40. **Console.ReadKey();**
41. **}**