# MODERN PROGRAMMING PRACTICES - QUIZ

Maharishi International University
Submitted to : Professor Assad Maalouf

## Table of Contents

## Question 1: **Explain polymorphism and why it is important**

**Answer:**

Polymorphism is one of the four principles of Object-Oriented Paradigm. It is the ability of an object to take on many forms.

Polymorphism in OOP occurs when a super class references a sub class object.

Java supports two types of polymorphism:

1. Static or compile-time: ***Overloading*** *a method with different sets of parameters*
2. Dynamic: *Within an inheritance hierarchy, a subclass can* ***override*** *a method of its superclass. If you instantiate the subclass, the JVM will always call the overridden method, even if you cast the subclass to its superclass*.

Polymorphism in Java makes it possible to write a method that can correctly process lots of different types of functionalities that have the same name. We can also gain consistency in our code by using polymorphism.

### Few advantages of Polymorphism in OOP include

1. It provides reusability to the code. The classes that are written, tested and implemented can be reused multiple times. This, in turn, saves a lot of time for the coder. Also, the code can be changed without affecting the original code.
2. A single variable can be used to store multiple data values. The value of a variable inherited from the superclass into the subclass can be changed without changing that variable's value in the superclass or any other subclasses.
3. With lesser lines of code, it becomes easier for the programmer to debug the code.

## Question 2: **Explain the open close principle and give an example**

**Answer:**

Open/ Closed Principle is one of the principles stated by the SOLID principles
Bertrand Meyer wrote about Open/ Closed Principle in 1988 in his book Object-Oriented Software Construction. He explained the Open/Closed Principle as:

***"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."***

The general idea of this principle is great. It tells you to write your code so that you will be able to add new functionality without changing the existing code. That prevents situations in which a change to one of your classes also requires you to adapt all depending classes.

Open/ Closed Principle uses interfaces instead of super classes to allow different implementations which you can easily substitute without changing the code that uses them. The interfaces are closed for modifications, and you can provide new implementations to extend the functionality of your software. Example:

```
public interface Shape {

     public Double calculateArea();

}

public class Rectangle implements Shape {

     Double length;
     Double width;

     public double calculateArea() {

           return length * width;
     }
}

public class Circle implements Shape {

     public Double radius;

     public Double calculateArea() {

           return (22 / 7) * radius * radius;
     }
}
```

As aforementioned, there is a base interface Shape. All shapes now implement the base interface Shape. Shape interface has an abstract method calculateArea(). Both circle & rectangle provide their own overridden implementation of calculateArea() method using their own attributes. If in future we want to calculate area of other shapes like triangle, square etc., we can implement the Shape interface without changing any class.

We have brought in a degree of extensibility as shapes are now an instance of Shape interfaces. This allows us to use Shape instead of individual classes.
The last point is the consumer of these shapes. The consumer will be the AreaCalculator class which would now look like this.

```
public class AreaCalculator {
     public Double calculateShapeArea(Shape shape) {
            return shape.calculateArea();
     }
}
```

This AreaCalculator class now fully removes our design flaws noted above and provides a clean solution which adheres to the Open-Closed Principle.

## Question 3: **Explain early binding and when it is possible.**

### Answer:

Static Binding or Early Binding in Java refers to a process where the compiler determines the type of object and resolves the method during the compile-time. Generally, the compiler binds the overloaded methods using static binding.

The early binding happens at the compile-time. Additionally, if there is any private, final, or static method in a class, there is static binding.

Example of early/ static binding:

```
class Dog{
    private void eat(){System.out.println("dog is eating...");}

    public static void main(String args[]){
    Dog d1=new Dog();
    d1.eat();
    }
}
```

## Question 4: **Explain late binding and why it is needed.**

### Answer:

When type of the object is determined at run-time, it is known as dynamic binding. Simply put, if the method is resolved at runtime, it is Dynamic Binding or late binding. The best example of Dynamic binding is the Method Overriding where both the Parent class and the derived classes have the same method. And therefore the type of the object determines which method is going to be executed.

The type of object is determined during the execution of the program; therefore it is called dynamic binding.

```
class Animal{
    void eat(){
    System.out.println("animal is eating...");
    }
}
class Dog extends Animal{
    void eat(){
    System.out.println("dog is eating...");
    }
    public static void main(String args[]){
        Animal a=new Dog();
        a.eat();
       }
}
```

```
Output: dog is eating...
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So, compiler doesn't know its type, only its base type.

Late binding is generally used in scenarios where an **exact object interface is unknown at design time**, or where interaction with multiple unknown servers invoke functions by names. It is also **used as a workaround for compatibility issues between multiple versions of an improperly modified component.** Thus, the trade-off between *flexibility* and *performance* must be weighed prior to application development.

## Question 5: **Explain programming to an interface and what are the advantages of doing so.**

**Answer:**

‘Programming to interfaces’ is a **technique to write classes based on** an interface; interface that defines what the behavior of the object should be. It involves creating an interface first, defining its methods and then creating the actual class with the implementation. It promotes loose coupling.

*The best example to understand ‘Programming to an Interface’ paradigm comes from a Stack Exchange thread:*

The main idea is that domains change far slower than software does. Say you have software to keep track of your grocery list. In the 80's, this software would work against a command line and some flat files on floppy disk. Then you got a UI. Then you maybe put the list in the database. Later, it may be moved to the cloud or mobile phones or Facebook integration.

If you designed your code specifically around the implementation (floppy disks and command lines) you would be ill-prepared for changes. If you designed your code around the interface (manipulating a grocery list) then the implementation is free to change.

Few notable benefits of ‘Programming to an Interface’ include:

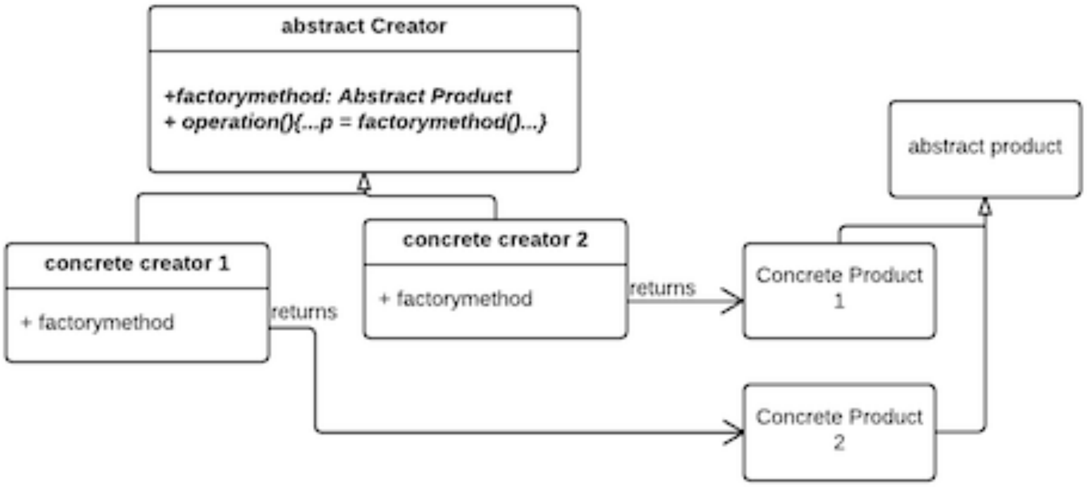
1. Clients remained decoupled and unaware of specific class they are using, as far as the underlying class adheres to the defined interface.
2. The client does not need to know with which concrete class its interacting.
3. Depending on the context, different implementation classes can be polymorphically provided without having to change client code. Makes it easy for the system to evolve.
4. During development even though the concrete class is not available, the client class can be developed. Helps parallelize work.

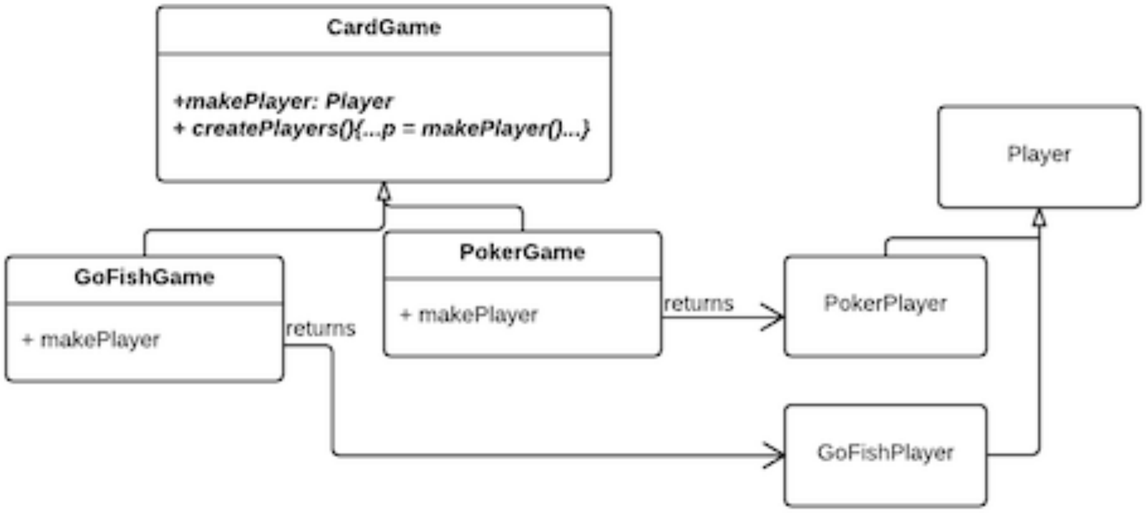
## Question 6: **Explain Factory design pattern and why is it important**

**Answer:**

A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.

In other words, Factory design pattern is used to create objects or Class in Java and it provides loose coupling and high cohesion. Factory pattern encapsulate object creation logic which makes it easy to change it later when you change how object gets created or you can even introduce new object with just change in one class.





The main advantages of Factory Design Pattern are:

1. Factory Method Pattern allows the sub-classes to choose the type of objects to create.
2. It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

## Question 7: **List at least three advantages of using a Factory method over using the constructor.**

**Answer:**

The three advantages of using a Factory method over using the constructor are:

1. A factory method doesn't necessarily return an object of the class in which it was called. Often these could be its subclasses, selected based on the arguments given to the method. Whereas constructor can't have any return type  not even void.

2. A factory method can have a better name that describes what and how it returns what it does, for example Troops::GetCrew(myTank) whereas constructor are restricted to the standard naming convention.

3. A factory method can return an already created object, unlike a constructor, which always creates a new instance.

## Question 8: **Explain Template Method design pattern and how it is useful**

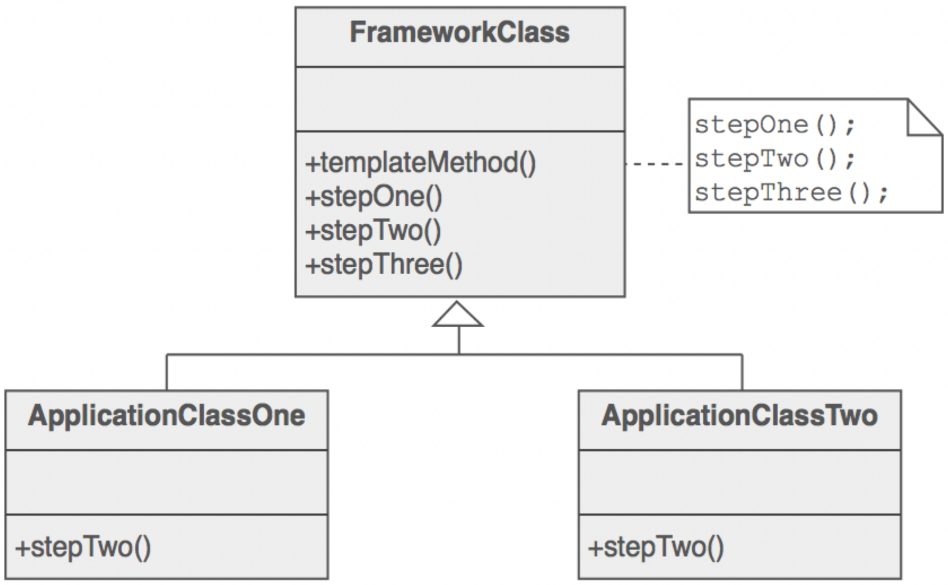
**Answer:**

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single template method. The steps may either be abstract or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).

Simply put, Template Method design Pattern:

a. Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
b. Base class declares algorithm 'placeholders', and derived classes implement the placeholders.



Template Method Pattern is useful for situations such as:

1. Let subclasses implement varying behavior (through method overriding), i.e., when you want to let client extend only particular steps of an algorithm but not the whole algorithm or its structure.
2. Avoid duplication in the code, the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in the subclasses.
3. Control at what points subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.

## Question 9: **Explain The Singleton design pattern and show how you implement it.**

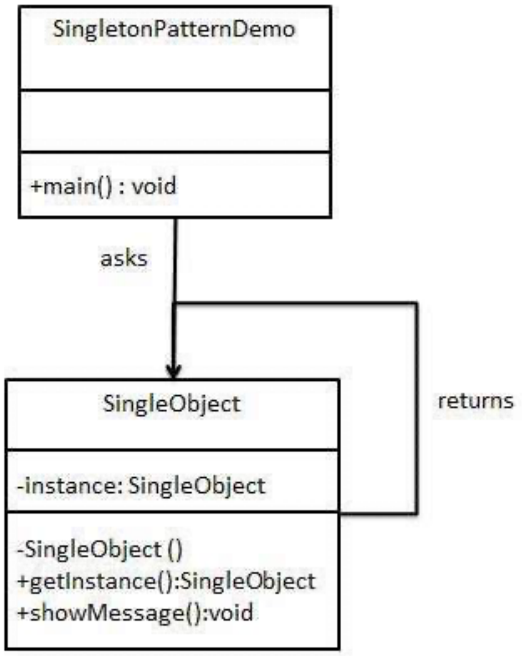
**Answer:**

The term comes from the mathematical concept of a singleton. In programming, the singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system.

The singleton design pattern solves problems by allowing it to:

1. Ensure that a class only has one instance
2. Easily access the sole instance of a class
3. Control its instantiation
4. Restrict the number of instances
5. Access a global variable

In essence, the singleton pattern forces it to be responsible for ensuring that it is only instantiated once. A hidden constructor—declared `private` or `protected`—ensures that the class can never be instantiated from outside the class. The public static operation can be accessed by using the class name and operation name, e.g., `Singleton.getInstance()`.



**Coding Solution:**

**Step 1:**

Create a Singleton Class.

*SingleObject.java*

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("This is singleton message!");
   }
}
```

**Step 2:**

Get the only object from the singleton class.

*SingletonPatternDemo.java*

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not
visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

**Step 3:**

Verify the output.

```
This is singleton message!
```