OOP

P. Shamoi

Main OOP concepts

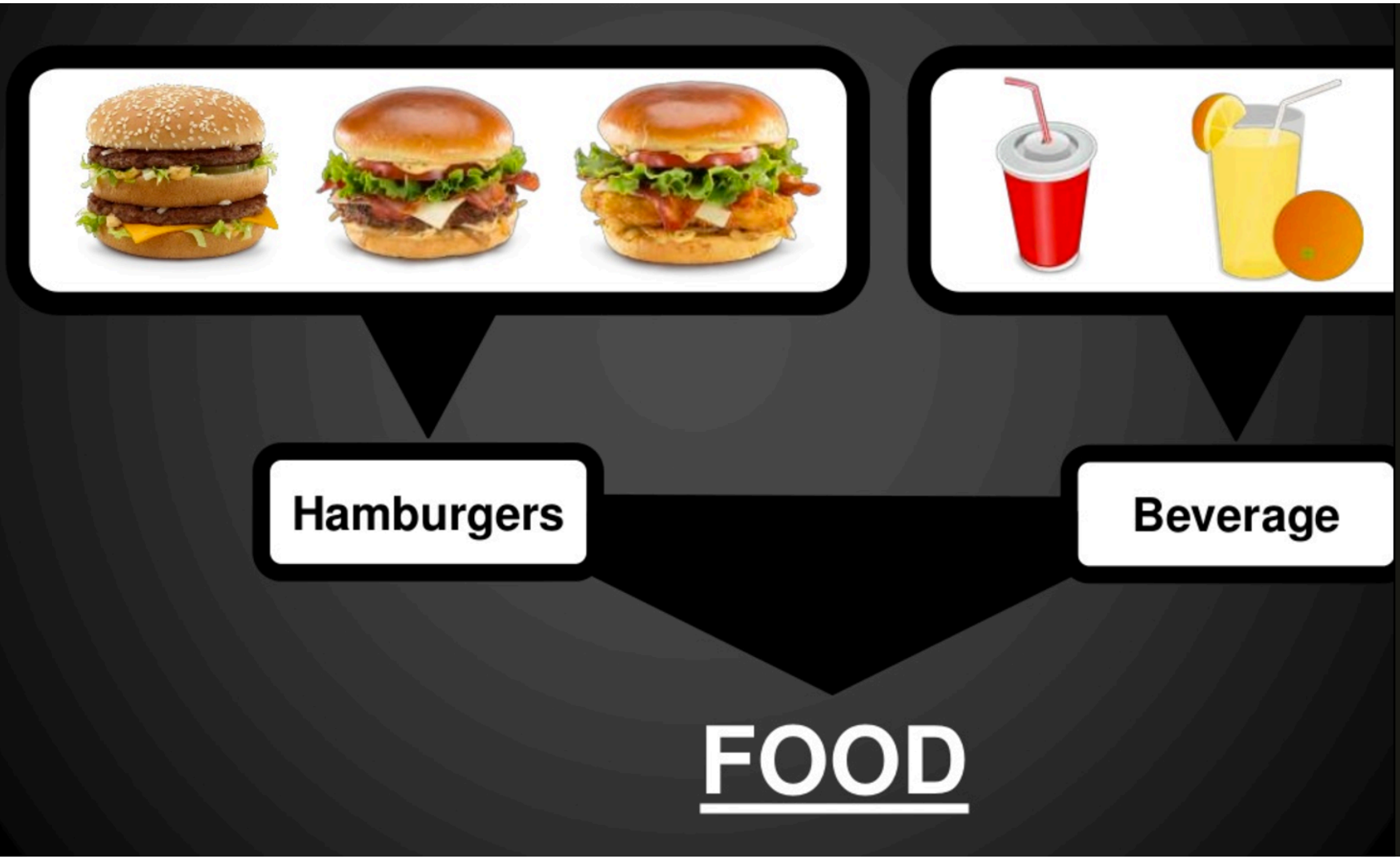
1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphisam

Abstraction

Hiding non-essential features and showing the essential features

(or)
Hiding unnecessary details from the users

Example:
TV Remote Button (you don't see the circuits in it, right?)


Hamburgers
Beverage
FOOD

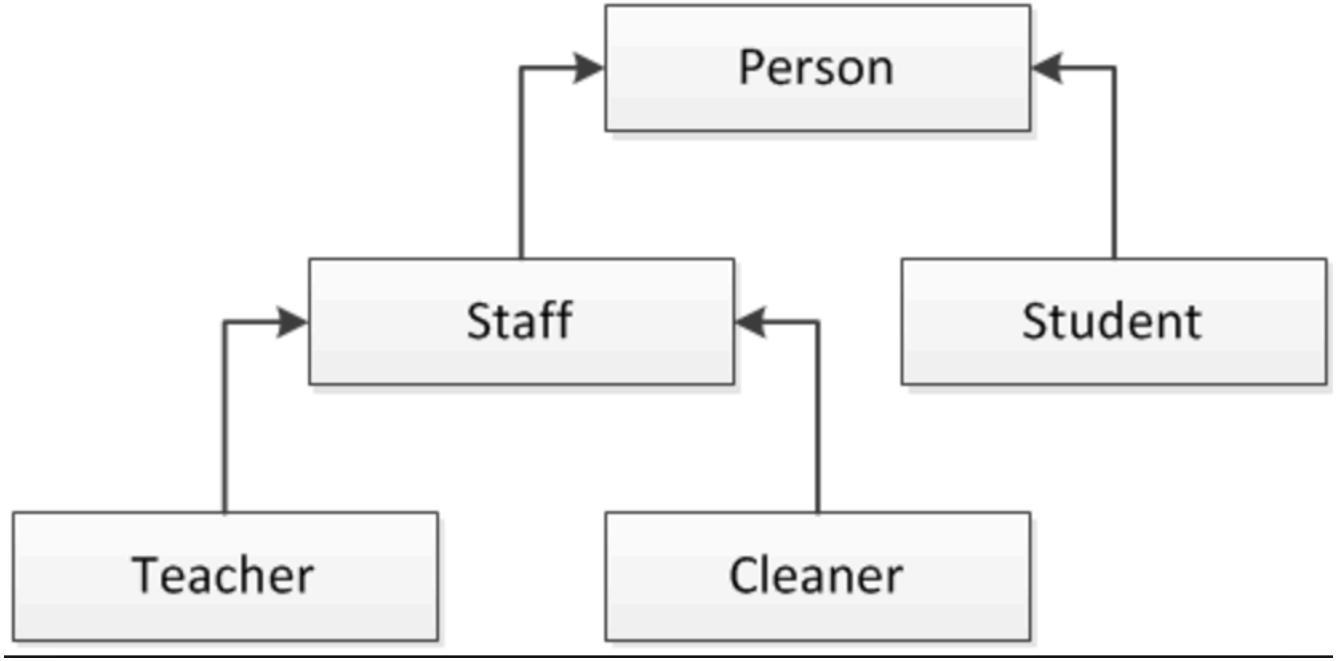
Encapsulation

Encapsulation is packaging the data and functions into a single component.

It allows selective hiding of properties and methods in an object by building an wall to protect the code from accidental corruption.

Inheritance

It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces.

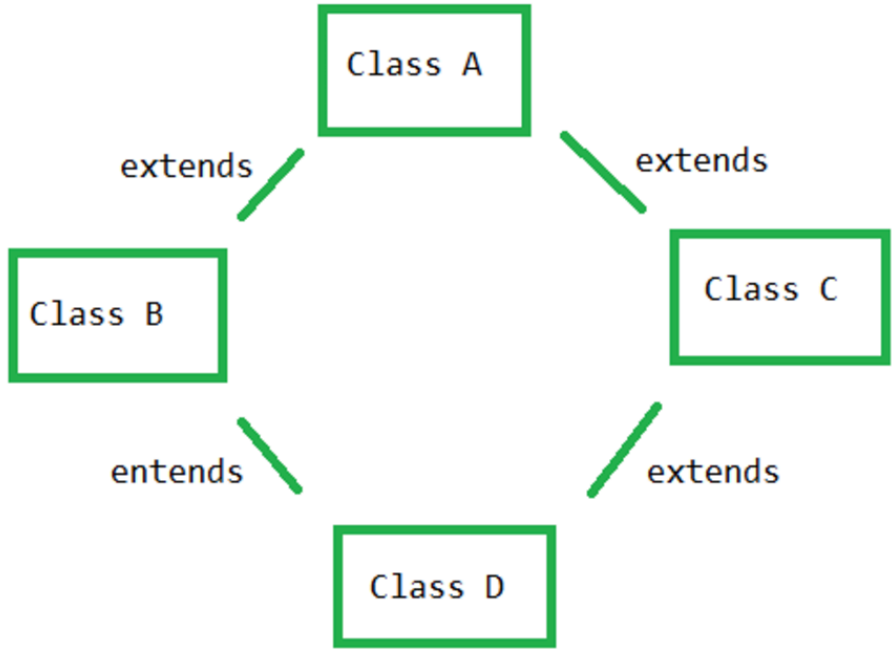


Inheritance

Types of Inheritance:

1. Multiple inheritance.
2. Multilevel inheritance.

"What do you do when you have multiple common base classes in the different superclasses?



Diamond problem:

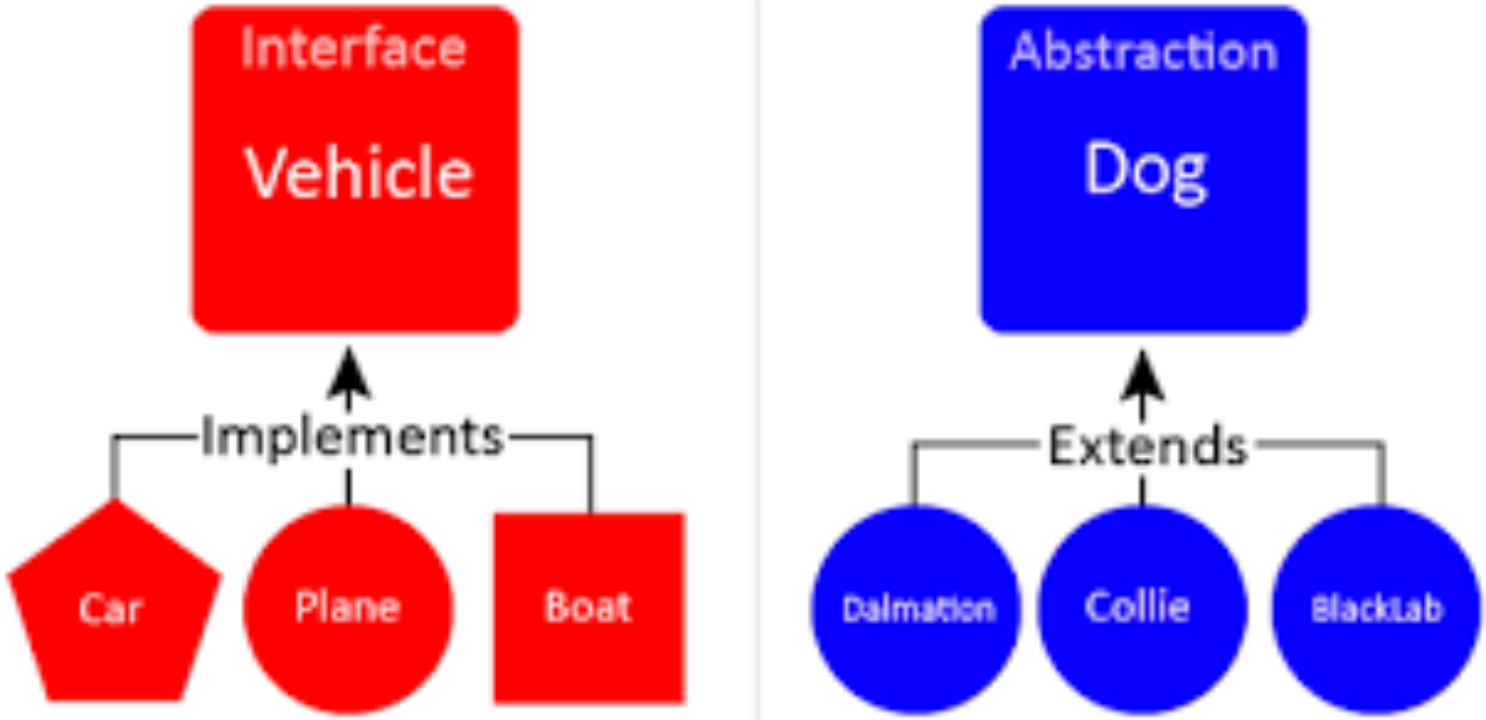
In multiple inheritance there is every chance of multiple properties of multiple objects with the same name available to the sub class object with same priorities leads for the ambiguity.

Abstract class

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

They are provided to define default behavior and ensure that client of that class should adore to those contract which are defined inside the abstract class. In order to use it, you must extend and implement their abstract methods.

Interfaces vs. Abstract Classes



Virtual vs Abstract methods

1813

An abstract function cannot have functionality. You're basically saying, any child class MUST give their own version of this method, however it's too general to even try to implement in the parent class.

A virtual function, is basically saying look, here's the functionality that may or may not be good enough for the child class. So if it is good enough, use this method, if not, then override me, and provide your own functionality.

```
public abstract class myBase
{
    //If you derive from this class you must implement this method.
    public abstract void YouMustImplement();

    //If you derive from this class you can change the behavior but
    public virtual void YouCanOverride()
    {
    }
}
```

```
public abstract class Shape
{
   public abstract void Paint(Graphics g, Rectangle r);
}
public class Ellipse: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
      g.DrawEllipse(r);
   }
}
public class Box: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
      g.DrawRect(r);
   }
}
```

Is there any sense in a virtual method without a body?

```
public virtual void CountX(){}
```

Is there any sense in a virtual method without a body?

Almost it is a rare occasion, yes. Because if the default behavior is to do nothing, but derived classes might want to do something. It's a perfectly valid structure.

If these methods (Pre, Post) were abstract, then all derived classes would need to implement them

```
abstract class Base
  {
      public void ProcessMessages(IMessage[] messages)
      {
          PreProcess(messages);

          // Process.

          PostProcess(messages);
      }

      public virtual void PreProcess(IMessage[] messages)
      {
          // Base class does nothing.
      }

      public virtual void PostProcess(IMessage[] messages)
      {
          // Base class does nothing.
      }
  }

  class Derived : Base
  {
      public override void PostProcess(IMessage[] messages)
      {
          // Do something, log or whatever.
      }

      // Don't want to bother with pre-process.
  }
```

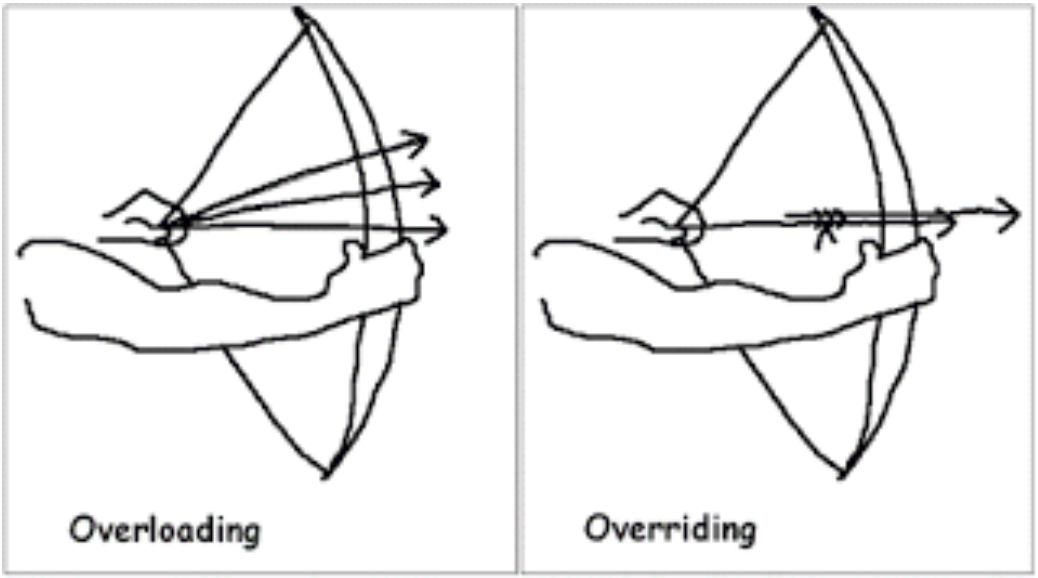
What is the output?

```
public class Program
{
    public static void Main(string[] args)
    {

        Animal a = new FlyingAnimal();

        a.startEating();
    }
}
class Animal {

   public void startEating(){

          eat();
       }

   public virtual void eat() {Console.WriteLine("Some animal is eating");}

}

class FlyingAnimal : Animal {

    public override void eat() {

    Console.WriteLine("Flying animal is eating!");

}
```

What about this code?

```
public class Program
{
    public static void Main(string[] args)
    {
        Animal a = new FlyingAnimal();
        a.startEating();
    }
}

   abstract class Animal {
   public void startEating(){
           eat();
       }
   public abstract void eat();
}
class FlyingAnimal : Animal {
    public override void eat() {
    Console.WriteLine("Flying animal is eating!");
}
```

Overloading and Overriding



Overloading is compile time Polymorphism and Overriding is runtime Polymorphism.

Preventing method overriding

- Final
- Private
- Static
- Why ?

Why preventing method overriding?

You use final when you don't want subclass changing the logic of your method by overriding it due to security reason.

This is why String class is final in Java. This concept is also used in template design pattern where template method is made final to prevent overriding.

Identify which oops concept were used in below scenario:

”In a group of 5 boys one boy never give any contribution when group goes for eating out, party or anything.
Suddenly one beautiful girl joins the same group and then the aforementioned boy spends lots of money for the group.”

Identify which oops concept is used in below scenario:

”In a group of 5 boys one boy never give any contribution when group goes for eating out, party or anything.
Suddenly one beautiful girl joins the same group and then the aforementioned boy spends lots of money for the group.”

Answer:
Runtime Polymorphism

Is there any sense in a private constructor?

Is there any sense in a private constructor?

Answer:
Sometimes, yes! E.g., for a Singleton pattern.

Singleton class

In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

```
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Singleton class

An implementation of the singleton pattern must:

- Ensure that only one instance of the singleton class ever exists;
- and provide global access to that instance.

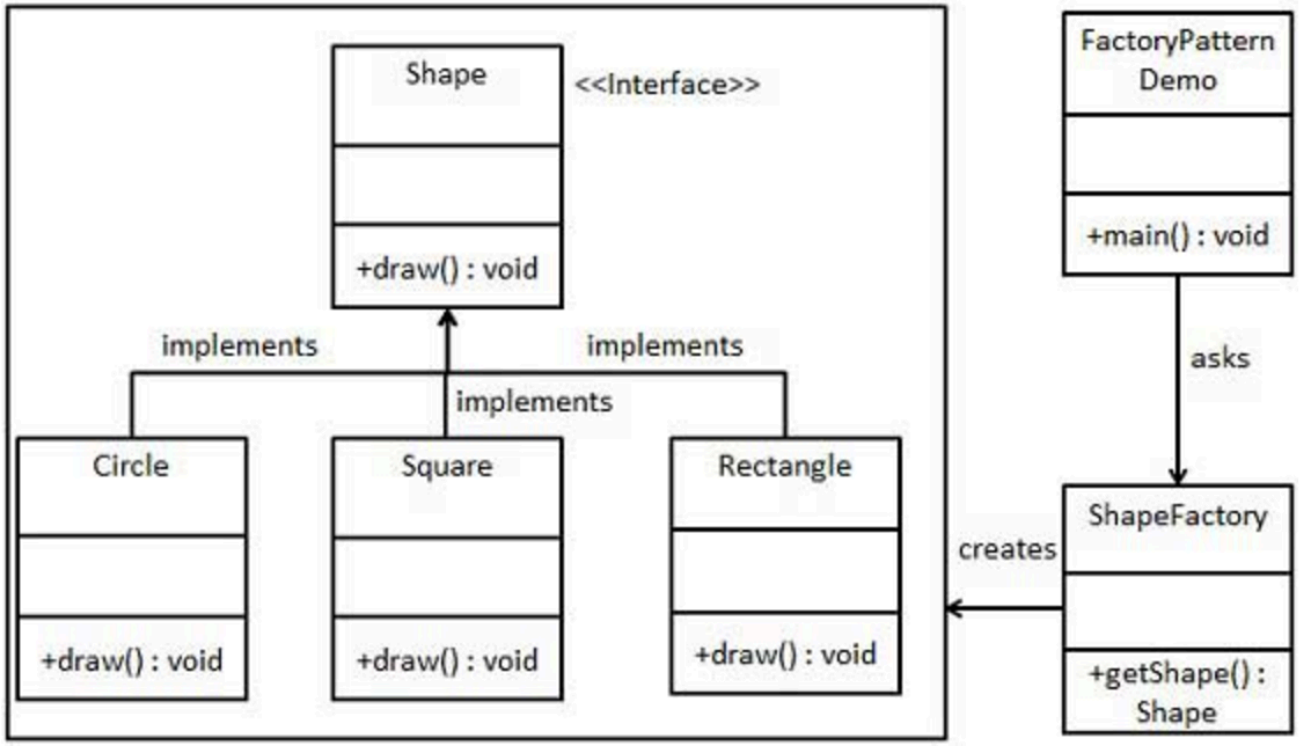
Typically, this is done by:

declaring all constructors of the class to be private; and
providing a static method that returns a reference to the instance.

Factory Design Pattern

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

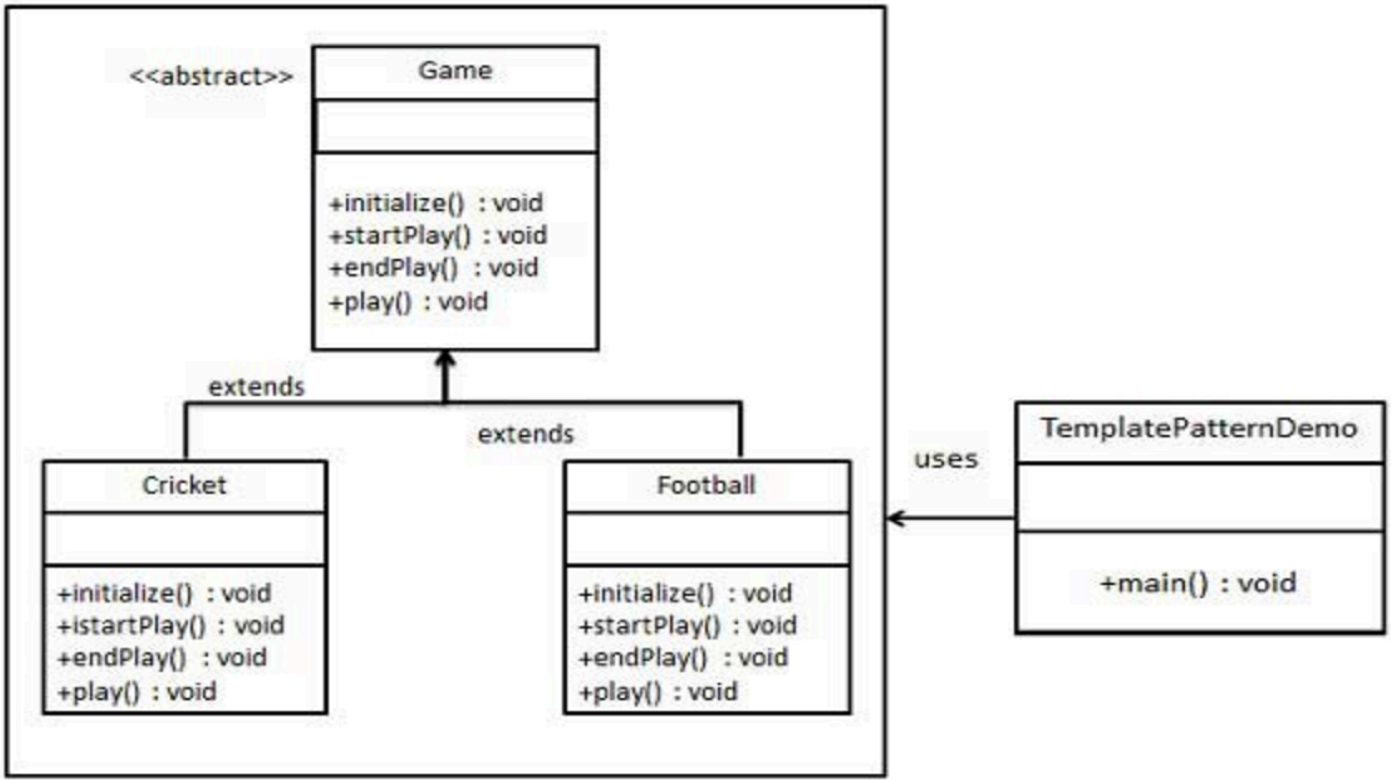
FactoryPatternDemo, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



```java
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

```java
ShapeFactory shapeFactory = new ShapeFactory();

//get an object of Circle and call its draw method.
Shape shape1 = shapeFactory.getShape("CIRCLE");

//call draw method of Circle
shape1.draw();

//get an object of Rectangle and call its draw method.
Shape shape2 = shapeFactory.getShape("RECTANGLE");

//call draw method of Rectangle
shape2.draw();
```

Template Pattern



In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.

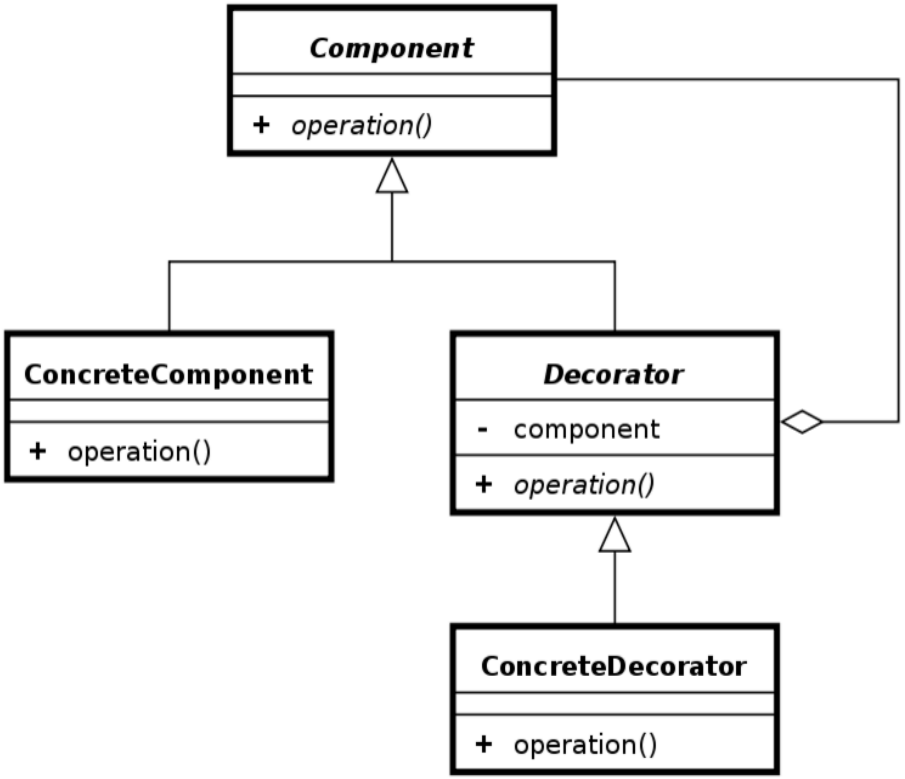
```java
public class Football extends Game {

   @Override
   void endPlay() {
      System.out.println("Football Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Football Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Football Game Started. Enjoy the game!");
   }
}
```

```java
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //template method
   public final void play(){

      //initialize the game
      initialize();

      //start game
      startPlay();

      //end game
      endPlay();
   }
}
```

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

      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

Decorator(Wrapper) Pattern

It is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class



The decorator pattern can be used to extend (decorate) the functionality of a certain object statically, or in some cases at run-time, independently of other instances of the same class

```java
public interface Coffee {
    public double getCost(); // Returns the cost
    public String getIngredients(); // Returns t
}

// Extension of a simple coffee without any extr
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1;
    }

    @Override
    public String getIngredients() {
        return "Coffee";
    }
}
```

```java
// Abstract decorator class - note that it implements Coffee interface
public abstract class CoffeeDecorator implements Coffee {
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    public double getCost() { // Implementing methods of the interface
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}

// Decorator WithMilk mixes milk into coffee.
// Note it extends CoffeeDecorator.
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    public double getCost() { // Overriding methods
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
}
```

```java
public class Main {
    public static void printInfo(Coffee c) {
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngr
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```

```
Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
```