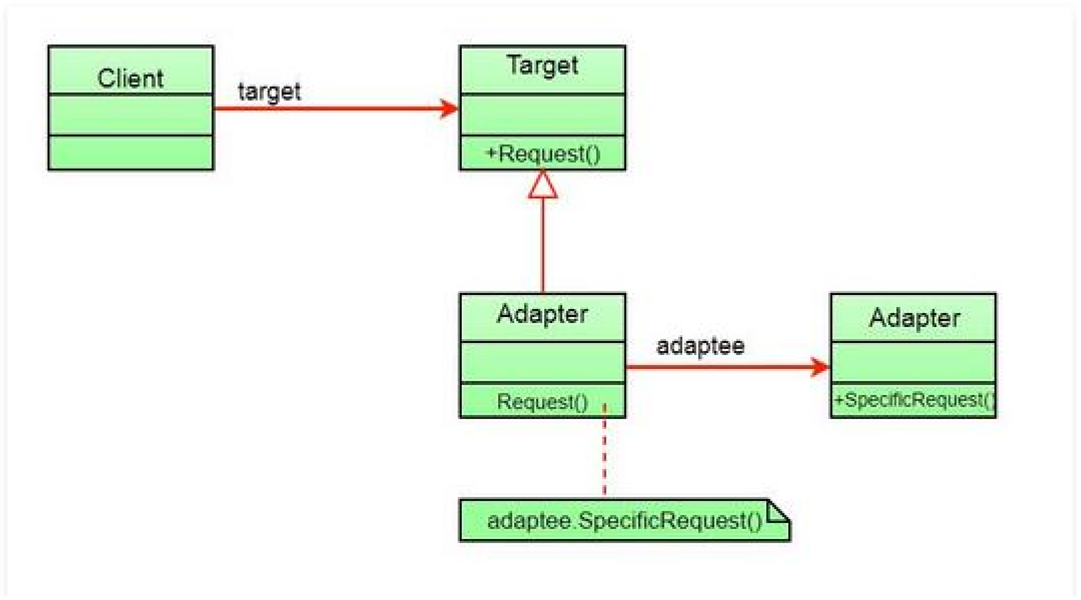
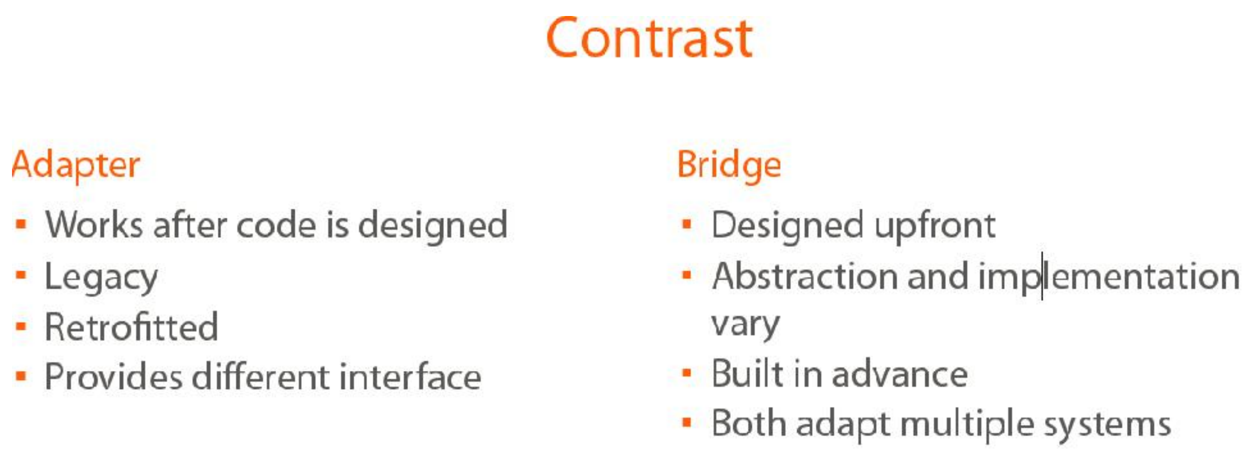
1. Adapter Pattern:

Why Adapter Pattern?

The adapter pattern is a great pattern for connecting new code to legacy code without having to change the working contract that was produced from the legacy code originally.

Class Diagram:





Contrast

Adapter

- Works after code is designed
- Legacy
- Retrofitted
- Provides different interface

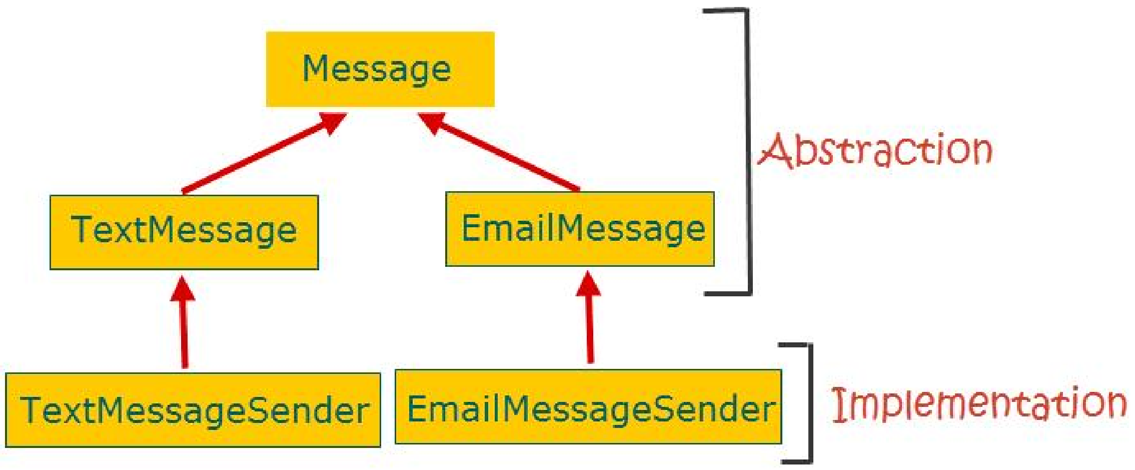
Bridge

- Designed upfront
- Abstraction and implementation vary
- Built in advance
- Both adapt multiple systems

2. Bridge Pattern

The bridge pattern is very similar to the adapter with the main difference being that the bridge works with new code whereas the adapter works with legacy code. Decouples an abstraction so two classes can vary independently.

To understand how the bridge pattern works, consider a messaging application that clients can use to send different types of messages, such as a text or an email message. The most intuitive approach is to first create an interface or an abstract base class, `Message`. Next, we create the derived classes: `TextMessage` and `EmailMessage`. Finally, to send messages, we create two message sender classes: `TextMessageSender` that extends `TextMessage` and `EmailMessageSender` that extends `EmailMessage`. This is how our inheritance hierarchy looks like.

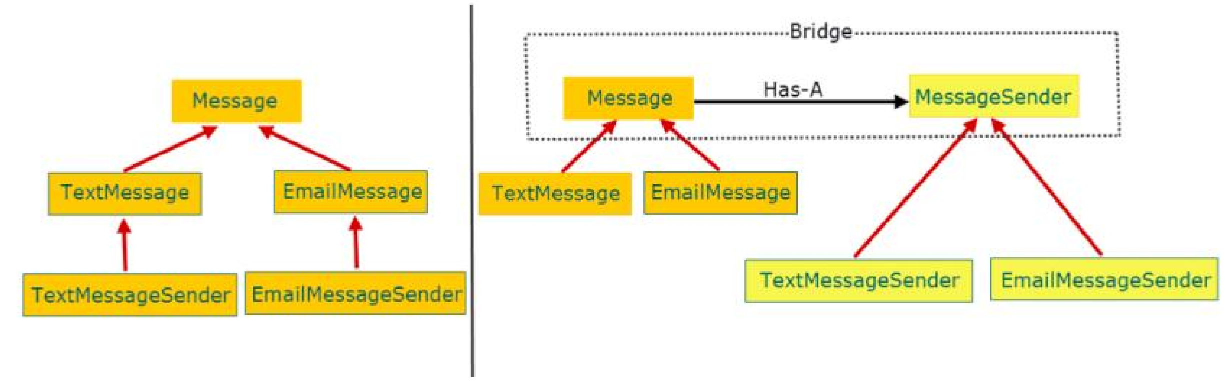


At first sight there appears nothing wrong in the design above. But if you look deep you will notice that, the abstraction part- the part that clients interact with, and the implementation part- the part that provides the core functionality of sending messages, are tightly integrated. Our design relies on inheritance and one inherent disadvantage is that it breaks encapsulation. As a developer of the `EmailMessageSender` subclass, you have to know about the internals of the `EmailMessage` superclass, which means the encapsulation in the superclass is broken.

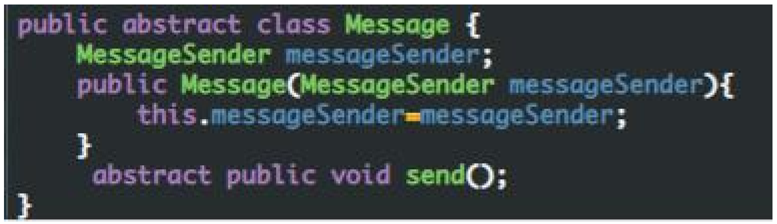
Our design is also fragile. As an example, if we change the implementation to allow clients to optionally encrypt message before sending, we will need to update the abstraction part to make the encryption functionality available to clients.

Another issue is reusability. If we want to reuse only the implementation (message sending) part in some other application, we will have to take along the abstraction part as extra baggage.

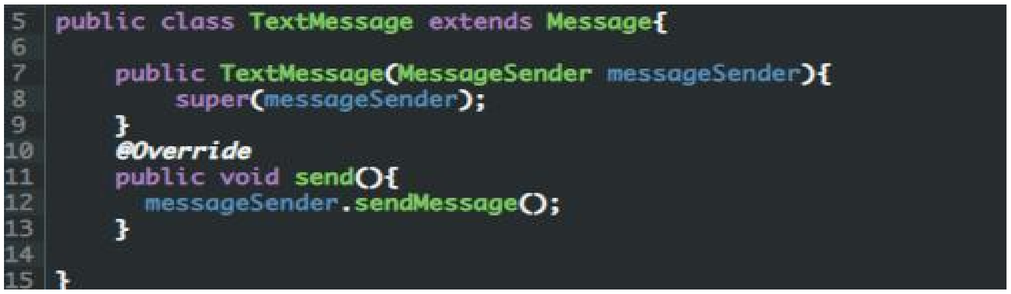
The bridge pattern addresses all such issues by separating the abstraction and implementation into two class hierarchies. This figure shows the design without and with the bridge pattern.



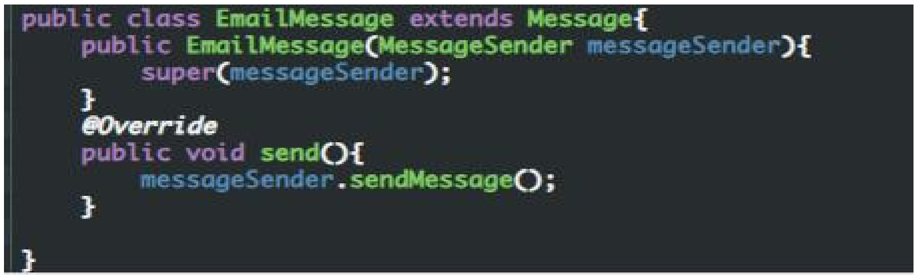
With the bridge pattern, the abstraction maintains a **Has-A** relationship with the implementation instead of a **IS-A** relationship. The **Has-A** relationship is achieved through composition where the abstraction maintains a reference of the implementation and forwards client requests to it.



```
public abstract class Message {
    MessageSender messageSender;
    public Message(MessageSender messageSender){
        this.messageSender=messageSender;
    }
     abstract public void send();
}
```

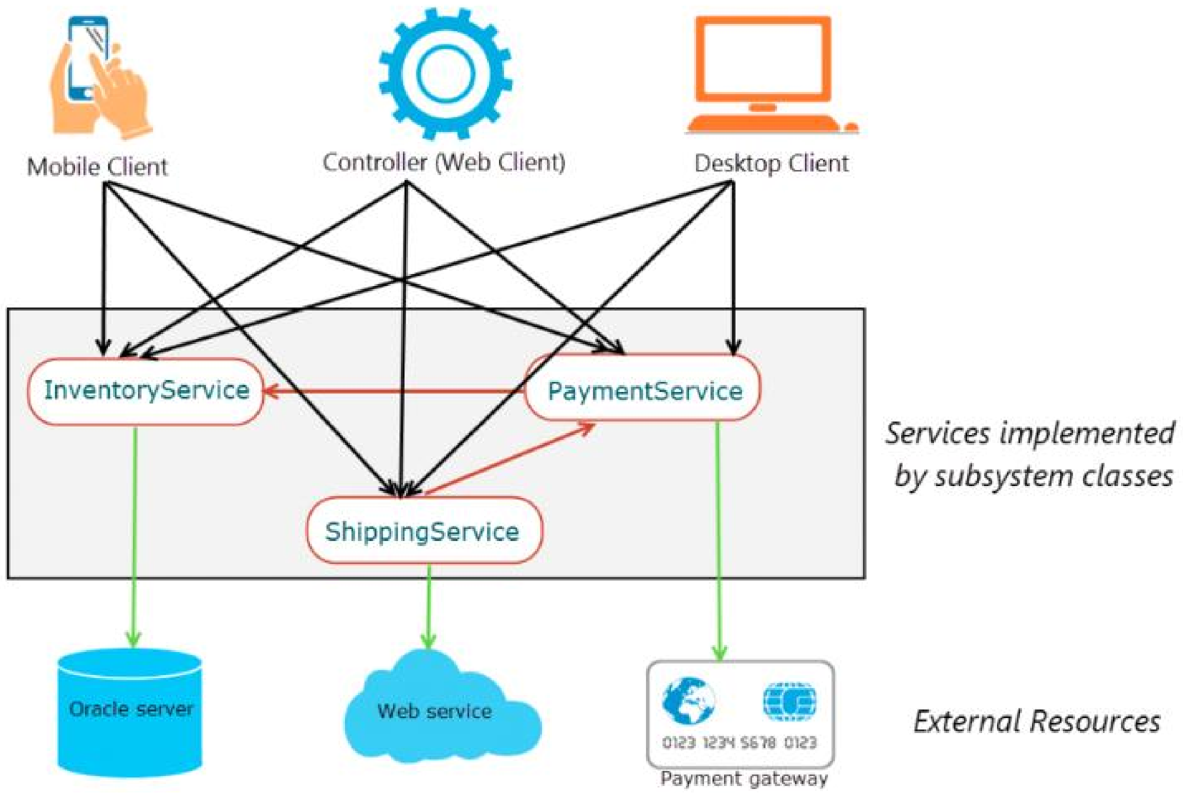


```
public class TextMessage extends Message{

    public TextMessage(MessageSender messageSender){
        super(messageSender);
    }
    @Override
    public void send(){
      messageSender.sendMessage();
    }

}
```



```
public class EmailMessage extends Message{
    public EmailMessage(MessageSender messageSender){
        super(messageSender);
    }
    @Override
    public void send(){
        messageSender.sendMessage();
    }

}
```

3. **Facade Pattern**



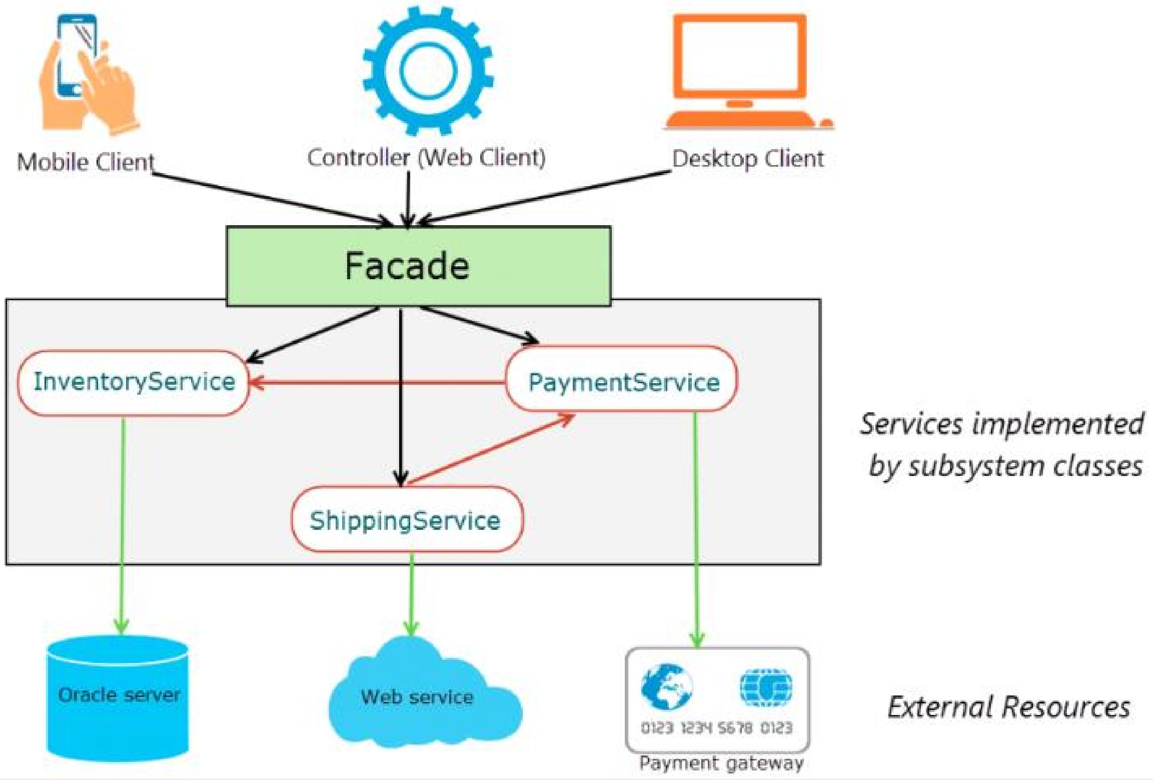
Problem with the above Design:

1. the clients need to make multiple interactions with the services implemented by subsystem classes in order to submit an order fulfillment process of an e-commerce store. It means, our clients are tightly coupled with the subsystem classes – a fundamental violation of the SOLID design principles

2. Things can get worse if a new **InvoicingService** is introduced in the service layer or the existing **ShippingService** is updated to make the logistic part internal to the organization

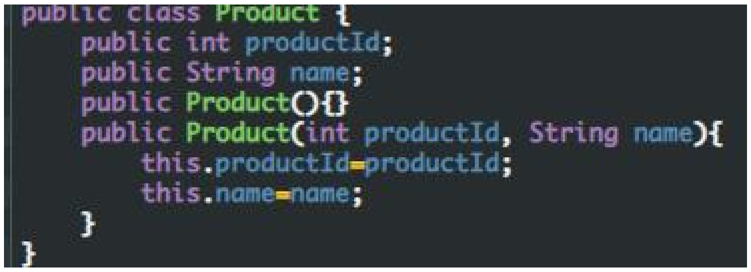
Solution:

Rather than having the clients tightly coupled to the subsystems, we need is an interface which makes the subsystems easier to use. In our example, our clients just want to place an order. They don't really need to care about dealing with inventory, shipping or payments. The Facade pattern is a way of providing a simple way for the clients to interact with the subsystems.

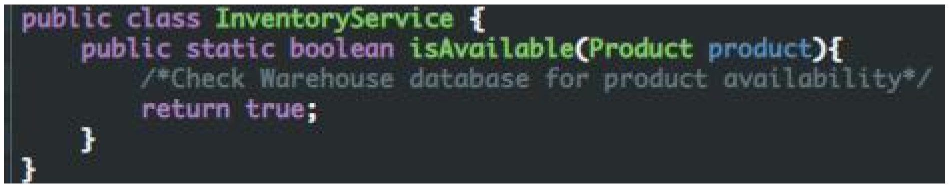


- **Facade**: Delegates client requests to appropriate subsystem classes.
- **Subsystem classes**: Implements subsystem functionalities. Subsystem classes are used by the facade, but not the other way around. We will come to it later in this post.
- **Client**: Requests the facade to perform some action.

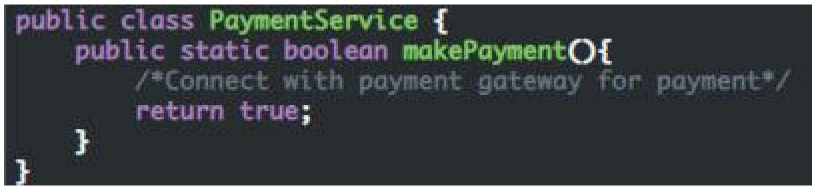
Applying the Facade Pattern



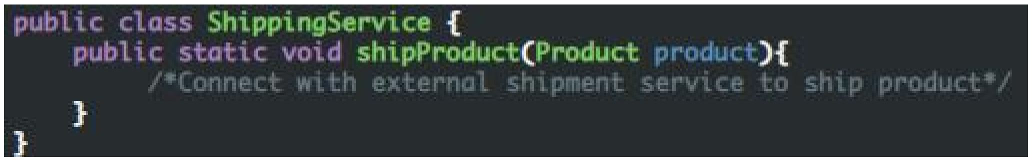
```java
public class Product {
    public int productId;
    public String name;
    public Product(){}
    public Product(int productId, String name){
        this.productId=productId;
        this.name=name;
    }
}
```



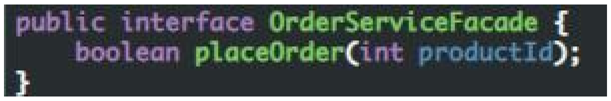
```java
public class InventoryService {
    public static boolean isAvailable(Product product){
        /*Check Warehouse database for product availability*/
        return true;
    }
}
```



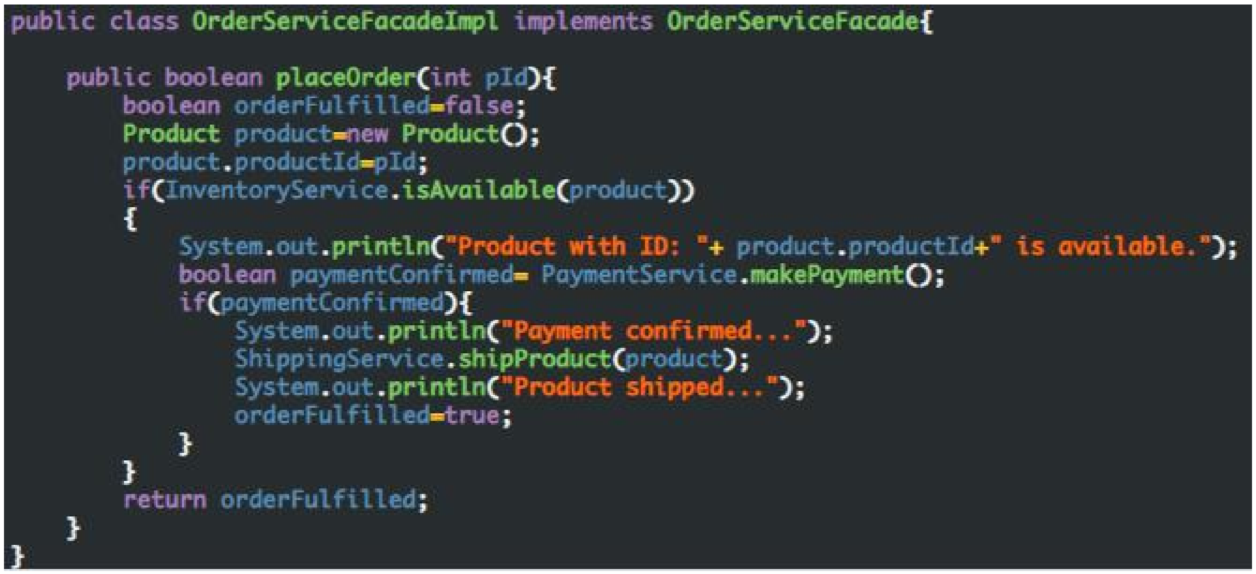
```java
public class PaymentService {
    public static boolean makePayment(){
        /*Connect with payment gateway for payment*/
        return true;
    }
}
```



```java
public class ShippingService {
    public static void shipProduct(Product product){
         /*Connect with external shipment service to ship product*/
    }
}
```



```java
public interface OrderServiceFacade {
    boolean placeOrder(int productId);
}
```

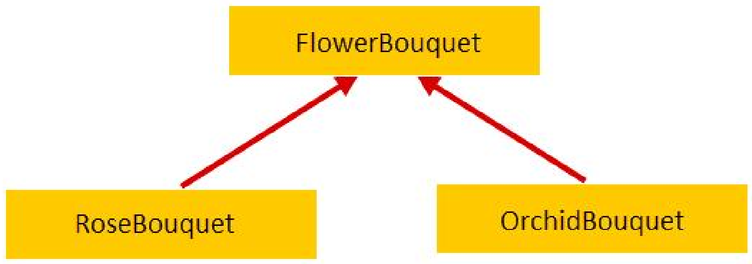


```java
public class OrderServiceFacadeImpl implements OrderServiceFacade{

    public boolean placeOrder(int pId){
        boolean orderFulfilled=false;
        Product product=new Product();
        product.productId=pId;
        if(InventoryService.isAvailable(product))
        {
            System.out.println("Product with ID: "+ product.productId+" is available.");
            boolean paymentConfirmed= PaymentService.makePayment();
            if(paymentConfirmed){
                System.out.println("Payment confirmed...");
                ShippingService.shipProduct(product);
                System.out.println("Product shipped...");
                orderFulfilled=true;
            }
        }
        return orderFulfilled;
    }
}
```

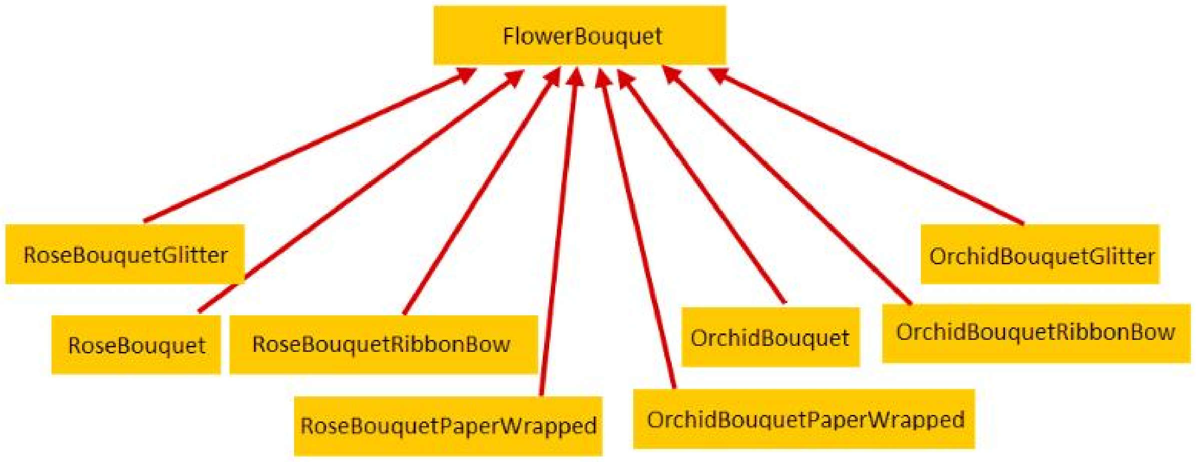
4. Decorator Pattern

Allows for an object's behavior to be extended dynamically at run time

Consider a florist who sells flower bouquets, such as rose bouquet, orchid bouquet, and so on. When a customer walks in, the florist describes the bouquets and tells their prices. We can model the requirements with a **FlowerBouquet** abstract base class and subclasses for specific bouquets.



Customers, in addition to a bouquet, can ask the florist to decorate it with paper wrappings, ribbon bows, and glitters for which the florist charges extra. To address the new requirement, we can add new subclasses of **FlowerBouquet**, one each to represent a bouquet with an additional decoration, and this is how our design looks like now.

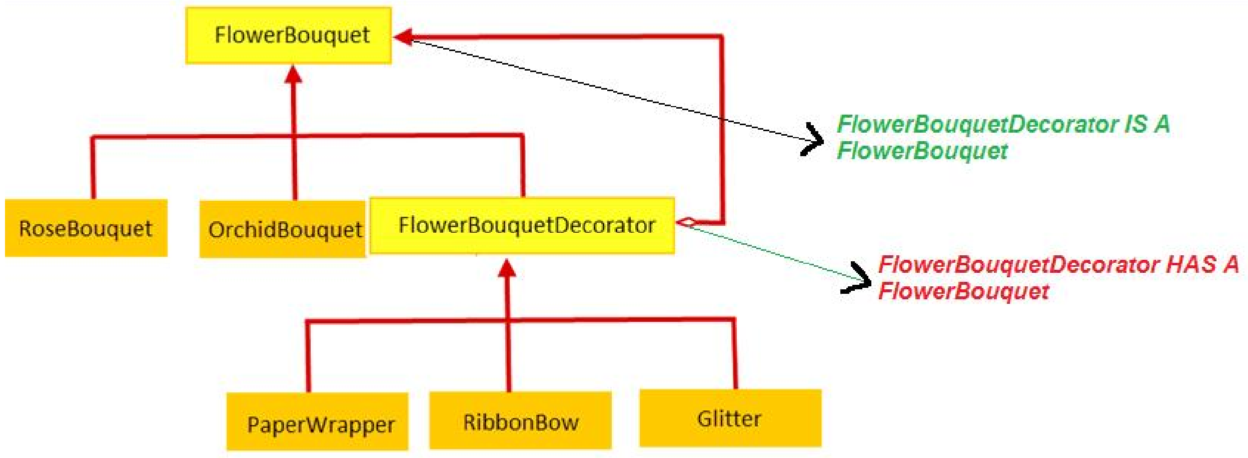


Problems:

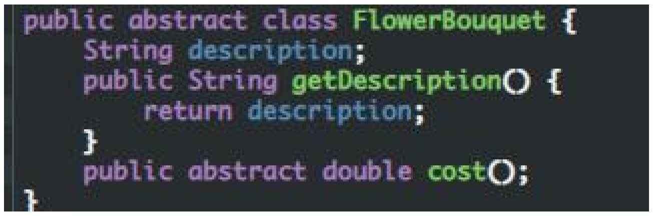
- *What if we want a rose bouquet with double paper wrap?*
- *What if we want to add a new lily bouquet?*
- *What if we want to add new ornamental leaves decoration?*

Solution:

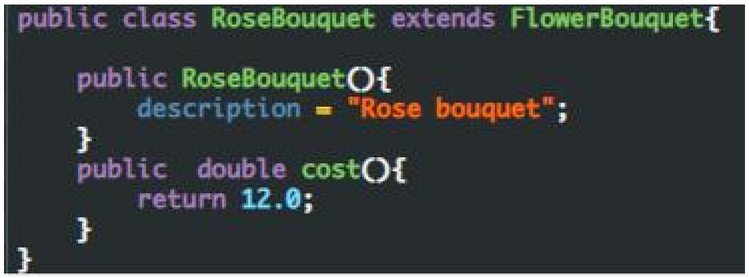
Clearly our design is flawed, but this is an ideal use case for the decorator pattern. By using the decorator pattern, we can create a flower bouquet and dynamically decorate it with any numbers of features at run time. Without the flower bouquet knowing it that is "*being decorated*". For that, we will create an abstract base class **FlowerBouquet** and specific subclasses **RoseBouquet** and **OrchidBouquet** to extend from it. We will also create an abstract **FlowerBouquetDecorator** class to extend **FlowerBouquet**. For each decorator, we will create the **Glitter**, **PaperWrapper**, and **RibbonBow** classes to extend from **FlowerBouquetDecorator**. This is how the design will be.



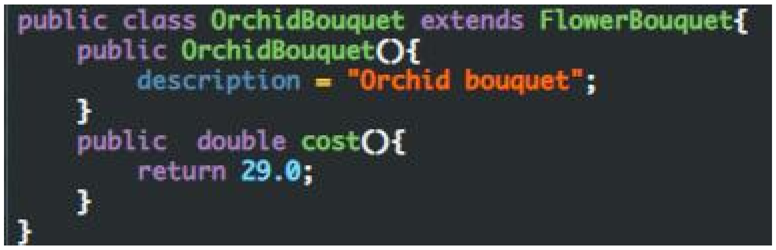
- **Component** (**FlowerBouquet**): Is an abstract base class that can be decorated with responsibilities dynamically.
- **ConcreteComponent**(**RoseBouquet** and **OrchidBouquet**): Are concrete classes that extends **Component** to represent objects to which additional responsibilities can be attached.
- **Decorator** (**FlowerBouquetDecorator**): Is an abstract class that extends **Component** and acts as the base class for concrete decorator classes.
- ConcreteDecorator (**PapperWrapper**, **RibbonBow**, and **Glitter**): Are concrete classes that extends **Decorator** to decorate **Components** with responsibilities.



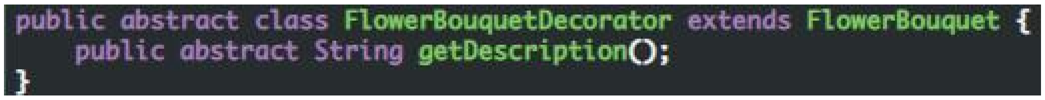
```java
public abstract class FlowerBouquet {
    String description;
    public String getDescription() {
        return description;
    }
    public abstract double cost();
}
```



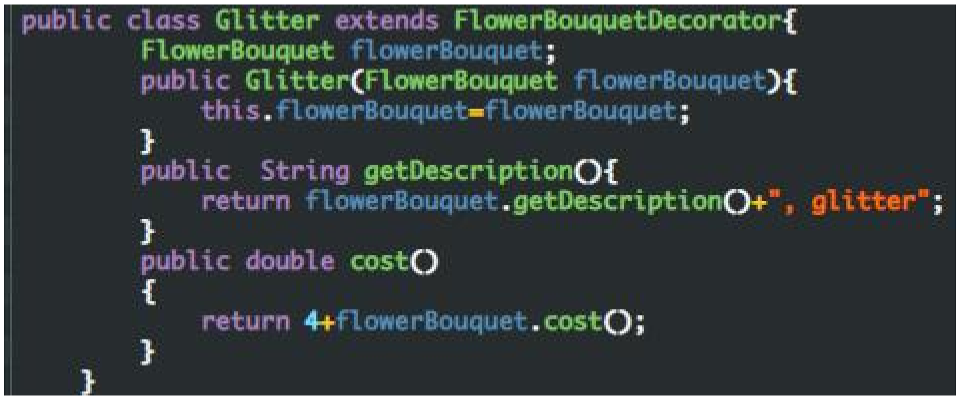
```java
public class RoseBouquet extends FlowerBouquet{

    public RoseBouquet(){
        description = "Rose bouquet";
    }
    public  double cost(){
        return 12.0;
    }
}
```



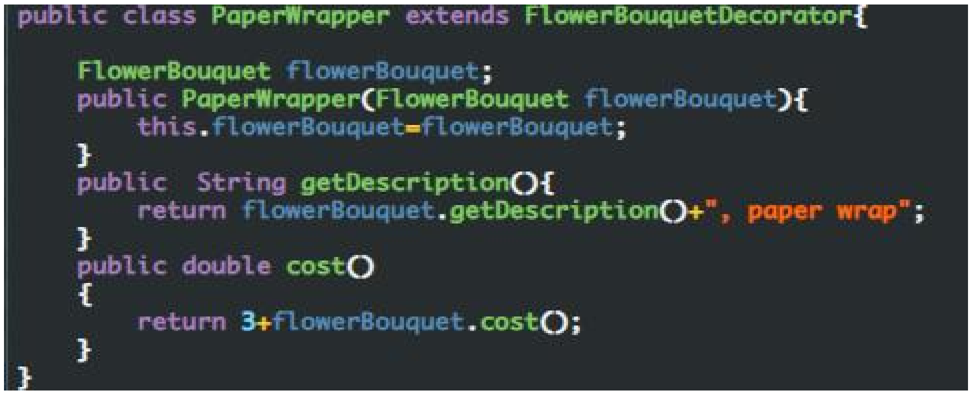
```java
public class OrchidBouquet extends FlowerBouquet{
    public OrchidBouquet(){
        description = "Orchid bouquet";
    }
    public  double cost(){
        return 29.0;
    }
}
```



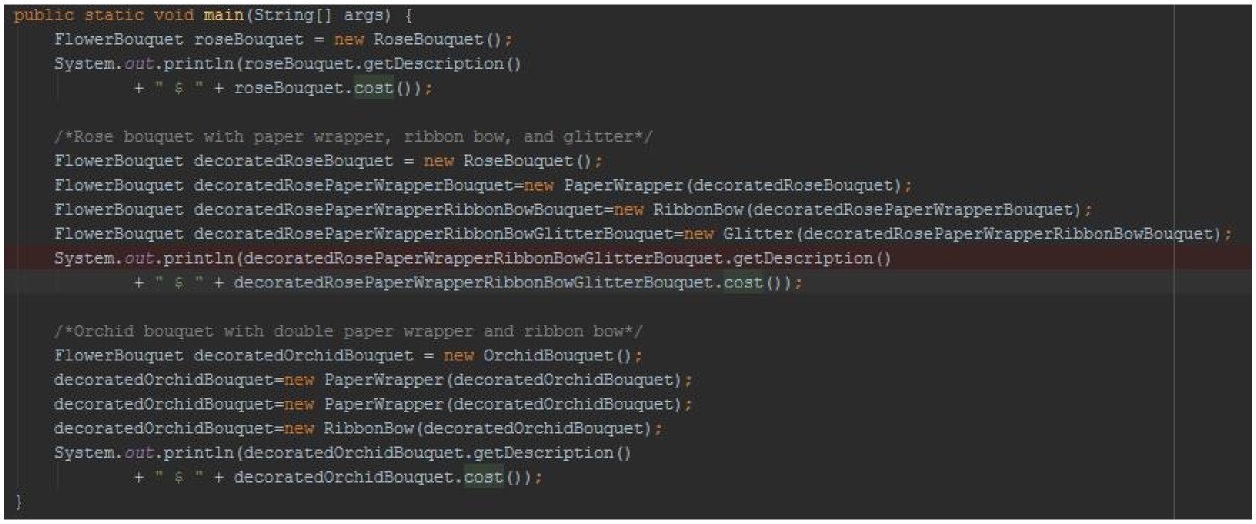
```java
public abstract class FlowerBouquetDecorator extends FlowerBouquet {
    public abstract String getDescription();
}
```



```java
public class Glitter extends FlowerBouquetDecorator{
        FlowerBouquet flowerBouquet;
        public Glitter(FlowerBouquet flowerBouquet){
            this.flowerBouquet=flowerBouquet;
        }
        public  String getDescription(){
            return flowerBouquet.getDescription()+", glitter";
        }
        public double cost()
        {
            return 4+flowerBouquet.cost();
        }
    }
```



```java
public class PaperWrapper extends FlowerBouquetDecorator{

    FlowerBouquet flowerBouquet;
    public PaperWrapper(FlowerBouquet flowerBouquet){
        this.flowerBouquet=flowerBouquet;
    }
    public  String getDescription(){
        return flowerBouquet.getDescription()+", paper wrap";
    }
    public double cost()
    {
        return 3+flowerBouquet.cost();
    }
}
```



```
public static void main(String[] args) {
    FlowerBouquet roseBouquet = new RoseBouquet();
    System.out.println(roseBouquet.getDescription()
            + " $ " + roseBouquet.cost());

    /*Rose bouquet with paper wrapper, ribbon bow, and glitter*/
    FlowerBouquet decoratedRoseBouquet = new RoseBouquet();
    FlowerBouquet decoratedRosePaperWrapperBouquet=new PaperWrapper(decoratedRoseBouquet);
    FlowerBouquet decoratedRosePaperWrapperRibbonBowBouquet=new RibbonBow(decoratedRosePaperWrapperBouquet);
    FlowerBouquet decoratedRosePaperWrapperRibbonBowGlitterBouquet=new Glitter(decoratedRosePaperWrapperRibbonBowBouquet);
    System.out.println(decoratedRosePaperWrapperRibbonBowGlitterBouquet.getDescription()
            + " $ " + decoratedRosePaperWrapperRibbonBowGlitterBouquet.cost());

    /*Orchid bouquet with double paper wrapper and ribbon bow*/
    FlowerBouquet decoratedOrchidBouquet = new OrchidBouquet();
    decoratedOrchidBouquet=new PaperWrapper(decoratedOrchidBouquet);
    decoratedOrchidBouquet=new PaperWrapper(decoratedOrchidBouquet);
    decoratedOrchidBouquet=new RibbonBow(decoratedOrchidBouquet);
    System.out.println(decoratedOrchidBouquet.getDescription()
            + " $ " + decoratedOrchidBouquet.cost());
}
```

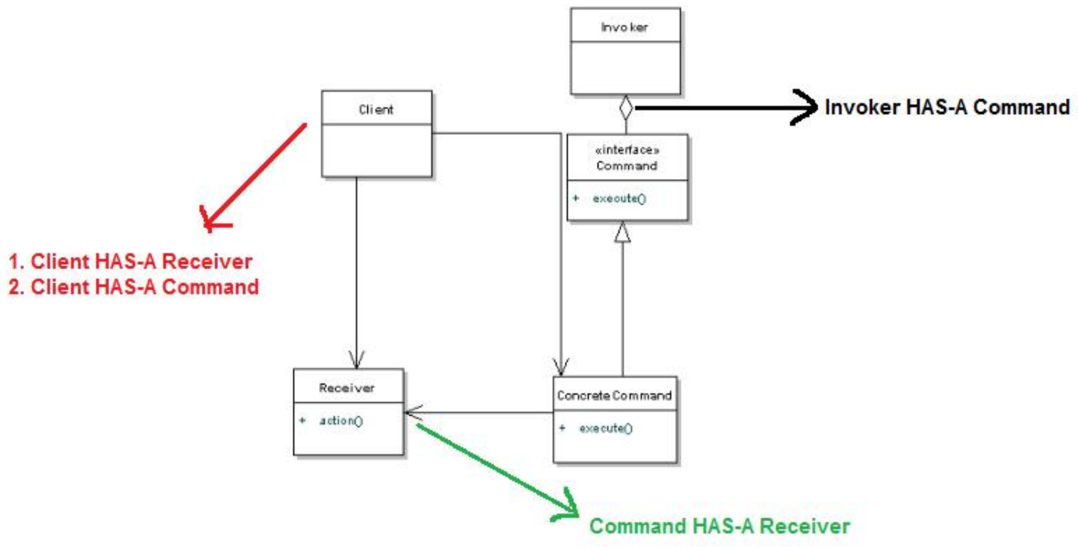
5. Command Pattern

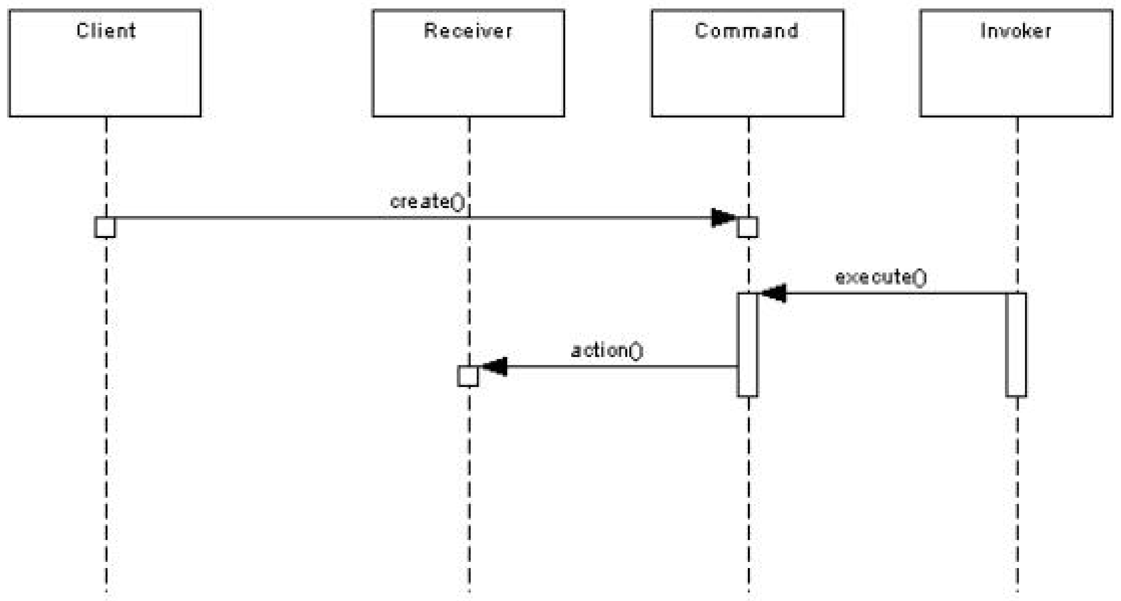
Communication between objects in enterprise applications starts from an object sending a request to another object. The object receiving the request can either process the request and send back a response or forward the request to another object for processing. Typically requests are made by the invoker (the object making the request) through method calls on the object that processes the request, which we will refer as the receiver. As a result, the invoker and the receiver are tightly coupled. This violates the SOLID design principles that advocates loosely coupled components to ensure that changes made to one component does not affect the other components of the application.

The Command Pattern is a proven solution that addresses such recurring problems of tightly coupled invoker and receiver components in applications. This pattern states that requests should be encapsulated as objects that like any other objects can be stored and passed around the application. Requests encapsulated as objects are known as commands.

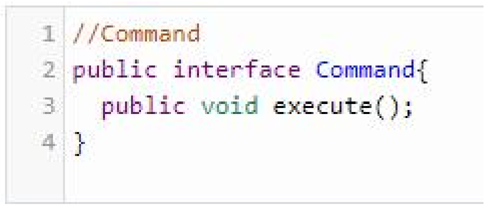
In the command pattern, the invoker issues commands without knowing anything about the receiver. In fact the invoker issuing the command doesn't even know what operation will be carried out on issuing a command. Let's look at it from programming point of view.

So what does this mean in a class diagram?

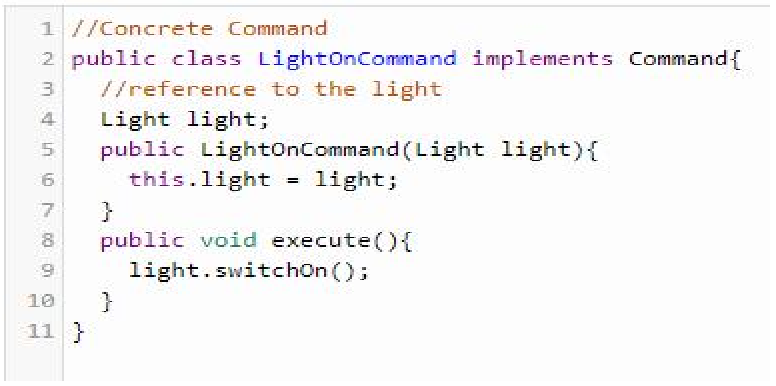




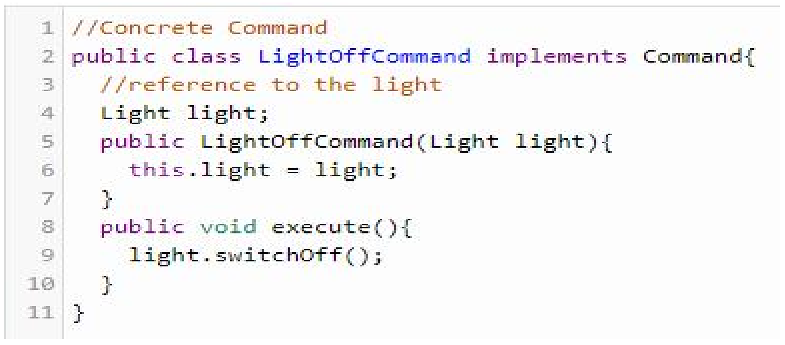
So How Does It Work In Java?



```java
//Command
public interface Command{
  public void execute();
}
```



```java
//Concrete Command
public class LightOnCommand implements Command{
  //reference to the light
  Light light;
  public LightOnCommand(Light light){
    this.light = light;
  }
  public void execute(){
    light.switchOn();
  }
}
```



```java
//Concrete Command
public class LightOffCommand implements Command{
  //reference to the light
  Light light;
  public LightOffCommand(Light light){
    this.light = light;
  }
  public void execute(){
    light.switchOff();
  }
}
```

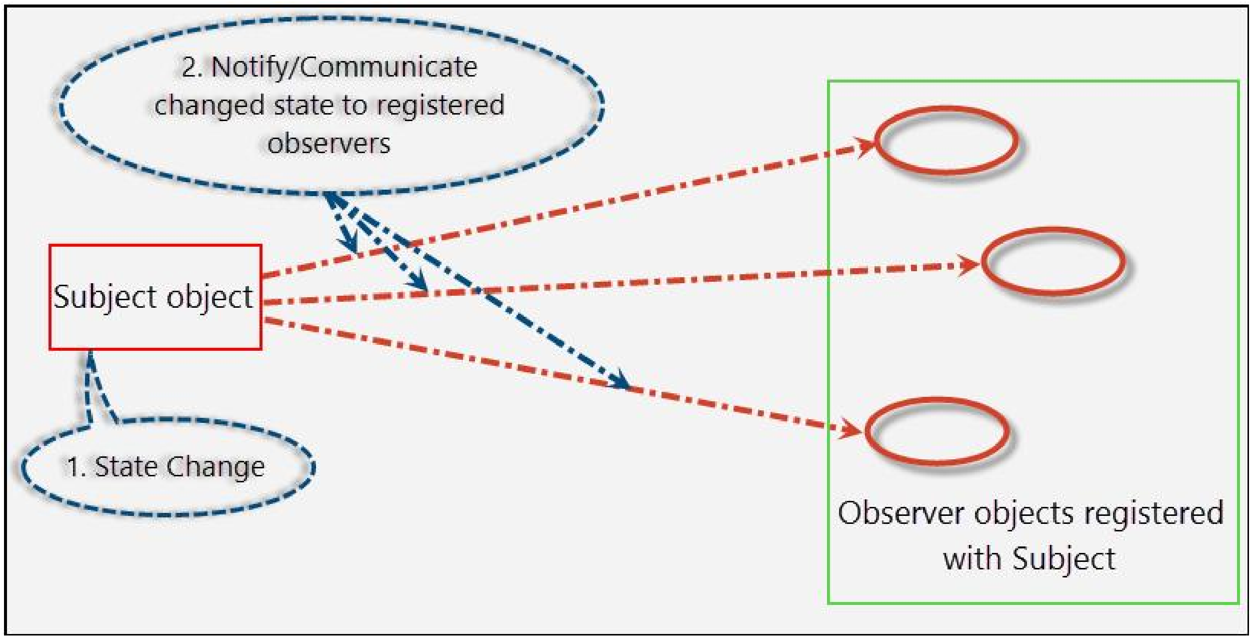
```java
//Receiver
public class Light{
  private boolean on;
  public void switchOn(){
    on = true;
  }
  public void switchOff(){
    on = false;
  }
}
```

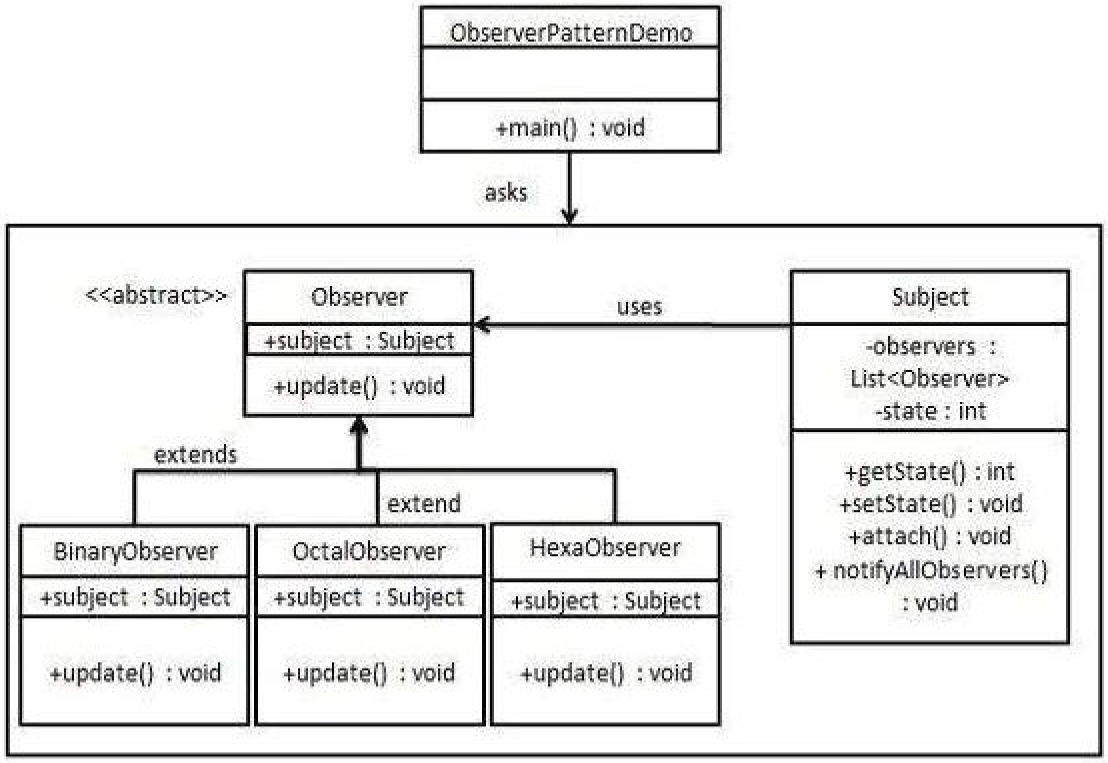
```java
//Invoker
public class RemoteControl{
  private Command command;
  public void setCommand(Command command){
    this.command = command;
  }
  public void pressButton(){
    command.execute();
  }
}
```

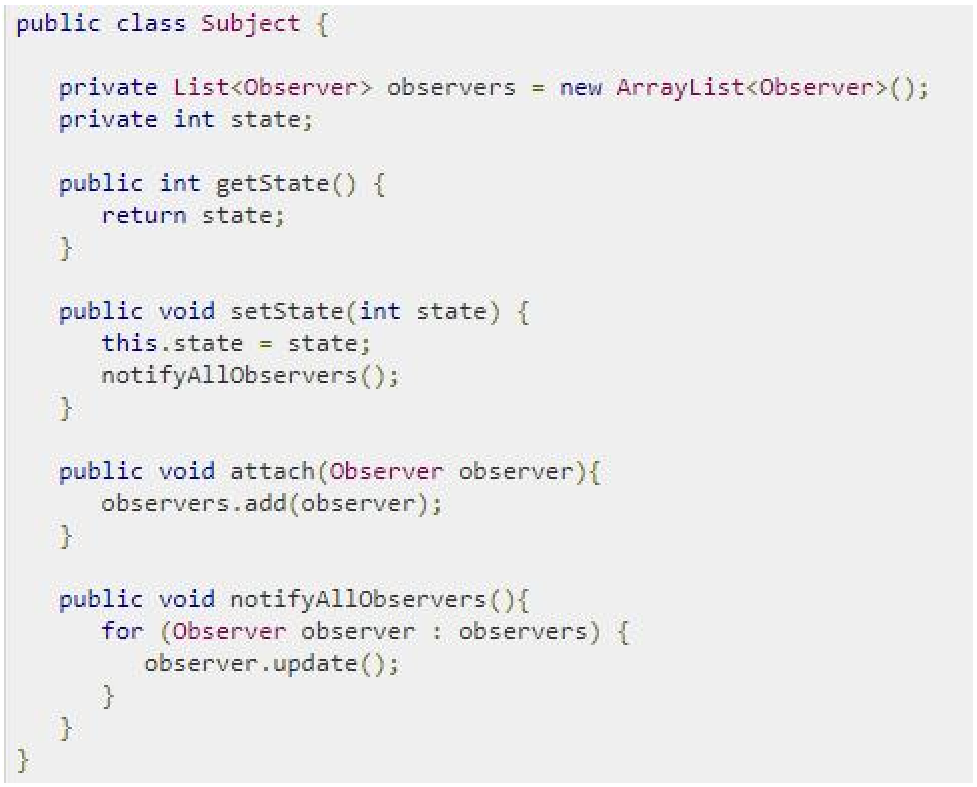
```java
//Client
public class Client{
  public static void main(String[] args)    {
    RemoteControl control = new RemoteControl();
    Light light = new Light();
    Command lightsOn = new LightsOnCommand(light);
    Command lightsOff = new LightsOffCommand(light);

    //switch on
    control.setCommand(lightsOn);
    control.pressButton();

    //switch off
    control.setCommand(lightsOff);
    control.pressButton();
  }
}
```

6. Observer Pattern

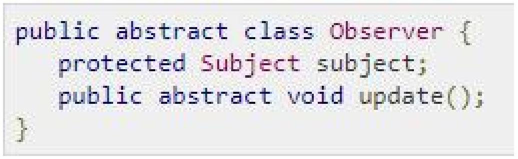
Observer Pattern is a publish/subscribe pattern which allows a number of observer objects to see an event. *In this pattern we will refer publisher as subject and subscriber as observer*.


2. Notify/Communicate changed state to registered observers
Subject object
1. State Change
Observer objects registered with Subject

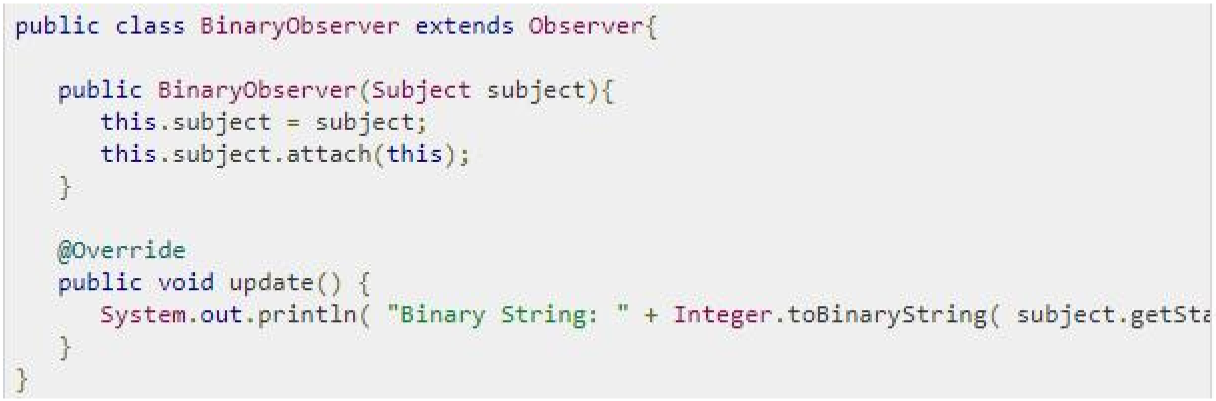




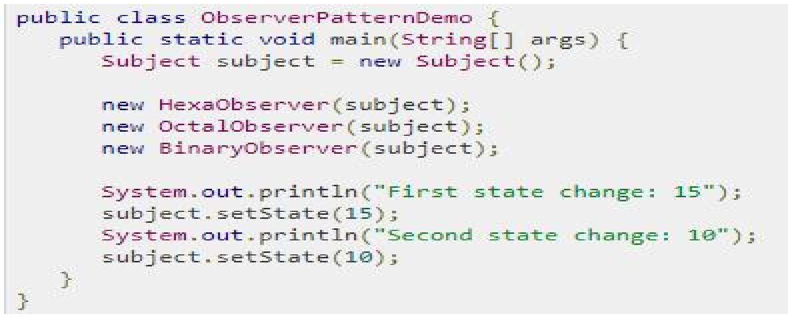
```java
public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }

   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```



```java
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```



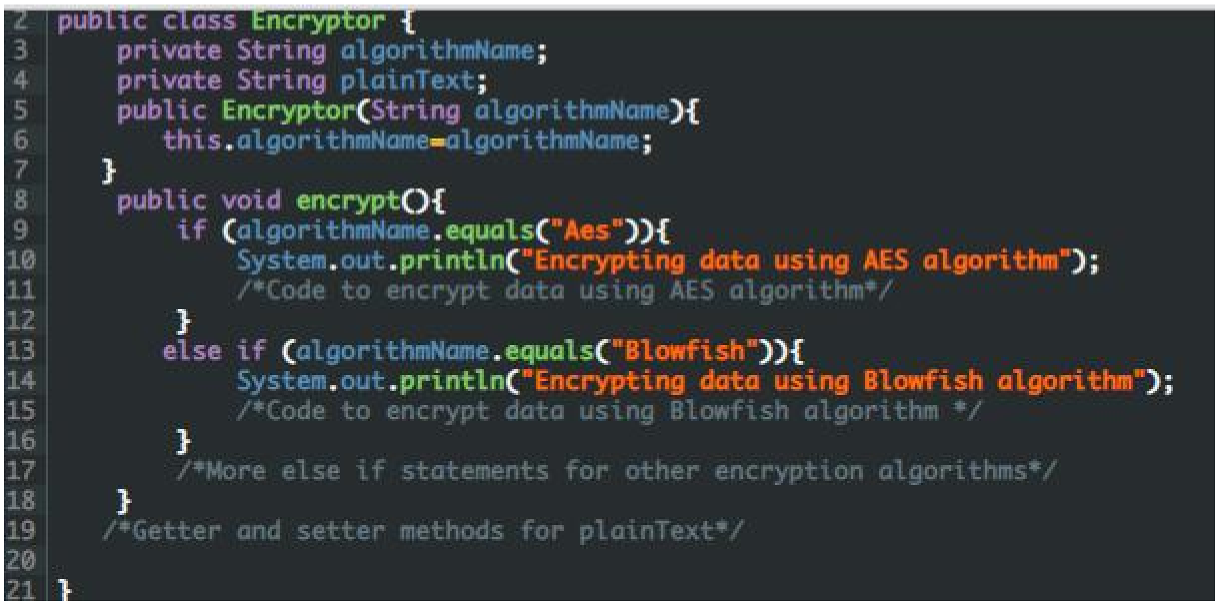
```java
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString( subject.getSta
   }
}
```



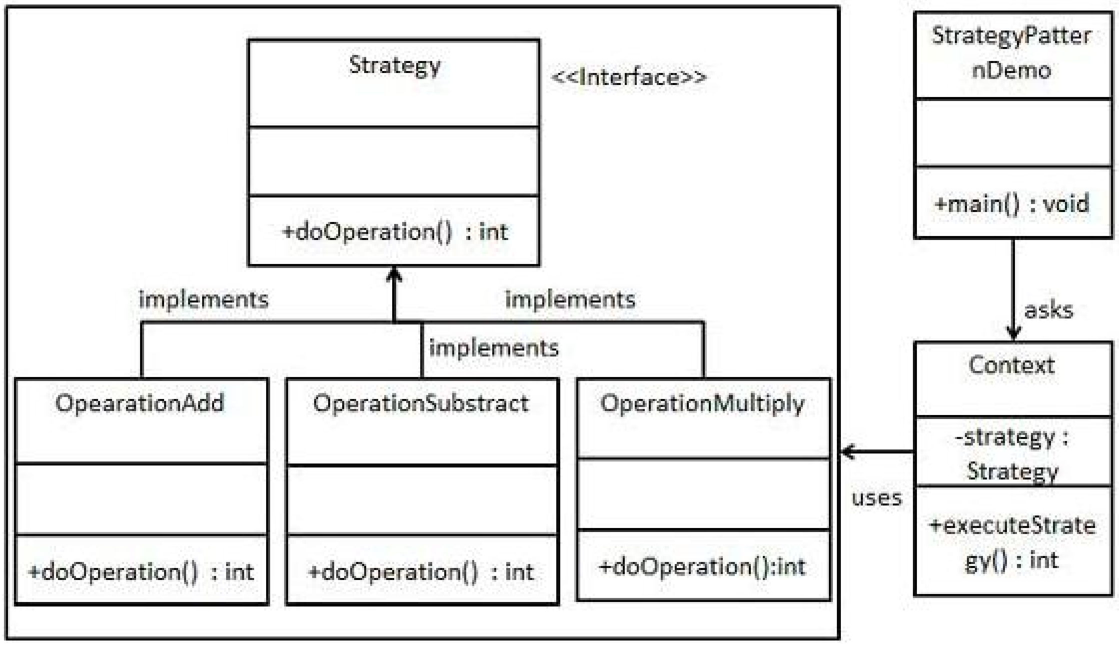
```java
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

7. Strategy Pattern:

In enterprise applications, you will often have objects that use multiple algorithms to implement some business requirements. A common example is a number sorting class that supports multiple sorting algorithms, such as bubble sort, merge sort, and quick sort. Similarly, a file compression class can support different compression algorithm, such as ZIP, GZIP, LZ4, or even a custom compression algorithm. Another example can be a data encryption class that encrypts data using different encryption algorithms, such as AES, TripleDES, and Blowfish. Typically, programmers tend to bundle all the algorithm logic in the host class, resulting in a monolithic class with multiple switch case or conditional statements. The following example shows the structure of one such class that supports multiple algorithms to encrypt data.



```java
public class Encryptor {
    private String algorithmName;
    private String plainText;
    public Encryptor(String algorithmName){
       this.algorithmName=algorithmName;
   }
    public void encrypt(){
        if (algorithmName.equals("Aes")){
            System.out.println("Encrypting data using AES algorithm");
            /*Code to encrypt data using AES algorithm*/
        }
       else if (algorithmName.equals("Blowfish")){
            System.out.println("Encrypting data using Blowfish algorithm");
            /*Code to encrypt data using Blowfish algorithm */
        }
        /*More else if statements for other encryption algorithms*/
    }
   /*Getter and setter methods for plainText*/

}
```



Note:

- **Strategy** (**Strategy**): Is an interface common to all supported algorithm-specific classes.
- **ConcreteStrategy** (**OperationAdd, OperationSubtract** and **OperationMultiply**): Implements the algorithm using the **Strategy** interface.
- **Context** (**Context**): Provides the interface to client for encrypting data.
  The **Context** maintains a reference to a **Strategy** object and is instantiated and initialized by clients with a **ConcreteStrategy** object.

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

```java
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

```java
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

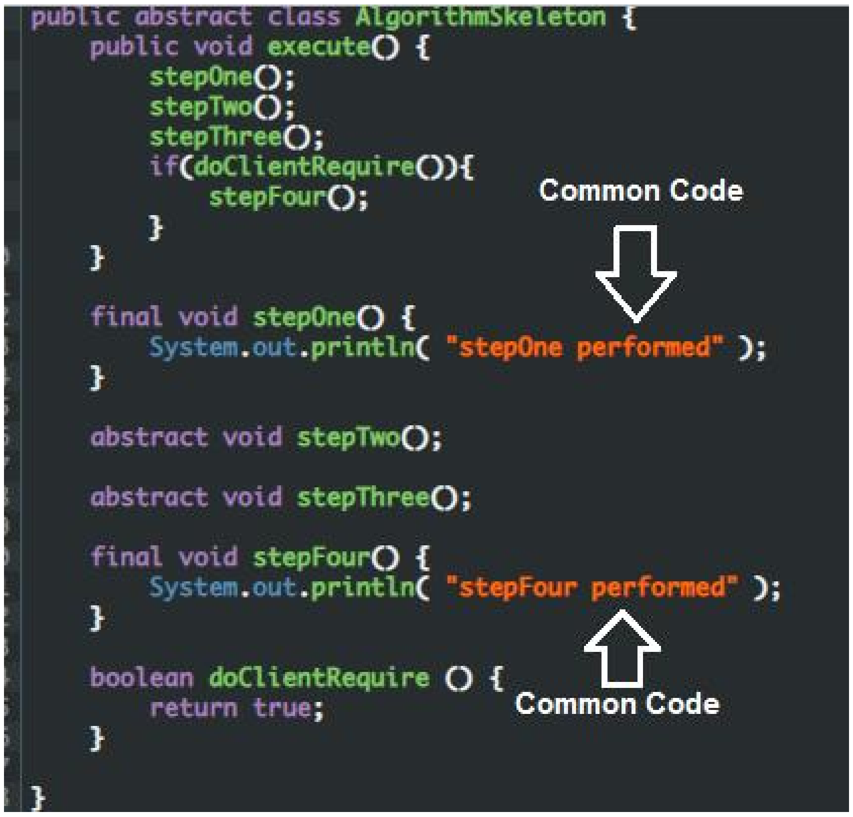
8. *Template Method Pattern*

If you have already gone through my post on Strategy pattern, understanding the Template Method pattern will be easy. If you haven't done that yet, I recommend you to do so. But, if you want to jump start with Template Method, let me start with a brief introduction on the need of this pattern.

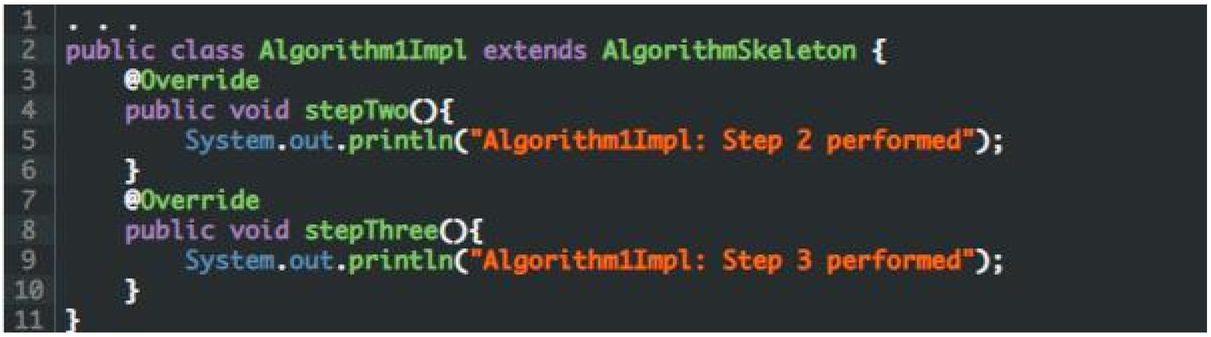
In enterprise applications, some classes need to support multiple similar algorithms to perform some business requirements. A number sorting class that supports multiple sorting algorithms, such as bubble sort, merge sort, and quick sort is an example of such a class. Another example is a data encryption class that encrypts data using different encryption algorithms, such as AES, TripleDES, and Blowfish.

So how does the Template Method pattern fits in? Also, why do we need it at all if the Strategy Pattern is getting all the right things done? The answer is – The Strategy Pattern is not the optimal solution for all types of algorithms.

This is the Template Method Pattern. In a nutshell, this pattern defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior. The interface method in the abstract class that clients call is the template method – In simple terms, a method that defines an algorithm as a series of steps.



```
public abstract class AlgorithmSkeleton {
    public void execute() {
        stepOne();
        stepTwo();
        stepThree();
        if(doClientRequire()){
            stepFour();
        }
    }

    final void stepOne() {
        System.out.println( "stepOne performed" );
    }

    abstract void stepTwo();

    abstract void stepThree();

    final void stepFour() {
        System.out.println( "stepFour performed" );
    }

    boolean doClientRequire () {
        return true;
    }

}
```



```java
. . .
public class Algorithm1Impl extends AlgorithmSkeleton {
    @Override
    public void stepTwo(){
        System.out.println("Algorithm1Impl: Step 2 performed");
    }
    @Override
    public void stepThree(){
        System.out.println("Algorithm1Impl: Step 3 performed");
    }
}
```

9. Singleton Method Pattern:

1. There should be only one instance allowed for a class
2. We should allow global point of access to that single instance.

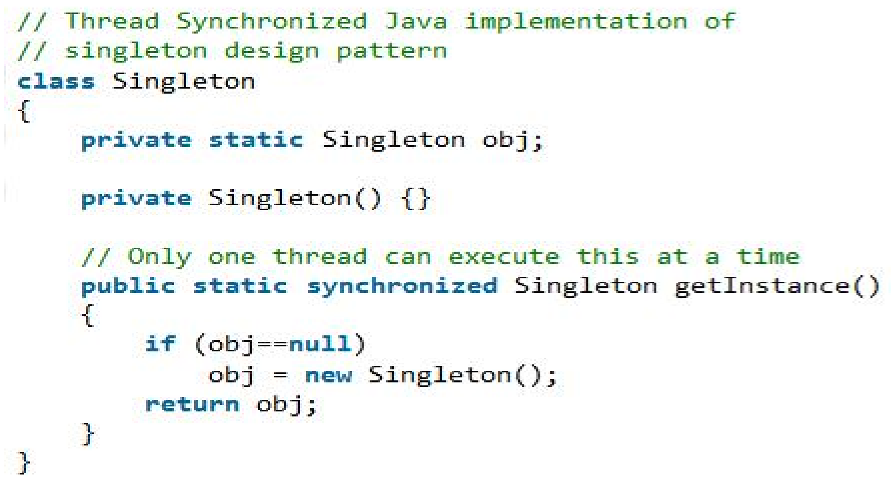
A. <u>Question?</u>

In singleton class, what are the advantages of using **Lazy Initialization** over **Eager Initialization**?

<u>Answer:</u>

Lazy Initialization will be beneficial when we want to delay the initialization until it is not needed. If we use eager initialization and if initialization fails, there is no chance to get the instance further while in Lazy initialization, we may get it in second chance. In Lazy Initialization, we will not get instance until we call.

<u>Example of Singleton Class</u>

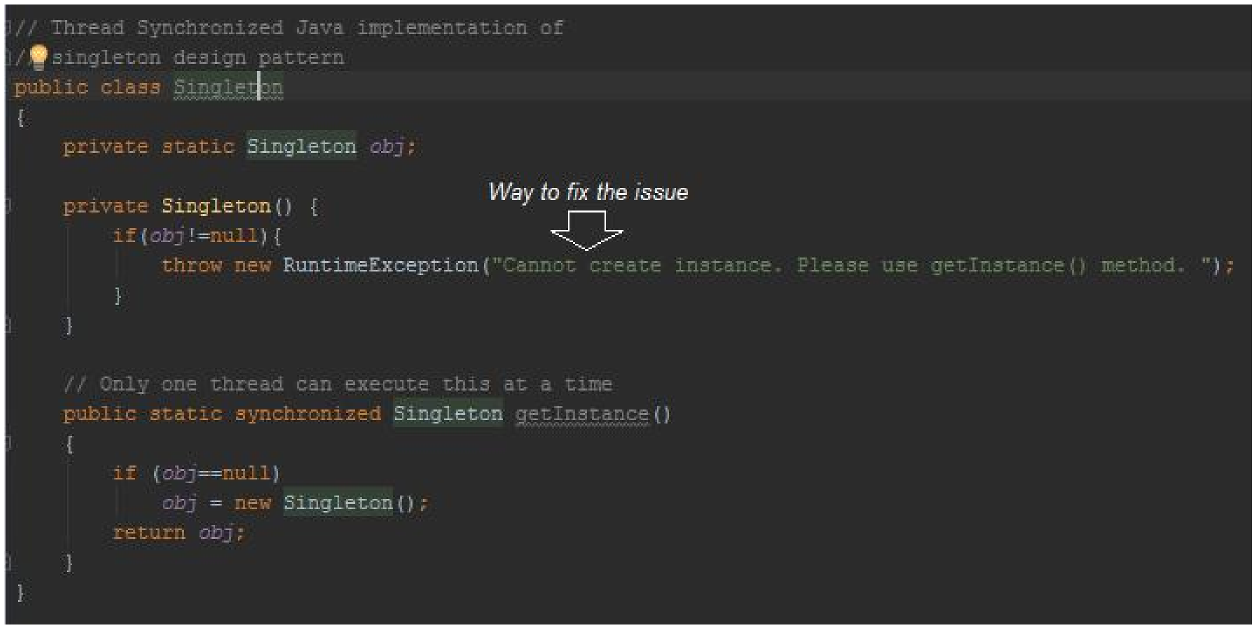


```java
// Thread Synchronized Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj;

    private Singleton() {}

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

B. Question?

Using reflection, we can set the private constructor to become accessible. How to fix this?

Answer:

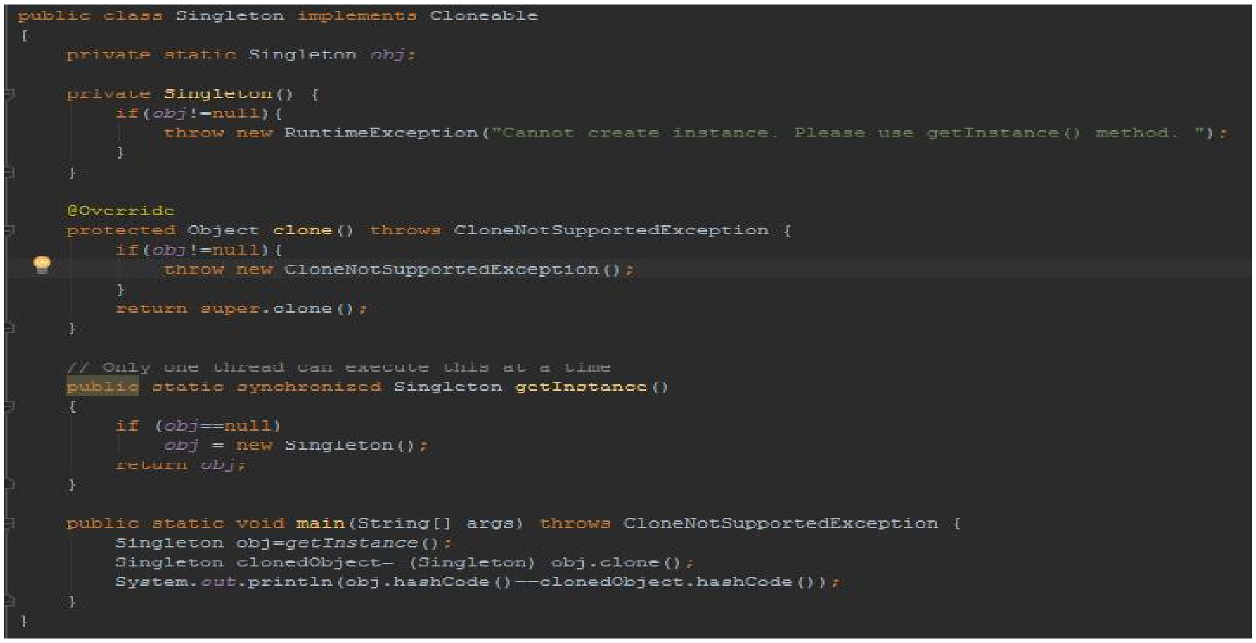


```java
// Thread Synchronized Java implementation of
// singleton design pattern
public class Singleton
{
    private static Singleton obj;

    private Singleton() {
        if(obj!=null){
            throw new RuntimeException("Cannot create instance. Please use getInstance() method. ");
        }
    }

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

C. Question?

If we try to make instance of a Singleton class by cloning it, the generated hash code of cloned copy doesn't match with the actual object. Thus it violates the Singleton Principle. How to fix this issue?

Answer:



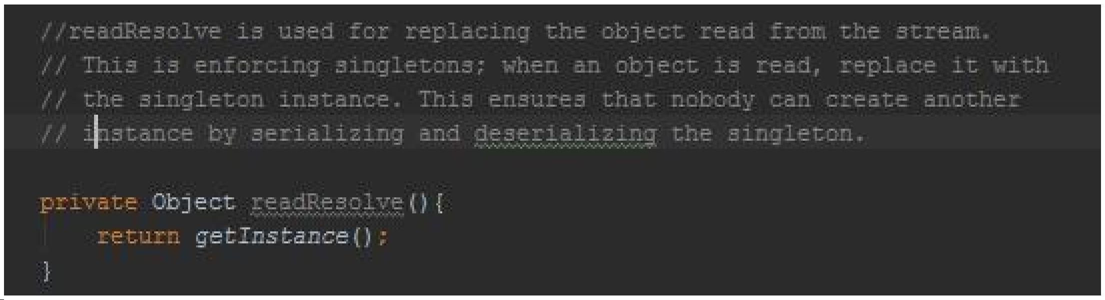
```java
public class Singleton implements Cloneable
{
    private static Singleton obj;

    private Singleton() {
        if(obj!=null){
            throw new RuntimeException("Cannot create instance. Please use getInstance() method. ");
        }
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        if(obj!=null){
            throw new CloneNotSupportedException();
        }
        return super.clone();
    }

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Singleton obj=getInstance();
        Singleton clonedObject= (Singleton) obj.clone();
        System.out.println(obj.hashCode()==clonedObject.hashCode());
    }
}
```

D. Question?

When we serialize an Object and de-serialize it again, there are different hash code values generated. So our Singleton principle breaks in case of object Serialization/Deserialization. How to fix this issue.

Answer:

Implement readResolve() method in the Singleton class as shown below.

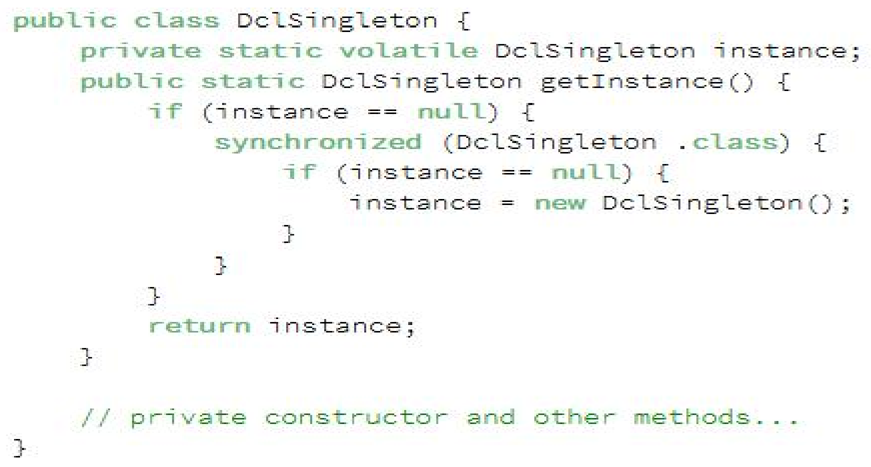


```
//readResolve is used for replacing the object read from the stream.
// This is enforcing singletons; when an object is read, replace it with
// the singleton instance. This ensures that nobody can create another
// instance by serializing and deserializing the singleton.

private Object readResolve(){
    return getInstance();
}
```

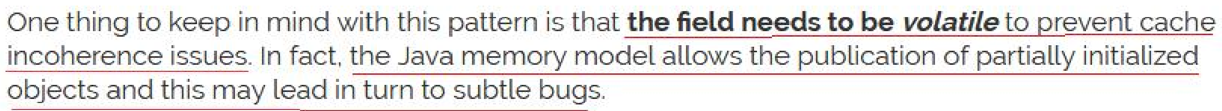
E. Question?

What is Double Check Locking on Singleton?

Answer:

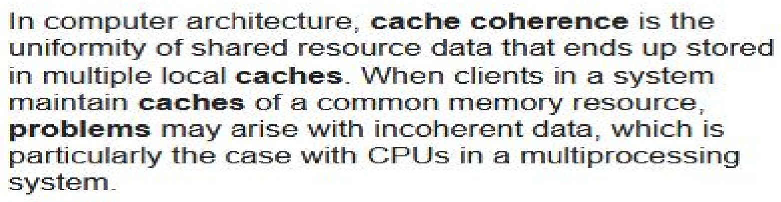


```
public class DclSingleton {
    private static volatile DclSingleton instance;
    public static DclSingleton getInstance() {
        if (instance == null) {
            synchronized (DclSingleton .class) {
                if (instance == null) {
                    instance = new DclSingleton();
                }
            }
        }
        return instance;
    }

    // private constructor and other methods...
}
```



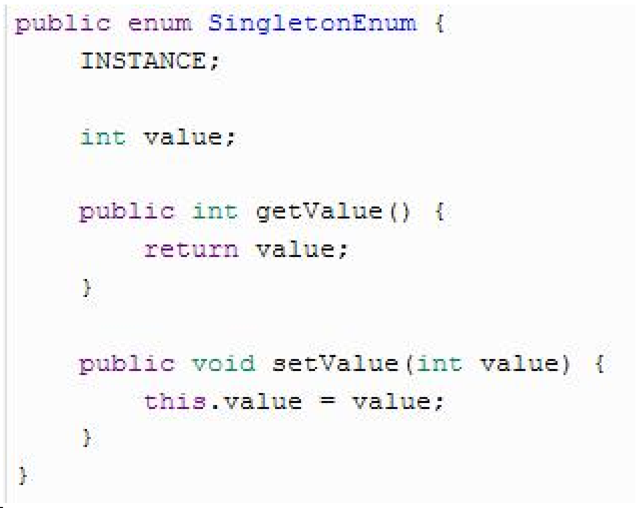
One thing to keep in mind with this pattern is that **the field needs to be *volatile*** to prevent cache incoherence issues. In fact, the Java memory model allows the publication of partially initialized objects and this may lead in turn to subtle bugs.

Note:



In computer architecture, **cache coherence** is the uniformity of shared resource data that ends up stored in multiple local **caches**. When clients in a system maintain **caches** of a common memory resource, **problems** may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

Singleton Class with Enum



```
public enum SingletonEnum {
    INSTANCE;

    int value;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

-

Note:

1. Since enums are inherently serializable, we don't need to implement it with a serializable interface.
2. The reflection problem is also not there. Therefore, it is 100% guaranteed that only one instance of the singleton is present within a JVM.
3. Thus, this method is recommended as the **best method of making singletons in Java.**

10. *Builder Design Patterns*

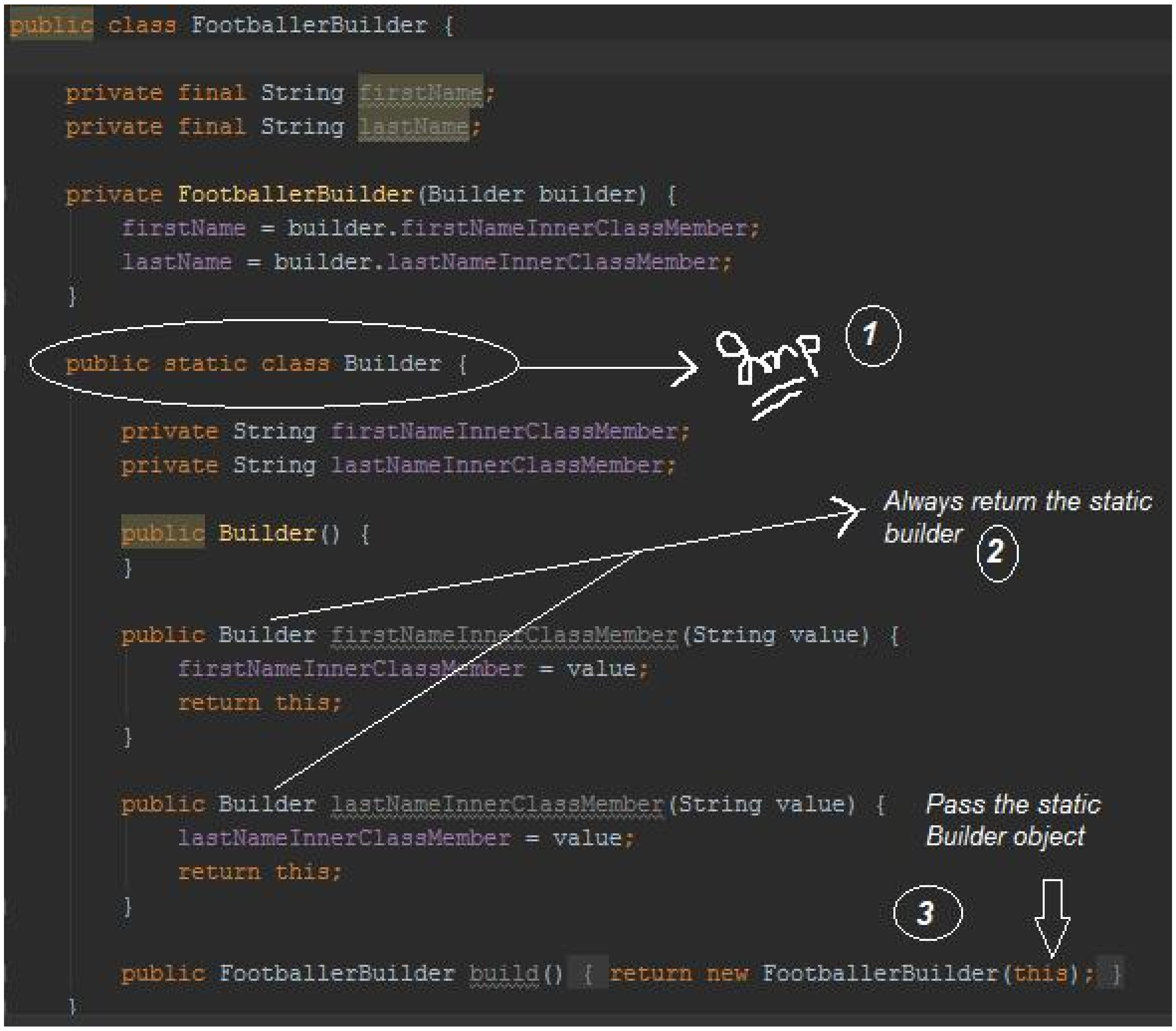
- Use Case:

    I. *Complex constructor.*

    Multiple constructor having combinations of multiple parameters (optional and Mandatory field)

    II. *Large number of parameters.*

    having large number of field parameter is also the key point to consider.


Imp
1
Always return the static builder
2
Pass the static Builder object
3

```java
public class FootballerBuilder {

    private final String firstName;
    private final String lastName;

    private FootballerBuilder(Builder builder) {
        firstName = builder.firstNameInnerClassMember;
        lastName = builder.lastNameInnerClassMember;
    }

    public static class Builder {

        private String firstNameInnerClassMember;
        private String lastNameInnerClassMember;

        public Builder() {
        }

        public Builder firstNameInnerClassMember(String value) {
            firstNameInnerClassMember = value;
            return this;
        }

        public Builder lastNameInnerClassMember(String value) {
            lastNameInnerClassMember = value;
            return this;
        }

        public FootballerBuilder build() { return new FootballerBuilder(this); }
    }
```

Cons

1) Builder pattern is verbose and requires code duplication as Builder needs to copy all fields from Original or Item class.

11. *Prototype Design Pattern*

<u>Why Prototype Design Pattern?</u>

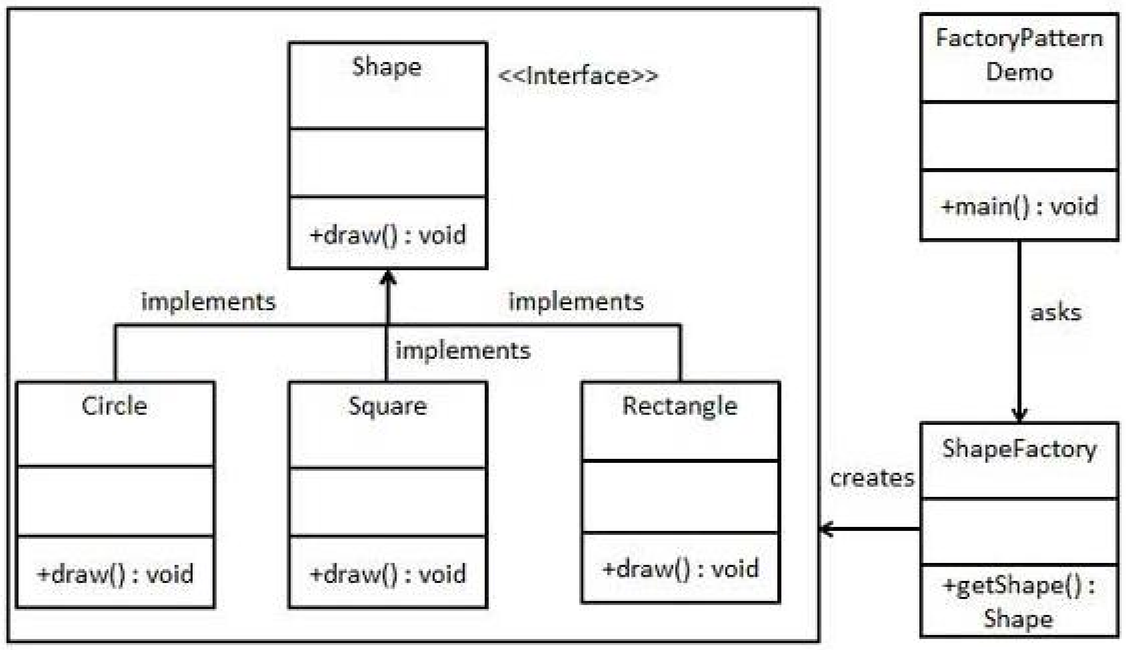
The prototype pattern has some benefits, for example:

- It eliminates the (potentially expensive) overhead of initializing an object
- It simplifies and can optimize the use case where multiple objects of the same type will have mostly the same data

For example, say your program uses objects that are created from data parsed from motley unchanging information retrieved over the network. Rather than retrieving the data and re-parsing it each time a new object is created, the prototype pattern can be used to simply duplicate the original object whenever a new one is needed.

Also, say that object may have data that uses up large amounts of memory, such as data representing images. Memory can be reduced by using a copy-on-write style inheritance, where the original, unduplicated data is shown until the code attempts to change that data. Then, the new data will mask to reference to the original data.

12. *Factory Pattern*



```
public interface Shape {
   void draw();
}
```

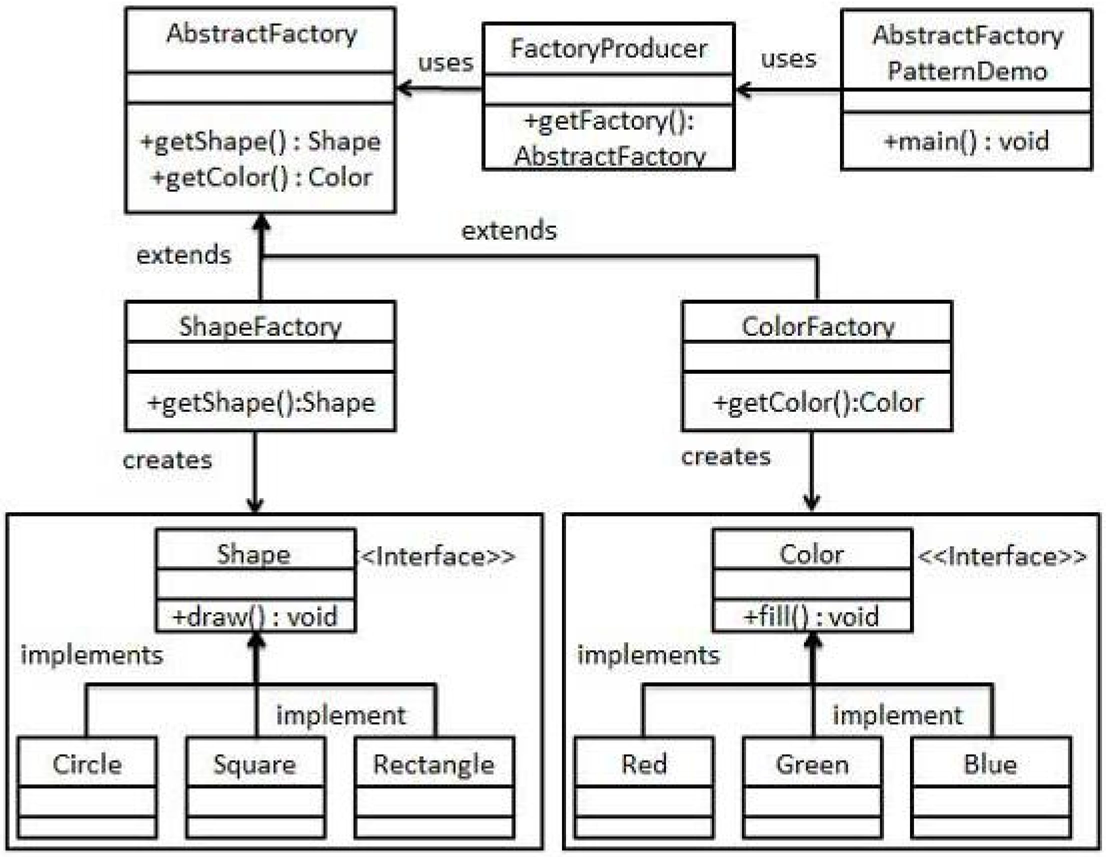
```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

```
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;
   }
}
```

13. *Abstract Factory Pattern:*



- **Difference between Factory Pattern and Abstract Factory Pattern?**

  Factory Method is used to **create one product** only **[Like Shape Product in the above Factory Pattern Example]** but **Abstract Factory** is about creating **families of related or dependent products** **[Like Shape (Circle/Square/Rectangle) or Color (Red/Green/Blue) Product in the above Factory Pattern Example]**.