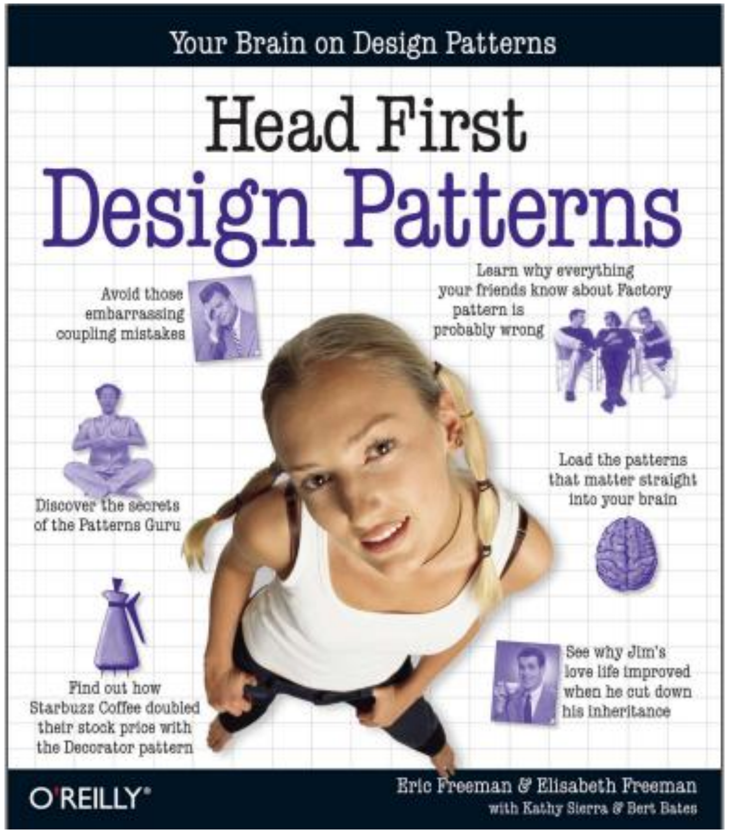
Design Patterns

Outline

- Introduction
- Definition and History
- Types of Design Patterns
- Creational Patterns
  - Factory Pattern
  - Singleton Pattern
  - Prototype Pattern

Book


Your Brain on Design Patterns
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

http://www.sws.bfh.ch/~amrhein/ADP/HeadFirstDesignPatterns.pdf

Introduction

- Design patterns represent the best practices used by experienced object-oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Definition

- A pattern is a recurring solution to a standard problem, in a context.
- “A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” (Christopher Alexander)

Why Patterns ?

- Mature engineering disciplines have handbooks describing successful solutions to known problems
- Automobile designers don't design cars from scratch using the laws of physics
- Instead, they reuse standard designs with successful track records, learning from experience

Some History

- Started in 1987 by Ward Cunningham and Ken Beck who were working with Smalltalk and designing GUIs.
- Popularized by Gamma, Helm, Johnson and Vlissides (The gang of four, Go4)
- The three of Go4 were working on frameworks (E++,Unidraw, HotDraw)
- Design pattern use a consistent documentation approach
- Design pattern are granular and applied at different levels such as frameworks, subsystems and sub-subsystems

Elements of Design Pattern

- Design patterns have 4 essential elements:
  - Pattern name: increases vocabulary of designers
  - Problem: intent, context, when to apply
  - Solution: UML-like structure, abstract code
  - Consequences: results and tradeoffs

Categorizing Pattern

Patterns, then, represent expert solutions to recurring problems in a context and thus have been captured at many levels of abstraction and in numerous domains. Numerous categories are:

- **Creational**
- **Structural**
- **Behavioral**

Creational Patterns

- These design patterns provide a way to create objects while hiding the creation logic.

- This gives program more flexibility in deciding which objects need to be created for a given use case.

Structural Pattern

- In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

Behavioral Pattern

- These design patterns are specifically concerned with communication between objects.

Creational Patterns

Factory Pattern

- Factory pattern is one of most used 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.

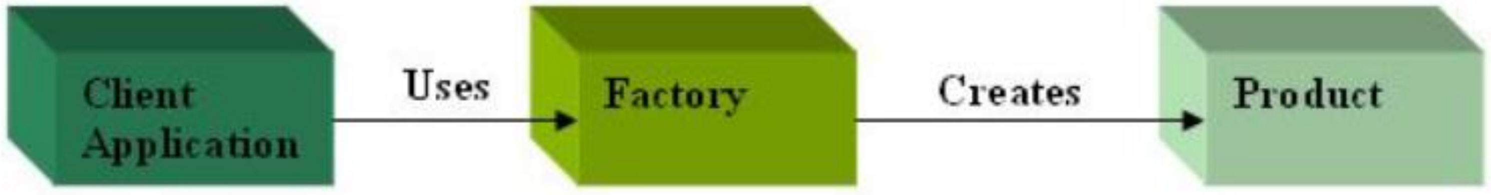
Cont.

- An important aspect of software design is the manner in which objects are created. Thus, it is not only important what an object does or what it models, but also in what manner it was created.
- Initially an object is created with the "new" operator. That basic mechanism of object creation could result in design problems or added complexity to the design. On each Object creation we must use the new keyword. The Factory helps you to reduce this practice and use the common interface to create an object.

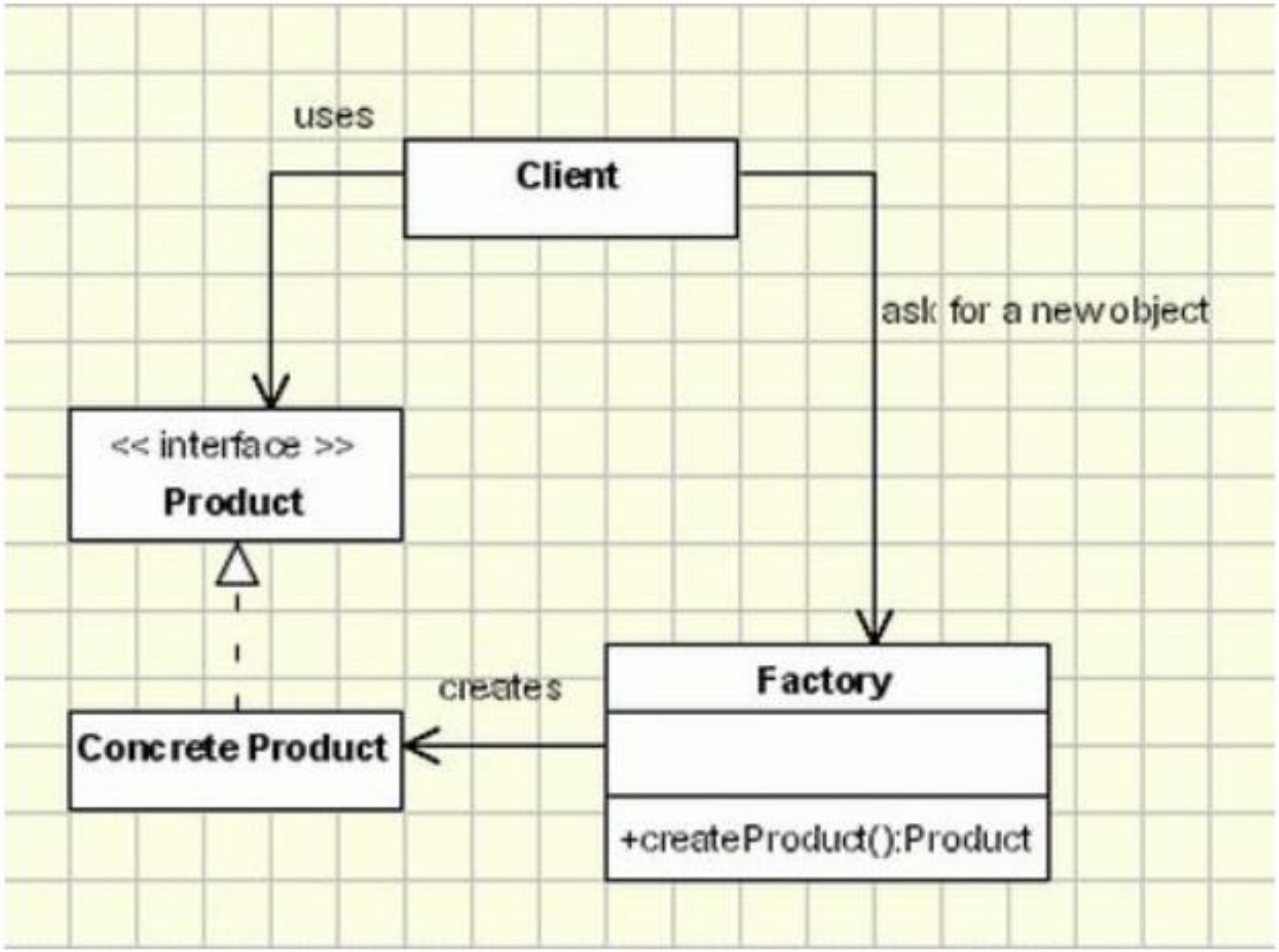
- GOF says: Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory Method lets a class defer instantiation to subclasses.
- The Factory Pattern is a Creational Pattern that simplifies object creation. You need not worry about the object creation; you just need to supply an appropriate parameter and factory to provide you a product as needed.

Factory Advantages

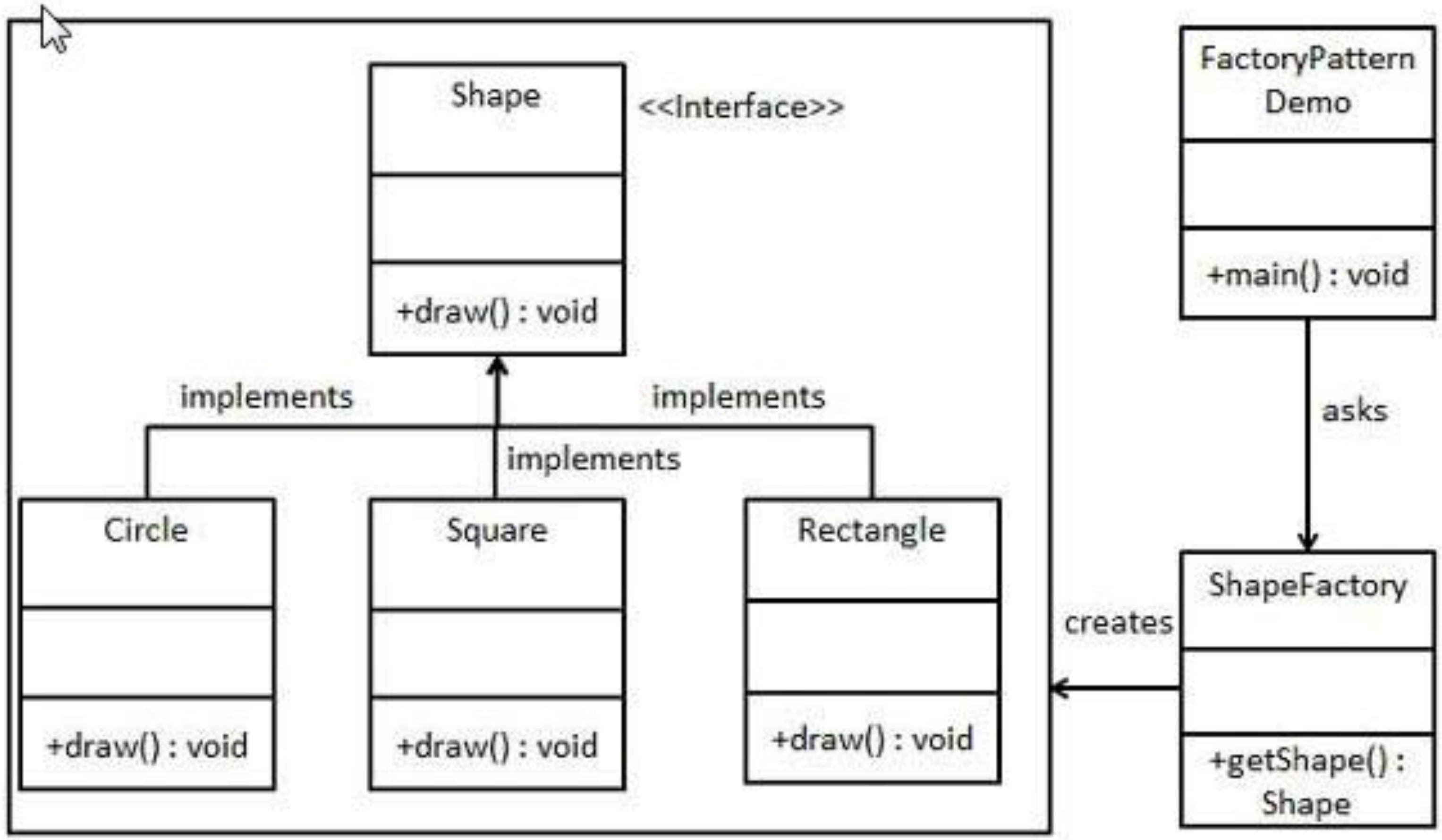
- Creating objects without exposing the instantiation logic to the client
- Referring to the newly created objects through a common interface



Overall Design



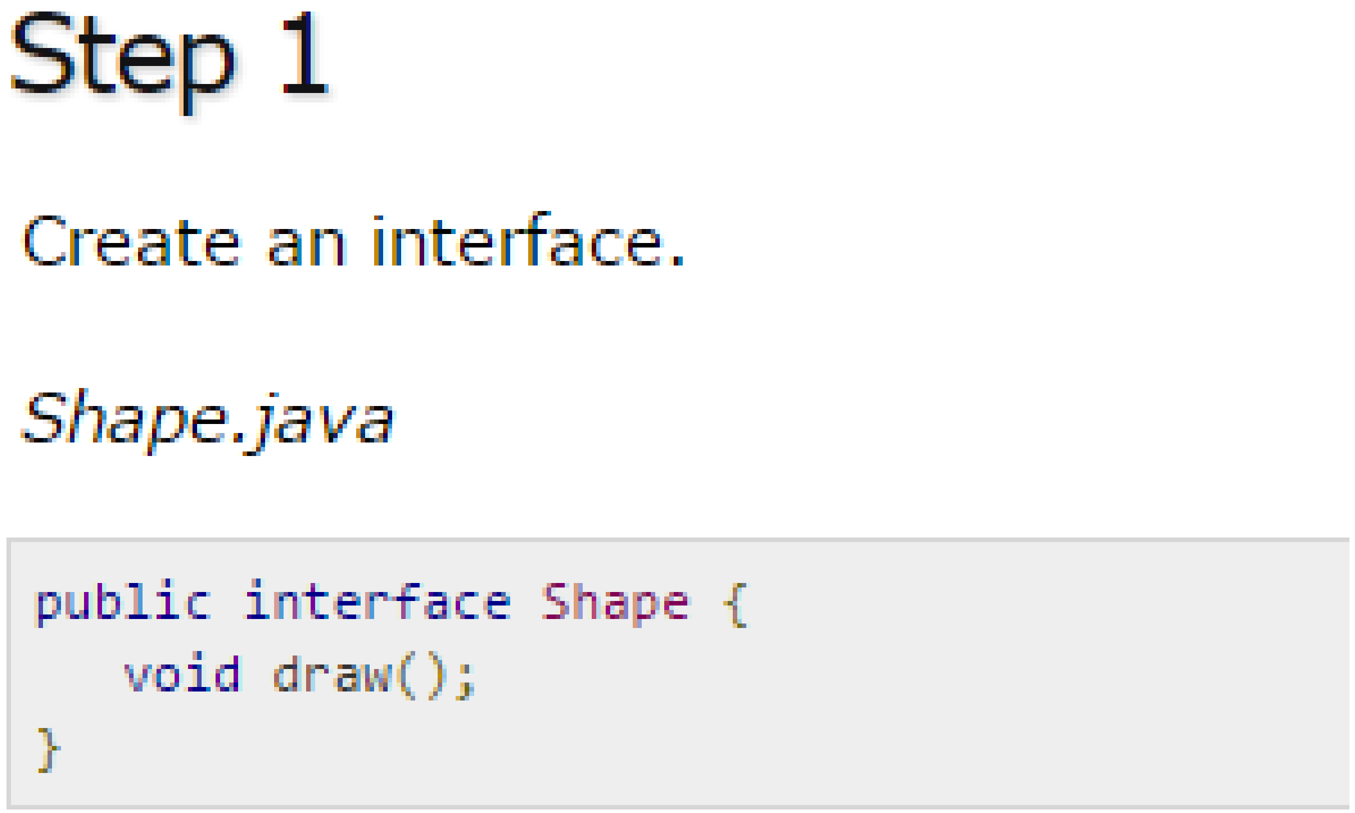
Example


Shape
<<Interface>>
+draw() : void
implements
implements
implements
Circle
+draw() : void
Square
+draw() : void
Rectangle
+draw() : void
FactoryPattern Demo
+main() : void
asks
ShapeFactory
+getShape() : Shape
creates

Description

- We're going to create a *Shape* interface and concrete classes implementing the*Shape* interface. A factory class *ShapeFactory* is defined as a next step.
- *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.

Implementation

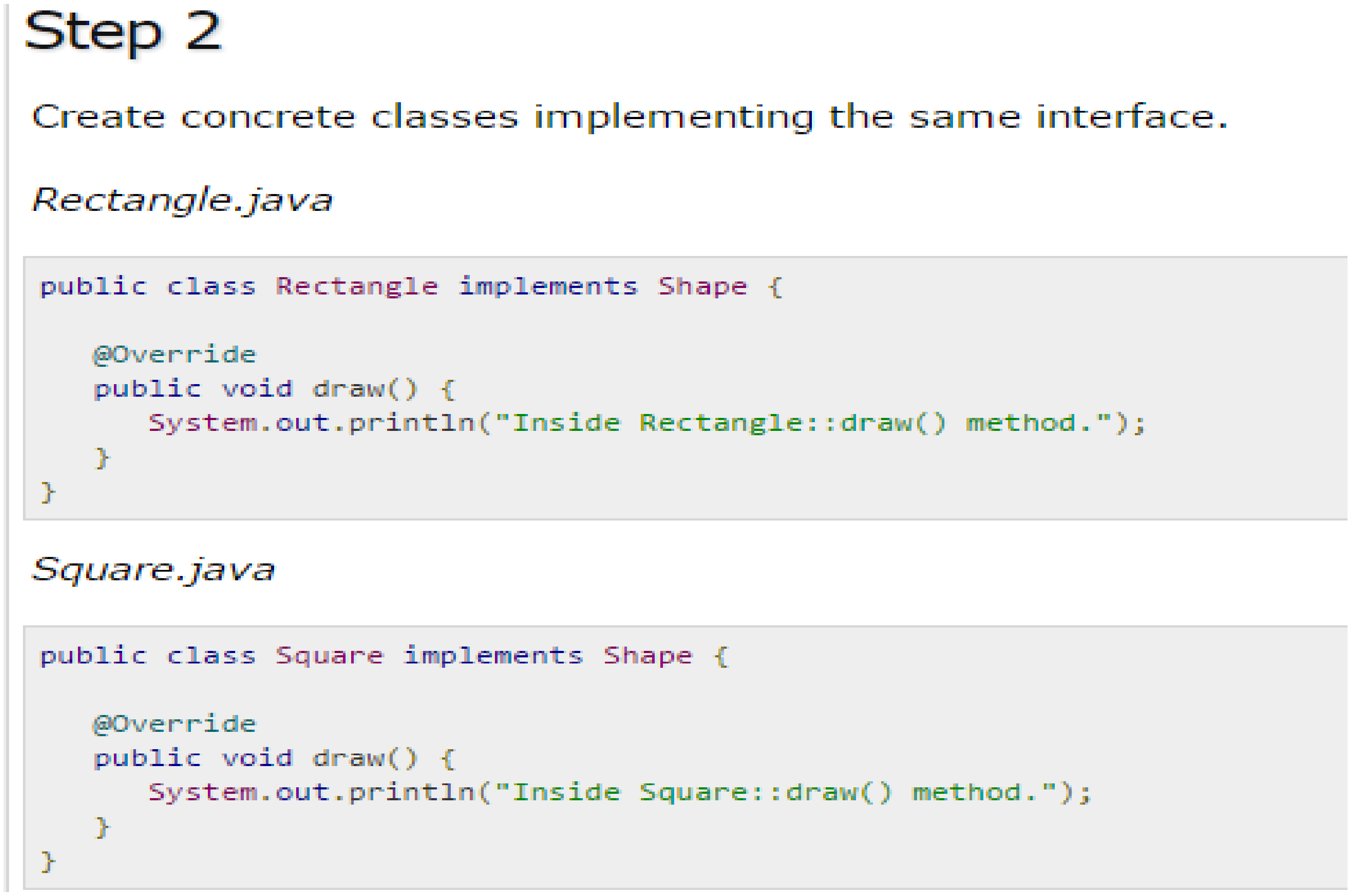


Step 1

Create an interface.

Shape.java

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



Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

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

Square.java

```
public class Square implements Shape {

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

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.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;
   }
}
```

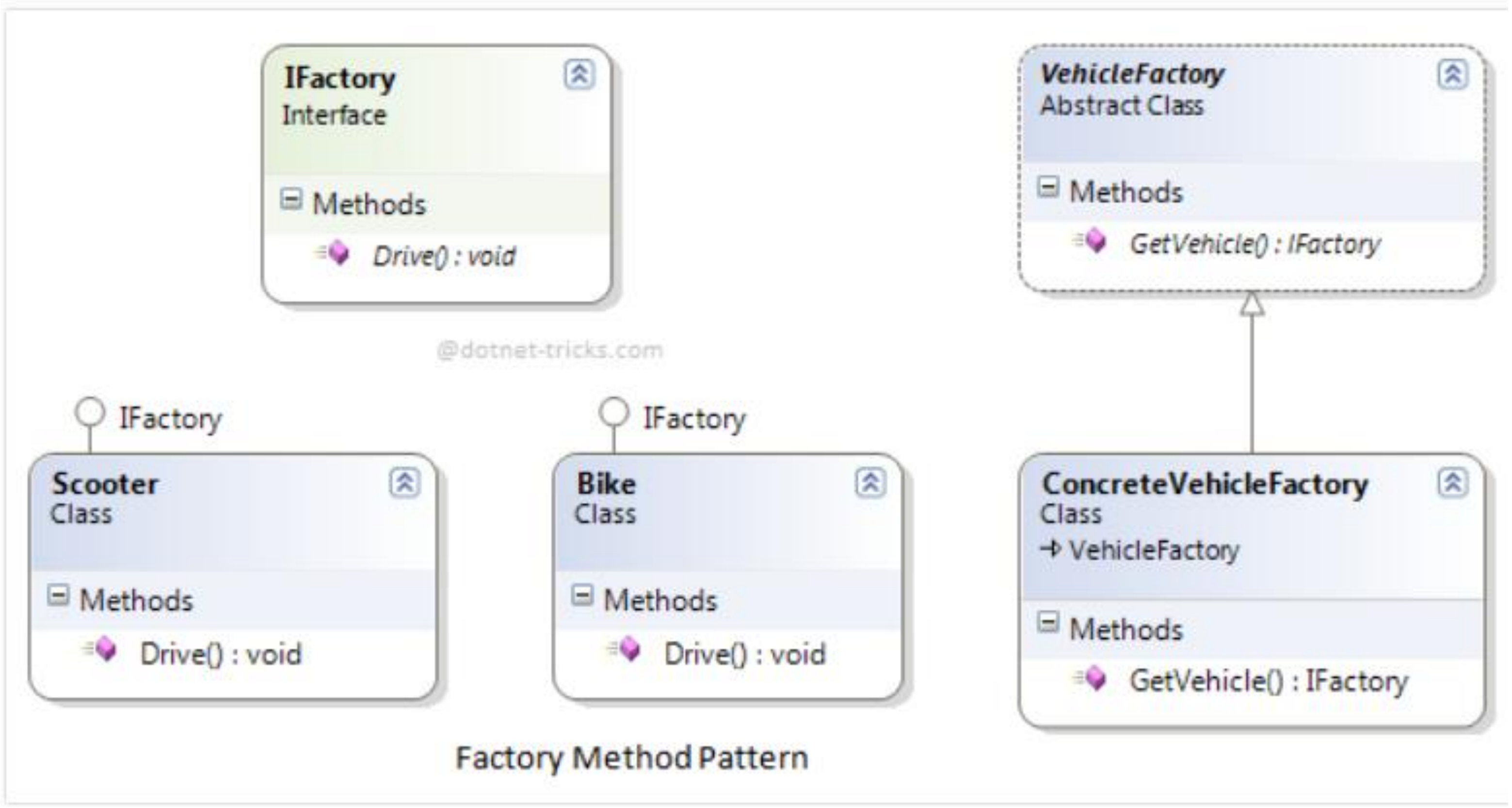
FactoryPatternDemo.java

```java
public class FactoryPatternDemo {

   public static void main(String[] args) {
      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();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

Advantages / Use

- New shapes can be added without changing a single line of code in the framework(the client code that uses the shapes from the factory)

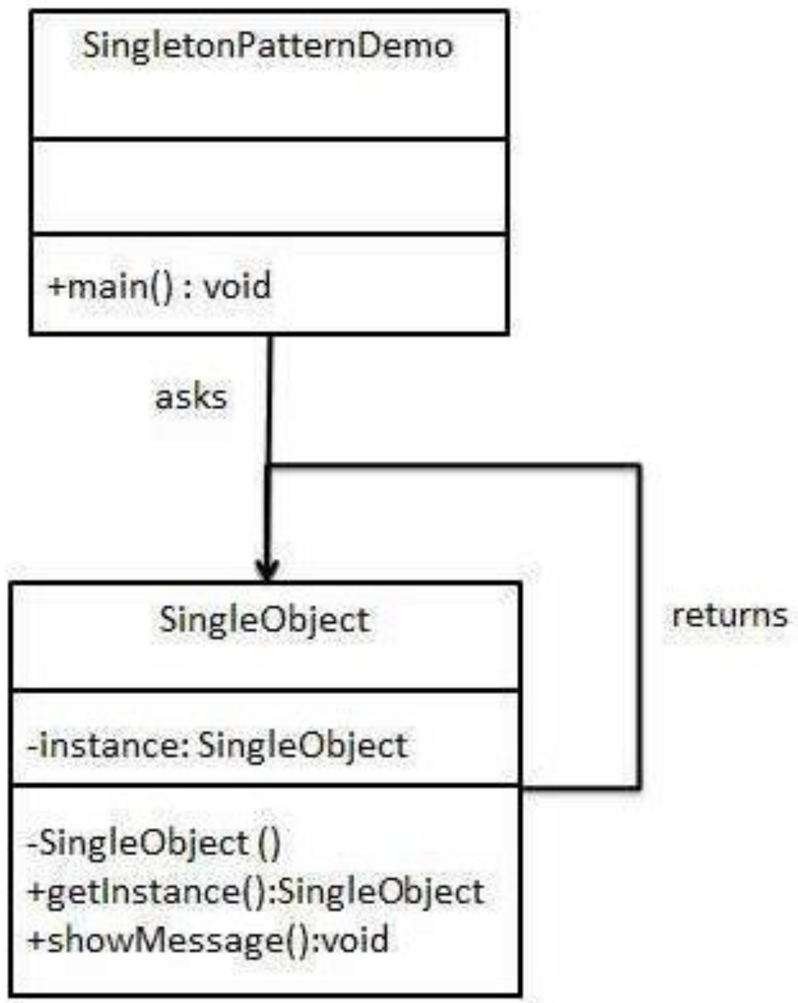
Example-2 (Factory Pattern)



Singleton Pattern

- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Example


SingletonPatternDemo
+main() : void
asks
returns
SingleObject
-instance: SingleObject
-SingleObject ()
+getInstance():SingleObject
+showMessage():void

SingleObject.java

```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("Hello World!");
   }
}
```

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();
   }
}
```

Use of Singleton Patterns

- It is useful when exactly one object is needed to coordinate actions across the system.
- This is one of the most commonly used patterns. There are some instances in the application where we have to use just one instance of a particular class. Let’s take up an example to understand this.
- A very simple example is say Logger, suppose we need to implement the logger and log it to some file according to date time. In this case, we cannot have more than one instances of Logger in the application otherwise the file in which we need to log will be created with every instance.

Prototype Pattern

- Prototype pattern refers to creating duplicate object while keeping performance in mind.
- This pattern involves implementing a prototype interface which tells to create a clone of the current object.
- This pattern is used when creation of object directly is costly.

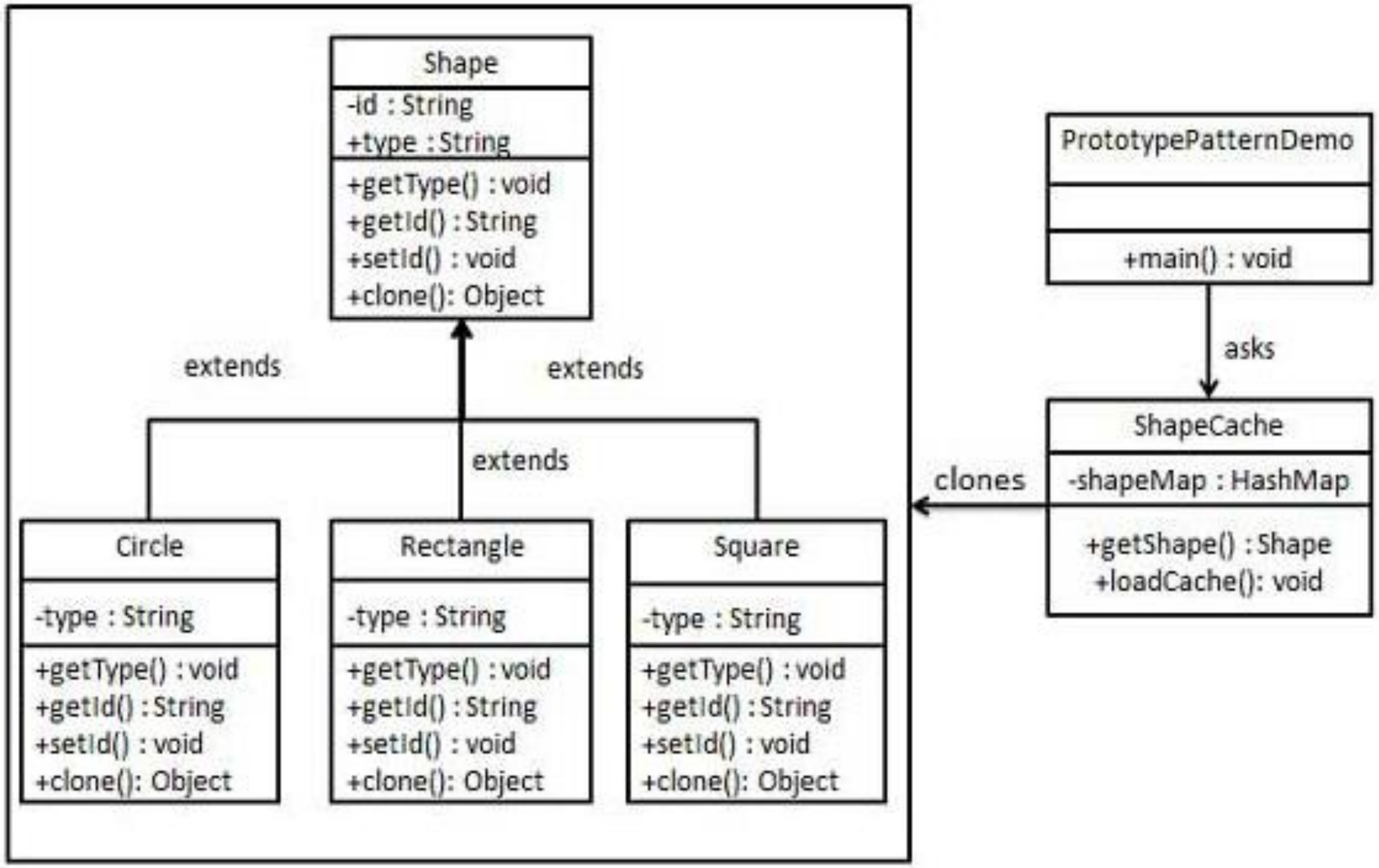
Clonable Interface

- The **Cloneable** interface defines no members.
- It is used to indicate that a class allows a bitwise copy of an object (that is, a *clone*) to be made.
- If you try to call **clone( )** on a class that does not implement**Cloneable**, a **CloneNotSupportedException** is thrown

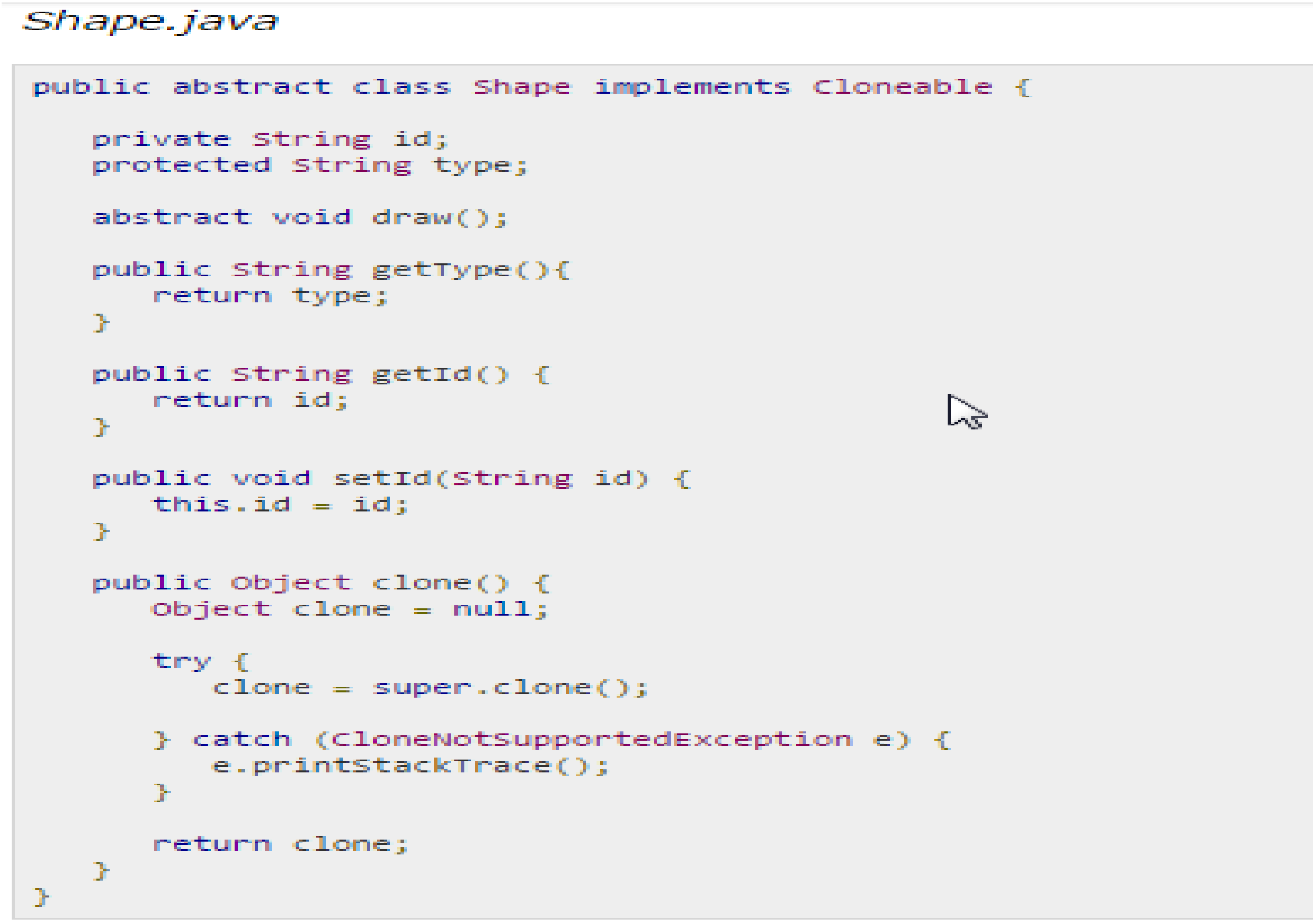
Description

- We're going to create an abstract class *Shape (Which implements Clonable interface)* and concrete classes extending the *Shape* class. A class *ShapeCache* is defined as a next step which stores shape objects in a *Hashtable* and returns their clone when requested.
- *PrototypPatternDemo*, our demo class will use *ShapeCache* class to get a *Shape*object.

Example


Shape
-id : String
+type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
extends
extends
extends
Circle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Rectangle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Square
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
PrototypePatternDemo
+main() : void
asks
ShapeCache
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void
clones

Implementation



Shape.java

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

Rectangle.java

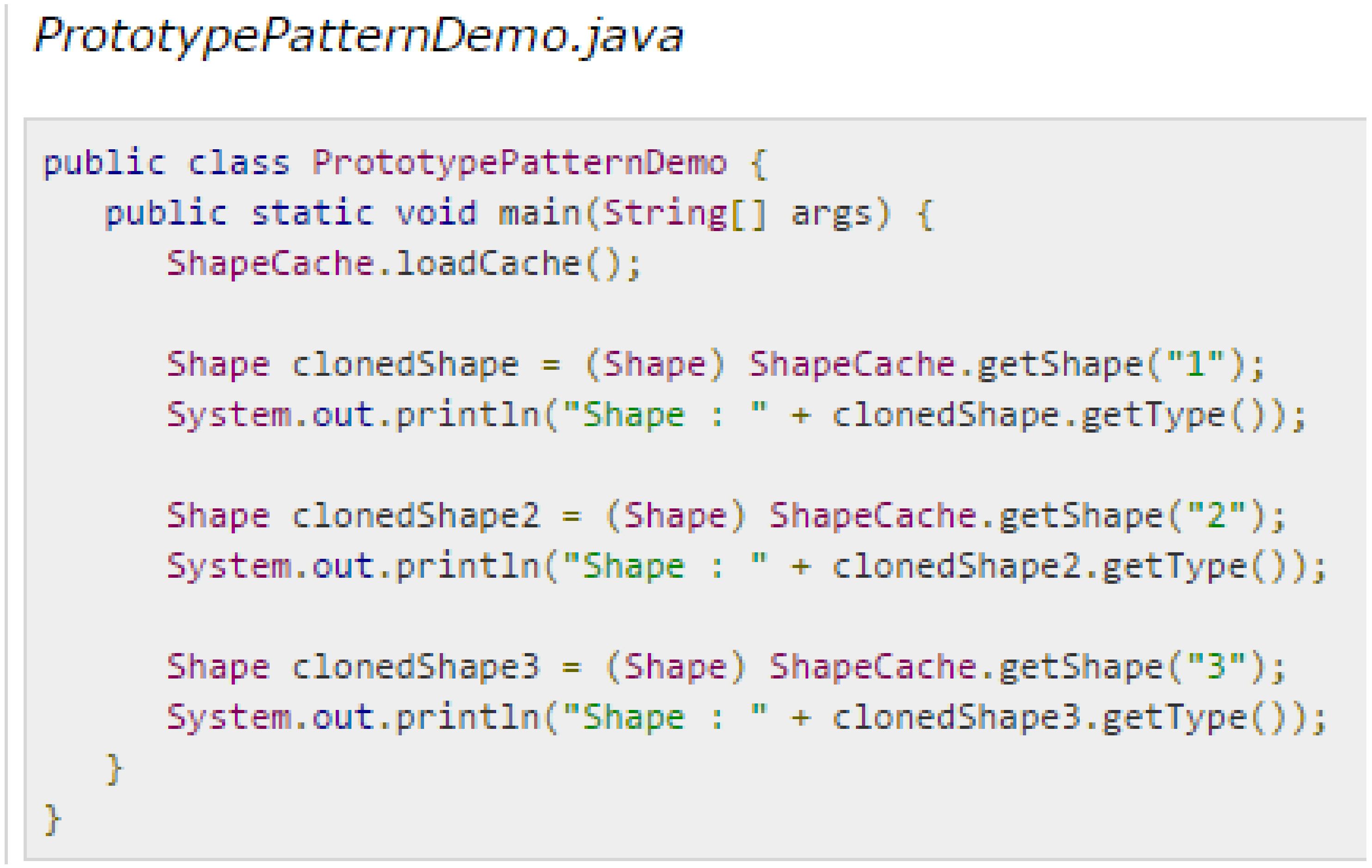
```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

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

Square.java

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

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

ShapeCache.java

```
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
```



PrototypePatternDemo.java

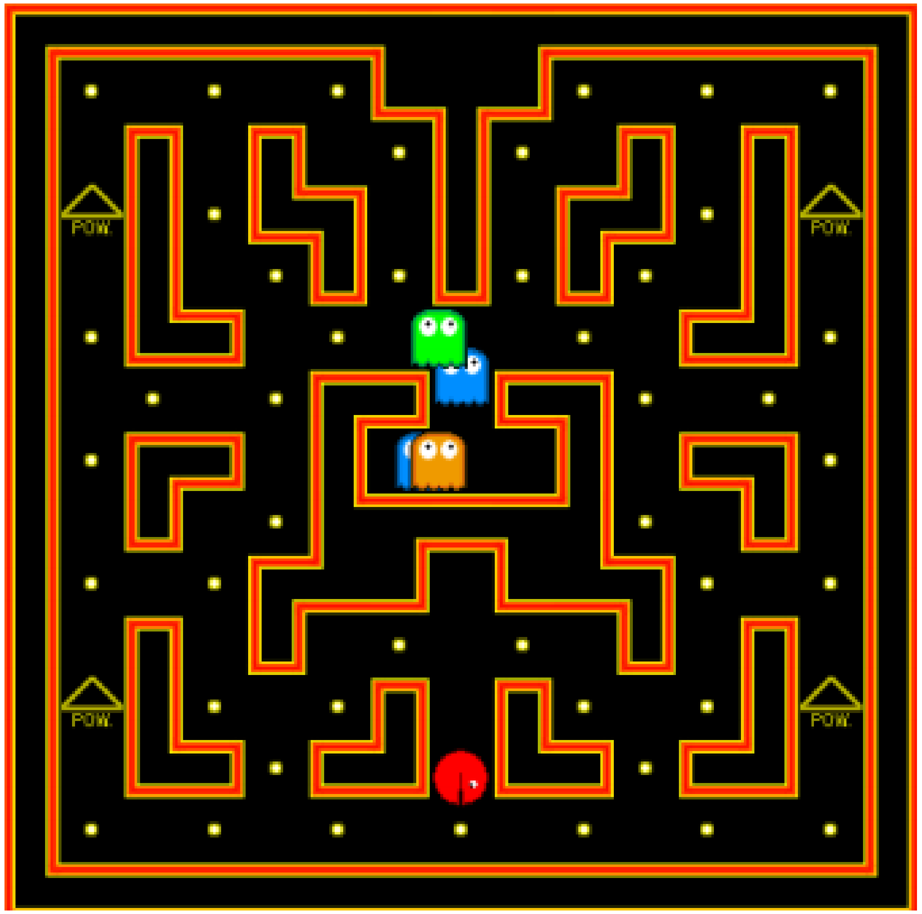
```java
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

Application

- If you have a requirement , where you need to populate or use a same data containing Object repeatable and it is not possible to build from existing Object.
- To build an Object is time consuming [Building a Big Object , by getting data from Database] then use this design pattern , as in this a Copy the existing Object is created ,this copy would be different from the Original Object and could be used just like Original one.

Example

- The walls, rooms, doors and other elements of a video game are simply repeated in different arrangements.
- To ensure smooth experience, a quick way to generate a haze map is required.



What is the difference between Factory Design pattern and Prototype Pattern ?

Object Pool Pattern

- The **object pool pattern** is a software creational design pattern that uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand.
- A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object to the pool rather than destroying it; this can be done manually or automatically.

Object Pool

- When it is necessary to work with a large number of objects that are particularly expensive to instantiate and each object is only needed for a short period of time, the performance of an entire application may be adversely affected. An object pool design pattern may be deemed desirable in cases such as these.

Example

- One example is the .NET Framework Data Provider for SQL Server. As SQL Server database connections can be slow to create, a pool of connections is maintained. When you close a connection it does not actually relinquish the link to SQL Server. Instead, the connection is held in a pool from which it can be retrieved when requesting a new connection. This substantially increases the speed of making connections.

Implementation

- Java supports thread pooling via java.util.concurrent.ExecutorService and other related classes. The executor service has a certain number of "basic" threads that are never discarded. If all threads are busy, the service allocates the allowed number of extra threads that are later discarded if not used for the certain expiration time.

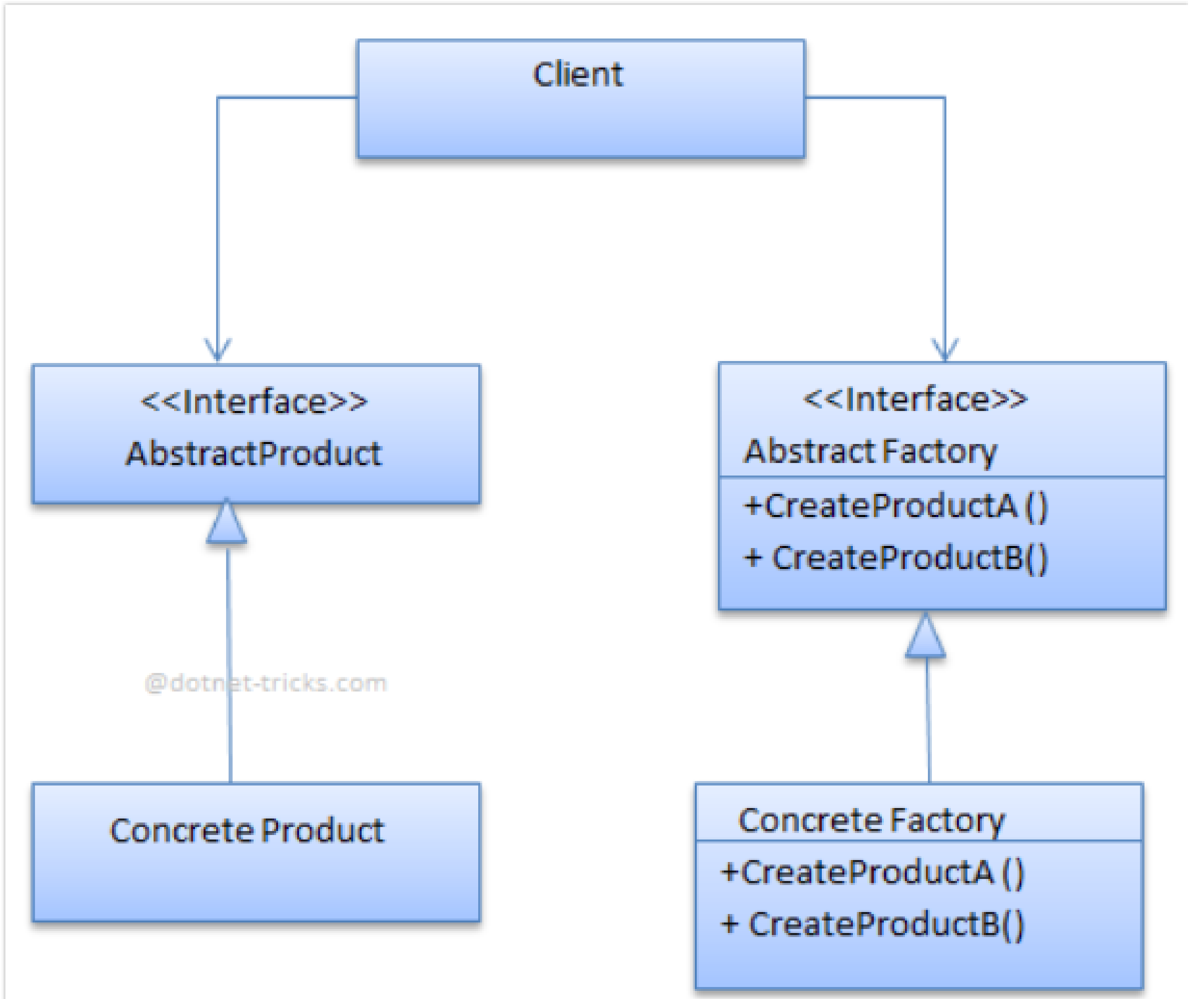
Abstract Factory Pattern

- **Abstract Factory patterns** work around a super-**factory** which creates other **factories**. This **factory** is also called as **factory** of **factories**.
- In **Abstract Factory pattern** an interface is responsible for creating a **factory** of related objects without explicitly specifying their classes.

Cont.

- In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

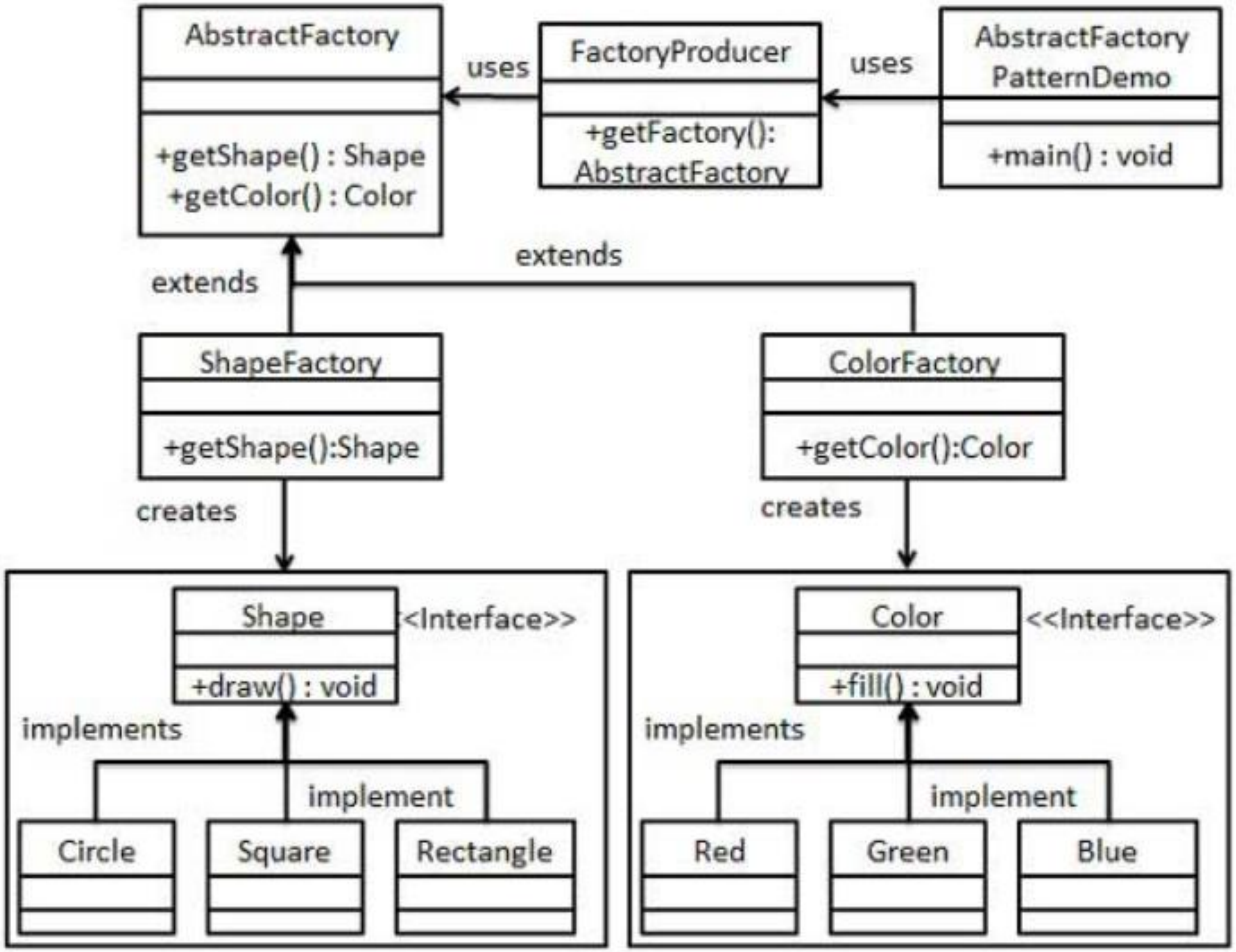
Overall Design



Abstract Factory Pattern

- We are going to create a *Shape* and *Color* interfaces and concrete classes implementing these interfaces. We create an abstract factory class *AbstractFactory* as next step. Factory classes *ShapeFactory* and *ColorFactory*are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* is created.

- *AbstractFactoryPatternDemo*, our demo class uses *FactoryProducer* to get a *AbstractFactory* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE* for *Shape*) to *AbstractFactory* to get the type of object it needs. It also passes information (*RED / GREEN / BLUE* for *Color*) to *AbstractFactory* to get the type of object it needs.



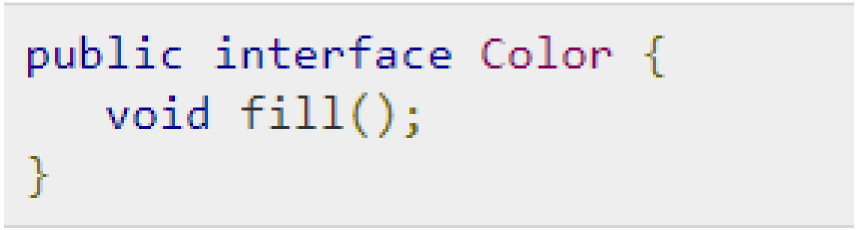
AbstractFactory
+getShape() : Shape
+getColor() : Color
uses
FactoryProducer
+getFactory(): AbstractFactory
uses
AbstractFactory PatternDemo
+main() : void
extends
extends
ShapeFactory
+getShape():Shape
ColorFactory
+getColor():Color
creates
creates
Shape
<<Interface>>
+draw() : void
implements
implement
Circle
Square
Rectangle
Color
<<Interface>>
+fill() : void
implements
implement
Red
Green
Blue

Shape Interface

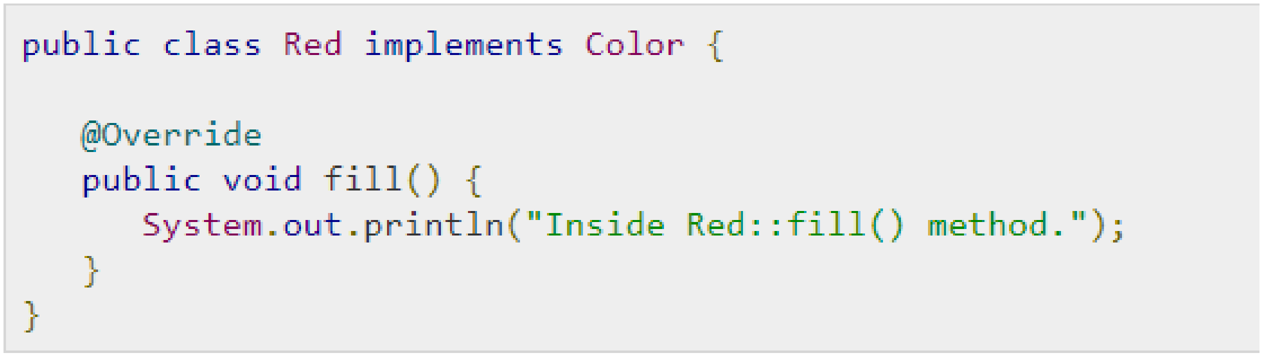
```java
public interface Shape {
   void draw();
}
```

```java
public class Rectangle implements Shape {

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

Color Interface

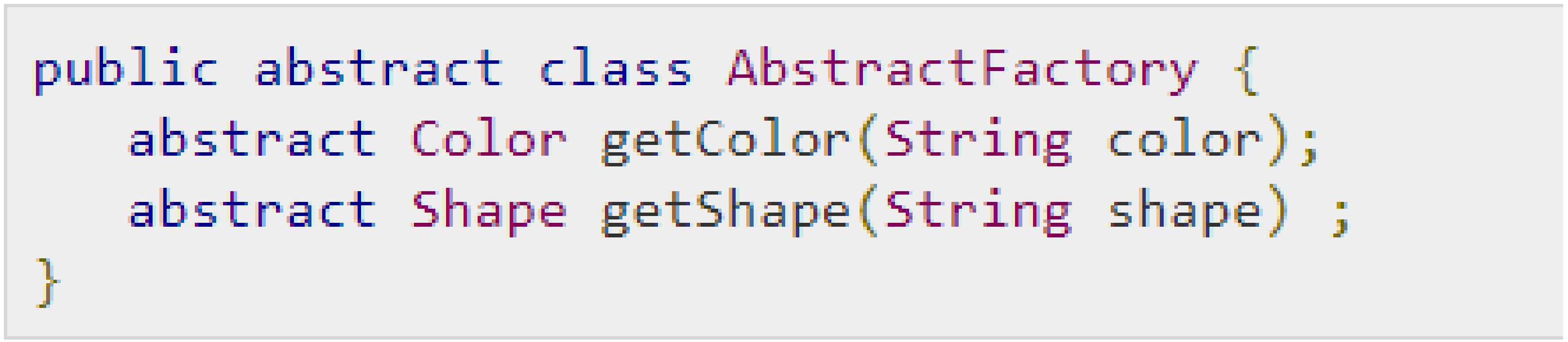


```
public interface Color {
   void fill();
}
```



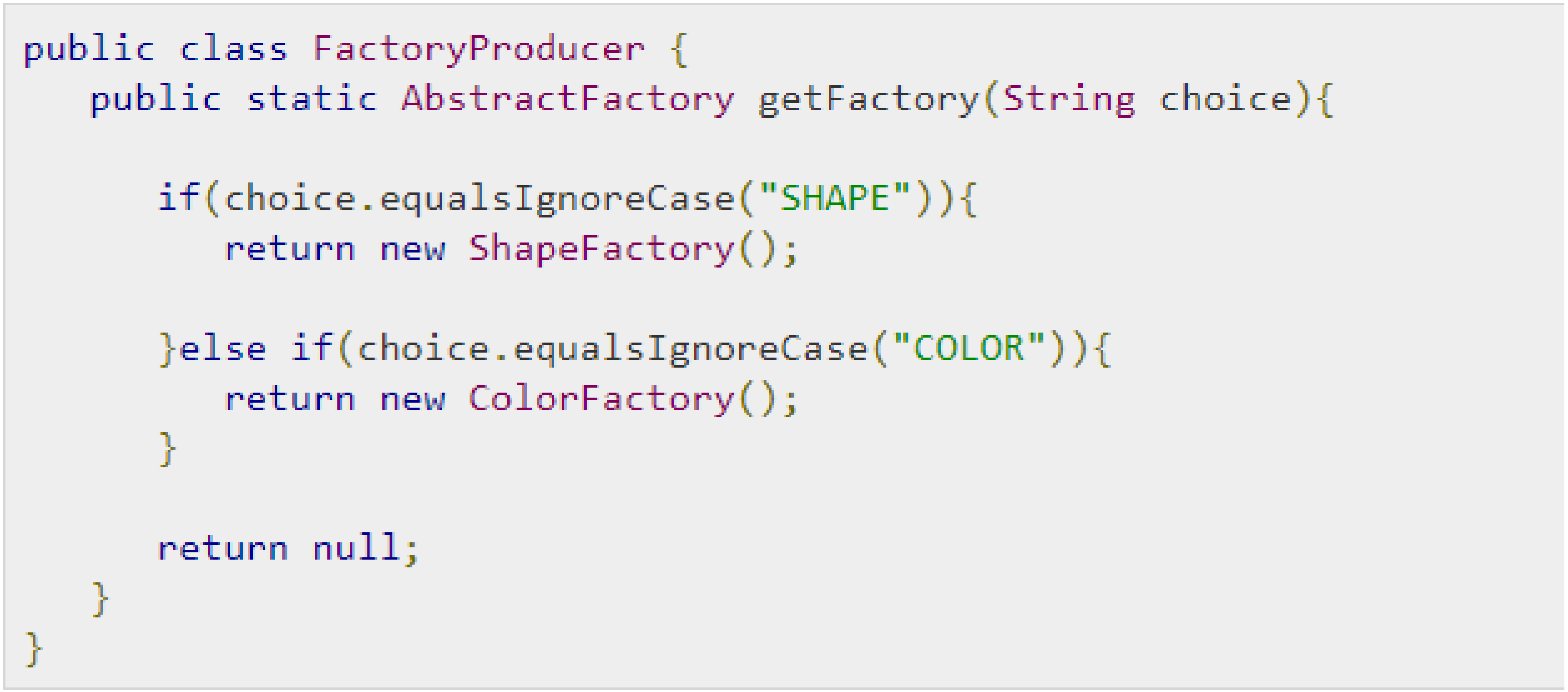
```
public class Red implements Color {

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

Abstract Factory Class



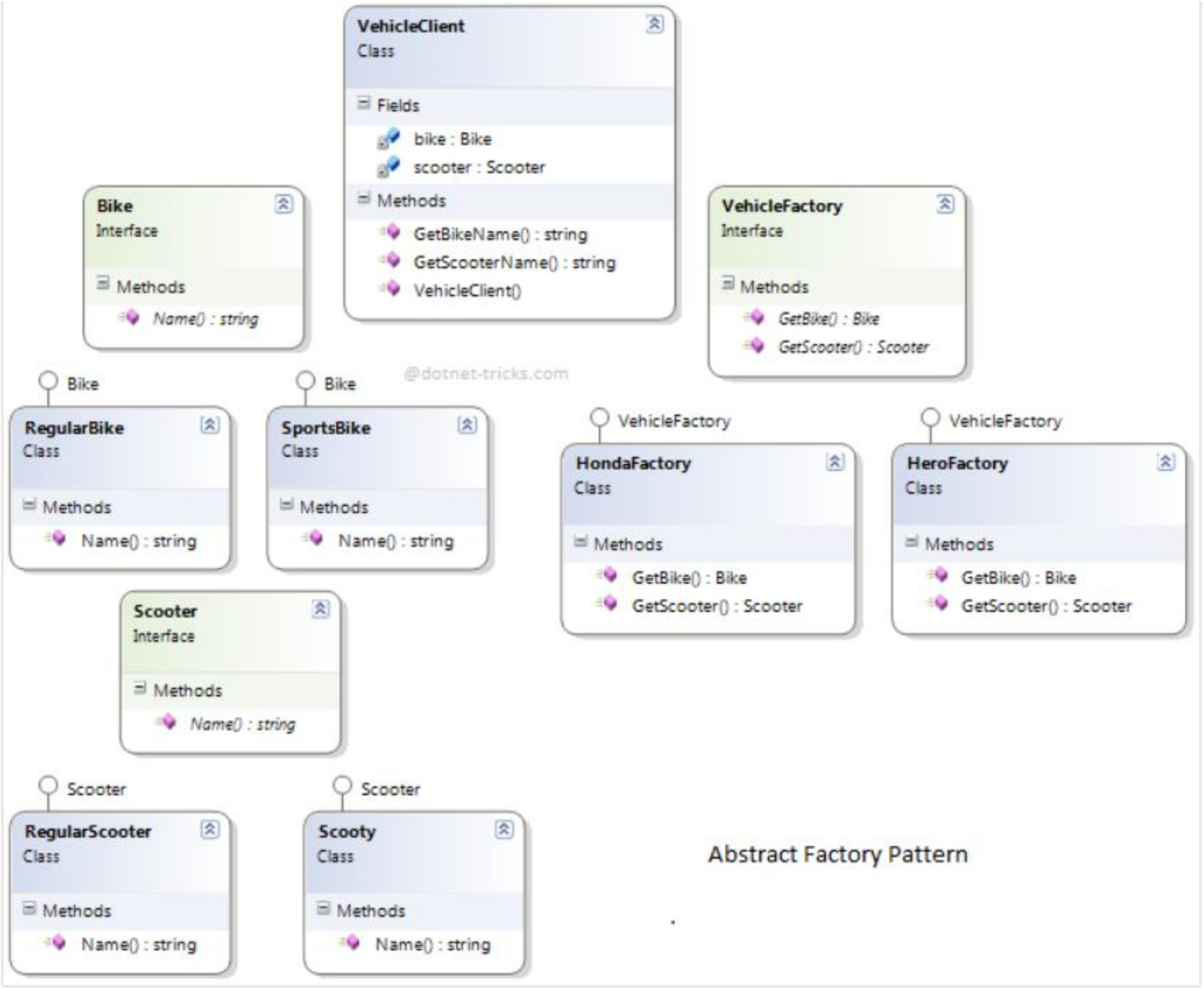
```java
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}
```

```java
public class ShapeFactory extends AbstractFactory {

   @Override
   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;
   }

   @Override
   Color getColor(String color) {
      return null;
   }
}
```



```java
public class FactoryProducer {
   public static AbstractFactory getFactory(String choice){

      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();

      }else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }

      return null;
   }
}
```

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

      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

      //get an object of Shape Circle
      Shape shape1 = shapeFactory.getShape("CIRCLE");

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

      //get an object of Shape Rectangle
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

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

      //get an object of Shape Square
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of Shape Square
      shape3.draw();

      //get color factory
      AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

      //get an object of Color Red
      Color color1 = colorFactory.getColor("RED");

      //call fill method of Red
      color1.fill();

      //get an object of Color Green
      Color color2 = colorFactory.getColor("Green");

      //call fill method of Green
      color2.fill();

      //get an object of Color Blue
      Color color3 = colorFactory.getColor("BLUE");

      //call fill method of Color Blue
      color3.fill();
   }
}
```

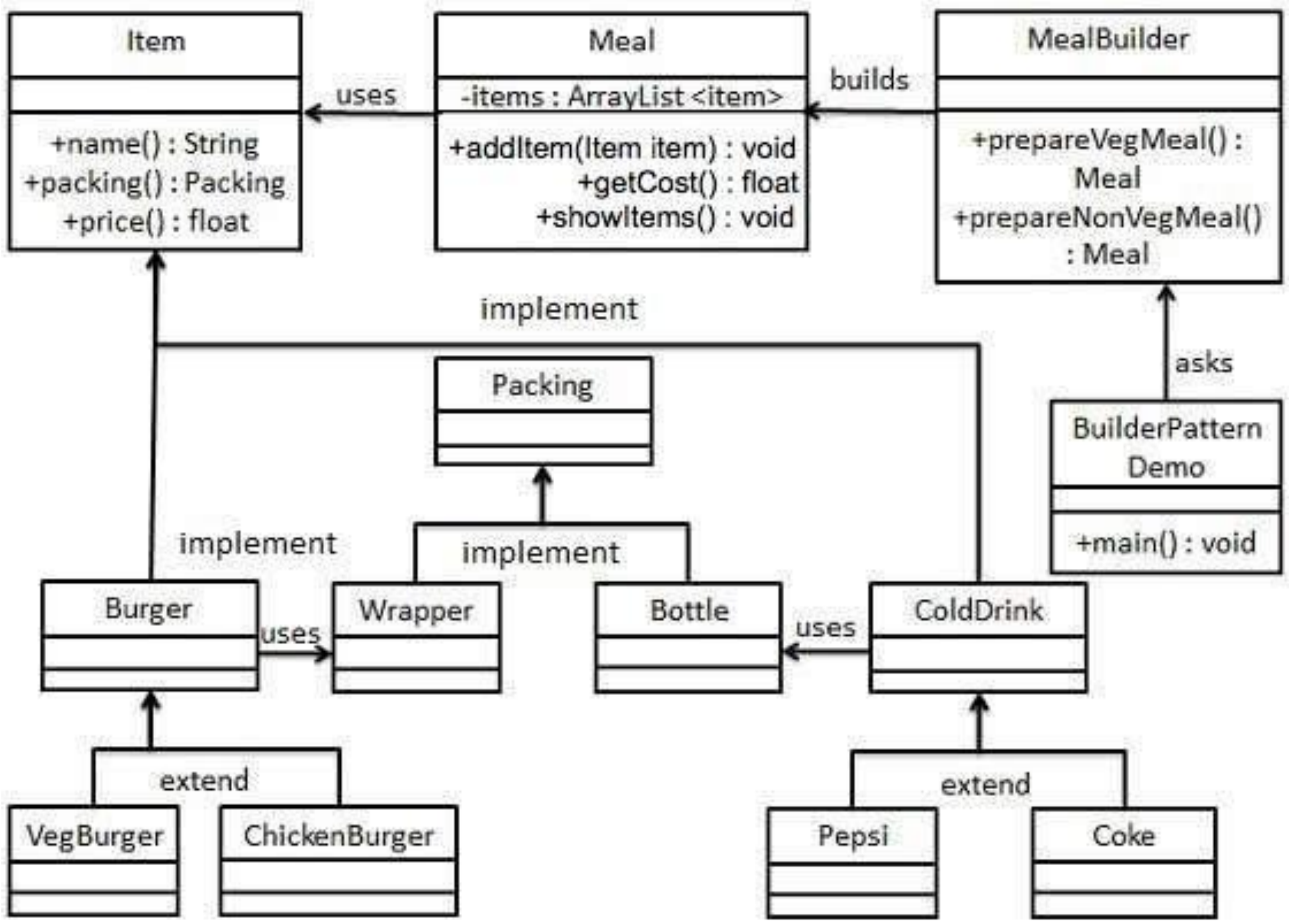

VehicleClient
Class
Fields
bike : Bike
scooter : Scooter
Methods
GetBikeName() : string
GetScooterName() : string
VehicleClient()
Bike
Interface
Methods
Name() : string
VehicleFactory
Interface
Methods
GetBike() : Bike
GetScooter() : Scooter
@dotnet-tricks.com
Bike
RegularBike
Class
Methods
Name() : string
Bike
SportsBike
Class
Methods
Name() : string
VehicleFactory
HondaFactory
Class
Methods
GetBike() : Bike
GetScooter() : Scooter
VehicleFactory
HeroFactory
Class
Methods
GetBike() : Bike
GetScooter() : Scooter
Scooter
Interface
Methods
Name() : string
Scooter
RegularScooter
Class
Methods
Name() : string
Scooter
Scooty
Class
Methods
Name() : string
Abstract Factory Pattern

Builder Pattern

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.

- Builder focuses on constructing a complex object step by step. Factory emphasizes a family of product objects (either simple or complex).
- Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.

Example


Item
+name() : String
+packing() : Packing
+price() : float
uses
Meal
-items : ArrayList <item>
+addItem(Item item) : void
+getCost() : float
+showItems() : void
builds
MealBuilder
+prepareVegMeal() : Meal
+prepareNonVegMeal() : Meal
asks
BuilderPattern Demo
+main() : void
implement
Packing
implement
implement
Burger
uses
Wrapper
Bottle
uses
ColdDrink
extend
VegBurger
ChickenBurger
extend
Pepsi
Coke

Item.java

```java
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

```java
public interface Packing {
   public String pack();
}
```

Wrapper.java

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

Burger.java

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

        @Override
        public Packing packing() {
       return new Bottle();
        }

        @Override
        public abstract float price();
}
```

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Meal.java

```java
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){

      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

MealBuilder.java

```java
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

BuilderPatternDemo.java

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

      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " + nonVegMeal.getCost());
   }
}
```

Behavioral Pattern

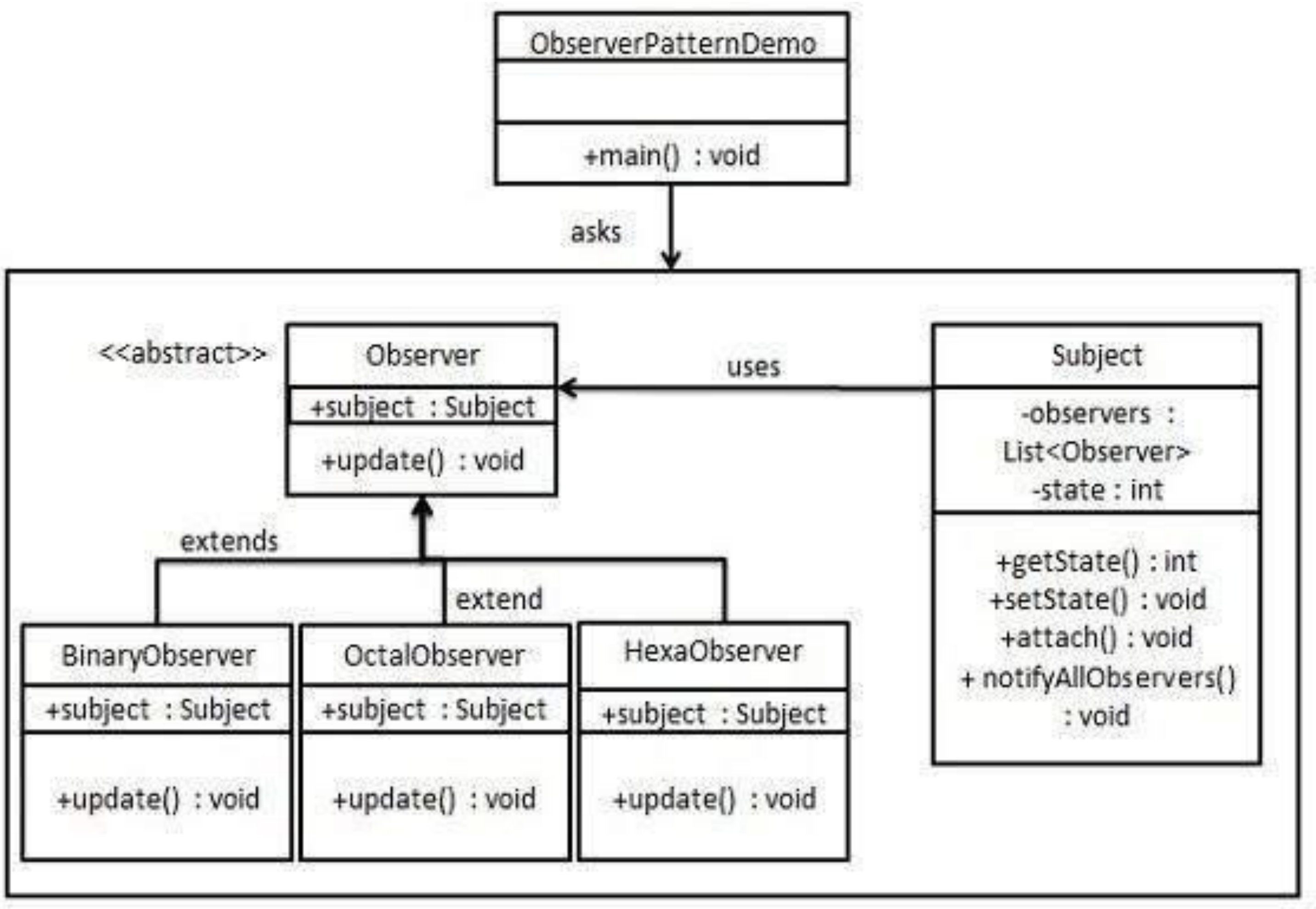
Observer Pattern

- Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically

Cont.

- Observer pattern uses three actor classes.
- Subject, Observer and Client.
- Subject is an object having methods to attach and detach observers to a client object.
- We have created an abstract class *Observer* and a concrete class*Subject* that is extending class *Observer*.

Example


ObserverPatternDemo
+main() : void
asks
<<abstract>>
Observer
+subject : Subject
+update() : void
uses
Subject
-observers : List<Observer>
-state : int
+getState() : int
+setState() : void
+attach() : void
+ notifyAllObservers() : void
extends
extend
BinaryObserver
+subject : Subject
+update() : void
OctalObserver
+subject : Subject
+update() : void
HexaObserver
+subject : Subject
+update() : void

Subject.java

```java
import java.util.ArrayList;
import java.util.List;

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();
      }
   }
}
```

Observer.java

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

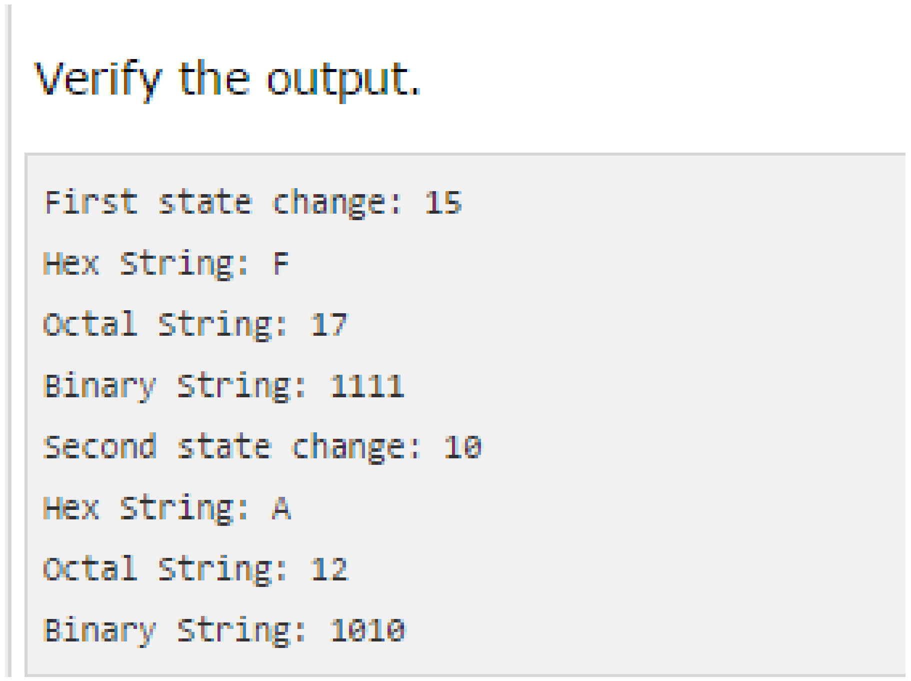
BinaryObserver.java

```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.getState() )
   }
}
```

ObserverPatternDemo.java

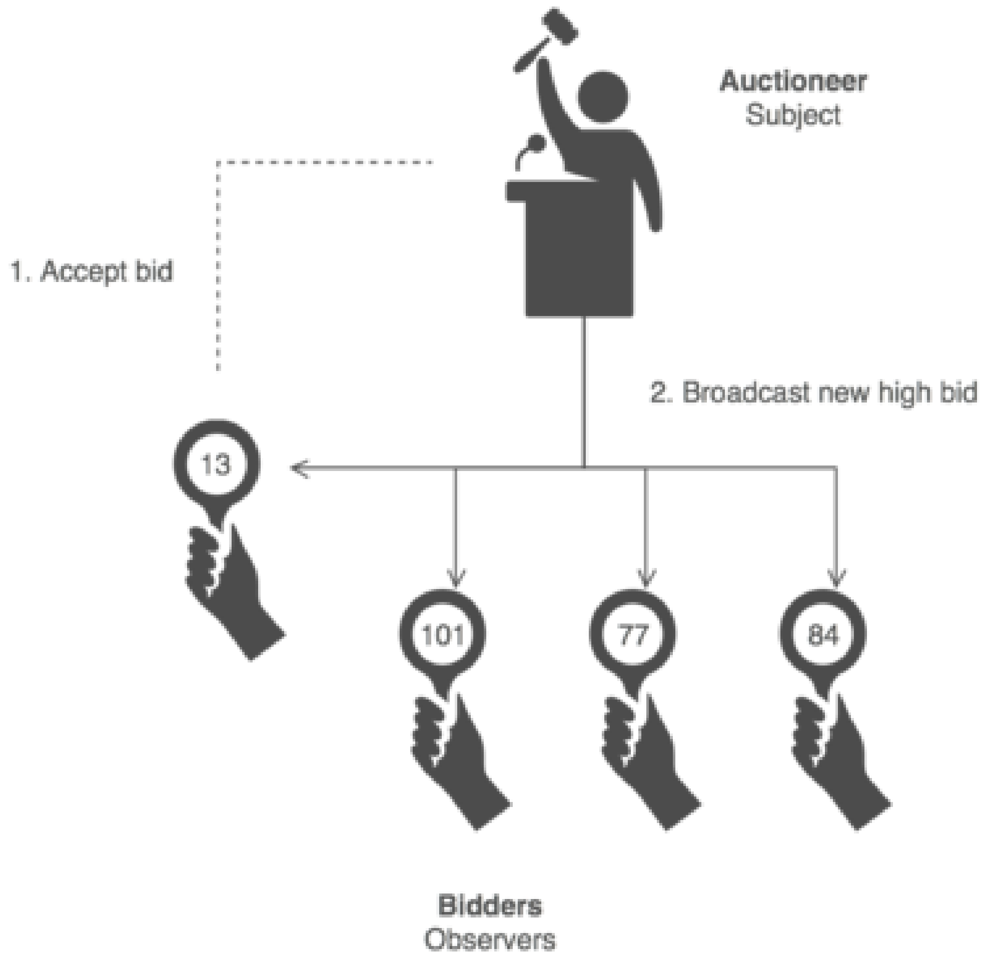
```java
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      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);
   }
}
```

Output


Verify the output.

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

Example


Auctioneer
Subject
1. Accept bid
2. Broadcast new high bid
13
101
77
84
Bidders
Observers

Mediator Pattern

- Mediator pattern is used to reduce communication complexity between multiple objects or classes.

- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Cont.

- This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling.

- What problems can the Mediator design pattern solve?
  - Tight coupling between a set of interacting objects should be avoided.
  - It should be possible to change the interaction between a set of objects independently.

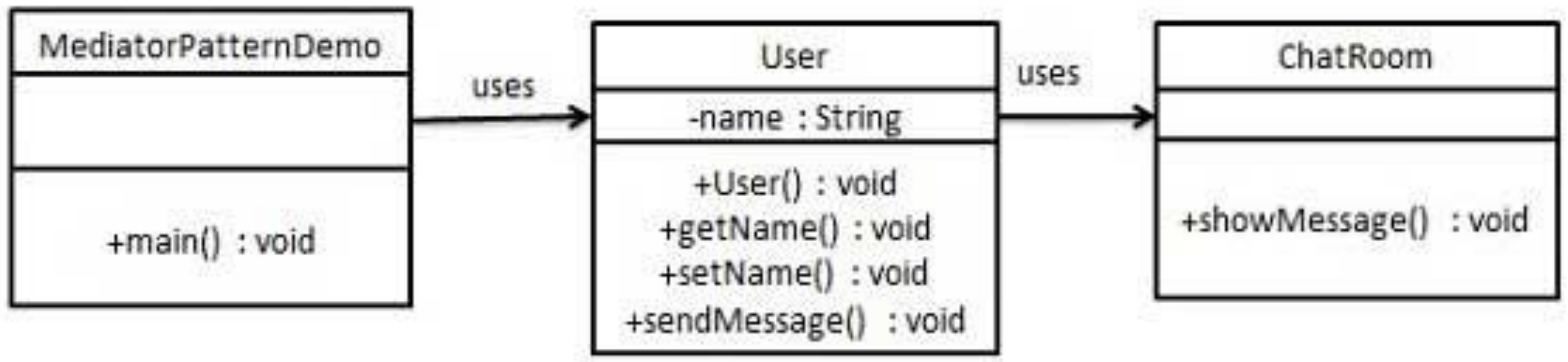
Example

- The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.

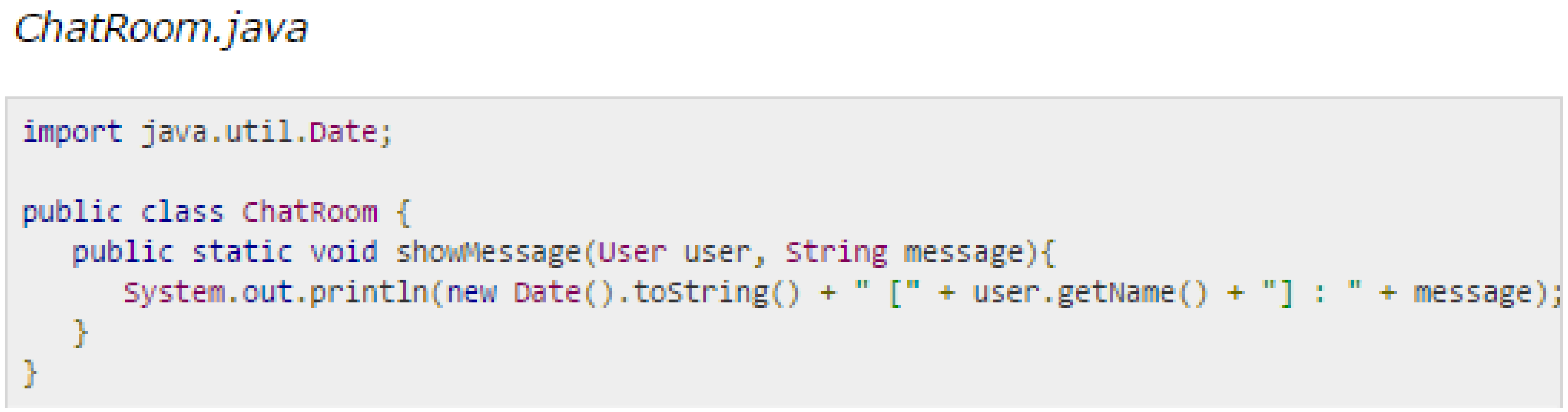
Example

- We are demonstrating mediator pattern by example of a chat room where multiple users can send message to chat room and it is the responsibility of chat room to show the messages to all users. We have created two classes*ChatRoom* and *User*. *User* objects will use *ChatRoom* method to share their messages.

Example


MediatorPatternDemo
+main() : void
uses
User
-name : String
+User() : void
+getName() : void
+setName() : void
+sendMessage() : void
uses
ChatRoom
+showMessage() : void

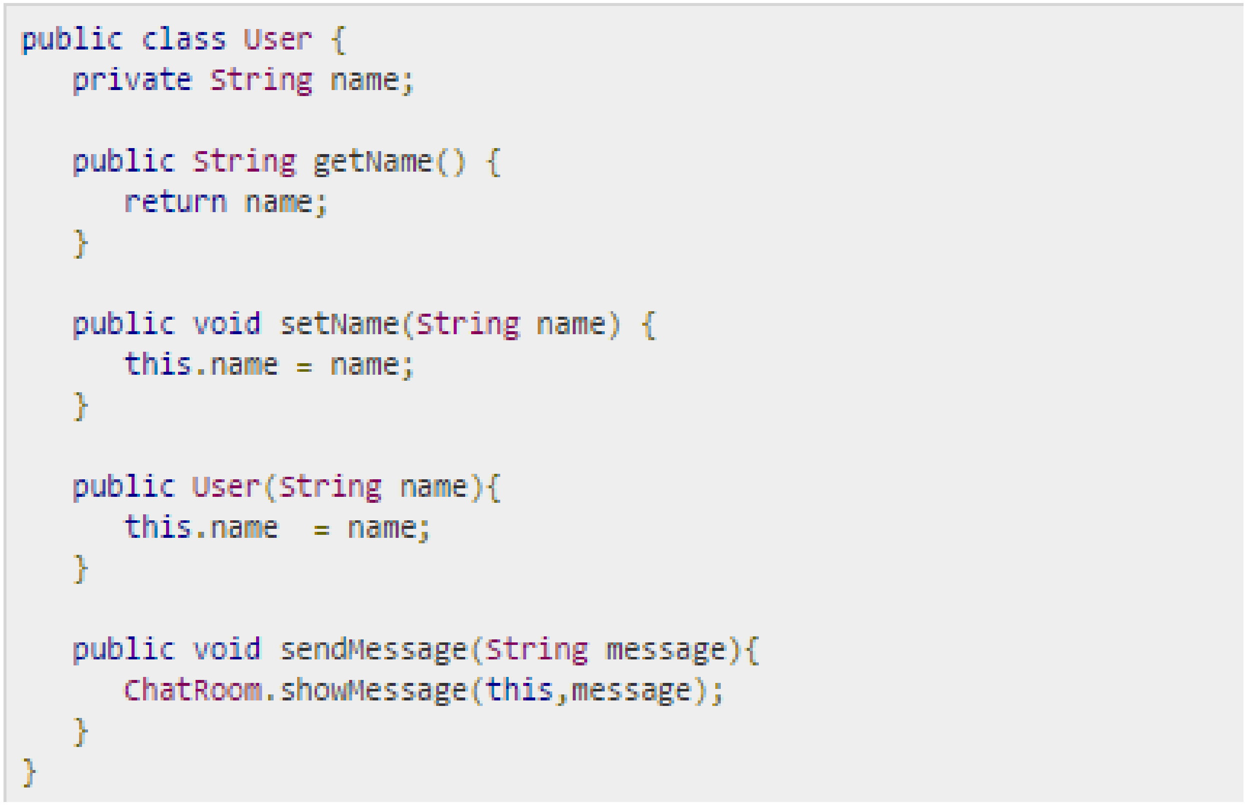
Example


ChatRoom.java

```
import java.util.Date;

public class ChatRoom {
   public static void showMessage(User user, String message){
      System.out.println(new Date().toString() + " [" + user.getName() + "] : " + message);
   }
}
```

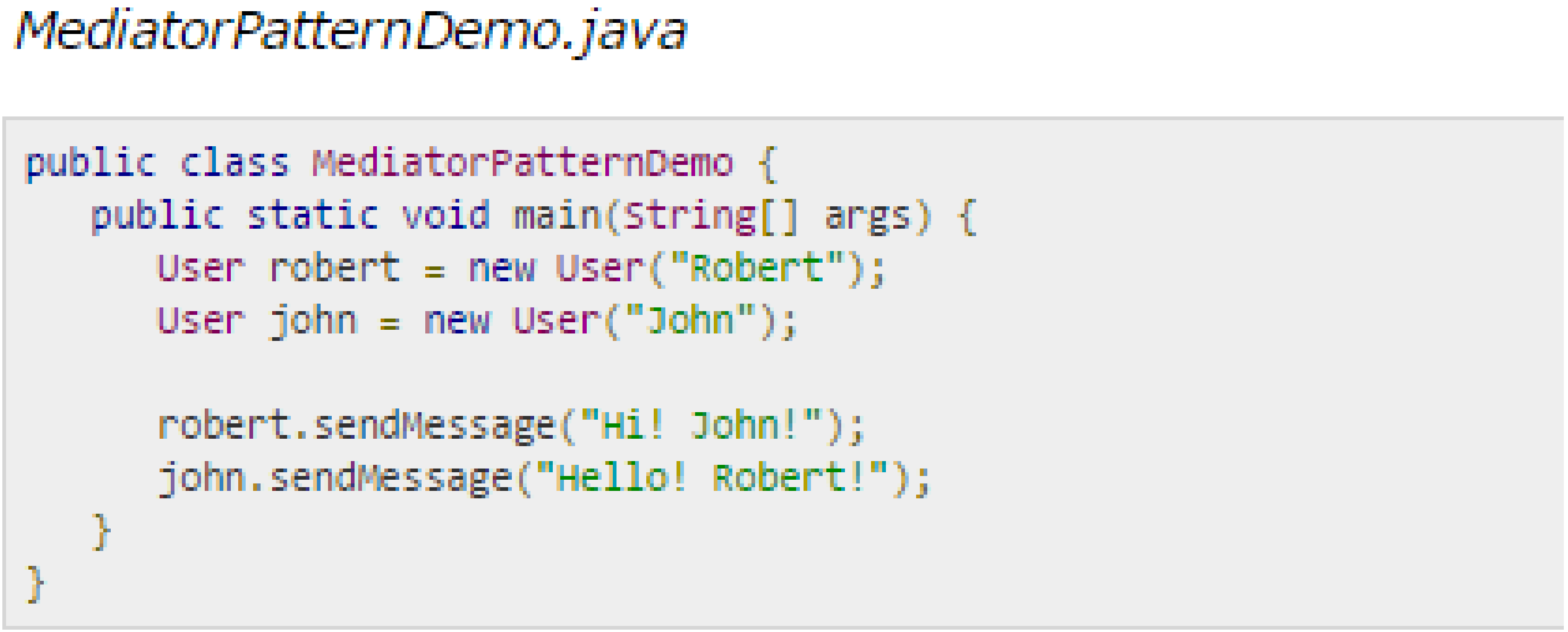
Example

User.java



```
public class User {
   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public User(String name){
      this.name  = name;
   }

   public void sendMessage(String message){
      ChatRoom.showMessage(this,message);
   }
}
```

Example



MediatorPatternDemo.java

```java
public class MediatorPatternDemo {
   public static void main(String[] args) {
      User robert = new User("Robert");
      User john = new User("John");

      robert.sendMessage("Hi! John!");
      john.sendMessage("Hello! Robert!");
   }
}
```

Memento Pattern

- Memento pattern is used to restore state of an object to a previous state.

- The client requests a Memento from the source object when it needs to checkpoint the source object's state.
- Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.

Memento Pattern

- Memento pattern uses three actor classes. Memento contains state of an object to be restored. Originator creates and stores states in Memento objects and Caretaker object is responsible to restore object state from Memento. We have created classes *Memento*, *Originator* and *CareTaker*

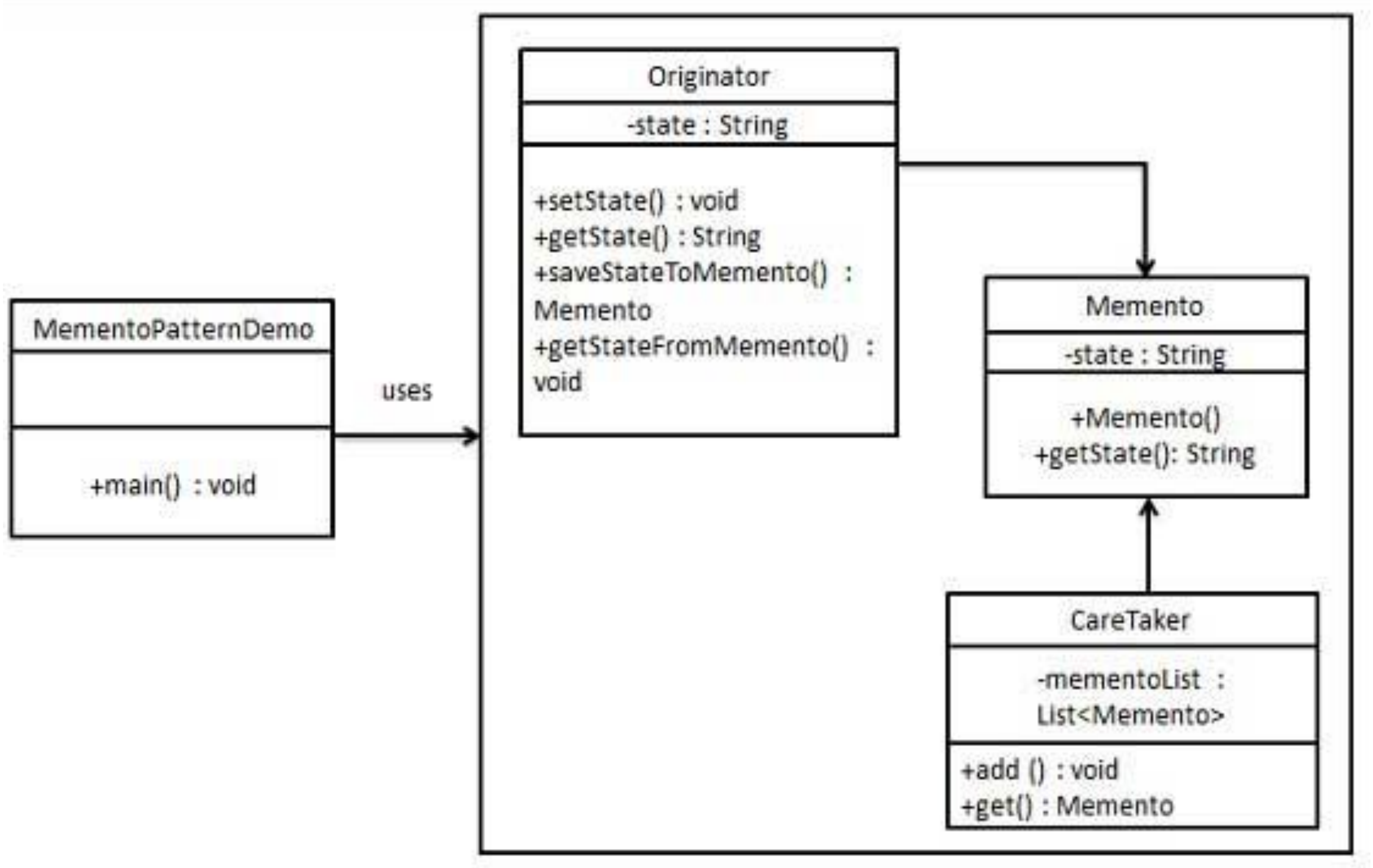
Memento Pattern

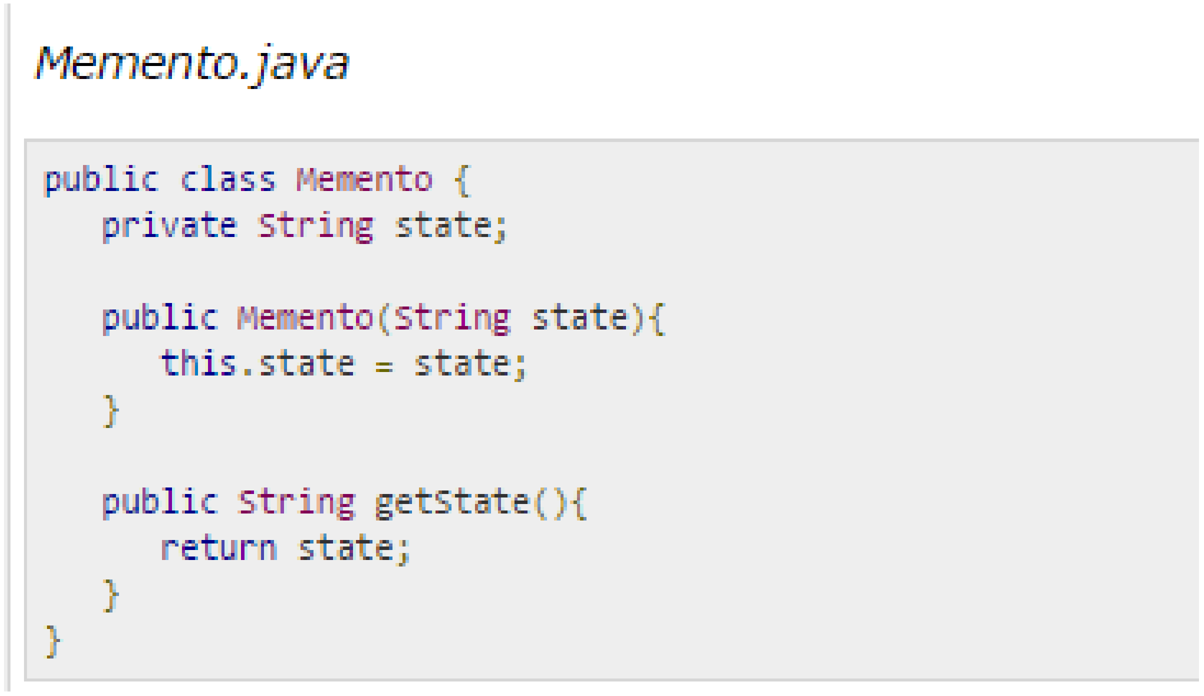
- *Originator* - the object that knows how to save itself.
- *Caretaker* - the object that knows why and when the Originator needs to save and restore itself.
- *Memento* - the lock box that is written and read by the Originator, and guided by the Caretaker.

Example

- This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled and the other serves as a Memento of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento.

Example


MementoPatternDemo
+main() : void
uses
Originator
-state : String
+setState() : void
+getState() : String
+saveStateToMemento() : Memento
+getStateFromMemento() : void
Memento
-state : String
+Memento()
+getState(): String
CareTaker
-mementoList : List<Memento>
+add () : void
+get() : Memento



Memento.java

```java
public class Memento {
   private String state;

   public Memento(String state){
      this.state = state;
   }

   public String getState(){
      return state;
   }
}
```

Originator.java

```java
public class Originator {
   private String state;

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

   public String getState(){
      return state;
   }

   public Memento saveStateToMemento(){
      return new Memento(state);
   }

   public void getStateFromMemento(Memento Memento){
      state = Memento.getState();
   }
}
```

CareTaker.java

```java
import java.util.ArrayList;
import java.util.List;

public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();

   public void add(Memento state){
      mementoList.add(state);
   }

   public Memento get(int index){
      return mementoList.get(index);
   }
}
```

MementoPatternDemo.java

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

      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();

      originator.setState("State #1");
      originator.setState("State #2");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #3");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #4");
      System.out.println("Current State: " + originator.getState());

      originator.getStateFromMemento(careTaker.get(0));
      System.out.println("First saved State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(1));
      System.out.println("Second saved State: " + originator.getState());
   }
}
```

State Pattern

- In State pattern a class behavior changes based on its state.
- In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

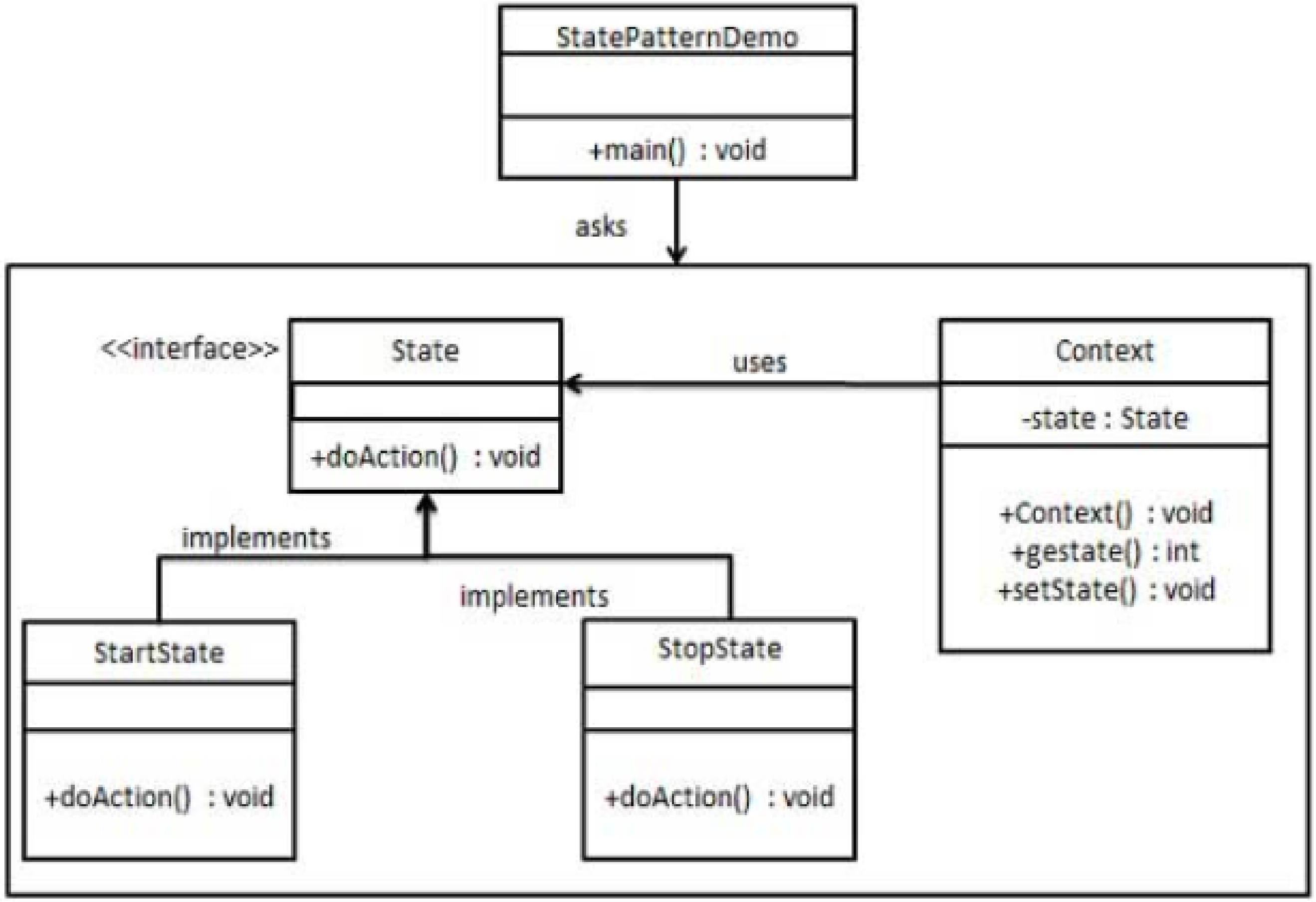
Example

- Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.

- There are times when we find some scenarios in our application when we need to maintain the state of a sub system. This state needs to be changed based on some conditions and/or user actions. One way to keep track of such states is by using the conditional logic in code.
- Using conditional logic will get us the desired result but that would result in a code that is less understandable and is harder to maintain. Also, if we need to add more states in the system then that is very difficult and could create problems in existing system too.

Components

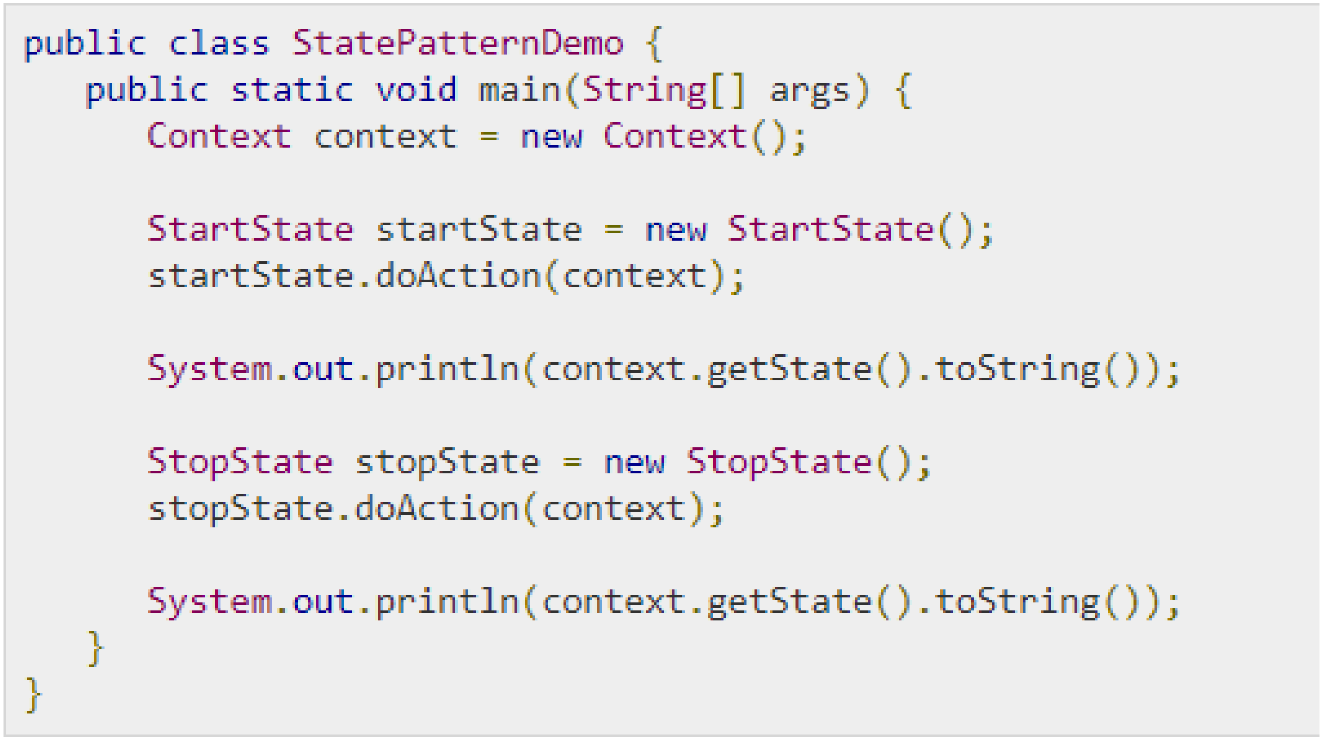
- Context: This is the actual object who is accessed by the client. The state of this object needs to be tracked.
- State: This is an interface or abstract class that defines the common behavior associated with all the possible states of the Context.
- ConcreteState: This class represent a state of the Context. Each state will be represented as one concrete class.


StatePatternDemo
+main() : void
asks
<<interface>>
State
+doAction() : void
uses
Context
-state : State
+Context() : void
+gestate() : int
+setState() : void
implements
implements
StartState
+doAction() : void
StopState
+doAction() : void

```java
public interface State {
   public void doAction(Context context);
}
```

```java
public class StartState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in start state");
      context.setState(this);
   }

   public String toString(){
      return "Start State";
   }
}
```

```java
public class Context {
   private State state;

   public Context(){
      state = null;
   }

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

   public State getState(){
      return state;
   }
}
```



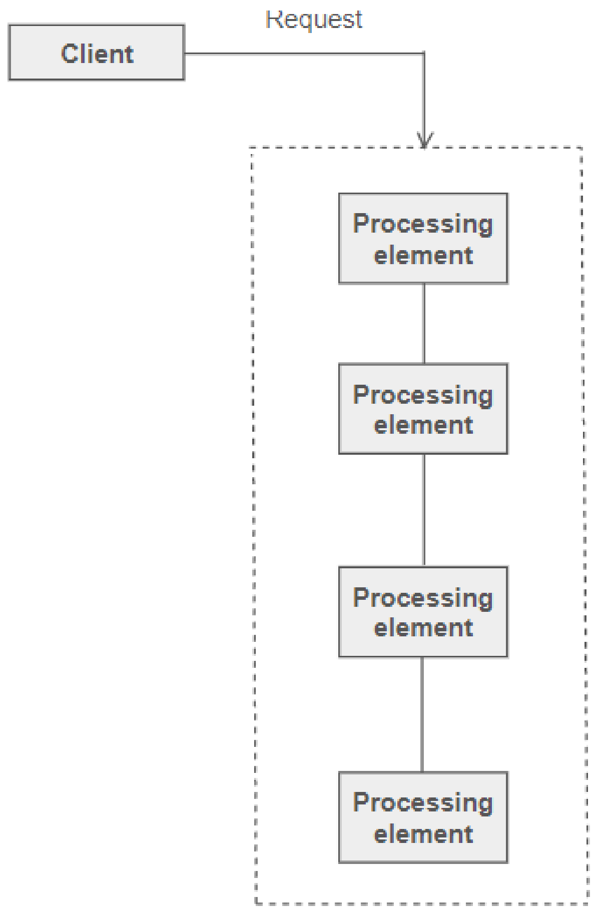
```java
public class StatePatternDemo {
   public static void main(String[] args) {
      Context context = new Context();

      StartState startState = new StartState();
      startState.doAction(context);

      System.out.println(context.getState().toString());

      StopState stopState = new StopState();
      stopState.doAction(context);

      System.out.println(context.getState().toString());
   }
}
```

Chain of Responsibility

- As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request.
- There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests

- The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically.

Scenario


Client
Request
Processing element
Processing element
Processing element
Processing element

- Chain of Responsibility simplifies object interconnections. Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.

Example

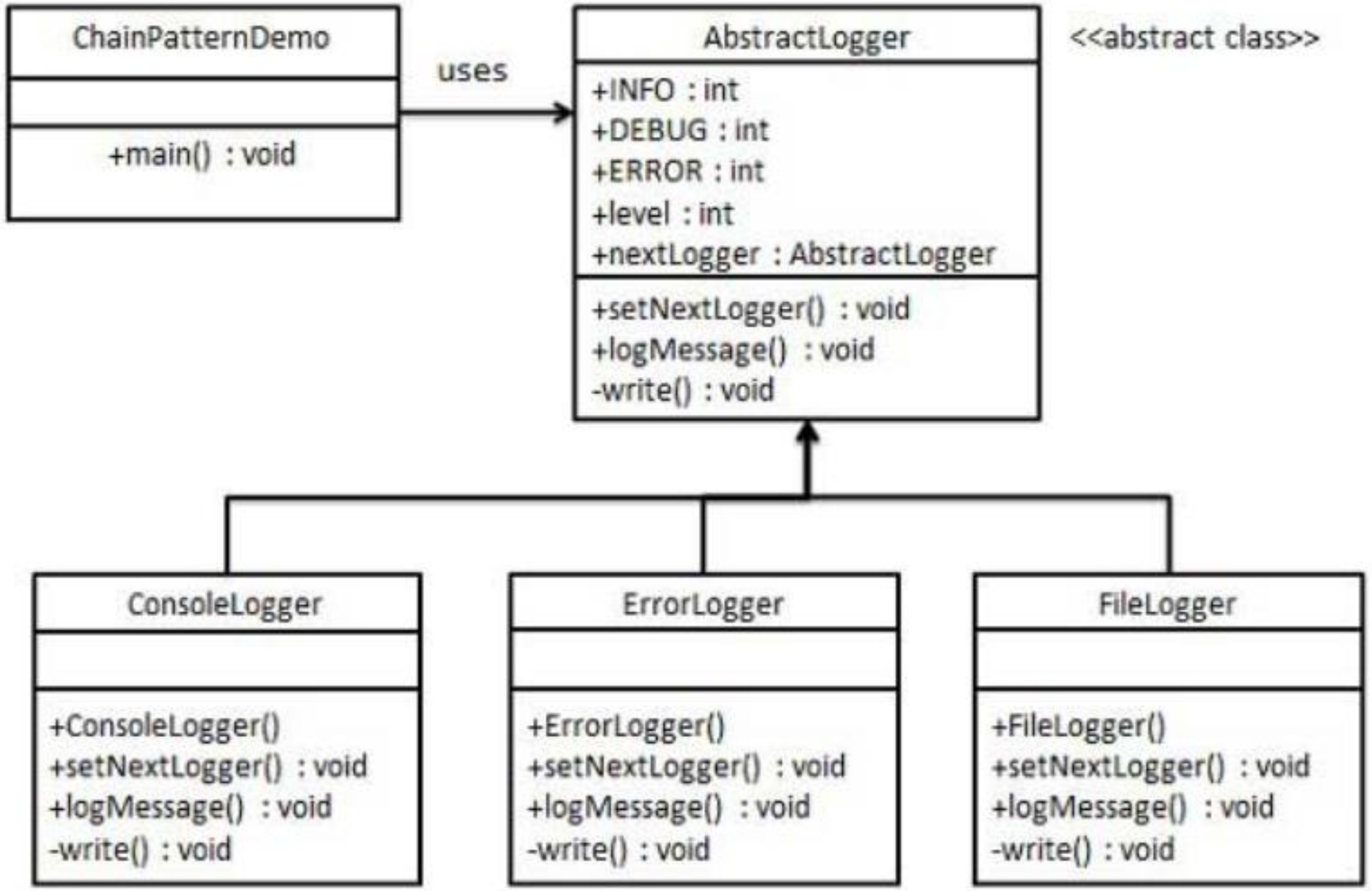
- The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.

Example

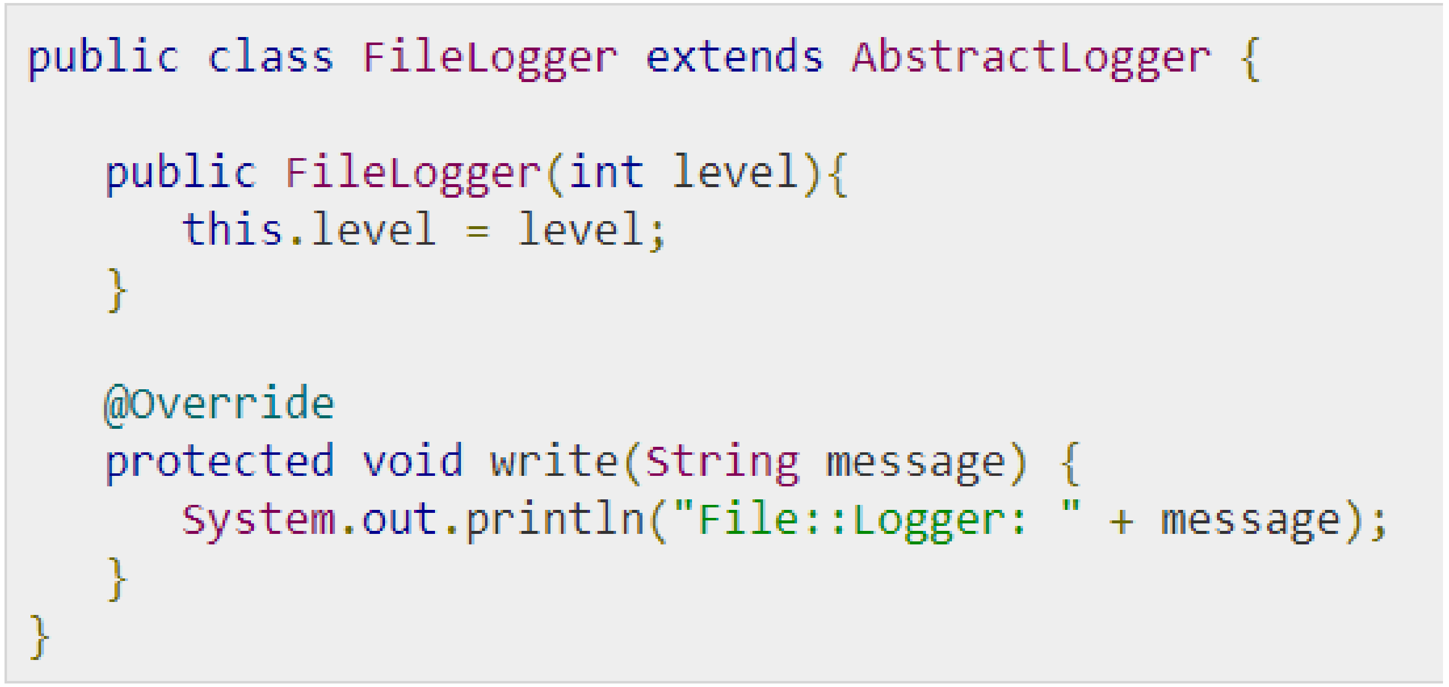
Example

- We have created an abstract class *AbstractLogger* with a level of logging. Then we have created three types of loggers extending the *AbstractLogger*. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.

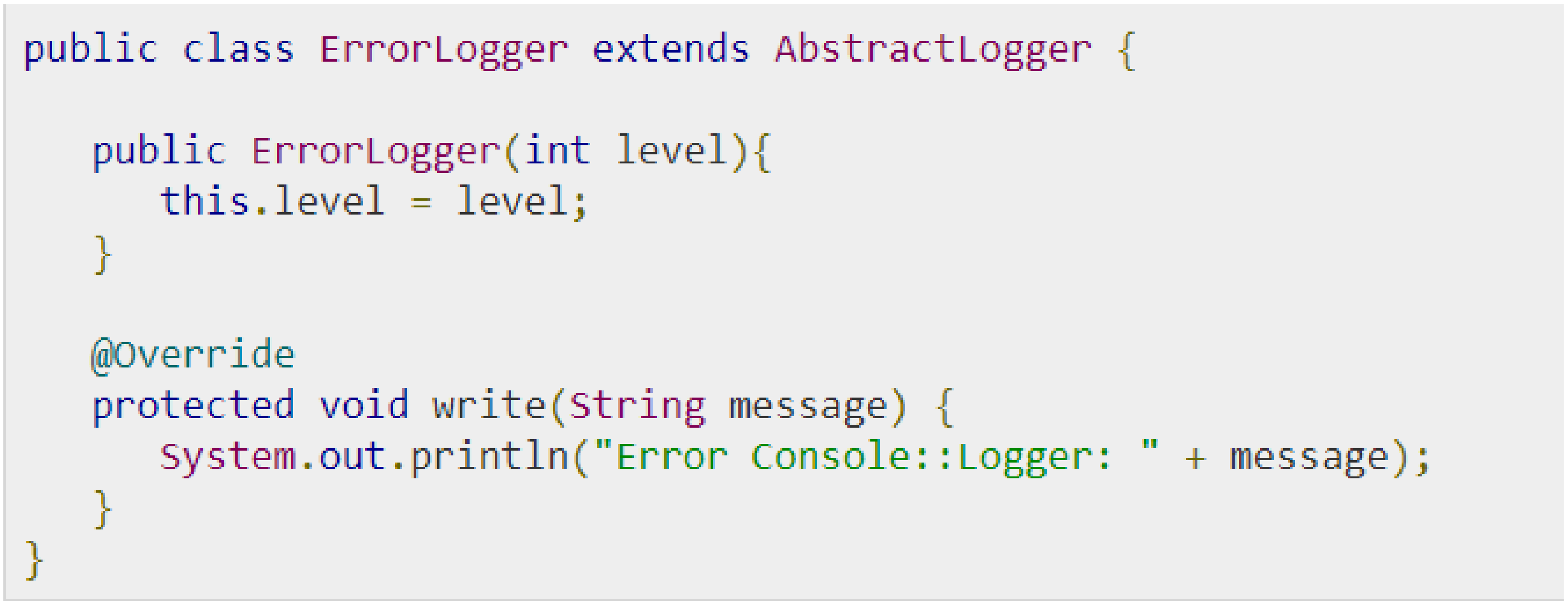
Example


ChainPatternDemo
+main() : void
uses
AbstractLogger
<<abstract class>>
+INFO : int
+DEBUG : int
+ERROR : int
+level : int
+nextLogger : AbstractLogger
+setNextLogger() : void
+logMessage() : void
-write() : void
ConsoleLogger
+ConsoleLogger()
+setNextLogger() : void
+logMessage() : void
-write() : void
ErrorLogger
+ErrorLogger()
+setNextLogger() : void
+logMessage() : void
-write() : void
FileLogger
+FileLogger()
+setNextLogger() : void
+logMessage() : void
-write() : void

```java
public class ChainPatternDemo {

   private static AbstractLogger getChainOfLoggers(){

      AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
      AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
      AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);

      errorLogger.setNextLogger(fileLogger);
      fileLogger.setNextLogger(consoleLogger);

      return errorLogger;
   }

   public static void main(String[] args) {
      AbstractLogger loggerChain = getChainOfLoggers();

      loggerChain.logMessage(AbstractLogger.INFO,
         "This is an information.");

      loggerChain.logMessage(AbstractLogger.DEBUG,
         "This is an debug level information.");

      loggerChain.logMessage(AbstractLogger.ERROR,
         "This is an error information.");
   }
}
```



```java
public class FileLogger extends AbstractLogger {

   public FileLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("File::Logger: " + message);
   }
}
```



```java
public class ErrorLogger extends AbstractLogger {

   public ErrorLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("Error Console::Logger: " + message);
   }
}
```

```java
public abstract class AbstractLogger {
   public static int INFO = 1;
   public static int DEBUG = 2;
   public static int ERROR = 3;

   protected int level;

   //next element in chain or responsibility
   protected AbstractLogger nextLogger;

   public void setNextLogger(AbstractLogger nextLogger){
      this.nextLogger = nextLogger;
   }

   public void logMessage(int level, String message){
      if(this.level <= level){
         write(message);
      }
      if(nextLogger !=null){
         nextLogger.logMessage(level, message);
      }
   }

   abstract protected void write(String message);

}
```

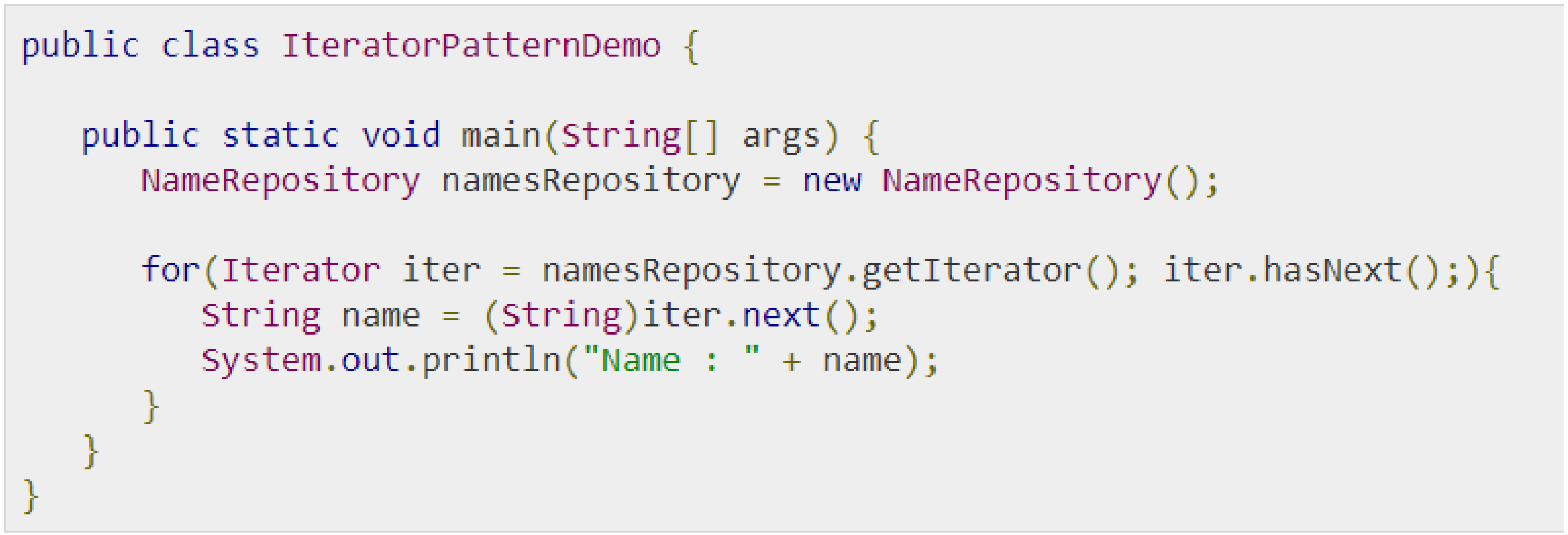
Iterator Pattern

Introduction

- Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

- We're going to create a *Iterator* interface which narrates navigation method and a *Container* interface which retruns the iterator . Concrete classes implementing the *Container* interface will be responsible to implement *Iterator*interface and use it
- *IteratorPatternDemo*, our demo class will use *NamesRepository*, a concrete class implementation to print a *Names* stored as a collection in *NamesRepository*.

Example



```java
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

```java
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {

         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {

         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

Iterator.java

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

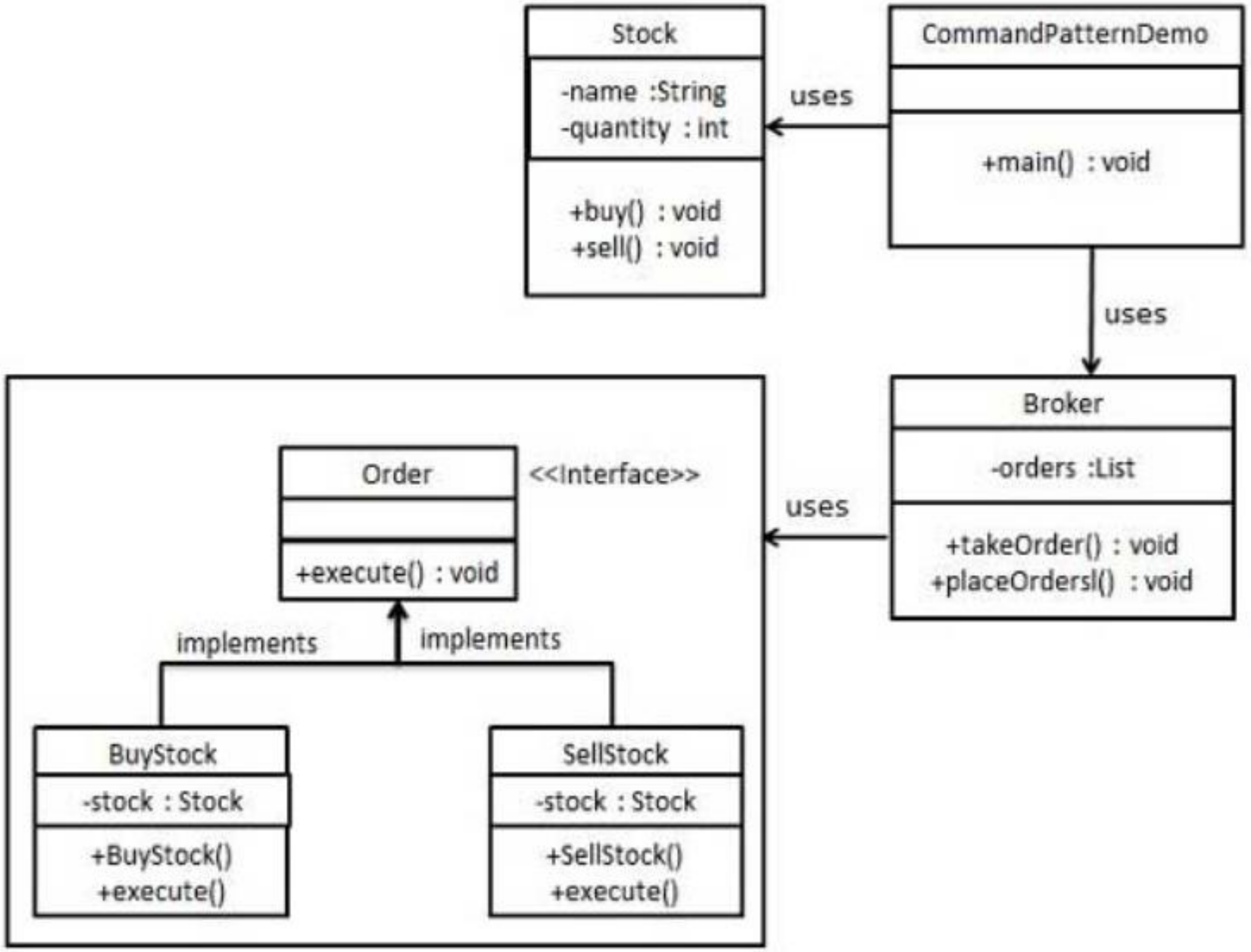
Command Pattern (Self Study – Not part of Syllabus)

- Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

Example

- We have created an interface *Order* which is acting as a command. We have created a *Stock* class which acts as a request. We have concrete command classes *BuyStock* and *SellStock* implementing *Order* interface which will do actual command processing. A class *Broker* is created which acts as an invoker object. It can take and place orders.

- *Broker* object uses command pattern to identify which object will execute which command based on the type of command. *CommandPatternDemo*, our demo class, will use *Broker* class to demonstrate command pattern.

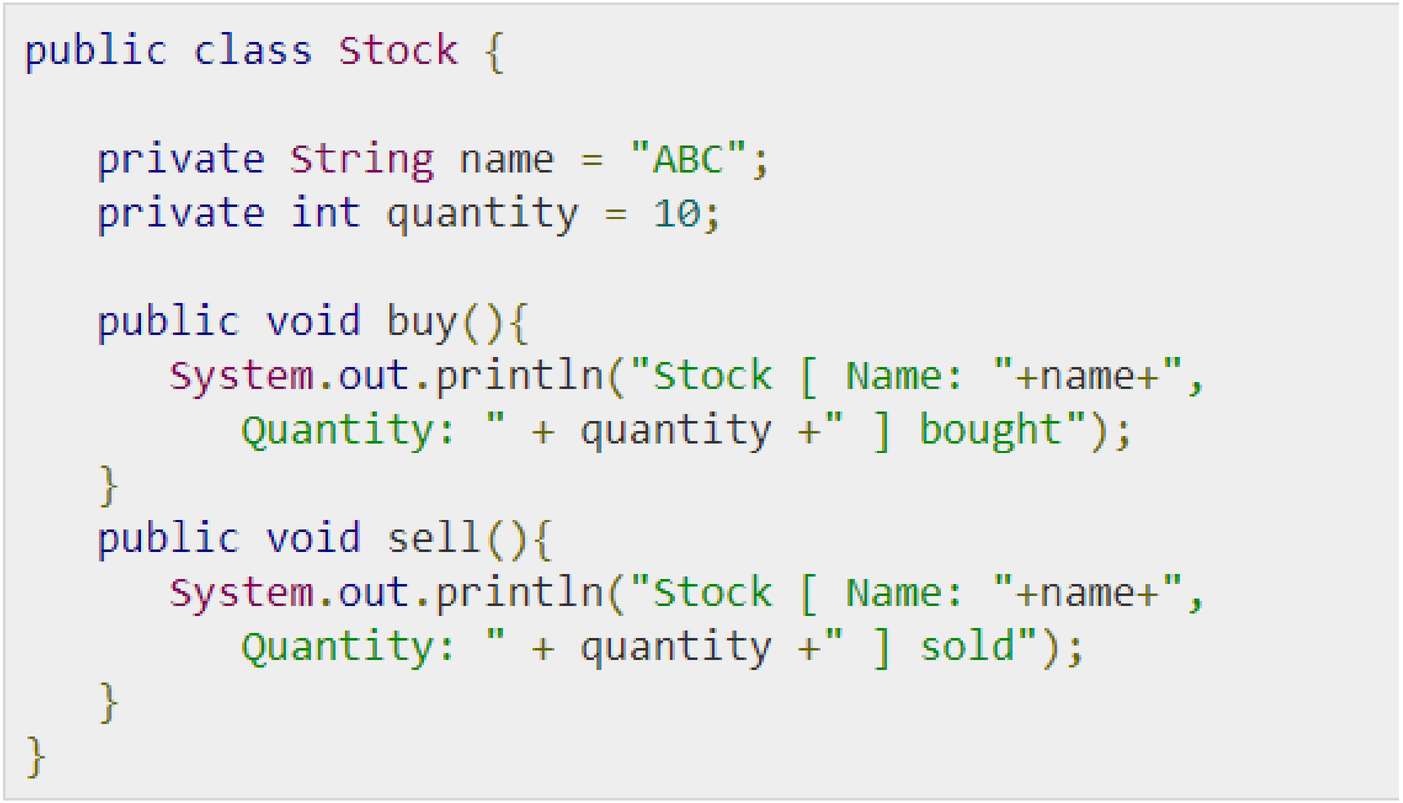

Stock
-name :String
-quantity : int
+buy() : void
+sell() : void
uses
CommandPatternDemo
+main() : void
uses
Broker
-orders :List
+takeOrder() : void
+placeOrdersl() : void
uses
Order
<<Interface>>
+execute() : void
implements
implements
BuyStock
-stock : Stock
+BuyStock()
+execute()
SellStock
-stock : Stock
+SellStock()
+execute()

```java
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();

      BuyStock buyStockOrder = new BuyStock(abcStock);
      SellStock sellStockOrder = new SellStock(abcStock);

      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);

      broker.placeOrders();
   }
}
```

```java
import java.util.ArrayList;
import java.util.List;

   public class Broker {
   private List<Order> orderList = new ArrayList<Order>();

   public void takeOrder(Order order){
      orderList.add(order);
   }

   public void placeOrders(){

      for (Order order : orderList) {
         order.execute();
      }
      orderList.clear();
   }
}
```

```java
public class SellStock implements Order {
   private Stock abcStock;

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

```java
public class BuyStock implements Order {
   private Stock abcStock;

   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.buy();
   }
}
```



```java
public class Stock {

   private String name = "ABC";
   private int quantity = 10;

   public void buy(){
      System.out.println("Stock [ Name: "+name+", 
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+", 
         Quantity: " + quantity +" ] sold");
   }
}
```

```
Stock [ Name: ABC, Quantity: 10 ] bought
Stock [ Name: ABC, Quantity: 10 ] sold
```