Design Patterns in Java

A design patterns are well-proved solution for solving the specific problem/task. Design patterns represent the best practices used by experienced object-oriented software developers.

But remember one-thing, design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

1. Creational Design Pattern

- Factory Pattern
- Abstract Factory Pattern
- Singleton Pattern
- Prototype Pattern
- Builder Pattern

2. Structural Design Pattern

- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- Proxy Pattern

3. Behavioural Design Pattern

- Chain Of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern
- Visitor Pattern

Creational Patterns

These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new opreator. This gives program more flexibility in deciding which objects need to be created for a given use case.

Structural Patterns

These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

Behavioral Patterns

These design patterns are specifically concerned with communication between objects.

Factory Pattern

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

The Factory Method Pattern is also known as **Virtual Constructor**.

JDK Example	String.valueOf(...) method
	
	This method is present in String class. valueOf() is used to convert any non String variable or Object such as int, char, double and others to a newly created String object. It returns the String representation of the argument passed
	
	Ex.
	
	`String s = String.valueOf(int i);`
	`String s = String.valueOf(char c);`
	`String s = String.valueOf(double d);`
	`String s = String.valueOf(boolean b);`

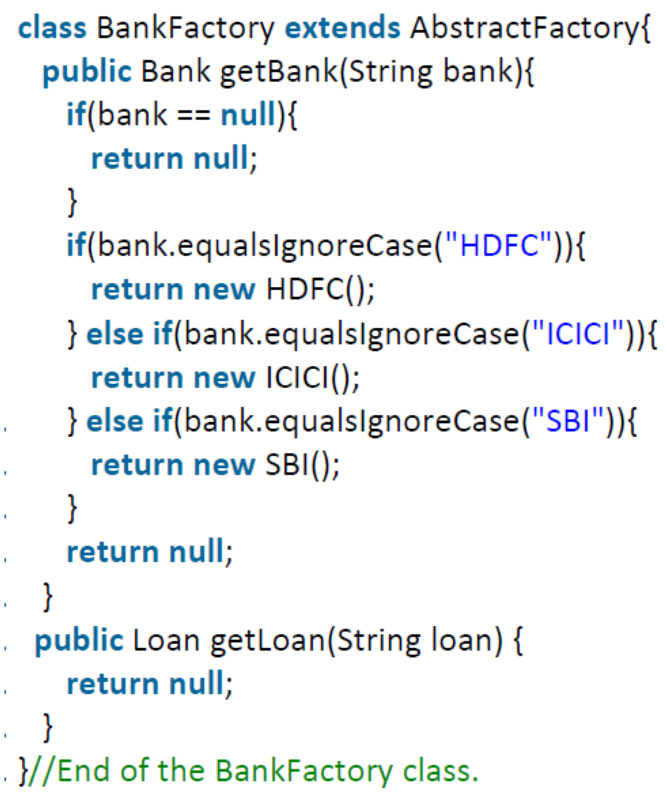
Java Example	Example of Factory Design Pattern can be mentioned by quoting BankFactory class below. Correlate this with Bank Payment Gateway where we can create object depending on the bank name selected.
	
	`class BankFactory`
	`{`
	`public Bank getBank(String bank)`
	`{`
	`if(bank.equalsIgnoreCase("HDFC")){`
	`return new HDFC();`
	`} else if(bank.equalsIgnoreCase("ICICI")){`
	`return new ICICI();`
	`} else if(bank.equalsIgnoreCase("SBI")){`
	`return new SBI();`
	`}`
	`}`

Abstract Factory Pattern

Abstract Factory lets a class returns a factory of classes. So, this is the reason that. Abstract Factory Pattern is one level higher than the Factory Pattern. An Abstract Factory Pattern is also known as Kit.

Java Example

```
abstract class AbstractFactory
{
public abstract Bank getBank(String bank);
public abstract Loan getLoan(String loan);
}
```



```
class BankFactory extends AbstractFactory{
   public Bank getBank(String bank){
      if(bank == null){
         return null;
      }
      if(bank.equalsIgnoreCase("HDFC")){
         return new HDFC();
      } else if(bank.equalsIgnoreCase("ICICI")){
         return new ICICI();
      } else if(bank.equalsIgnoreCase("SBI")){
         return new SBI();
      }
      return null;
   }
  public Loan getLoan(String loan) {
      return null;
   }
}//End of the BankFactory class.
```

```
class LoanFactory extends AbstractFactory{
         public Bank getBank(String bank){
             return null;
        }

    public Loan getLoan(String loan){
     if(loan == null){
       return null;
     }
     if(loan.equalsIgnoreCase("Home")){
       return new HomeLoan();
     } else if(loan.equalsIgnoreCase("Business")){
       return new BussinessLoan();


    } else if(loan.equalsIgnoreCase("Education")){
      return new EducationLoan();
    }
    return null;
  }
```

Singleton Pattern

Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

To create the singleton class, we need to have static member of class, private constructor and static factory method.

- **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
- **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
- **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

```
class A{
 private static A obj=new A();//Early, instance will be created at load time
 private A(){}

 public static A getA(){
  return obj;
 }
}
```

```
class A{
 private static A obj;
 private A(){}

 public static A getA(){
   if (obj == null){
      synchronized(Singleton.class){
        if (obj == null){
            obj = new Singleton();//instance will be created at request time
        }
     }
     }
  return obj;
 }
}
```

Override clone also.

Significance of Classloader in Singleton Pattern

If singleton class is loaded by two classloaders, two instance of singleton class will be created, one for each classloader.

Significance of Serialization in Singleton Pattern

If singleton class is Serializable, you can serialize the singleton instance. Once it is serialized, you can deserialize it but it will not return the singleton object.

To resolve this issue, you need to override the **readResolve() method** that enforces the singleton. It is called just after the object is deserialized. It returns the singleton object.

```
public class A implements Serializable {
    //your code of singleton
    protected Object readResolve() {
        return getA();
    }

  }
```

Next is the implementation using Volatile keyword.

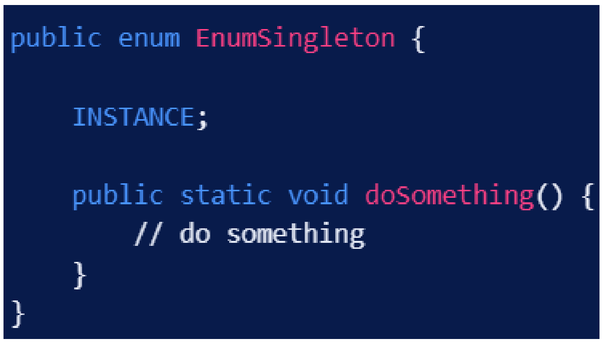
All the above implementations can fail when we use Reflection to get access to constructor and make the accessibility to true with below code.

```
Constructor[] constructors = Singleton.class.getDeclaredConstructors();

for (Constructor constructor: constructors)
{
// This code will destroy the singleton design pattern
constructor.setAccessible(true);
objectTwo = constructor.newInstance();
}
```

Enum Singleton

To overcome this situation with Reflection, we can use enum to implement Singleton as Java ensures that any enum value is instantiated only once in a Java program.

The drawback is that the enum type is somewhat inflexible (for example, it does not allow lazy initialization).



```java
public enum EnumSingleton {

    INSTANCE;

    public static void doSomething() {
        // do something
    }
}
```

__**JDK Example**__

[java.lang.Runtime#getRuntime()]

Prototype Design Pattern

Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**. This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

Builder Design Pattern

Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**. It is mostly used when object can't be created in single step like in the deserialization of a complex object.

When we have many nulls and defaults in a class we can use Builder Design Pattern.
Ex. AVRO schema we can use Builder Design Pattern.

Similarly for Booking Visibility testing to create event payload - we can use Builder Design Pattern.

Builder pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes. There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.
   We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters. The problem with this approach is that the Object state will be **inconsistent** until unless all the attributes are set explicitly. Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

JDK Example

Spring Batch example.

Let’s see how we can implement builder design pattern in java.

1. First of all you need to create a static nested class and then copy all the arguments from the outer class to the Builder class. We should follow the naming convention and if the class name is Computer then builder class should be named as ComputerBuilder.
2. Java Builder class should have a public constructor with all the required attributes as parameters.
3. Java Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
4. The final step is to provide a build() method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

Here is the sample builder pattern example code where we have a Computer class and ComputerBuilder class to build it.

```
public class Computer {

        //required parameters
        private String HDD;
        private String RAM;

        //optional parameters
        private boolean isGraphicsCardEnabled;
        private boolean isBluetoothEnabled;


        public String getHDD() {
                return HDD;
        }

        public String getRAM() {
                return RAM;
        }

        public boolean isGraphicsCardEnabled() {
                return isGraphicsCardEnabled;
        }

        public boolean isBluetoothEnabled() {
                return isBluetoothEnabled;
        }
        private Computer(ComputerBuilder builder) {
                this.HDD=builder.HDD;
                this.RAM=builder.RAM;
                this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;
                this.isBluetoothEnabled=builder.isBluetoothEnabled;
        }

        //Builder Class
        public static class ComputerBuilder{

                // required parameters
                private String HDD;
                private String RAM;

                // optional parameters
                private boolean isGraphicsCardEnabled;
                private boolean isBluetoothEnabled;

                public ComputerBuilder(String hdd, String ram){
                        this.HDD=hdd;
                        this.RAM=ram;
                }

public ComputerBuilder setGraphicsCardEnabled(
                                        boolean isGraphicsCardEnabled)
                {
                this.isGraphicsCardEnabled = isGraphicsCardEnabled;
                return this;
                }

public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled)
{
                        this.isBluetoothEnabled = isBluetoothEnabled;
                        return this;
                }

                public Computer build(){
                        return new Computer(this);
                }
        }

}
```

Notice that Computer class has only getter methods and no public constructor. So the only way to get a Computer object is through the ComputerBuilder class. Here is a builder pattern example test program showing how to use Builder class to get the object.

```
// Using builder to get the object in a single line of code and
// without any inconsistent state or arguments management
// issues

Computer comp = new Computer.ComputerBuilder("500 GB", "2GB")
                             .setBluetoothEnabled(true)
                             .setGraphicsCardEnabled(true)
                             .build();
```

Builder Design Pattern Example in JDK

Some of the builder pattern example in Java classes are;

- java.lang.StringBuilder#append() (unsynchronized)
- java.lang.StringBuffer#append() (synchronized)

Object Pool Pattern

Mostly, performance is the key issue during the software development and the object creation, which may be a costly step. Object Pool Pattern says that **"to reuse the object that are expensive to create".** Basically, an Object pool is a container which contains a specified amount of objects. When an object is taken from the pool, it is not available in the pool until it is put back. **Objects in the pool have a lifecycle: creation, validation and destroy.**

NOTE: Object pool design pattern is essentially used in Web Container of the server for creating thread pools and data source pools to process the requests.

JDK Example

ExectutorService.newFixedThreadPool(n)
ExecutorService.cachedThreadPool()
ConnectionPool

Structural Design Pattern

Adapter Pattern

Adapter design pattern is one of the **structural design pattern** and using this two unrelated interfaces can work together. The object that joins these unrelated interface is called an **Adapter**.

Adapter pattern is also known as Wrapper

Advantage of Adapter Pattern

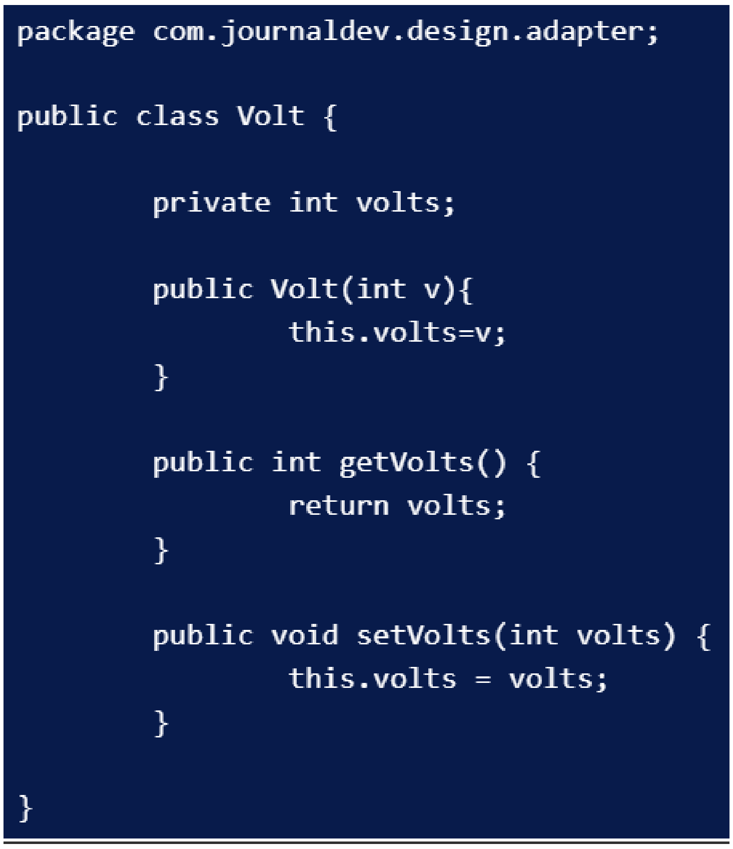
- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

One of the great real life example of Adapter design pattern is mobile charger. Mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India).

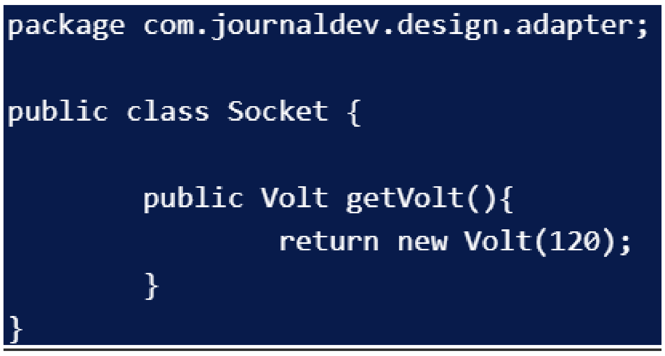
So the mobile charger works as an adapter between mobile charging socket and the wall socket.

We will try to implement multi-adapter using adapter design pattern in this tutorial.

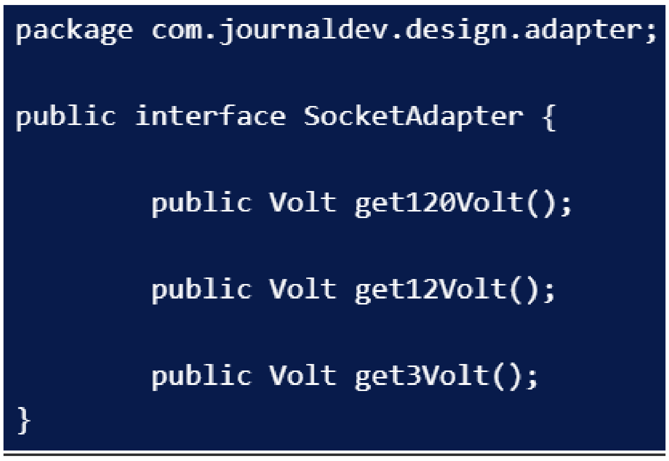
So first of all we will have two classes - Volt (to measure volts) and Socket (producing constant volts of 120V).



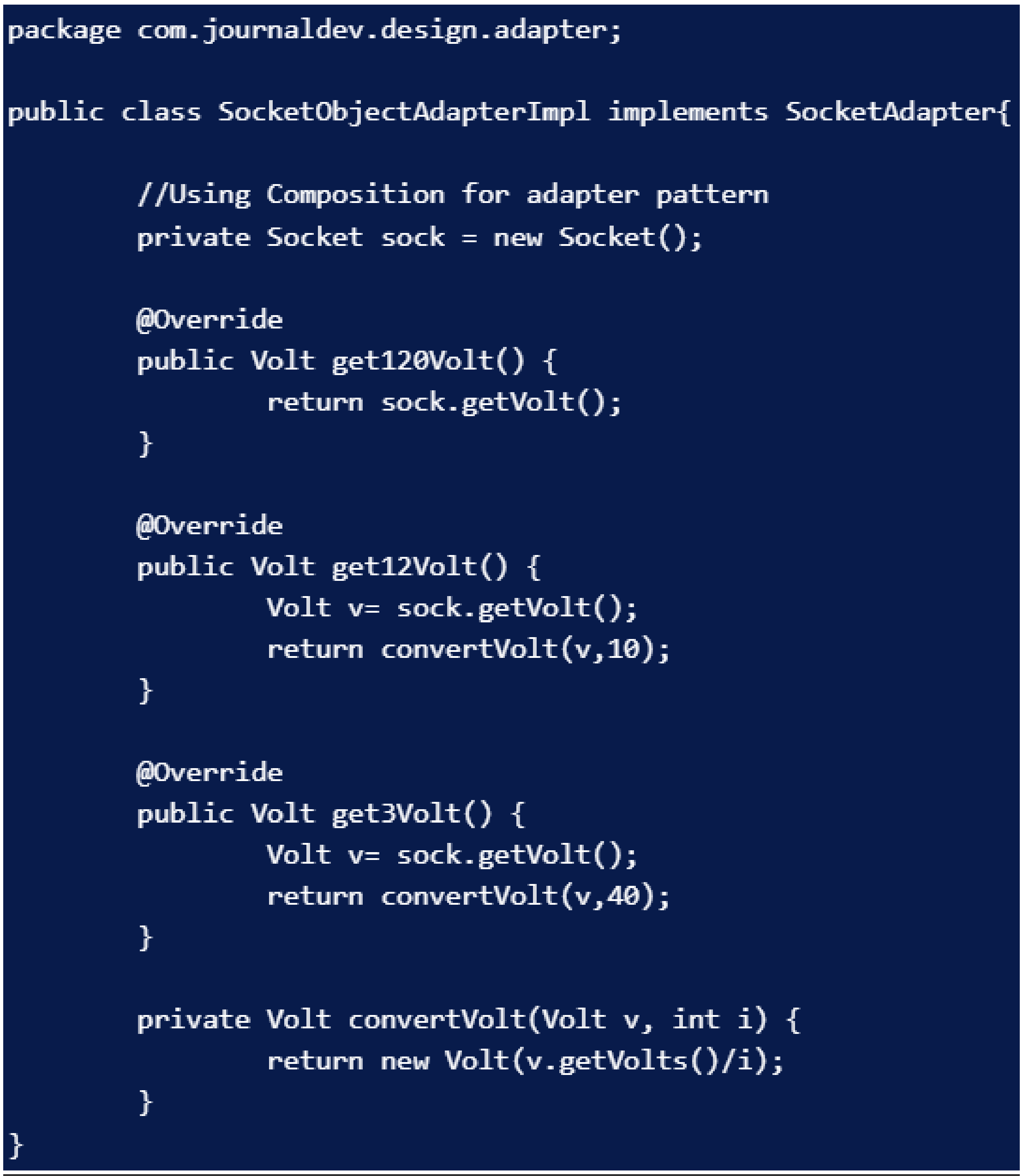
```
package com.journaldev.design.adapter;

public class Volt {

        private int volts;

        public Volt(int v){
                this.volts=v;
        }

        public int getVolts() {
                return volts;
        }

        public void setVolts(int volts) {
                this.volts = volts;
        }

}
```



```java
package com.journaldev.design.adapter;

public class Socket {

        public Volt getVolt(){
                return new Volt(120);
        }
}
```



```java
package com.journaldev.design.adapter;

public interface SocketAdapter {

        public Volt get120Volt();

        public Volt get12Volt();

        public Volt get3Volt();
}
```



```java
package com.journaldev.design.adapter;

public class SocketObjectAdapterImpl implements SocketAdapter{

        //Using Composition for adapter pattern
        private Socket sock = new Socket();

        @Override
        public Volt get120Volt() {
                return sock.getVolt();
        }

        @Override
        public Volt get12Volt() {
                Volt v= sock.getVolt();
                return convertVolt(v,10);
        }

        @Override
        public Volt get3Volt() {
                Volt v= sock.getVolt();
                return convertVolt(v,40);
        }

        private Volt convertVolt(Volt v, int i) {
                return new Volt(v.getVolts()/i);
        }
}
```

```java
SocketAdapter sockAdapter = new SocketObjectAdapterImpl();
Volt v3         =       sockAdapter.get3Volt();
Volt v12        =       sockAdapter.get12Volt();
Volt v120       =       sockAdapter.get120Volt();
```

<u>Adapter Design Pattern Example in JDK</u>

Some of the adapter design pattern example I could easily find in JDK classes are;

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)

Bridge Pattern

When we have interface hierarchies in both interfaces as well as implementations, then **bridge design pattern** is used to decouple the interfaces from implementation and hiding the implementation details from the client programs.

Example

Let us take the example of Video & VideoProcessing

In YouTube also, we can see the videos from NetFlix, we can see the videos from YouTube

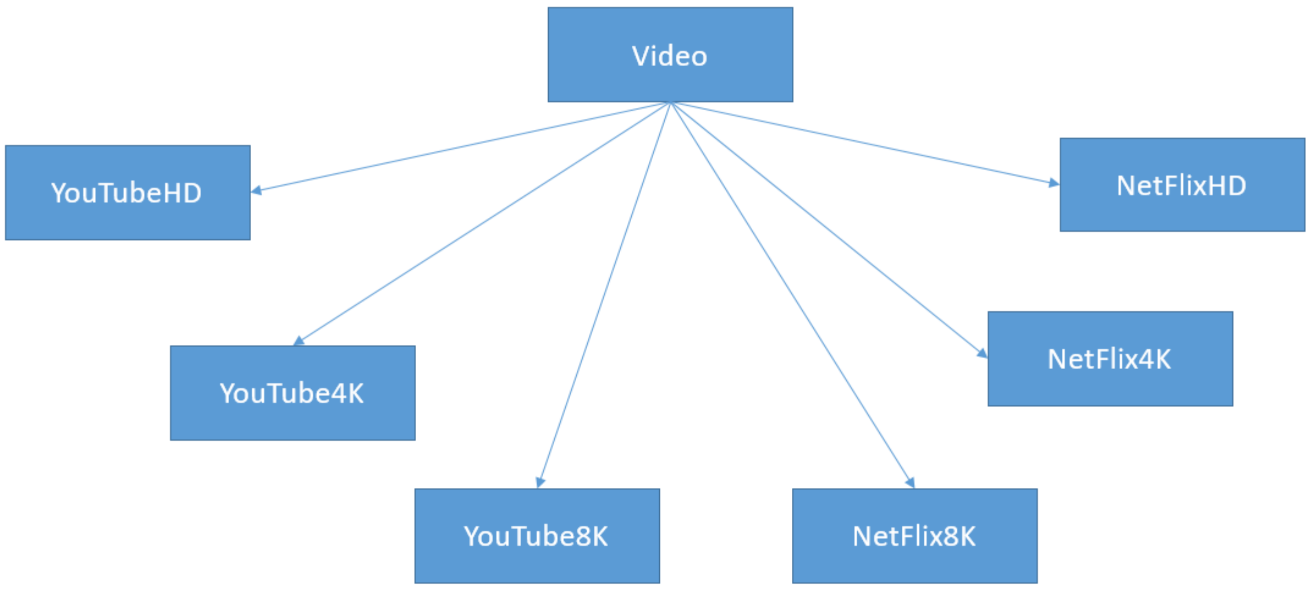
And we can play the video in different quality (360/480/720/HD etc.). So if we want to have that functionality also then all these combinations will become sub classes like –

YouTube360, YouTube480, YouTube720, YouTubeHD, YouTube4K, YouTube8K etc.

NetFlix360, NetFlix480, NetFlix720, NetFlixHD, NetFlix4K, NetFlix8K etc.

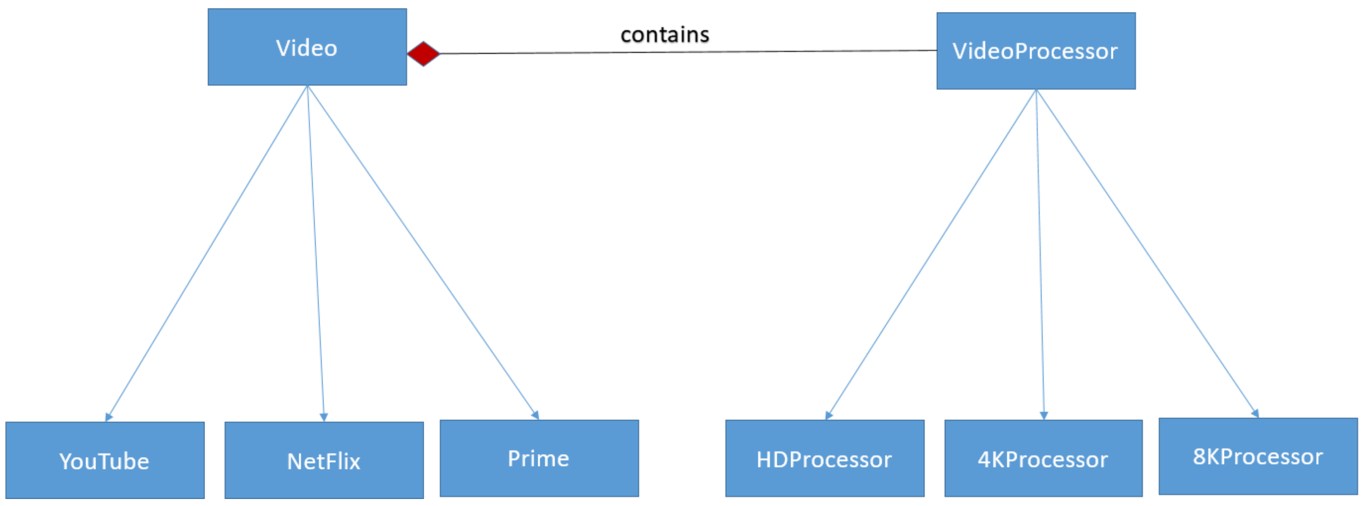
If we want to add Prime also then it will again increase the number of sub classes

So as and when we want to on-board a new functionality, we are exponentially increasing the number of classes. How can we avoid it? Using Bridge pattern.



With Bridge design pattern, entire above Is-A relationship can be converted to a Composition (Has-A relationship).

https://www.youtube.com/watch?v=7pvhfHN1zY0



Video class will contain the VideoProcessor.

Now as you see, we are not creating all hierarchies as sub classes. We are simply going to create them using composition.

Video.java

```
public abstract class Video {
    protected VideoProcessor processor;
    public Video(VideoProcessor processor){
        this.processor = processor;
    }
    public abstract void play(String videoFile);
}
```

YoutubeVideo.java

```
public class YoutubeVideo extends Video{
    public YoutubeVideo(VideoProcessor processor) {
        super(processor);
    }

    @Override
    public void play(String videoFile) {
        processor.process(videoFile); //Processed as per given processor
        //Now play
    }
}
```

NetflixVideo.java

```java
public class NetflixVideo extends Video{
    public NetflixVideo(VideoProcessor processor) {
        super(processor);
    }

    @Override
    public void play(String videoFile) {
        processor.process(videoFile);
    }
}
```

VideoProcessor.java

```java
public interface VideoProcessor {
    void process(String videoFile);
}
```

HDProcessor.java

```java
public class HDProcessor implements VideoProcessor {
    @Override
    public void process(String videoFile) {
        //Process
    }
}
```

UHD4KProcessor.java

```java
public class UHD4KProcessor implements VideoProcessor{
    @Override
    public void process(String videoFile) {
        //process
    }
}
```

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Video youtubeVideo = new YoutubeVideo(new HDProcessor());
        youtubeVideo.play("abc.mp4");
        Video netflixVideo = new NetflixVideo(new UHD4KProcessor());
        netflixVideo.play("abc.mp4");
    }
}
```

Composite Pattern

Composite pattern is used where we need to treat a group of objects in similar way as a single object.

Composite pattern composes objects in terms of a tree structure to represent part as well as whole hierarchy.

This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

Example

Let us take Order system and packaging.

Say we can package

- Item 1 -> in a small box,
- Item 2 -> in a medium box,
- Item 3 -> in a large box.

Say we have some space left in large box. So within this large box, you add Item 3 and small box as well. We can see that we are adding a box inside a box and within that small box we have item as well.

So it means, within the Box you can add Item (or) you can add another box within a box which in turn contains Item.

So we can see that it is creating a hierarchy here. Box within a box within a box and item as leaf …

So when we have this kind of relationship we can implement composite design pattern

Composite design pattern contains 3 components

- **Base Component**
  - Base Component is the actual base component which you can refer to as an object. In our example, it is Box
- **Leaf**
  - It doesn't have references to other Components. In our example, it is Item. Within the Box, we can add Item. But within the Item we can't add anything else
- **Composite**
  - It consists of leaf elements and implements the operations in base component. In our example, it can contain methods to add item inside a box, add box inside a box etc.

The composite pattern is meant to allow treating individual objects and compositions of objects, or “composites” in the same way.

Now, let's dive into the implementation. Let's suppose **we want to build a hierarchical structure of departments in a company.**

Base Component

```
public interface Department {
    void printDepartmentName();
}
```

Leafs

For the leaf components, let's define classes for financial and sales departments:

```
public class FinancialDepartment implements Department {

    private Integer id;
    private String name;

    public void printDepartmentName() {
        System.out.println(getClass().getSimpleName());
    }

    // standard constructor, getters, setters
}
```

```
public class SalesDepartment implements Department {

    private Integer id;
    private String name;

    public void printDepartmentName() {
        System.out.println(getClass().getSimpleName());
    }

    // standard constructor, getters, setters
}
```

Both classes implement the *printDepartmentName()* method from the base component, where they print the class names for each of them. Also, as they're leaf classes, they don't contain other *Department* objects.

Next, let's see a composite class as well.

The Composite Element

As a composite class, let's create a *HeadDepartment* class:

```java
public class HeadDepartment implements Department {
    private Integer id;
    private String name;

    private List<Department> childDepartments;

    public HeadDepartment(Integer id, String name) {
        this.id = id;
        this.name = name;
        this.childDepartments = new ArrayList<>();
    }

    public void printDepartmentName() {
        childDepartments.forEach(Department::printDepartmentName);
    }

    public void addDepartment(Department department) {
        childDepartments.add(department);
    }

    public void removeDepartment(Department department) {
        childDepartments.remove(department);
    }
}
```

This is a composite class as it holds a collection of *Department* components, as well as methods for adding and removing elements from the list.

The composite *printDepartmentName()* method is implemented by iterating over the list of leaf elements and invoking the appropriate method for each one.

Decorator Pattern

Decorator design pattern is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behaviour. Decorator design pattern is one of the structural design pattern

- Decorator - It is a design pattern categorized in structural pattern. It helps to decorate the object meaning
- It basically keeps expanding behaviour of the object. The decorator design pattern is used to change an object’s functionality during runtime.
- Other instances of the same class will be unaffected, therefore each object will have its behaviour changed.

The Decorator Pattern is also known as **Wrapper**.

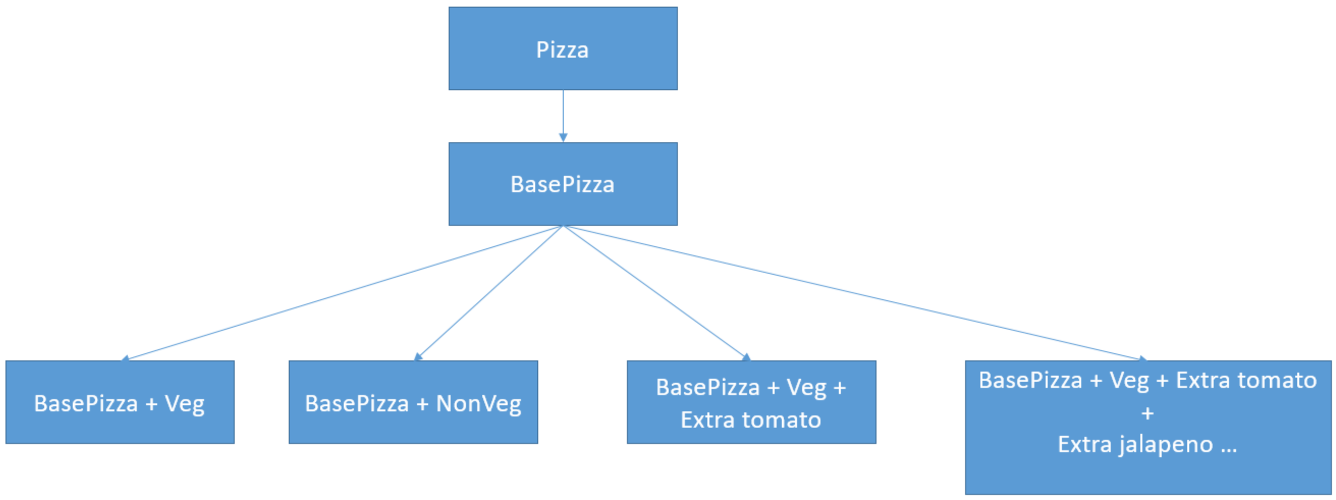
A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".**

Example

Say we want to have Pizza class. As you see we can have multiple variations (sub classes) of Pizza like VegPizza, NonVegPizza, VegPizzaExtraCheese, VegPizzaExtraCheeseExtraJalapeno etc.

With all these different permutations and combinations we can very well run into huge number of classes.

This is not ideal.


Pizza
BasePizza
BasePizza + Veg
BasePizza + NonVeg
BasePizza + Veg + Extra tomato
BasePizza + Veg + Extra tomato + Extra jalapeno …

So as you see, Base class is the same but with different behaviour.
Decorator pattern allows us to decorate a base class.

Pizza.java

```
public interface Pizza {
    public String bake();
}
```

BasePizza.java

```
public class BasePizza implements Pizza{
    @Override
    public String bake() {
        return "Base Pizza";
    }
}
```

PizzaDecorator.java

```
public abstract class PizzaDecorator implements Pizza {
    protected Pizza pizza;

    public PizzaDecorator(Pizza pizza) {
        this.pizza = pizza;
    }

    public String bake() {
        return pizza.bake();
    }
}
```

JalapenoDecorator.java

```
public class JalepanoDecorator extends PizzaDecorator{
    public JalepanoDecorator(Pizza pizza) {
        super(pizza);
    }
    public String bake() {
        return pizza.bake() + addJalepano();
    }

    public String addJalepano(){
        return "jalepeno";
    }
}
```

CheeseBurstDecorator.java

```
public class CheeseBurstDecorator extends PizzaDecorator{
    public CheeseBurstDecorator(Pizza pizza) {
        super(pizza);
    }
    public String bake() {
        return pizza.bake() + addCheese();
    }

    public String addCheese(){
        return "Cheese";
    }

}
```

```
// We got pizza with different topings , we can keep adding topings
Pizza pizza = new JalepanoDecorator(new CheeseBurstDecorator(new BasePizza()));
System.out.println(pizza.bake());
```

JDK Example

Decorator pattern is used a lot in Java IO classes, such as FileReader, BufferedReader etc.

Flyweight Pattern

Flyweight design pattern is a **Structural design pattern** like Facade pattern, Adapter Pattern and Decorator pattern.

A Flyweight Pattern says that just "**to reuse already existing similar kind of objects by storing them and create new object when no matching object is found**".

Flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects.

Before we apply flyweight design pattern, we need to consider following factors:

- The number of Objects to be created by application should be huge.
- The object creation is heavy on memory and it can be time consuming too.
- The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.

JDK Example

The best example is Java String class String Pool implementation.

Immutable class

To create an immutable class in Java, you need to follow these general principles:

1. Declare the class as final so it can't be extended.
2. Make all of the fields private so that direct access is not allowed.
3. Don't provide setter methods for variables.
4. Make all mutable fields final so that a field's value can be assigned only once.
5. Initialize all fields using a constructor method performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

```
import java.util.HashMap;
import java.util.Iterator;

public final class FinalClassExample {

        // fields of the FinalClassExample class
        private final int id;

        private final String name;

        private final HashMap<String,String> testMap;


        public int getId() {
                return id;
        }

        public String getName() {
                return name;
        }

        // Getter function for mutable objects

        public HashMap<String, String> getTestMap() {
                return (HashMap<String, String>) testMap.clone();
        }
```

```
// Constructor method performing deep copy

public FinalClassExample(int i, String n, HashMap<String,String> hm){
        System.out.println("Performing Deep Copy for Object initialization");

        // "this" keyword refers to the current object
        this.id=i;
        this.name=n;

        HashMap<String,String> tempMap=new HashMap<String,String>();
        String key;
        Iterator<String> it = hm.keySet().iterator();
        while(it.hasNext()){
                key=it.next();
                tempMap.put(key, hm.get(key));
        }
        this.testMap=tempMap;
}

// Test the immutable class

public static void main(String[] args) {
        HashMap<String, String> h1 = new HashMap<String,String>();
        h1.put("1", "first");
        h1.put("2", "second");

        String s = "original";

        int i=10;

        FinalClassExample ce = new FinalClassExample(i,s,h1);

        // print the ce values
        System.out.println("ce id: "+ce.getId());
        System.out.println("ce name: "+ce.getName());
        System.out.println("ce testMap: "+ce.getTestMap());
```

```java
// change the local variable values
i=20;
s="modified";
h1.put("3", "third");

// print the values again
System.out.println("ce id after local variable change: "+ce.getId());
System.out.println("ce name after local variable change: "+ce.getName());
System.out.println("ce testMap after local variable change: "+ce.getTestMap());

HashMap<String, String> hmTest = ce.getTestMap();
hmTest.put("4", "new");

System.out.println("ce testMap after changing variable from getter methods:
"+ce.getTestMap());

                }

}
```

Output

```
Performing Deep Copy for Object initialization
ce id: 10
ce name: original
ce testMap: {1=first, 2=second}
ce id after local variable change: 10
ce name after local variable change: original
ce testMap after local variable change: {1=first, 2=second}
ce testMap after changing variable from getter methods: {1=first, 2=second}
```

Facade Design Pattern

A Facade Pattern says that "just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".

In other words, Facade Pattern describes a higher-level interface that makes the sub-system easier to use.

Practically, **every Abstract Factory** is a type of **Facade.**

Facade pattern –

- This is also a structural pattern where it defines how classes needs to be structured in a way that for complex functionalities there is single entry point to that function.
- We should use facade pattern when we have complex subsystems , calling each leads to a single operation for client.

Advantage of Facade Pattern

- It shields the clients from the complexities of the sub-system components.
- It promotes loose coupling between subsystems and its clients.

Example

Zomato is the Food delivery app. It consists of food order, delivery person assignment, pickup, order delivery

All these sub-systems complexity can be hidden using Facade design pattern. Zomato is our Facade in this case.

ZomatoFacade.java

```
public class ZomatoFacade {
    private Restaurant restaurant;
    private  DeliveryBoy deliveryBoy;
    private DeliveryTeam deliveryTeam;

    public void placeOrder() {
        restaurant.prepareOrder();
        deliveryTeam.assignDeliveryBoy();
        deliveryBoy.pickUpOrder();
        deliveryBoy.deliverOrder();
    }
}
```

JDK Example

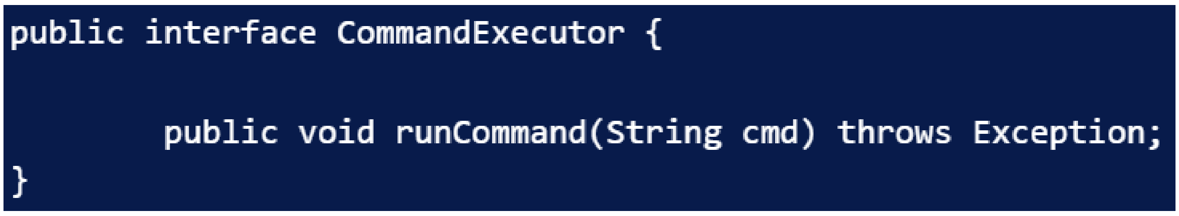
JDBC Driver Manager class to get the database connection is a wonderful example of facade design pattern

Proxy Design Pattern

Proxy Design pattern is one of the Structural design pattern. A Proxy Pattern "provides the control for accessing the original object".

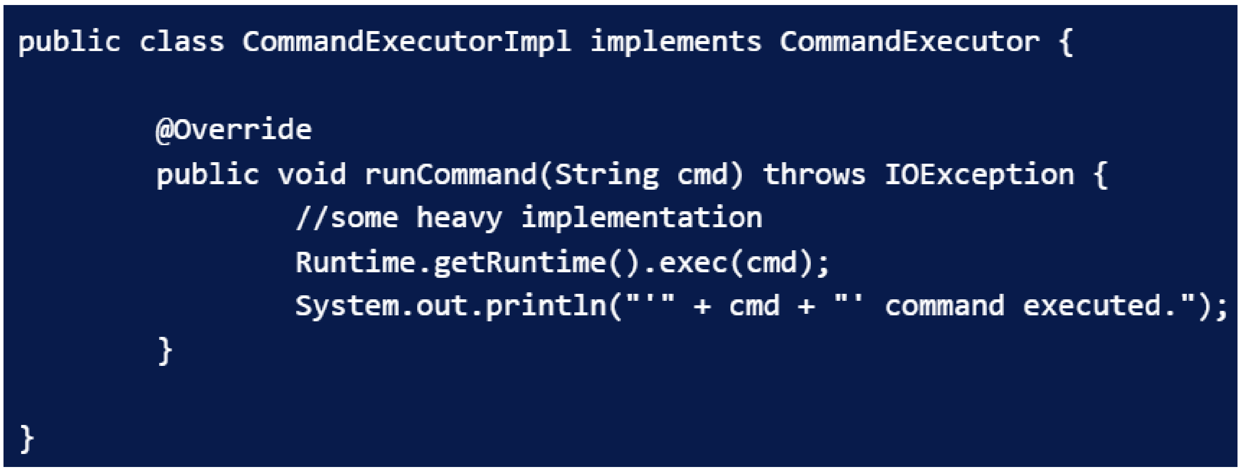
Let’s say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don’t want. Here a proxy class can be created to provide controlled access of the program.

CommandExecutor.java



```java
public interface CommandExecutor {

	public void runCommand(String cmd) throws Exception;
}
```

CommandExecutorImpl.java



```java
public class CommandExecutorImpl implements CommandExecutor {

	@Override
	public void runCommand(String cmd) throws IOException {
		//some heavy implementation
		Runtime.getRuntime().exec(cmd);
		System.out.println("'" + cmd + "' command executed.");
	}

}
```

Proxy Design Pattern - Proxy Class

Now we want to provide only admin users to have full access of above class, if the user is not admin then only limited commands will be allowed. Here is our very simple proxy class implementation. CommandExecutorProxy.java

CommandExecutorProxy.java

```
public class CommandExecutorProxy implements CommandExecutor {

        private boolean isAdmin;
        private CommandExecutor executor;

        public CommandExecutorProxy(String user, String pwd){

                if("Pankaj".equals(user) && "J@urnalD$v".equals(pwd))
                  isAdmin=true;

                executor = new CommandExecutorImpl();
        }

        @Override
        public void runCommand(String cmd) throws Exception {
                if(isAdmin){
                        executor.runCommand(cmd);
                }else{
                        if(cmd.trim().startsWith("rm")){
                                throw new Exception("rm command is not
                                                    allowed for non-admin
                                                    users.");
                        }else{
                                executor.runCommand(cmd);
                        }
                }
        }

}
```

JDK Example

Java RMI package uses proxy design pattern. Stub and Skeleton are two proxy objects used in RMI.

Chain Of Responsibility Pattern

In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.

A Chain of Responsibility Pattern says that just "avoid coupling the sender of a request to it's receiver by giving multiple objects a chance to handle the request".

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them. Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

Chain of Responsibility Pattern Example in JDK

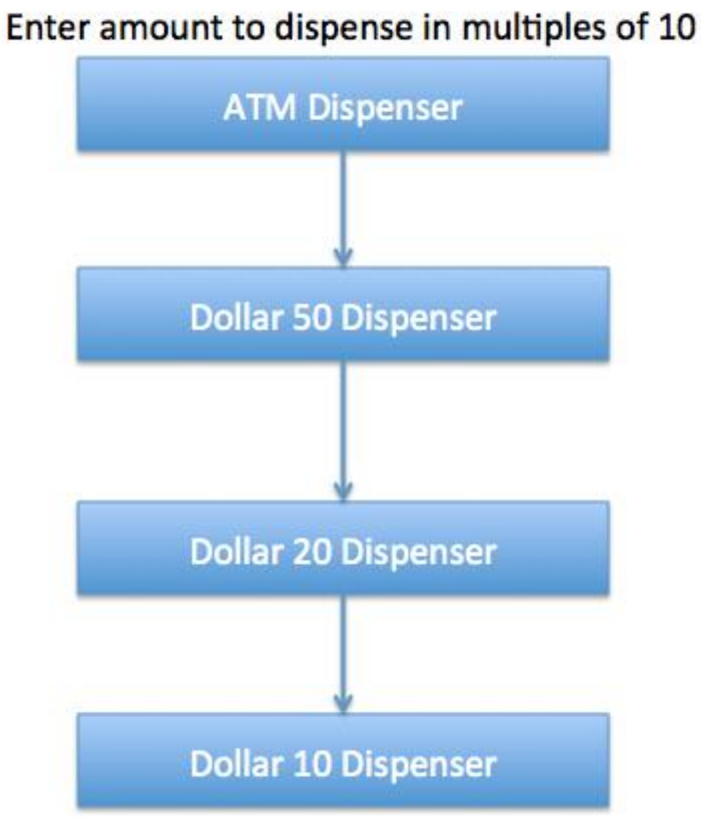
Let's see the example of chain of responsibility pattern in JDK and then we will proceed to implement a real life example of this pattern.

We know that we can have multiple catch blocks in a **try-catch block** code. Here every catch block is kind of a processor to process that particular exception. So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e. next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.

Chain of Responsibility Design Pattern Example

One of the great example of Chain of Responsibility pattern is **ATM Dispense machine**. The user enters the amount to be dispensed and the machine dispense amount in terms of defined currency bills such as 50$, 20$, 10$ etc.

If the user enters an amount that is not multiples of 10, it throws error. We will use Chain of Responsibility pattern to implement this solution. The chain will process the request in the same order as below image.

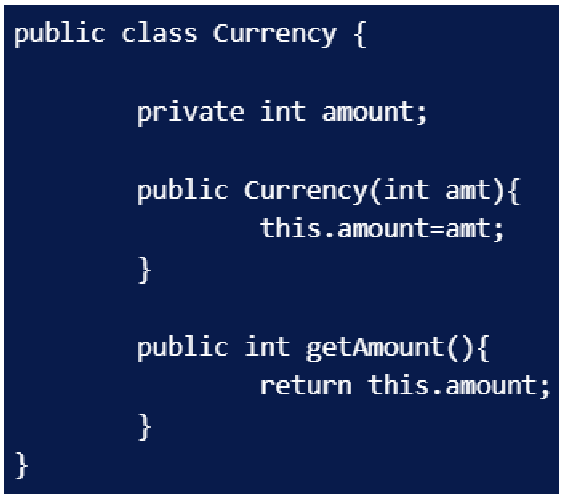


Note that we can implement this solution easily in a single program itself but then the complexity will increase and the solution will be tightly coupled. So we will create a chain of dispense systems to dispense bills of 50$, 20$ and 10$.

Chain of Responsibility Design Pattern - Base Classes and Interface

We can create a class Currency that will store the amount to dispense and used by the chain implementations. Currency.java

Currency.java



```
public class Currency {

	private int amount;

	public Currency(int amt){
		this.amount=amt;
	}

	public int getAmount(){
		return this.amount;
	}
}
```

DispenseChain.java

```
public interface DispenseChain {

        void setNextChain(DispenseChain nextChain);

        void dispense(Currency cur);
}
```

Chain of Responsibilities Pattern - Chain Implementations

Dollar50Dispenser.java

```
public class Dollar50Dispenser implements DispenseChain {

        private DispenseChain chain;

        @Override
        public void setNextChain(DispenseChain nextChain) {
                this.chain=nextChain;
        }

        @Override
        public void dispense(Currency cur) {
                if(cur.getAmount() >= 50){
                        int num = cur.getAmount()/50;
                        int remainder = cur.getAmount() % 50;
                        System.out.println("Dispensing "+num+" 50$ note");
                        if(remainder !=0) this.chain.dispense(new Currency(remainder));
                }else{
                        this.chain.dispense(cur);
                }
        }

}
```

Dollar20Dispenser.java

```java
public class Dollar20Dispenser implements DispenseChain{

	private DispenseChain chain;

	@Override
	public void setNextChain(DispenseChain nextChain) {
		this.chain=nextChain;
	}

	@Override
	public void dispense(Currency cur) {
		if(cur.getAmount() >= 20){
			int num = cur.getAmount()/20;
			int remainder = cur.getAmount() % 20;
			System.out.println("Dispensing "+num+" 20$ note");
			if(remainder !=0) this.chain.dispense(new Currency(remainder));
		}else{
			this.chain.dispense(cur);
		}
	}

}
```

Dollar10Dispenser.java

```java
public class Dollar10Dispenser implements DispenseChain {

	private DispenseChain chain;

	@Override
	public void setNextChain(DispenseChain nextChain) {
		this.chain=nextChain;
	}

	@Override
	public void dispense(Currency cur) {
		if(cur.getAmount() >= 10){
			int num = cur.getAmount()/10;
			int remainder = cur.getAmount() % 10;
			System.out.println("Dispensing "+num+" 10$ note");
			if(remainder !=0) this.chain.dispense(new Currency(remainder));
		}else{
			this.chain.dispense(cur);
		}
	}

}
```

The important point to note here is the implementation of dispense method. You will notice that every implementation is trying to process the request and based on the amount, it might process some or full part of it. If one of the chain not able to process it fully, it sends the request to the next processor in chain to process the remaining request. If the processor is not able to process anything, it just forwards the same request to the next chain.

Chain of Responsibilities Design Pattern - Creating the Chain

This is a very important step and we should create the chain carefully, otherwise a processor might not be getting any request at all.

For example, in our implementation if we keep the first processor chain as Dollar10Dispenser and then Dollar20Dispenser, then the request will never be forwarded to the second processor and the chain will become useless.

Here is our ATM Dispenser implementation to process the user requested amount.

ATMDispenseChain.java

```java
import java.util.Scanner;

public class ATMDispenseChain {

        private DispenseChain c1;

        public ATMDispenseChain() {
                // initialize the chain
                this.c1 = new Dollar50Dispenser();
                DispenseChain c2 = new Dollar20Dispenser();
                DispenseChain c3 = new Dollar10Dispenser();

                // set the chain of responsibility
                c1.setNextChain(c2);
                c2.setNextChain(c3);
        }
```

```
public static void main(String[] args) {
        ATMDispenseChain atmDispenser = new ATMDispenseChain();
        while (true) {
                int amount = 0;
                System.out.println("Enter amount to dispense");
                Scanner input = new Scanner(System.in);
                amount = input.nextInt();
                if (amount % 10 != 0) {
                        System.out.println("Amount should be in multiple of 10s.");
                        return;
                }
                // process the request
                atmDispenser.c1.dispense(new Currency(amount));
        }

}
```

Chain of Responsibility Pattern Examples in JDK

- java.util.logging.Logger#log()
- javax.servlet.Filter#doFilter()

Command Design Pattern

Command Pattern is one of the Behavioural Design Pattern. Command design pattern is used to implement **loose coupling** in a request-response model.

Advantage of command pattern

- It separates the object that invokes the operation from the object that actually performs the operation.
- It makes easy to add new commands, because existing classes remain unchanged.

In command pattern, the request is send to the invoker and invoker pass it to the encapsulated command object. Command object passes the request to the appropriate method of Receiver to perform the specific action.

Example

If we are opening a file – then File is the Receiver in this case.

Command Design Pattern Example

We will look at a real life scenario where we can implement Command pattern. Let's say we want to provide a **File System utility with methods to open, write and close file**. This file system utility should support multiple operating systems such as Windows and Unix.

To implement our File System utility, first of all we need to create the receiver classes that will actually do all the work. Since we code in terms of interface in java, we can have FileSystemReceiver interface and it's implementation classes for different operating system flavors such as Windows, Unix, Solaris etc.

Command Pattern Receiver Classes

FileSystemReceiver.java

```
public interface FileSystemReceiver {

	void openFile();
	void writeFile();
	void closeFile();
}
```

FileSystemReceiver interface defines the contract for the implementation classes. For simplicity, I am creating two flavors of receiver classes to work with Unix and Windows systems.

UnixFileSystemReceiver.java

```
public class UnixFileSystemReceiver implements FileSystemReceiver {

        @Override
        public void openFile() {
                System.out.println("Opening file in unix OS");
        }

        @Override
        public void writeFile() {
                System.out.println("Writing file in unix OS");
        }

        @Override
        public void closeFile() {
                System.out.println("Closing file in unix OS");
        }

}
```

WindowsFileSystemReceiver.java

```
public class WindowsFileSystemReceiver implements FileSystemReceiver {

        @Override
        public void openFile() {
                System.out.println("Opening file in Windows OS");

        }

        @Override
        public void writeFile() {
                System.out.println("Writing file in Windows OS");
        }

        @Override
        public void closeFile() {
                System.out.println("Closing file in Windows OS");
        }

}
```

Command Pattern Interface and Implementations

We can use interface or abstract class to create our base Command.

Command.java

```java
public interface Command {

        void execute();
}
```

Now we need to create implementations for all the different types of action performed by the receiver. Since we have three actions we will create three Command implementations. Each Command implementation will forward the request to the appropriate method of receiver.

OpenFileCommand.java

```java
public class OpenFileCommand implements Command {

        private FileSystemReceiver fileSystem;  Receiver

        public OpenFileCommand(FileSystemReceiver fs){
                this.fileSystem=fs;
        }
        @Override
        public void execute() {
                //open command is forwarding request to openFile method
                this.fileSystem.openFile();
        }

}
```

WriteFileCommand.java

```java
public class WriteFileCommand implements Command {

	private FileSystemReceiver fileSystem;

	public WriteFileCommand(FileSystemReceiver fs){
		this.fileSystem=fs;
	}
	@Override
	public void execute() {
		this.fileSystem.writeFile();
	}

}
```

CloseFileCommand.java

```java
public class CloseFileCommand implements Command {

	private FileSystemReceiver fileSystem;

	public CloseFileCommand(FileSystemReceiver fs){
		this.fileSystem=fs;
	}
	@Override
	public void execute() {
		this.fileSystem.closeFile();
	}

}
```

Now we have receiver and command implementations ready, so we can move to implement the invoker class.

<u>Command Pattern Invoker Class</u>

Invoker is a simple class that encapsulates the Command and passes the request to the command object to process it.

FileInvoker.java

```
public class FileInvoker {

	public Command command;

	public FileInvoker(Command c){
		this.command=c;
	}

	public void execute(){
		this.command.execute();
	}
}
```

Our file system utility implementation is ready and we can move to write a simple command pattern client program.

But before that I will provide a utility to create appropriate FileSystemReceiver object.

Since we can use System class to get the operating system information, we will use this or else we can use Factory pattern to return appropriate type based on the input.

FileSystemReceiverUtil.java

```
public class FileSystemReceiverUtil {

	public static FileSystemReceiver getUnderlyingFileSystem(){
		 String osName = System.getProperty("os.name");
		 System.out.println("Underlying OS is:"+osName);
		 if(osName.contains("Windows")){
			 return new WindowsFileSystemReceiver();
		 }else{
			 return new UnixFileSystemReceiver();
		 }
	}

}
```

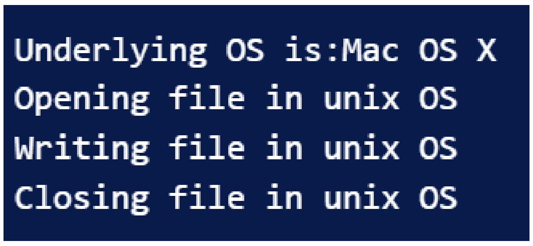
Let’s move now to create our command pattern example client program that will consume our file system utility.

FileSystemClient.java

```
public class FileSystemClient {

public static void main(String[] args) {

        //Creating the receiver object
        FileSystemReceiver fs = FileSystemReceiverUtil.getUnderlyingFileSystem();

        //creating command and associating with receiver
        OpenFileCommand openFileCommand = new OpenFileCommand(fs);

        //Creating invoker and associating with Command
        FileInvoker file = new FileInvoker(openFileCommand);

        //perform action on invoker object
        file.execute();

        WriteFileCommand writeFileCommand = new WriteFileCommand(fs);
        file = new FileInvoker(writeFileCommand);
        file.execute();

        CloseFileCommand closeFileCommand = new CloseFileCommand(fs);
        file = new FileInvoker(closeFileCommand);
        file.execute();
        }

}
```

Notice that client is responsible to create the appropriate type of command object. For example if you want to write a file you are not supposed to create CloseFileCommand object.

Client program is also responsible to attach receiver to the command and then command to the invoker class. Output of the above command pattern example program is:


```
Underlying OS is:Mac OS X
Opening file in unix OS
Writing file in unix OS
Closing file in unix OS
```

The drawback with Command design pattern is that the code gets huge and confusing with high number of action methods and because of so many associations.

<u>Command Design Pattern JDK Example</u>

- Runnable interface (java.lang.Runnable) (Thread creation cay vary based on OS. It will take care)
- Swing Action (javax.swing.Action) uses command pattern.

Interpreter Design Pattern

Interpreter design pattern is part of behavioural design pattern and it will allow us to interpret things. Evaluates expressions.

SQL Parsing uses interpreter design pattern.

Iterator Design Pattern

According to GoF, Iterator Pattern is used "to access the elements of an aggregate object sequentially without exposing its underlying implementation".

The Iterator pattern is also known as Cursor.

In collection framework, we are now using Iterator that is preferred over Enumeration.

java.util.Iterator interface uses Iterator Design Pattern.

Mediator Design Pattern

Mediator design pattern is one of the behavioural design pattern, so it deals with the behaviours of objects.

Mediator design pattern is **used to provide a centralized communication medium between different objects in a system**.

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other.

Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other.

If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and hard to extend.

Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.

Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights.

Mediator works as a router between objects and it can have its own logic to provide way of communication.

The system objects that communicate each other are called **Colleagues**. Usually we have an interface or abstract class that provides the contract for communication and then we have concrete implementation of mediators.

For our example, we will try to implement a chat application where users can do group chat. Every user will be identified by its name and they can send and receive messages. The message sent by any user should be received by all the other users in the group.

Mediator Pattern Interface

First of all we will create Mediator interface that will define the contract for concrete mediators. ChatMediator.java

ChatMediator.java

```
public interface ChatMediator {

        public void sendMessage(String msg, User user);

        void addUser(User user);
}
```

Mediator Pattern Colleague Interface

Users can send and receive messages, so we can have User interface or abstract class. I am creating User as abstract class like below. User.java

User.java

```
public abstract class User {
        protected ChatMediator mediator;
        protected String name;

        public User(ChatMediator med, String name){
                this.mediator=med;
                this.name=name;
        }

        public abstract void send(String msg);

        public abstract void receive(String msg);
}
```

Notice that User has a reference to the mediator object, it’s required for the communication between different users.

Mediator Design Pattern Concrete Colleague

Now we can create concrete User classes to be used by client system. UserImpl.java

UserImpl.java

```
public class UserImpl extends User {

        public UserImpl(ChatMediator med, String name) {
                super(med, name);
        }

        @Override
        public void send(String msg){
                System.out.println(this.name+": Sending Message="+msg);
                mediator.sendMessage(msg, this);
        }
        @Override
        public void receive(String msg) {
                System.out.println(this.name+": Received Message:"+msg);
        }

}
```

Notice that send() method is using mediator to send the message to the users and it has no idea how it will be handled by the mediator.

Concrete Mediator

Now we will create concrete mediator class, it will have a list of users in the group and provide logic for the communication between the users. ChatMediatorImpl.java

ChatMediatorImpl.java

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

public class ChatMediatorImpl implements ChatMediator {

        private List<User> users;

        public ChatMediatorImpl(){
                this.users=new ArrayList<>();
        }
        @Override
        public void addUser(User user){
                this.users.add(user);
        }

        @Override
        public void sendMessage(String msg, User user) {
                for(User u : this.users){
                        //message should not be received by the user sending it
                        if(u != user){
                                u.receive(msg);
                        }
                }
        }

}
```

Mediator Pattern Example Client Program Code
Let’s test this our chat application with a simple program where we will create mediator and add users to the group and one of the user will send a message. ChatClient.java

ChatClient.java

```
public class ChatClient {

        public static void main(String[] args) {
                ChatMediator mediator = new ChatMediatorImpl();
                User user1 = new UserImpl(mediator, "Pankaj");
                User user2 = new UserImpl(mediator, "Lisa");
                User user3 = new UserImpl(mediator, "Saurabh");
                User user4 = new UserImpl(mediator, "David");
                mediator.addUser(user1);
                mediator.addUser(user2);
                mediator.addUser(user3);
                mediator.addUser(user4);

                user1.send("Hi All");

        }

}
```

Mediator Pattern Example in JDK

- java.util.Timer class scheduleXXX() methods
- Java Concurrency Executor execute() method.
- java.lang.reflect.Method invoke() method.

Mediator Design Pattern Important Points

- Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.
- Java Message Service (JMS) uses Mediator pattern along with **Observer pattern** to allow applications to subscribe and publish data to other applications.
- We should not use mediator pattern just to achieve lose-coupling because if the number of mediators will grow, then it will become hard to maintain them.

Observer Design Pattern

Observer pattern is one of the behavioural design pattern.

Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change.

In observer pattern, the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject**.

Subject contains a list of observers to notify of any change in its state, so it should provide methods using which observers can register and unregister themselves. Subject also contain a method to notify all the observers of any change.

Java Message Service (JMS) uses **Observer design pattern** along with **Mediator pattern** to allow applications to subscribe and publish data to other applications. Model-View-Controller (MVC) frameworks also use Observer pattern where Model is the Subject and Views are observers that can register to get notified of any change to the model.

Kafka Topic is also an example.

Observer Pattern Java Example

For our observer pattern java program example, we would implement a simple topic and observers can register to this topic.

Whenever any new message will be posted to the topic, all the registers observers will be notified and they can consume the message.

Based on the requirements of Subject, here is the base Subject interface that defines the contract methods to be implemented by any concrete subject.

Subject.java

```
public interface Subject {
		//methods to register and unregister observers
		public void register(Observer obj);
		public void unregister(Observer obj);

		//method to notify observers of change
		public void notifyObservers();

		//method to get updates from subject
		public Object getUpdate(Observer obj);


}
```

Observer.java

```
public interface Observer {

	//method to update the observer, used by subject
	public void update();

	//attach with subject to observe
	public void setSubject(Subject sub);
}
```

Now our contract is ready, let's proceed with the concrete implementation of our topic.

MyTopic.java

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

public class MyTopic implements Subject {

	private List<Observer> observers;
	private String message;


	public MyTopic(){
		this.observers=new ArrayList<>();
	}
	@Override
	public void register(Observer obj) {
	observers.add(obj);
	}

	@Override
	public void unregister(Observer obj) {
		observers.remove(obj);
	}
	@Override
	public void notifyObservers() {
		List<Observer> observersLocal = null;
		for (Observer obj : observersLocal) {
			obj.update();
		}
	}

	@Override
	public Object getUpdate(Observer obj) {
		return this.message;
	}

	//method to post message to the topic
	public void postMessage(String msg){
		this.message=msg;
		notifyObservers();
	}

}
```

MyTopicSubscriber.java

```
public class MyTopicSubscriber implements Observer {

	private String name;
	private Subject topic;

	public MyTopicSubscriber(String nm){
		this.name=nm;
	}
	@Override
	public void update() {
		String msg = (String) topic.getUpdate(this);
		System.out.println(name+":: Consuming message::"+msg);
	}
```

```
          @Override
          public void setSubject(Subject sub) {
                    this.topic=sub;
          }

}
```

Notice the implementation of *update()* method where it’s calling Subject *getUpdate()* method to get the message to consume. We could have avoided this call by passing message as argument to *update()* method.

Testing

```
                    //create subject
                    MyTopic topic = new MyTopic();

                    //create observers
                    Observer obj1 = new MyTopicSubscriber("Obj1");
                    Observer obj2 = new MyTopicSubscriber("Obj2");
                    Observer obj3 = new MyTopicSubscriber("Obj3");

                    //register observers to the subject
                    topic.register(obj1);
                    topic.register(obj2);
                    topic.register(obj3);

                    //attach observer to subject
                    obj1.setSubject(topic);
                    obj2.setSubject(topic);
                    obj3.setSubject(topic);

                    //check if any update is available
                    obj1.update();

                    //now send message to subject
                    topic.postMessage("New Message");
```

Output

```
Obj1:: Consuming message::New Message
Obj2:: Consuming message::New Message
Obj3:: Consuming message::New Message
```

State Design Pattern

If we have to change the behaviour of an object based on its state, we can have a state variable in the Object and use **if-else** condition block to perform different actions based on the state.

Suppose we want to implement a TV Remote with a simple button to perform action, if the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

We can implement it using if-else condition like below

```
public class TVRemoteBasic {

	private String state="";

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

	public void doAction(){
		if(state.equalsIgnoreCase("ON")){
			System.out.println("TV is turned ON");
		}else if(state.equalsIgnoreCase("OFF")){
			System.out.println("TV is turned OFF");
		}
	}

	public static void main(String args[]){
		TVRemoteBasic remote = new TVRemoteBasic();

		remote.setState("ON");
		remote.doAction();

		remote.setState("OFF");
		remote.doAction();
	}

}
```

Notice that client code should know the specific values to use for setting the state of remote. Furthermore if number of states increase then the tight coupling between implementation and the client code will be very hard to maintain and extend. Now we will use State pattern to implement above TV Remote example.

State design pattern is used to provide a systematic and loosely coupled way to achieve this through Context and State implementations. State Pattern **Context** is the class that has a State reference to one of the concrete implementations of the State. Context forwards the request to the state object for processing.

State Design Pattern Interface

First of all we will create State interface that will define the method that should be implemented by different concrete states and context class. State.java

State.java

```
public interface State {

public void doAction();

}
```

State Design Pattern Concrete State Implementations
In our example, we can have two states - one for turning TV on and another to turn it off. So we will create two concrete state implementations for these behaviours. TVStartState.java

TVStartState.java

```
public class TVStartState implements State {

            @Override
            public void doAction() {
                        System.out.println("TV is turned ON");
            }

}
```

TVStopState.java

```
public class TVStopState implements State {
            @Override
            public void doAction() {
                        System.out.println("TV is turned OFF");
            }
}
```

Now we are ready to implement our Context object that will change its behavior based on its internal state.

State Design Pattern Context Implementation

TVContext.java

```
public class TVContext implements State {

		private State tvState;

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

		public State getState() {
				return this.tvState;
		}

		@Override
		public void doAction() {
				this.tvState.doAction();
		}
}
```

Notice that Context also implements State and keep a reference of its current state and forwards the request to the state implementation.

State Design Pattern Test Program

Now let’s write a simple program to test our state pattern implementation of TV Remote.

TVRemote.java

```
public class TVRemote {

	public static void main(String[] args) {
		TVContext context = new TVContext();
		State tvStartState = new TVStartState();
		State tvStopState = new TVStopState();

		context.setState(tvStartState);
		context.doAction();


		context.setState(tvStopState);
		context.doAction();

	}

}
```

State Pattern is very similar to Strategy Pattern.

Strategy Design Pattern

Strategy design pattern is one of the **behavioural design pattern**. Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

Strategy pattern is also known as Policy Pattern. We defines multiple algorithms and let client application pass the algorithm to be used as a parameter.

One of the best example of this pattern is **Collections.sort()** method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.

Strategy.java

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

OperationAdd.java

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

OperationSubtract.java

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

OperationMultiply.java

```java
public class OperationMultiply implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 * num2;
    }
}
```

Context.java

```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);
    }
}
```

StrategyApplication.java

```java
public class StrategyApplication {

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

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

}
```

Strategy Pattern is very similar to **State Pattern**. One of the difference is that Context contains state as instance variable and there can be multiple tasks whose implementation can be dependent on the state whereas in strategy pattern strategy is passed as argument to the method and context object doesn't have any variable to store it.

Template Design Pattern

Template Design Pattern is a **behavioural design pattern** and it's used to create a method stub and deferring some of the steps of implementation to the subclasses.

It defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses.

Let's understand this pattern with an example, suppose we want to provide an algorithm to build a house.

The steps need to be performed to build a house are –
Building foundation,
Building pillars,
Building walls and
Windows.

The important point is that we can't change the order of execution because we can't build windows before building the foundation.

So in this case we can create a template method that will use different methods to build the house. Now building the foundation for a house is same for all type of houses, whether it's a wooden house or a glass house.

So we can provide base implementation for this, if subclasses want to override this method, they can but mostly it's common for all the types of houses. **To make sure that subclasses don't override the template method, we should make it final**.

Example

Game.java

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

    //template method
    public final void play(){

        //initialize the game
        initialize();

        //start game
        startPlay();

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

Cricket.java

```java
public class Cricket extends Game {

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

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

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

FootBall.java

```java
public class Football extends Game {

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

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

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

TemplateApplication.java

```java
public class TemplateApplication {

    public static void main(String[] args) {

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

}
```

<u>Template Method Design Pattern in JDK</u>

- All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer.

Visitor Design Pattern

Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects.

With the help of visitor pattern, we can move the operational logic from the objects to another class.

For example, think of a Shopping cart where we can add different type of items (Elements). When we click on checkout button, it calculates the total amount to be paid.

Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern.

Let's implement this in our example of visitor pattern

Visitor Design Pattern Java Example
To implement visitor pattern, first of all we will create different type of items (Elements) to be used in shopping cart.

ItemElement.java

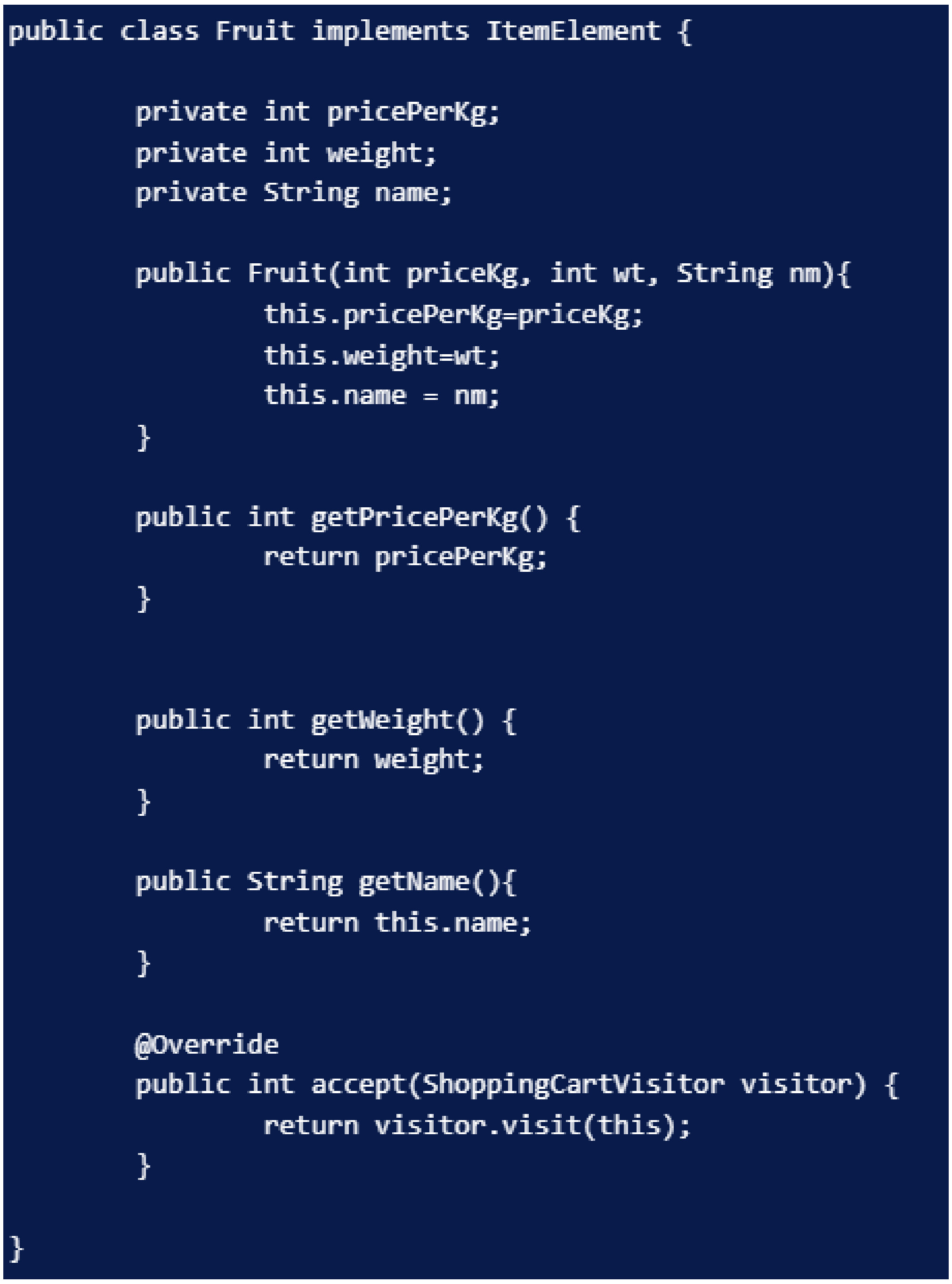
```
public interface ItemElement {

        public int accept(ShoppingCartVisitor visitor);
}
```

Let's create some concrete classes for different types of items.

Book.java

```
public class Book implements ItemElement {

        private int price;
        private String isbnNumber;

        public Book(int cost, String isbn){
                this.price=cost;
                this.isbnNumber=isbn;
        }

        public int getPrice() {
                return price;
        }

        public String getIsbnNumber() {
                return isbnNumber;
        }

        @Override
        public int accept(ShoppingCartVisitor visitor) {
                return visitor.visit(this);
        }

}
```

Fruit.java



```java
public class Fruit implements ItemElement {

        private int pricePerKg;
        private int weight;
        private String name;

        public Fruit(int priceKg, int wt, String nm){
                this.pricePerKg=priceKg;
                this.weight=wt;
                this.name = nm;
        }

        public int getPricePerKg() {
                return pricePerKg;
        }


        public int getWeight() {
                return weight;
        }

        public String getName(){
                return this.name;
        }

        @Override
        public int accept(ShoppingCartVisitor visitor) {
                return visitor.visit(this);
        }

}
```

Notice the implementation of accept() method in concrete classes, its calling visit() method of Visitor and passing itself as argument. We have visit() method for different type of items in Visitor interface that will be implemented by concrete visitor class.

ShoppingCartVisitor.java

```java
public interface ShoppingCartVisitor {

        int visit(Book book);
        int visit(Fruit fruit);
}
```

ShoppingCartVisitorImpl.java

```java
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {

        @Override
        public int visit(Book book) {
                int cost=0;
                //apply 5$ discount if book price is greater than 50
                if(book.getPrice() > 50){
                        cost = book.getPrice()-5;
                }else cost = book.getPrice();
                System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);
                return cost;
        }

        @Override
        public int visit(Fruit fruit) {
                int cost = fruit.getPricePerKg()*fruit.getWeight();
                System.out.println(fruit.getName() + " cost = "+cost);
                return cost;
        }

}
```

ShoppingCartClient.java

```java
public class ShoppingCartClient {

        public static void main(String[] args) {
                ItemElement[] items = new ItemElement[]{
                                        new Book(20, "1234"),
                                        new Book(100, "5678"),
                                        new Fruit(10, 2, "Banana"),
                                        new Fruit(5, 5, "Apple")};
```

```
		int total = calculatePrice(items);
		System.out.println("Total Cost = "+total);
	}

	private static int calculatePrice(ItemElement[] items) {
		ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
		int sum=0;
		for(ItemElement item : items){
			sum = sum + item.accept(visitor);
		}
		return sum;
	}

}
```

Visitor Pattern Benefits

The benefit of this pattern is that if the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes. Another benefit is that adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

Visitor Pattern Limitations

The drawback of visitor pattern is that we should know the return type of visit() methods at the time of designing otherwise we will have to change the interface and all of its implementations. Another drawback is that if there are too many implementations of visitor interface, it makes it hard to extend.

Memento Design Pattern

Memento pattern is one of the **behavioural design pattern**.

The memento design pattern is used when we want to save the state of an object so that we can restore it later on. This pattern is used to implement this in such a way that the saved state data of the object is not accessible outside of the Object, this protects the integrity of saved state data.

Memento pattern is implemented with two Objects – *originator* and *caretaker*. The originator is the Object whose state needs to be saved and restored, and it uses an inner class to save the state of Object. The inner class is called "Memento", and it's private so that it can't be accessed from other objects.

Caretaker is the helper class that is responsible for storing and restoring the Originator's state through Memento object. Since Memento is private to Originator, Caretaker can't access it and it's stored as an Object within the caretaker.

One of the best **real life example** is the text editors where we can save its data anytime and use undo to restore it to previous saved state. We will implement the same feature and provide a utility where we can write and save contents to a File anytime and we can restore it to last saved state.

Originator class

```
public class FileWriterUtil
{
private String fileName;
private StringBuilder content;

public FileWriterUtil(String file)
{
this.fileName=file;
this.content=new StringBuilder();
}

@Override
public String toString(){
return this.content.toString();
}

public void write(String str){
content.append(str);
}
public Memento save(){
return new Memento(this.fileName,this.content);
}

public void undoToLastSave(Object obj){
Memento memento = (Memento) obj;
this.fileName= memento.fileName;
this.content=memento.content;
}

private class Memento{
private String fileName;
private StringBuilder content;
public Memento(String file, StringBuilder content){
this.fileName=file;
//notice the deep copy so that Memento and FileWriterUtil content variables don't refer to
//same object
this.content=new StringBuilder(content);
}
}
}
```

Notice the Memento inner class and implementation of save and undo methods. Now we can continue to implement Caretaker class.

Caretaker class

```
public class FileWriterCaretaker {

private Object obj;

public void save(FileWriterUtil fileWriter){

this.obj=fileWriter.save();

}

public void undo(FileWriterUtil fileWriter){

fileWriter.undoToLastSave(obj);

}

}
```

Notice that caretaker object contains the saved state in the form of Object, so it can’t alter its data and also it has no knowledge of its structure.

Memento Test Class

```
public class FileWriterClient

{

public static void main(String[] args)

{

FileWriterCaretaker caretaker = new FileWriterCaretaker();

FileWriterUtil fileWriter = new FileWriterUtil("data.txt");

fileWriter.write("First Set of Data\n");

System.out.println(fileWriter+"\n\n");

// lets save the file

caretaker.save(fileWriter);

//now write something else

fileWriter.write("Second Set of Data\n");

//checking file contents

System.out.println(fileWriter+"\n\n");

//lets undo to last save

caretaker.undo(fileWriter);

//checking file content again

System.out.println(fileWriter+"\n\n");

}

}
```

DAO Design Pattern

The Data Access Object (DAO) design pattern is used to decouple the data persistence logic to a separate layer.

DAO is a very popular pattern when we design systems to work with databases.

The idea is to keep the service layer separate from the data access layer. This way we implement the separation of logic in our application.

Dependency Injection Pattern

The dependency injection pattern allows us to remove the hard-coded dependencies and make our application loosely-coupled, extendable, and maintainable.

We can implement dependency injection in Java **to move the dependency resolution from compile-time to runtime**.

Spring framework is built on the principle of dependency injection.

MVC Pattern

Model-View-Controller (MVC) Pattern is one of the oldest architectural patterns for creating web applications