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.

## **What is Gang of Four (GOF)**

**Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

According to these authors design patterns are primarily based on the following principles of object orientated design.

* Program to an interface not an implementation

List myList = new ArrayList(); instead of ArrayList myList = new ArrayList();

* Favor object composition over inheritance

In Inheritance, a new class, which wants to reuse code, inherit an existing class, known as super class.

On composition, a class, which desire to use functionality of an existing class, doesn't inherit, instead it holds a reference of that class in a member variable, that’s why the name composition. Inheritance and composition relationships are also referred as IS-A and HAS-A relationships.

## **Usage of Design Pattern**

Design Patterns have two main usages in software development.

### **Common platform for developers**

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

### **Best Practices**

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development.

## **Types of Design Patterns**

There are many design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. We'll also discuss another category of design pattern: J2EE design patterns.

|  |  |
| --- | --- |
| 1 | **Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case. |
| 2 | **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. |
| 3 | **Behavioral Patterns** These design patterns are specifically concerned with communication between objects. |
| 4 | **J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center. |

[Creational Design Patterns](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#creational-patterns)

=====================================================================

### Singleton Pattern

**Java Singleton Pattern** is one of the **Gangs of Four Design patterns** and comes in the **Creational**[**Design Pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) category. From the definition, it seems to be a very simple design pattern but when it comes to implementation, it comes with a lot of implementation concerns. The implementation of Java Singleton pattern has always been a controversial topic among developers

## Java Singleton

* Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.
* The singleton class must provide a global access point to get the instance of the class.
* Singleton pattern is used for [logging](https://www.journaldev.com/977/logger-in-java-logging-example), drivers objects, caching and [thread pool](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice).
* Singleton design pattern is also used in other design patterns like [Abstract Factory](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java), [Builder](https://www.journaldev.com/1425/builder-design-pattern-in-java), [Prototype](https://www.journaldev.com/1440/prototype-design-pattern-in-java), [Facade](https://www.journaldev.com/1557/facade-design-pattern-in-java) etc.
* Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

### Java Singleton Pattern

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

In further sections, we will learn different approaches of Singleton pattern implementation and design concerns with the implementation.

1. [Eager initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#eager-initialization)
2. [Static block initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#static-block-initialization)
3. [Lazy Initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#lazy-initialization)
4. [Thread Safe Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#thread-safe-singleton)
5. [Bill Pugh Singleton Implementation](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#bill-pugh-singleton)
6. [Using Reflection to destroy Singleton Pattern](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#reflection-and-singleton)
7. [Enum Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#enum-singleton)
8. [Serialization and Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#serialization-and-singleton)

### Eager initialization

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

Here is the implementation of static initialization singleton class.

package com.journaldev.singleton;

public class EagerInitializedSingleton {

private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();

//private constructor to avoid client applications to use constructor

private EagerInitializedSingleton(){}

public static EagerInitializedSingleton getInstance(){

return instance;

}

}

If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc and we should avoid the instantiation until unless client calls the getInstance method. Also this method doesn’t provide any options for exception handling.

### Static block initialization

[Static block](https://www.journaldev.com/1365/static-keyword-in-java) initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for [exception handling](https://www.journaldev.com/1696/exception-handling-in-java).

package com.journaldev.singleton;

public class StaticBlockSingleton {

private static StaticBlockSingleton instance;

private StaticBlockSingleton(){}

//static block initialization for exception handling

static{

try{

instance = new StaticBlockSingleton();

}catch(Exception e){

throw new RuntimeException("Exception occured in creating singleton instance");

}

}

public static StaticBlockSingleton getInstance(){

return instance;

}

}

Both eager initialization and static block initialization creates the instance even before it’s being used and that is not the best practice to use. So in further sections, we will learn how to create Singleton class that supports lazy initialization.

### Lazy Initialization

Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this approach.

package com.journaldev.singleton;

public class LazyInitializedSingleton {

private static LazyInitializedSingleton instance;

private LazyInitializedSingleton(){}

public static LazyInitializedSingleton getInstance(){

if(instance == null){

instance = new LazyInitializedSingleton();

}

return instance;

}

}

The above implementation works fine incase of single threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class. In next section, we will see different ways to create a [thread-safe](https://www.journaldev.com/1061/thread-safety-in-java) singleton class.

### Thread Safe Singleton

The easier way to create a thread-safe singleton class is to make the global access method [synchronized](https://www.journaldev.com/1061/thread-safety-in-java), so that only one thread can execute this method at a time. General implementation of this approach is like the below class.

package com.journaldev.singleton;

public class ThreadSafeSingleton {

private static ThreadSafeSingleton instance;

private ThreadSafeSingleton(){}

public static synchronized ThreadSafeSingleton getInstance(){

if(instance == null){

instance = new ThreadSafeSingleton();

}

return instance;

}

}

Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances (Read: [Java Synchronization](https://www.journaldev.com/1061/thread-safety-in-java)). To avoid this extra overhead every time, **double checked locking** principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of singleton class is created.

Below code snippet provides the double checked locking implementation.

public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){

if(instance == null){

synchronized (ThreadSafeSingleton.class) {

if(instance == null){

instance = new ThreadSafeSingleton();

}

}

}

return instance;

}

### Bill Pugh Singleton Implementation

Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using a [inner static helper class](https://www.journaldev.com/996/java-inner-class). The Bill Pugh Singleton implementation goes like this;

package com.journaldev.singleton;

public class BillPughSingleton {

private BillPughSingleton(){}

private static class SingletonHelper{

private static final BillPughSingleton INSTANCE = new BillPughSingleton();

}

public static BillPughSingleton getInstance(){

return SingletonHelper.INSTANCE;

}

}

Notice the **private inner static class** that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance method, this class gets loaded and creates the Singleton class instance.

This is the most widely used approach for Singleton class as it doesn’t require synchronization. I am using this approach in many of my projects and it’s easy to understand and implement also.

### Using Reflection to destroy Singleton Pattern

Reflection can be used to destroy all the above singleton implementation approaches. Let’s see this with an example class.

package com.journaldev.singleton;

import java.lang.reflect.Constructor;

public class ReflectionSingletonTest {

public static void main(String[] args) {

EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();

EagerInitializedSingleton instanceTwo = null;

try {

Constructor[] constructors = EagerInitializedSingleton.class.getDeclaredConstructors();

for (Constructor constructor : constructors) {

//Below code will destroy the singleton pattern

constructor.setAccessible(true);

instanceTwo = (EagerInitializedSingleton) constructor.newInstance();

break;

}

} catch (Exception e) {

e.printStackTrace();

}

System.out.println(instanceOne.hashCode());

System.out.println(instanceTwo.hashCode());

}

}

When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate, do check out [**Java Reflection Tutorial**](https://www.journaldev.com/1789/java-reflection-example-tutorial).

### Enum Singleton

To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program. Since [Java Enum](https://www.journaldev.com/716/java-enum) values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

package com.journaldev.singleton;

public enum EnumSingleton {

INSTANCE;

public static void doSomething(){

//do something

}

}

### Serialization and Singleton

Sometimes in distributed systems, we need to implement Serializable interface in Singleton class so that we can store it’s state in file system and retrieve it at later point of time. Here is a small singleton class that implements Serializable interface also.

package com.journaldev.singleton;

import java.io.Serializable;

public class SerializedSingleton implements Serializable{

private static final long serialVersionUID = -7604766932017737115L;

private SerializedSingleton(){}

private static class SingletonHelper{

private static final SerializedSingleton instance = new SerializedSingleton();

}

public static SerializedSingleton getInstance(){

return SingletonHelper.instance;

}

}

The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Let’s see it with a simple program.

package com.journaldev.singleton;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInput;

import java.io.ObjectInputStream;

import java.io.ObjectOutput;

import java.io.ObjectOutputStream;

public class SingletonSerializedTest {

public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {

SerializedSingleton instanceOne = SerializedSingleton.getInstance();

ObjectOutput out = new ObjectOutputStream(new FileOutputStream(

"filename.ser"));

out.writeObject(instanceOne);

out.close();

//deserailize from file to object

ObjectInput in = new ObjectInputStream(new FileInputStream(

"filename.ser"));

SerializedSingleton instanceTwo = (SerializedSingleton) in.readObject();

in.close();

System.out.println("instanceOne hashCode="+instanceOne.hashCode());

System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());

}

}

Output of the above program is;

instanceOne hashCode=2011117821

instanceTwo hashCode=109647522

So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.

protected Object readResolve() {

return getInstance();

}

After this you will notice that hashCode of both the instances are same in test program.

## Factory Design Pattern

[1 Factory Design Pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java#factory-design-pattern)

* [1.1 Factory Design Pattern Super Class](https://www.journaldev.com/1392/factory-design-pattern-in-java#factory-design-pattern-super-class)
* [1.2 Factory Design Pattern Sub Classes](https://www.journaldev.com/1392/factory-design-pattern-in-java#factory-design-pattern-sub-classes)
* [1.3 Factory Class](https://www.journaldev.com/1392/factory-design-pattern-in-java#factory-class)
* [1.4 Factory Design Pattern Advantages](https://www.journaldev.com/1392/factory-design-pattern-in-java#factory-design-pattern-advantages)
* [1.5 Factory Design Pattern Examples in JDK](https://www.journaldev.com/1392/factory-design-pattern-in-java#factory-design-pattern-examples-in-jdk)
* [1.6 Factory Design Pattern YouTube Video Tutorial](https://www.journaldev.com/1392/factory-design-pattern-in-java#factory-design-pattern-youtube-video-tutorial)

Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class.

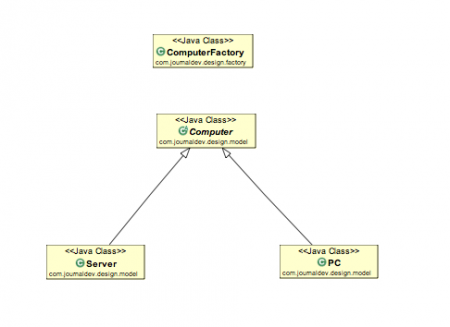
Let’s first learn how to implement factory design pattern in java and then we will look into factory pattern advantages. We will see some of factory design pattern usage in JDK. Note that this pattern is also known as **Factory Method Design Pattern**.

### Factory Design Pattern Super Class

Super class in factory design pattern can be an interface, [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java) or a normal java class. For our factory design pattern example, we have abstract super class with [overridden](https://www.journaldev.com/817/java-override-annotation) toString() method for testing purpose.

Some important points about Factory Design Pattern method are;

1. We can keep Factory class [Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples) or we can keep the method that returns the subclass as [static](https://www.journaldev.com/1365/static-keyword-in-java).
2. Notice that based on the input parameter, different subclass is created and returned. getComputer is the factory method.

[](https://cdn.journaldev.com/wp-content/uploads/2013/05/factory-pattern-java.png)

Here is a simple test client program that uses above factory design pattern implementation.

### Factory Design Pattern Advantages

1. Factory design pattern provides approach to code for interface rather than implementation.
2. Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
3. Factory pattern provides abstraction between implementation and client classes through inheritance.

### Factory Design Pattern Examples in JDK

1. java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern.
2. valueOf() method in wrapper classes like Boolean, Integer etc.

# Abstract Factory Design Pattern in Java

[1 Abstract Factory](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java#abstract-factory)

* [1.1 Abstract Factory Design Pattern Super Class and Subclasses](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java#abstract-factory-design-pattern-super-class-and-subclasses)
* [1.2 Factory Class for Each subclass](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java#factory-class-for-each-subclass)
* [1.3 Abstract Factory Design Pattern Banefits](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java#abstract-factory-design-pattern-banefits)
* [1.4 Abstract Factory Design Pattern Examples in JDK](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java#abstract-factory-design-pattern-examples-in-jdk)

If you are familiar with [**factory design pattern in java**](https://www.journaldev.com/1392/factory-design-pattern-in-java), you will notice that we have a single Factory class. This factory class returns different subclasses based on the input provided and factory class uses if-else or switch statement to achieve this.

In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class. Then an Abstract Factory class that will return the sub-class based on the input factory class. At first it seems confusing but once you see the implementation, its really easy to grasp and understand the minor difference between Factory and Abstract Factory pattern.

Notice that its a simple class and getComputer method is accepting ComputerAbstractFactory argument and returning Computer object. At this point the implementation must be getting clear.

### Abstract Factory Design Pattern Banefits

* Abstract Factory design pattern provides approach to code for interface rather than implementation.
* Abstract Factory pattern is “factory of factories” and can be easily extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory.
* Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.

### Abstract Factory Design Pattern Examples in JDK

* javax.xml.parsers.DocumentBuilderFactory#newInstance()
* javax.xml.transform.TransformerFactory#newInstance()
* javax.xml.xpath.XPathFactory#newInstance()

### Builder Pattern

* [1 Builder Design Pattern](https://www.journaldev.com/1425/builder-design-pattern-in-java#builder-design-pattern)
* [2 Builder Design Pattern in Java](https://www.journaldev.com/1425/builder-design-pattern-in-java#builder-design-pattern-in-java)
  + [2.1 Builder Design Pattern Video Tutorial](https://www.journaldev.com/1425/builder-design-pattern-in-java#builder-design-pattern-video-tutorial)
  + [2.2 Builder Design Pattern Example in JDK](https://www.journaldev.com/1425/builder-design-pattern-in-java#builder-design-pattern-example-in-jdk)

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](https://www.journaldev.com/1392/factory-design-pattern-in-java), 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.

## Builder Design Pattern in Java

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

1. First of all you need to create a [static nested class](https://www.journaldev.com/996/java-inner-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 Computerthen 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.

# Prototype Design Pattern in Java

Prototype design pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing.

Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype design pattern uses java cloning to copy the object.

### Prototype Design Pattern Example

It would be easy to understand prototype design pattern with an example. Suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it’s not a good idea to create the Object using new keyword and load all the data again from database.

The better approach would be to clone the existing object into a new object and then do the data manipulation.

Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and its a design decision.

[Structural Design Patterns](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#structural-patterns)

1. [Adapter Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#adapter-pattern)
2. [Composite Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#composite-pattern)
3. [Proxy Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#proxy-pattern)
4. [Flyweight Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#flyweight-pattern)
5. [Facade Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#facade-pattern)
6. [Bridge Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#bridge-pattern)
7. [Decorator Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#decorator-pattern)

## Structural Design Patterns

Structural patterns provide different ways to create a class structure, for example using inheritance and composition to create a large object from small objects.

# Adapter Design Pattern in Java

Adapter [design pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is one of the **structural design pattern** and its used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an **Adapter**.

[1 Adapter Design Pattern](https://www.journaldev.com/1487/adapter-design-pattern-java#adapter-design-pattern)

* [1.1 Two Way Adapter Pattern](https://www.journaldev.com/1487/adapter-design-pattern-java#two-way-adapter-pattern)
* [1.2 Adapter Design Pattern – Class Adapter](https://www.journaldev.com/1487/adapter-design-pattern-java#adapter-design-pattern-8211-class-adapter)
* [1.3 Adapter Design Pattern – Object Adapter Implementation](https://www.journaldev.com/1487/adapter-design-pattern-java#adapter-design-pattern-8211-object-adapter-implementation)
* [1.4 Adapter Design Pattern Class Diagram](https://www.journaldev.com/1487/adapter-design-pattern-java#adapter-design-pattern-class-diagram)
* [1.5 Adapter Design Pattern Example in JDK](https://www.journaldev.com/1487/adapter-design-pattern-java#adapter-design-pattern-example-in-jdk)

## Adapter Design Pattern

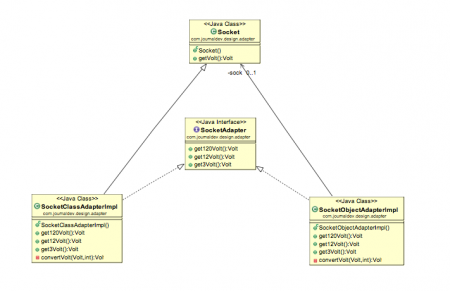
One of the great real life example of Adapter design pattern is mobile charger. Mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

### Two Way Adapter Pattern

While implementing Adapter pattern, there are two approaches – class adapter and object adapter – however both these approaches produce same result.

1. **Class Adapter** – This form uses [**java inheritance**](https://www.journaldev.com/644/inheritance-java-example) and extends the source interface, in our case Socket class.
2. **Object Adapter** – This form uses [**Java Composition**](https://www.journaldev.com/1325/composition-in-java-example) and adapter contains the source object.

### Adapter Design Pattern Class Diagram

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/adapter-pattern-java-class-diagram.png)

### Adapter Design Pattern Example in JDK

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)

# Composite Design Pattern in Java

[1 Composite Design Pattern](https://www.journaldev.com/1535/composite-design-pattern-in-java#composite-design-pattern)

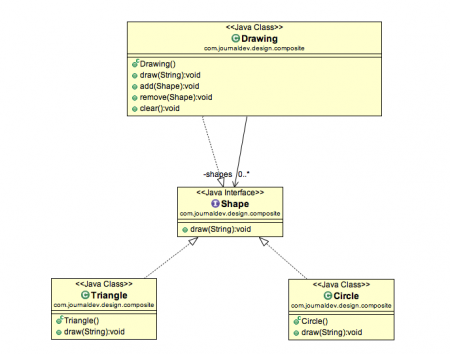
* [1.1 Composite Pattern Base Component](https://www.journaldev.com/1535/composite-design-pattern-in-java#composite-pattern-base-component)
* [1.2 Composite Design Pattern Leaf Objects](https://www.journaldev.com/1535/composite-design-pattern-in-java#composite-design-pattern-leaf-objects)
* [1.3 Composite object](https://www.journaldev.com/1535/composite-design-pattern-in-java#composite-object)
* [1.4 Composite Design Pattern Client Program](https://www.journaldev.com/1535/composite-design-pattern-in-java#composite-design-pattern-client-program)
* [1.5 Composite Pattern Important Points](https://www.journaldev.com/1535/composite-design-pattern-in-java#composite-pattern-important-points)

## Composite Design Pattern

When we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern.

Lets understand it with a real life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc. When we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations.

Composite Pattern consists of following objects.

1. **Base Component** – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java) with some methods common to all the objects.
2. **Leaf** – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn’t have references to other Components.
3. **Composite** – It consists of leaf elements and implements the operations in base component.
4. Notice that composite also implements component and behaves similar to leaf except that it can contain group of leaf elements.
5. [](https://cdn.journaldev.com/wp-content/uploads/2013/07/Composite-Pattern-java.png)

### Composite Pattern Important Points

* Composite pattern should be applied only when the group of objects should behave as the single object.
* Composite design pattern can be used to create a tree like structure.

java.awt.Container#add(Component) is a great example of Composite pattern in java and used a lot in

# Proxy Design Pattern

[1 Proxy Design Pattern](https://www.journaldev.com/1572/proxy-design-pattern#proxy-design-pattern)

* [1.1 Proxy Design Pattern – Main Class](https://www.journaldev.com/1572/proxy-design-pattern#proxy-design-pattern-8211-main-class)
* [1.2 Proxy Design Pattern – Proxy Class](https://www.journaldev.com/1572/proxy-design-pattern#proxy-design-pattern-8211-proxy-class)
* [1.3 Proxy Design Pattern Client Program](https://www.journaldev.com/1572/proxy-design-pattern#proxy-design-pattern-client-program)

**Provide a surrogate or placeholder for another object to control access to it.**

The definition itself is very clear and proxy design pattern is used when we want to provide controlled access of a functionality.

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.

Proxy design pattern common uses are to control access or to provide a wrapper implementation for better performance.

Java RMI package uses proxy pattern. That’s all for proxy design pattern in java.

# Flyweight Design Pattern in Java

[1 Flyweight Design Pattern](https://www.journaldev.com/1562/flyweight-design-pattern-java#flyweight-design-pattern)

* [1.1 Flyweight Design Pattern Interface and Concrete Classes](https://www.journaldev.com/1562/flyweight-design-pattern-java#flyweight-design-pattern-interface-and-concrete-classes)
* [1.2 Flyweight Factory](https://www.journaldev.com/1562/flyweight-design-pattern-java#flyweight-factory)
* [1.3 Flyweight Design Pattern Client Example](https://www.journaldev.com/1562/flyweight-design-pattern-java#flyweight-design-pattern-client-example)
* [1.4 Flyweight Design Pattern Example in JDK](https://www.journaldev.com/1562/flyweight-design-pattern-java#flyweight-design-pattern-example-in-jdk)
* [1.5 Flyweight Design Pattern Important Points](https://www.journaldev.com/1562/flyweight-design-pattern-java#flyweight-design-pattern-important-points)

## Flyweight Design Pattern

According to GoF, **flyweight design pattern** intent is:

Use sharing to support large numbers of fine-grained objects efficiently

Flyweight design pattern is a **Structural design pattern** like [Facade pattern](https://www.journaldev.com/1557/facade-design-pattern-in-java), [Adapter Pattern](https://www.journaldev.com/1487/adapter-design-pattern-java) and [Decorator pattern](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example).

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:

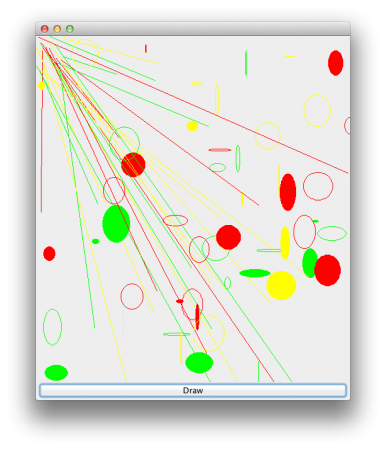
* The number of Objects to be created by application should be huge.
* The object creation is heavy on memory and it can be time consuming too.
* The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.

To apply flyweight pattern, we need to divide Object property into **intrinsic** and **extrinsic** properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations. For example, an Object Circle can have extrinsic properties such as color and width.

For applying flyweight pattern, we need to create a **Flyweight factory** that returns the shared objects. For our example, lets say we need to create a drawing with lines and Ovals. I have used [random number generation](https://www.journaldev.com/515/generate-random-number-in-java) to generate different type of Shapes in our frame.

If you run above client program, you will notice the delay in creating first Line Object and Oval objects with fill as true and false. After that the program executes quickly since its using the shared objects.

After clicking “Draw” button multiple times, the frame looks like below image.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/flyweight-pattern-example.png)

And you will see following output in command line confirming that Objects are shared.

Creating Line object

Creating Oval object with fill=true

Creating Oval object with fill=false

Thats all for flyweight pattern, we will look into more design patterns in future posts. If you liked it, please share your thoughts in comments section and share it with others too.

### Flyweight Design Pattern Example in JDK

All the [wrapper classes](https://www.journaldev.com/1002/wrapper-class-in-java) valueOf() method uses cached objects showing use of Flyweight design pattern. The best example is [Java String](https://www.journaldev.com/1321/java-string-interview-questions-and-answers) class [String Pool](https://www.journaldev.com/797/what-is-java-string-pool) implementation.

### Flyweight Design Pattern Important Points

1. In our example, the client code is not forced to create object using Flyweight factory but we can force that to make sure client code uses flyweight pattern implementation but its a complete design decision for particular application.
2. Flyweight pattern introduces complexity and if number of shared objects are huge then there is a trade of between memory and time, so we need to use it judiciously based on our requirements.
3. Flyweight pattern implementation is not useful when the number of intrinsic properties of Object is huge, making implementation of Factory class complex.

That’s all for Flyweight design pattern in java.

Oval class will have intrinsic property to determine whether to fill the Oval with given color or not whereas Line will not have any intrinsic property.

# Facade Design Pattern in Java

**Facade**[**Design Pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is one of the **Structural design patterns** (such as [Adapter pattern](https://www.journaldev.com/1487/adapter-design-pattern-java) and [Decorator pattern](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example)). Facade design pattern is used to help client applications to easily interact with the system.

[1 Facade Design Pattern](https://www.journaldev.com/1557/facade-design-pattern-in-java#facade-design-pattern)

* [1.1 Facade Design Pattern – Set of Interfaces](https://www.journaldev.com/1557/facade-design-pattern-in-java#facade-design-pattern-8211-set-of-interfaces)
* [1.2 Facade Design Pattern Interface](https://www.journaldev.com/1557/facade-design-pattern-in-java#facade-design-pattern-interface)
* [1.3 Facade Design Pattern Client Program](https://www.journaldev.com/1557/facade-design-pattern-in-java#facade-design-pattern-client-program)
* [1.4 Facade Design Pattern Important Points](https://www.journaldev.com/1557/facade-design-pattern-in-java#facade-design-pattern-important-points)

## Facade Design Pattern

According to GoF Facade design pattern is:

Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.

Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc.

So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports.

But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it.

So we can apply Facade design pattern here and provide a [wrapper](https://www.journaldev.com/1002/wrapper-class-in-java) interface on top of the existing interface to help client application.

As you can see that using Facade pattern interface is a lot easier and cleaner way to avoid having a lot of logic at client side. JDBC Driver Manager class to get the database connection is a wonderful example of facade design pattern.

### Facade Design Pattern Important Points

* Facade design pattern is more like a helper for client applications, it doesn’t hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.
* Facade design pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.
* Subsystem interfaces are not aware of Facade and they shouldn’t have any reference of the Facade interface.
* Facade design pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.
* We can use [Factory pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java) with Facade to provide better interface to client systems.

Thats all for Facade design pattern, stay tuned for more design pattern articles.

# Bridge Design Pattern in Java

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.

## Bridge Design Pattern

Just like [Adapter pattern](https://www.journaldev.com/1487/adapter-design-pattern-java), bridge design pattern is one of the **Structural design pattern**.

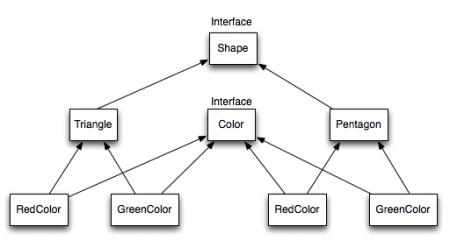
According to GoF bridge design pattern is:

Decouple an abstraction from its implementation so that the two can vary independently

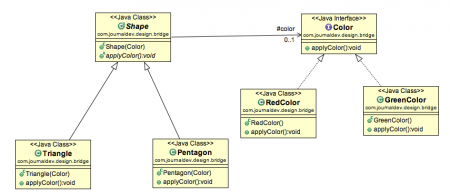
The implementation of bridge design pattern follows the notion to prefer [Composition](https://www.journaldev.com/1325/composition-in-java-example) over [inheritance](https://www.journaldev.com/644/inheritance-java-example).

### Bridge Design Pattern in Java Example

If we look into bridge design pattern with example, it will be easy to understand. Lets say we have an interface hierarchy in both interfaces and implementations like below image.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/Bridge-Interface-Hierarchy.png)

Now we will use bridge design pattern to decouple the interfaces from implementation. UML diagram for the classes and interfaces after applying bridge pattern will look like below image.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/bridge-design-pattern.png)

Notice the bridge between Shape and Color interfaces and use of composition in implementing the bridge pattern.

Here is the java code for Shape and Color interfaces

Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.

# Decorator Design Pattern in Java Example

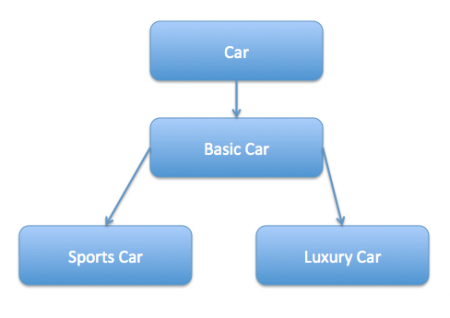
**Decorator**[**design pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. Decorator design pattern is one of the structural design pattern (such as [Adapter Pattern](https://www.journaldev.com/1487/adapter-design-pattern-java), [Bridge Pattern](https://www.journaldev.com/1491/bridge-design-pattern-java), [Composite Pattern](https://www.journaldev.com/1535/composite-design-pattern-in-java)) and uses abstract classes or interface with [composition](https://www.journaldev.com/1325/composition-in-java-example) to implement.

* [1 Decorator Design Pattern](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example#decorator-design-pattern)
* [2 Decorator Design Pattern – Class Diagram](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example#decorator-design-pattern-8211-class-diagram)
  + [2.1 Decorator Design Pattern Test Program](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example#decorator-design-pattern-test-program)
  + [2.2 Decorator Design Pattern – Important Points](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example#decorator-design-pattern-8211-important-points)

## Decorator Design Pattern

We use [inheritance](https://www.journaldev.com/644/inheritance-java-example) or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can’t add any new functionality of remove any existing behavior at runtime – this is when Decorator pattern comes into picture.

Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.

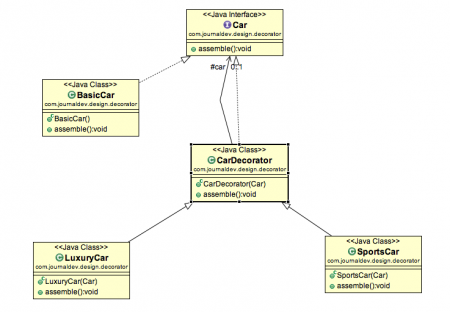
[](https://cdn.journaldev.com/wp-content/uploads/2013/07/inheritance-hierarchy.png)

But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now imagine if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern in java.

We need to have following types to implement decorator design pattern.

1. **Component Interface** – The interface or [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java) defining the methods that will be implemented. In our case Car will be the component interface.
2. package com.journaldev.design.decorator;
3. public interface Car {
4. public void assemble();
5. }
6. **Component Implementation** – The basic implementation of the component interface. We can have BasicCar class as our component implementation.
7. package com.journaldev.design.decorator;
8. public class BasicCar implements Car {
9. @Override
10. public void assemble() {
11. System.out.print("Basic Car.");
12. }
13. }
14. **Decorator** – Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.
15. package com.journaldev.design.decorator;
16. public class CarDecorator implements Car {
17. protected Car car;
19. public CarDecorator(Car c){
20. this.car=c;
21. }
23. @Override
24. public void assemble() {
25. this.car.assemble();
26. }
27. }
28. **Concrete Decorators** – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.
29. package com.journaldev.design.decorator;
30. public class SportsCar extends CarDecorator {
31. public SportsCar(Car c) {
32. super(c);
33. }
34. @Override
35. public void assemble(){
36. super.assemble();
37. System.out.print(" Adding features of Sports Car.");
38. }
39. }
40. package com.journaldev.design.decorator;
41. public class LuxuryCar extends CarDecorator {
42. public LuxuryCar(Car c) {
43. super(c);
44. }
46. @Override
47. public void assemble(){
48. super.assemble();
49. System.out.print(" Adding features of Luxury Car.");
50. }
51. }

## Decorator Design Pattern – Class Diagram

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/decorator-pattern.png)

[Behavioral Design Patterns](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#behavioral-patterns)

1. [Template Method Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#template-method-pattern)
2. [Mediator Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#mediator-pattern)
3. [Chain of Responsibility Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#chain-of-responsibility-pattern)
4. [Observer Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#observer-pattern)
5. [Strategy Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#strategy-pattern)
6. [Command Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#command-pattern)
7. [State Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#state-pattern)
8. [Visitor Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#visitor-pattern)
9. [Interpreter Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#interpreter-pattern)
10. [Iterator Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#iterator-pattern)
11. [Memento Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#memento-pattern)

Behavioral patterns provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

# Template Method Design Pattern in Java

Template Method is a **behavioral**[**design pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial). Template Method design pattern is used to create a method stub and deferring some of the steps of implementation to the subclasses.

[1 Template Method Design Pattern](https://www.journaldev.com/1763/template-method-design-pattern-in-java#template-method-design-pattern)

* [1.1 Template Method Abstract Class](https://www.journaldev.com/1763/template-method-design-pattern-in-java#template-method-abstract-class)
* [1.2 Template Method Concrete Classes](https://www.journaldev.com/1763/template-method-design-pattern-in-java#template-method-concrete-classes)
* [1.3 Template Method Design Pattern Client](https://www.journaldev.com/1763/template-method-design-pattern-in-java#template-method-design-pattern-client)
* [1.4 Template Method Class Diagram](https://www.journaldev.com/1763/template-method-design-pattern-in-java#template-method-class-diagram)
* [1.5 Template Method Design Pattern in JDK](https://www.journaldev.com/1763/template-method-design-pattern-in-java#template-method-design-pattern-in-jdk)
* [1.6 Template Method Design Pattern Important Points](https://www.journaldev.com/1763/template-method-design-pattern-in-java#template-method-design-pattern-important-points)

## Template Method Design Pattern

Template method 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 the 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 its 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.

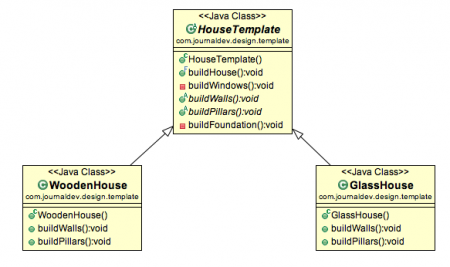
To make sure that subclasses don’t override the template method, we should make it final.

### Template Method Abstract Class

Since we want some of the methods to be implemented by subclasses, we have to make our base class as [abstract class](https://www.journaldev.com/1582/abstract-class-in-java).

Notice that client is invoking the template method of base class and depending of implementation of different steps, it’s using some of the methods from base class and some of them from subclass.

### Template Method Class Diagram

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/template-method-pattern.png)

### Template Method Design Pattern in JDK

* All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer.
* All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.

### Template Method Design Pattern Important Points

1. Template method should consists of certain steps whose order is fixed and for some of the methods, implementation differs from base class to subclass. Template method should be final.
2. Most of the times, subclasses calls methods from super class but in template pattern, superclass template method calls methods from subclasses, this is known as [Hollywood Principle](http://en.wikipedia.org/wiki/Hollywood_principle) – “don’t call us, we’ll call you.”.
3. Methods in base class with default implementation are referred as **Hooks** and they are intended to be overridden by subclasses, if you want some of the methods to be not overridden, you can make them final, for example in our case we can make buildFoundation() method final because if we don’t want subclasses to override it.

# Mediator Design Pattern in Java

Mediator [design pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is one of the behavioral design pattern, so it deals with the behaviors of objects. Mediator design pattern is used to provide a centralized communication medium between different objects in a system.

[1 Mediator Design Pattern](https://www.journaldev.com/1730/mediator-design-pattern-java#mediator-design-pattern)

* [1.1 Mediator Pattern Interface](https://www.journaldev.com/1730/mediator-design-pattern-java#mediator-pattern-interface)
* [1.2 Mediator Pattern Colleague Interface](https://www.journaldev.com/1730/mediator-design-pattern-java#mediator-pattern-colleague-interface)
* [1.3 Concrete Mediator](https://www.journaldev.com/1730/mediator-design-pattern-java#concrete-mediator)
* [1.4 Mediator Design Pattern Concrete Colleague](https://www.journaldev.com/1730/mediator-design-pattern-java#mediator-design-pattern-concrete-colleague)
* [1.5 Mediator Pattern Example Client Program Code](https://www.journaldev.com/1730/mediator-design-pattern-java#mediator-pattern-example-client-program-code)
* [1.6 Mediator Pattern Class Diagram](https://www.journaldev.com/1730/mediator-design-pattern-java#mediator-pattern-class-diagram)
* [1.7 Mediator Pattern Example in JDK](https://www.journaldev.com/1730/mediator-design-pattern-java#mediator-pattern-example-in-jdk)
* [1.8 Mediator Design Pattern Important Points](https://www.journaldev.com/1730/mediator-design-pattern-java#mediator-design-pattern-important-points)

## Mediator Design Pattern

According to GoF, mediator pattern intent is:

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

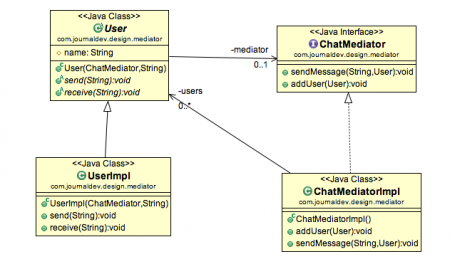
Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend. Mediator 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 it’s 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](https://www.journaldev.com/1607/difference-between-abstract-class-and-interface-in-java) 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 it’s 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 Class Diagram

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/mediator-pattern.png)

### Mediator Pattern Example in JDK

* [java.util.Timer](https://www.journaldev.com/1050/java-timer-timertask-example) class scheduleXXX() methods
* [Java Concurrency Executor](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice) 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**](https://www.journaldev.com/1739/observer-design-pattern-in-java) 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.

# Chain of Responsibility Design Pattern in Java

[1 Chain of Responsibility Design Pattern](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibility-design-pattern)

* [1.1 Chain of Responsibility Pattern Example in JDK](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibility-pattern-example-in-jdk)
* [1.2 Chain of Responsibility Design Pattern Example](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibility-design-pattern-example)
* [1.3 Chain of Responsibility Design Pattern – Base Classes and Interface](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibility-design-pattern-8211-base-classes-and-interface)
* [1.4 Chain of Responsibilities Pattern – Chain Implementations](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibilities-pattern-8211-chain-implementations)
* [1.5 Chain of Responsibilities Design Pattern – Creating the Chain](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibilities-design-pattern-8211-creating-the-chain)
* [1.6 Chain of Responsibilities Design Pattern Class Diagram](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibilities-design-pattern-class-diagram)
* [1.7 Chain of Responsibility Design Pattern Important Points](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibility-design-pattern-important-points)
* [1.8 Chain of Responsibility Pattern Examples in JDK](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java#chain-of-responsibility-pattern-examples-in-jdk)

## Chain of Responsibility Design Pattern

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](https://www.journaldev.com/592/java-try-with-resources) code. Here every catch block is kind of a processor to process that particular exception.

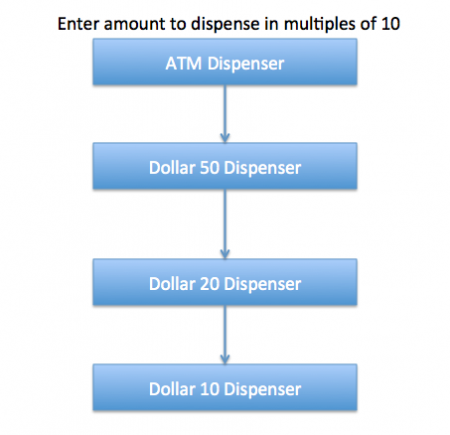
So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the 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.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/Chain-of-Responsibility-Pattern.png)

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$.

v

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.