**Java Design Patterns**

Some of the benefits of using design patterns are:

1. Design Patterns are already defined and provides **industry standard approach** to solve a recurring problem, so it saves time if we sensibly use the design pattern. There are many java design patterns that we can use in our java based projects.
2. Using design patterns promotes **reusability** that leads to more **robust** and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

**Java Design Patterns** are divided into three categories – **creational**, **structural**, and **behavioral** design patterns. This post serves as an index for all the java design patterns articles I have written so far.

* [Creational Design Patterns](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#creational-patterns)
  1. [Singleton Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#singleton-pattern)
  2. [Factory Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#factory-pattern)
  3. [Abstract Factory Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#abstract-factory-pattern)
  4. [Builder Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#builder-pattern)
  5. [Prototype Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#prototype-pattern)
* [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)
* [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)

# Creational Design Patterns

Creational design patterns provide solution to instantiate a object in the best possible way for specific situations.

1. **[Singleton Pattern](#_Java_Singleton_Design)**

[Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples) pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine. 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 Singleton pattern has always been a controversial topic among developers. Check out [**Singleton Design Pattern**](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples)to learn about different ways to implement Singleton pattern and pros and cons of each of the method. This is one of the most discussed java design patterns.

1. **[Factory Pattern](#_Factory_Design_Pattern)**

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. We can apply Singleton pattern on Factory class or make the factory method static. Check out [**Factory Design Pattern**](https://www.journaldev.com/1392/factory-design-pattern-in-java) for example program and [factory pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java) benefits. This is one of the most widely used java design pattern.

1. **[Abstract Factory Pattern](#_Abstract_Factory_Design)**

Abstract Factory pattern is similar to Factory pattern and it’s factory of factories. If you are familiar with factory design pattern in java, you will notice that we have a single Factory class that returns the different sub-classes 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 and then an Abstract Factory class that will return the sub-class based on the input factory class. Check out [**Abstract Factory Pattern**](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java) to know how to implement this pattern with example program.

1. **[Builder Pattern](#_Builder_Design_Pattern)**

This pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes. 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. Check out [**Builder Pattern**](https://www.journaldev.com/1425/builder-design-pattern-in-java) for example program and classes used in JDK.

1. **[Prototype Pattern](#_Prototype_Design_Pattern)**

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

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. Check out [**Prototype Pattern**](https://www.journaldev.com/1440/prototype-design-pattern-in-java) for sample program.

**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.

1. **Adapter Pattern**

Adapter design pattern 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. As a real life example, we can think of a mobile charger as an adapter because 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. Check out [**Adapter Pattern**](https://www.journaldev.com/1487/adapter-design-pattern-java)for example program and it’s usage in Java.

1. **Composite Pattern**

Composite pattern is one of the Structural design pattern and is used when we have to represent a part-whole hierarchy. 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 and 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. Check out [**Composite Pattern**](https://www.journaldev.com/1535/composite-design-pattern-in-java) article for different component of composite pattern and example program.

1. **Proxy Pattern**

Proxy pattern intent is to “Provide a surrogate or placeholder for another object to control access to it”. The definition itself is very clear and proxy 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. Check out [**Proxy Pattern**](https://www.journaldev.com/1572/proxy-design-pattern) post for the example program with implementation details.

1. **Flyweight Pattern**

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. String Pool implementation in java is one of the best example of Flyweight pattern implementation. Check out [**Flyweight Pattern**](https://www.journaldev.com/1562/flyweight-design-pattern-java) article for sample program and implementation process.

1. **Facade Pattern**

Facade Pattern is used to help client applications to easily interact with the system. 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 pattern here and provide a wrapper interface on top of the existing interface to help client application. Check out [**Facade Pattern**](https://www.journaldev.com/1557/facade-design-pattern-in-java) post for implementation details and sample program.

1. **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. Like Adapter pattern, its one of the Structural design pattern.

The implementation of bridge design pattern follows the notion to prefer Composition over inheritance. Check out [Bridge Pattern](https://www.journaldev.com/1491/bridge-design-pattern-java) post for implementation details and sample program.

1. **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 behavior. Decorator design pattern is one of the structural design pattern (such as Adapter Pattern, Bridge Pattern, Composite Pattern) and uses abstract classes or interface with composition to implement.

We use inheritance 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. Check out [**Decorator Pattern**](https://www.journaldev.com/1540/decorator-design-pattern-in-java-example) post for sample program and implementation details.

**Behavioral Design Patterns**

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

1. **Template Method Pattern**

Template Method is a behavioral design pattern and it’s used to create a method stub and deferring some of the steps of implementation to the subclasses. 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.

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. Check out [**Template Method Pattern**](https://www.journaldev.com/1763/template-method-design-pattern-in-java) post for implementation details with example program.

1. **Mediator Pattern**

Mediator design pattern is used to provide a centralized communication medium between different objects in a system. 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 maintainability cost higher and not flexible to extend easily. 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. Check out [**Mediator Pattern**](https://www.journaldev.com/1730/mediator-design-pattern-java) post for implementation details with example program.

1. **Chain of Responsibility Pattern**

Chain of responsibility pattern is used to achieve lose 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.

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.

ATM dispense machine logic can be implemented using [**Chain of Responsibility Pattern**](https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java), check out the linked post.

1. **Observer 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**.

Java provides inbuilt platform for implementing Observer pattern through java.util.Observable class and java.util.Observer interface. However it’s not widely used because the implementation is really simple and most of the times we don’t want to end up extending a class just for implementing Observer pattern as java doesn’t provide multiple inheritance in classes.

Java Message Service (JMS) uses Observer pattern along with Mediator pattern to allow applications to subscribe and publish data to other applications. Check out [**Observer Pattern**](https://www.journaldev.com/1739/observer-design-pattern-in-java) post for implementation details and example program.

1. **Strategy 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.

Check out [**Strategy Pattern**](https://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial) post for implementation details and example program.

1. **Command Pattern**

Command Pattern is used to implement lose coupling in a request-response model. 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.

Let’s say we want to provide a File System utility with methods to open, write and close file and it 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 java interfaces, we can have FileSystemReceiver interface and it’s implementation classes for different operating system flavors such as Windows, Unix, Solaris etc. Check out [**Command Pattern**](https://www.journaldev.com/1624/command-design-pattern) post for the implementation details with example program.

1. **State Pattern**

State design pattern is used when an Object change it’s behavior based on it’s internal state.

If we have to change the behavior of an object based on it’s state, we can have a state variable in the Object and use if-else condition block to perform different actions based on the state. State pattern is used to provide a systematic and lose-coupled way to achieve this through Context and State implementations.

Check out [**State Pattern**](https://www.journaldev.com/1751/state-design-pattern-java) post for implementation details with example program.

1. **Visitor 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. Check out [**Visitor Pattern**](https://www.journaldev.com/1769/visitor-design-pattern-java) post for implementation details.

1. **Interpreter Pattern**

is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

The best example of this pattern is java compiler that interprets the java source code into byte code that is understandable by JVM. Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.

Check out [**Interpreter Pattern**](https://www.journaldev.com/1635/interpreter-design-pattern-java) post for example program.

1. **Iterator Pattern**

Iterator pattern in one of the behavioral pattern and it’s used to provide a standard way to traverse through a group of Objects. Iterator pattern is widely used in [Java Collection Framework](https://www.journaldev.com/1260/collections-in-java-tutorial) where Iterator interface provides methods for traversing through a collection.

Iterator pattern is not only about traversing through a collection, we can provide different kind of iterators based on our requirements. Iterator pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods. Check out [**Iterator Pattern**](https://www.journaldev.com/1716/iterator-design-pattern-java) post for example program and implementation details.

1. **Memento Pattern**

Memento design pattern is used when we want to save the state of an object so that we can restore later on. Memento 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. 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 its private, so that it can’t be accessed from other objects.

Check out [**Memento Pattern**](https://www.journaldev.com/1734/memento-design-pattern-java) for sample program and implementation details.

# [Java Singleton Design Pattern Best Practices with Examples](#_Creational_Design_Patterns)

**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. Here we will learn about Singleton design pattern principles, different ways to implement Singleton design pattern and some of the best practices for it’s usage.

**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.

**Read**: [Java static](https://www.journaldev.com/1365/static-keyword-in-java)

**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;

}

**Read**: [Thread Safe Singleton Class](https://www.journaldev.com/171/thread-safety-in-java-singleton-classes-with-example-code)

**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.

**Read**: [Java Nested Classes](https://www.journaldev.com/996/java-inner-class)

**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

}

}

**Read**: [Java Enum](https://www.journaldev.com/716/java-enum)

**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 in Java](#creational-patterns)

Welcome to the Factory [Design Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) in [Java tutorial](https://www.journaldev.com/7153/core-java-tutorial). **Factory Pattern** is one of the **Creational Design pattern** and it’s widely used in JDK as well as frameworks like Spring and Struts.

**Table of Contents**[[hide](https://www.journaldev.com/1392/factory-design-pattern-in-java)]

* [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)

**Factory Design Pattern**

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.

package com.journaldev.design.model;

public abstract class Computer {

public abstract String getRAM();

public abstract String getHDD();

public abstract String getCPU();

@Override

public String toString(){

return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

}

}

**Factory Design Pattern Sub Classes**

Let’s say we have two sub-classes PC and Server with below implementation.

package com.journaldev.design.model;

public class PC extends Computer {

private String ram;

private String hdd;

private String cpu;

public PC(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

Notice that both the classes are extending Computer super class.

package com.journaldev.design.model;

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

**Factory Class**

Now that we have super classes and sub-classes ready, we can write our factory class. Here is the basic implementation.

package com.journaldev.design.factory;

import com.journaldev.design.model.Computer;

import com.journaldev.design.model.PC;

import com.journaldev.design.model.Server;

public class ComputerFactory {

public static Computer getComputer(String type, String ram, String hdd, String cpu){

if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);

else if("Server".equalsIgnoreCase(type)) return new Server(ram, hdd, cpu);

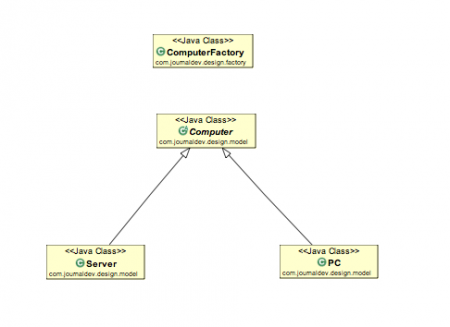
return null;

}

}

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://www.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.

package com.journaldev.design.test;

import com.journaldev.design.factory.ComputerFactory;

import com.journaldev.design.model.Computer;

public class TestFactory {

public static void main(String[] args) {

Computer pc = ComputerFactory.getComputer("pc","2 GB","500 GB","2.4 GHz");

Computer server = ComputerFactory.getComputer("server","16 GB","1 TB","2.9 GHz");

System.out.println("Factory PC Config::"+pc);

System.out.println("Factory Server Config::"+server);

}

}

Output of above program is:

Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

**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](#creational-patterns)

Welcome to Abstract Factory [Design Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) in java example. Abstract Factory design pattern is one of the Creational pattern. Abstract [Factory pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java) is almost similar to [**Factory Pattern**](https://www.journaldev.com/1392/factory-design-pattern-in-java) except the fact that its more like factory of factories.

**Table of Contents**[[hide](https://www.journaldev.com/1418/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)

**Abstract Factory**

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.

Like our factory pattern post, we will use the same super class and sub-classes.

**Abstract Factory Design Pattern Super Class and Subclasses**

Computer.java

package com.journaldev.design.model;

public abstract class Computer {

public abstract String getRAM();

public abstract String getHDD();

public abstract String getCPU();

@Override

public String toString(){

return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

}

}

PC.java

package com.journaldev.design.model;

public class PC extends Computer {

private String ram;

private String hdd;

private String cpu;

public PC(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

Server.java

package com.journaldev.design.model;

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

**Factory Class for Each subclass**

First of all we need to create a Abstract Factory interface or [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java).

ComputerAbstractFactory.java

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public interface ComputerAbstractFactory {

public Computer createComputer();

}

Notice that createComputer() method is returning an instance of super class Computer. Now our factory classes will implement this interface and return their respective sub-class.

PCFactory.java

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

import com.journaldev.design.model.PC;

public class PCFactory implements ComputerAbstractFactory {

private String ram;

private String hdd;

private String cpu;

public PCFactory(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public Computer createComputer() {

return new PC(ram,hdd,cpu);

}

}

Similarly we will have a factory class for Server subclass.

ServerFactory.java

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

import com.journaldev.design.model.Server;

public class ServerFactory implements ComputerAbstractFactory {

private String ram;

private String hdd;

private String cpu;

public ServerFactory(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public Computer createComputer() {

return new Server(ram,hdd,cpu);

}

}

Now we will create a consumer class that will provide the entry point for the client classes to create sub-classes.

ComputerFactory.java

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public class ComputerFactory {

public static Computer getComputer(ComputerAbstractFactory factory){

return factory.createComputer();

}

}

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.

Lets write a simple test method and see how to use the abstract factory to get the instance of sub-classes.

TestDesignPatterns.java

package com.journaldev.design.test;

import com.journaldev.design.abstractfactory.PCFactory;

import com.journaldev.design.abstractfactory.ServerFactory;

import com.journaldev.design.factory.ComputerFactory;

import com.journaldev.design.model.Computer;

public class TestDesignPatterns {

public static void main(String[] args) {

testAbstractFactory();

}

private static void testAbstractFactory() {

Computer pc = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new PCFactory("2 GB","500 GB","2.4 GHz"));

Computer server = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new ServerFactory("16 GB","1 TB","2.9 GHz"));

System.out.println("AbstractFactory PC Config::"+pc);

System.out.println("AbstractFactory Server Config::"+server);

}

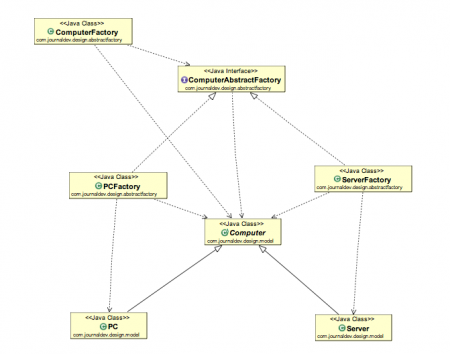
}

Output of the above program will be:

AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

AbstractFactory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

Here is the class diagram of abstract factory design pattern implementation.

[](https://www.journaldev.com/wp-content/uploads/2013/06/Abstract-Factory-Pattern.png)

**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 Design Pattern in Java](#creational-patterns)

Today we will look into Builder pattern in java. Builder [design pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is a **creational design pattern** like [**Factory Pattern**](https://www.journaldev.com/1392/factory-design-pattern-in-java) and [**Abstract Factory Pattern**](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java).

**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](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.

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

package com.journaldev.design.builder;

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.

package com.journaldev.design.test;

import com.journaldev.design.builder.Computer;

public class TestBuilderPattern {

public static void main(String[] args) {

//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", "2 GB").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)

# [Prototype Design Pattern in Java](#creational-patterns)

Prototype [design pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is one of the Creational Design pattern, so it provides a mechanism of object creation.

**Prototype Design Pattern**

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.

Here is a sample program showing Prototype design pattern example in java.

Employees.java

package com.journaldev.design.prototype;

import java.util.ArrayList;

import java.util.List;

public class Employees implements Cloneable{

private List<String> empList;

public Employees(){

empList = new ArrayList<String>();

}

public Employees(List<String> list){

this.empList=list;

}

public void loadData(){

//read all employees from database and put into the list

empList.add("Pankaj");

empList.add("Raj");

empList.add("David");

empList.add("Lisa");

}

public List<String> getEmpList() {

return empList;

}

@Override

public Object clone() throws CloneNotSupportedException{

List<String> temp = new ArrayList<String>();

for(String s : this.getEmpList()){

temp.add(s);

}

return new Employees(temp);

}

}

Notice that the clone method is overridden to provide a deep copy of the employees list.

Here is the prototype design pattern example test program that will show the benefit of prototype pattern.

PrototypePatternTest.java

package com.journaldev.design.test;

import java.util.List;

import com.journaldev.design.prototype.Employees;

public class PrototypePatternTest {

public static void main(String[] args) throws CloneNotSupportedException {

Employees emps = new Employees();

emps.loadData();

//Use the clone method to get the Employee object

Employees empsNew = (Employees) emps.clone();

Employees empsNew1 = (Employees) emps.clone();

List<String> list = empsNew.getEmpList();

list.add("John");

List<String> list1 = empsNew1.getEmpList();

list1.remove("Pankaj");

System.out.println("emps List: "+emps.getEmpList());

System.out.println("empsNew List: "+list);

System.out.println("empsNew1 List: "+list1);

}

}

Output of the above prototype design pattern example program is:

emps HashMap: [Pankaj, Raj, David, Lisa]

empsNew HashMap: [Pankaj, Raj, David, Lisa, John]

empsNew1 HashMap: [Raj, David, Lisa]

If the object cloning was not provided, we will have to make database call to fetch the employee list every time. Then do the manipulations that would have been resource and time consuming.

# 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**.

**Table of Contents**[[hide](https://www.journaldev.com/1487/adapter-design-pattern-java)]

* [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.

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;

}

}

package com.journaldev.design.adapter;

public class Socket {

public Volt getVolt(){

return new Volt(120);

}

}

Now we want to build an adapter that can produce 3 volts, 12 volts and default 120 volts. So first of all we will create an adapter interface with these methods.

package com.journaldev.design.adapter;

public interface SocketAdapter {

public Volt get120Volt();

public Volt get12Volt();

public Volt get3Volt();

}

### 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 Adapter

Here is the **class adapter** approach implementation of our adapter.

package com.journaldev.design.adapter;

//Using inheritance for adapter pattern

public class SocketClassAdapterImpl extends Socket implements SocketAdapter{

@Override

public Volt get120Volt() {

return getVolt();

}

@Override

public Volt get12Volt() {

Volt v= getVolt();

return convertVolt(v,10);

}

@Override

public Volt get3Volt() {

Volt v= getVolt();

return convertVolt(v,40);

}

private Volt convertVolt(Volt v, int i) {

return new Volt(v.getVolts()/i);

}

}

### Adapter Design Pattern – Object Adapter Implementation

Here is the **Object adapter** implementation of our adapter.

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);

}

}

Notice that both the adapter implementations are almost same and they implement the SocketAdapterinterface. The adapter interface can also be an [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java).

Here is a test program to consume our adapter design pattern implementation.

package com.journaldev.design.test;

import com.journaldev.design.adapter.SocketAdapter;

import com.journaldev.design.adapter.SocketClassAdapterImpl;

import com.journaldev.design.adapter.SocketObjectAdapterImpl;

import com.journaldev.design.adapter.Volt;

public class AdapterPatternTest {

public static void main(String[] args) {

testClassAdapter();

testObjectAdapter();

}

private static void testObjectAdapter() {

SocketAdapter sockAdapter = new SocketObjectAdapterImpl();

Volt v3 = getVolt(sockAdapter,3);

Volt v12 = getVolt(sockAdapter,12);

Volt v120 = getVolt(sockAdapter,120);

System.out.println("v3 volts using Object Adapter="+v3.getVolts());

System.out.println(" v12 volts using Object Adapter="+v12.getVolts());

System.out.println(" v120 volts using Object Adapter="+v120.getVolts());

}

private static void testClassAdapter() {

SocketAdapter sockAdapter = new SocketClassAdapterImpl();

Volt v3 = getVolt(sockAdapter,3);

Volt v12 = getVolt(sockAdapter,12);

Volt v120 = getVolt(sockAdapter,120);

System.out.println(" v3 volts using Class Adapter="+v3.getVolts());

System.out.println(" v12 volts using Class Adapter="+v12.getVolts());

System.out.println(" v120 volts using Class Adapter="+v120.getVolts());

}

private static Volt getVolt(SocketAdapter sockAdapter, int i) {

switch (i){

case 3: return sockAdapter.get3Volt();

case 12: return sockAdapter.get12Volt();

case 120: return sockAdapter.get120Volt();

default: return sockAdapter.get120Volt();

}

}

}

When we run above test program, we get following output.

v3 volts using Class Adapter=3

v12 volts using Class Adapter=12

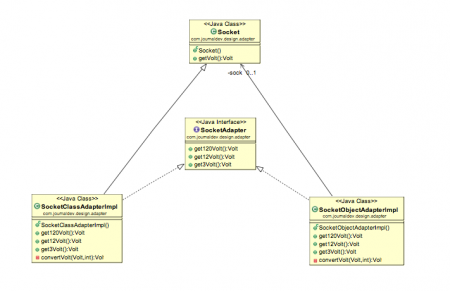
v120 volts using Class Adapter=120

v3 volts using Object Adapter=3

v12 volts using Object Adapter=12

v120 volts using Object Adapter=120

### Adapter Design Pattern Class Diagram

[](https://www.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)