Java Design Patterns

Some of the benefits of using design patterns are:

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.

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.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

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:

Singleton Pattern

Factory Pattern

Abstract Factory Pattern

Builder Pattern

Prototype Pattern

**Singleton Pattern**

1. Singleton 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 the Singleton pattern has always been a controversial topic among developers.

Check out Singleton Design Pattern 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.

**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, drivers objects, caching and thread pool.

Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc.

Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

**Java Singleton Pattern**

To implement a Singleton pattern, we have different approaches but all of them have the 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.

Eager initialization

Static block initialization

Lazy Initialization

Thread Safe Singleton

Bill Pugh Singleton Implementation

Using Reflection to destroy Singleton Pattern

Enum Singleton

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 the 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. 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 a 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 in case of the single-threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if condition at the same time. It will destroy the singleton pattern and both threads will get the different instances of the 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 the 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 a 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 the 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 an [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 is 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 its state in the file system and retrieve it at a 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 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());

}

}

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 is same in the test program.

### Factory 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.

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 or a normal java class. For our factory design pattern example, we have abstract super class with overridden 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.a

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;

We can keep Factory class Singleton or we can keep the method that returns the subclass as static.

Notice that based on the input parameter, different subclass is created and returned. getComputer is the factory method.

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

Copy

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);

}

}

Factory Design Pattern Advantages

Factory design pattern provides approach to code for interface rather than implementation.

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.

Factory pattern provides abstraction between implementation and client classes through inheritance.

Factory Design Pattern Examples in JDK

java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern.

valueOf() method in wrapper classes like Boolean, Integer etc.

Abstract Factory Pattern

Abstract Factory pattern is similar to Factory pattern and it’s a factory of factories. If you are familiar with the 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 to know how to implement this pattern with example program.

If you are familiar with 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 the 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, it’s 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 superclass 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.

Let’s 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);

}

}

**Abstract Factory Design Pattern Benefits**

* 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

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.

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.

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.

Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.

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.

YOU ARE HERE: [HOME](https://www.journaldev.com/) » [JAVA](https://www.journaldev.com/java) » [DESIGN PATTERNS](https://www.journaldev.com/java/design-patterns) » BUILDER DESIGN PATTERN IN JAVA

# Builder Design Pattern in Java

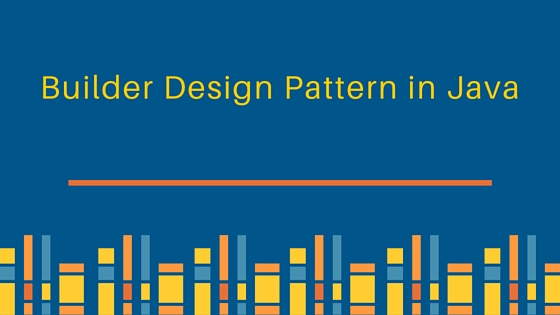
[PANKAJ](https://www.journaldev.com/author/pankaj) [39 COMMENTS](https://www.journaldev.com/1425/builder-design-pattern-in-java#comments)

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).

**Table of Contents**[[hide](https://www.journaldev.com/1425/builder-design-pattern-in-java)]

* [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 Design Pattern

[](https://cdn.journaldev.com/wp-content/uploads/2013/06/builder-design-pattern.jpg)

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();

}

}