### What is Design Pattern in Java?

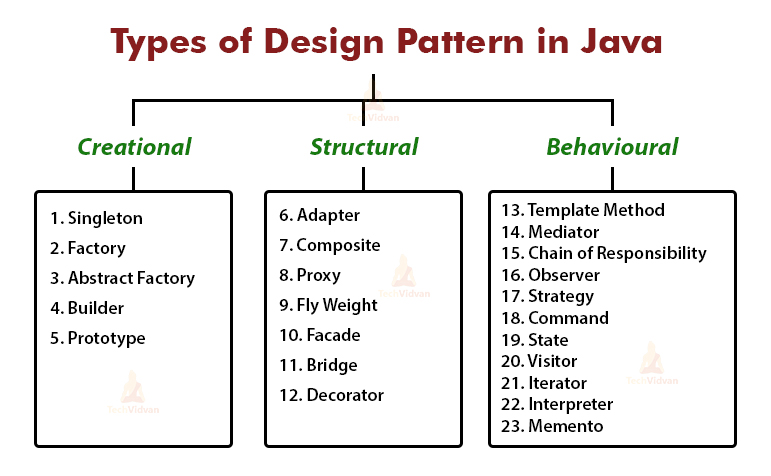
We can define Design patterns as a well-proven solution for solving a specific/particular problem or task. Design patterns provide solutions to problems that programmers generally face during software development.

These design patterns are guidelines to solve a particular problem. None of these patterns force you anything in regard to implementation.

### Types of Java Design patterns

**1.** Creational Design Patterns  
**2.** Structural Design Patterns  
3. Behavioral Design Patterns

**These are again divided into following subcategories:**

[](https://i0.wp.com/techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/06/Types-of-Design-Pattern-in-Java.jpg?ssl=1)

### 1. Creational Design Patterns in Java

Creational design patterns work on the creation of the object. They provide a solution for instantiating an object in the best possible ways. We use Creational patterns often in place of direct instantiation with constructors.

The creational design patterns make the creation process more adaptable and dynamic. They can particularly deal with the flexibility of object creation and the way of creating objects and initializing them.

#### **a. Singleton Design Patterns in Java**

Singleton pattern is a creational design pattern that restricts the number of objects of a class. It ensures that there is only one instance of the class in the JVM.

It is a very simple design pattern but when we need to implement it, there are a lot of implementation concerns. Singleton pattern allows an application to have one and only one instance or object of a class per JVM.

#### **b. Factory Design Pattern in Java**

A factory design pattern is useful when we have a superclass with multiple subclasses and we need to return one of the sub-class based on the input. We can apply a Singleton design pattern on the Factory class or declare the factory method as static.

A factory design pattern is most suitable when there is an involvement of complex object creation steps. The factory pattern ensures that these steps work in a centralized manner and not exposed to composing classes.

#### **c. Abstract Factory Design Pattern in Java**

Abstract Factory pattern is almost similar to the Factory design pattern and we can say it as a factory of factories. In the Abstract Factory pattern, there is no need to use the if-else block of the factory pattern.

We have a factory class for each subclass and then we use an Abstract Factory class that returns the child class on the basis of the input Factory class.

Abstract factory pattern is useful whenever we need another level of abstraction over a group of factories produced using the factory pattern.

#### **d. Java Builder Pattern**

The Builder Design Pattern came into the picture to solve the problems of having many attributes in an Object. Builder Design pattern solves the problem of optional parameters and inconsistent state.

It solves the problem by providing a step-by-step way to build the object and a method that returns the final Object. A builder design pattern is an alternative way to build complex objects.

We should use it only when we want to build different types of immutable objects using the same object building process.

#### **e. Java Prototype Pattern**

The prototype pattern is useful when the process of object creation is very costly and time-consuming.

So using this pattern we can copy the original object to a new object and then change it according to our requirements. This pattern uses a Java cloning mechanism to copy the object.

# **Structural design patterns**

**Structural design patterns** are concerned with how classes and objects can be composed, to form larger structures.

The structural design patterns **simplifies the structure by identifying the relationships**.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

### Structural Design Patterns in Java

Structural design patterns provide ways to create a class structure. For example, creating a large object from smaller ones using inheritance and composition.

Structural design patterns show us how to connect different pieces of a system together in a flexible and extensible manner. These patterns also guarantee that when one of the parts changes, the entire application structure does not need to change.

Structural patterns are the patterns that deal with the composition of objects and answer questions like:

* What is the content of the class?
* What is the relationship between a class with other classes?

#### **a. Java Adapter Pattern**

The Adapter design pattern is a structural design pattern and it ensures that two unrelated interfaces can work together. An adapter converts an interface of a class into another interface.

It allows different classes to work together that would not work together due to incompatible interfaces. The Adapter is the object that joins these unrelated interfaces.

Let’s take a real-life example. Think of a mobile charger that works as an adapter between the charging socket of the mobile and the wall socket of a mobile charger as an adapter.

Because the mobile battery needs 3 V to charge but the normal socket produces either 120V (US) or 240V (India).

#### **b. Composite Design Pattern in Java**

Composite pattern is the Structural design pattern that represents a part-whole hierarchy. The composite design pattern composes the objects into tree structures. It represents them as whole-part hierarchies.

The composite pattern allows the clients to treat individual objects and compositions of objects in a uniform way. We can apply the composite design pattern when we need to create a structure such that the objects in the structure that need treatment in the same way

Let’s understand the composite design pattern with a real-life example. A diagram is a structure that contains objects like Lines, Circle, Triangle, Square, etc.  
When we fill the drawing with color (like Blue), all other Objects in the drawing also fill with the same color.

#### **c. Proxy Pattern in Java**

The objective of the Proxy pattern is to “Provide a surrogate or placeholder for another object to provide control access to it”. The definition itself is very clear and a proxy pattern is useful when we want to provide controlled access to functionality.

Proxy is heavily used to implement lazy loading related use cases where we do not want to create a full object until it is actually needed.

Suppose we have a class that can run commands on the system. If we are using this class, it is fine. But, when we give it to a client application, then there can be severe issues.

These issues are because the client program can execute a command to change settings or delete some system files in your system that you do not want.

#### **d. Java Flyweight Design Pattern**

A flyweight design pattern creates a lot of Objects of a class. Flyweight design pattern allows us to share objects to support large numbers of fine-grained objects. A flyweight is a shared object that we can use in multiple contexts simultaneously.

We can apply the flyweight design pattern to reduce the load on memory by sharing objects. As every object consumes memory space that can be important for low memory devices, such as mobile devices or embedded systems.

One of the best examples of Flyweight pattern implementation is the String Pool implementation in Java.

#### **e. Facade Design Pattern in Java**

Facade Pattern helps client applications to easily interact with the system. A facade design pattern provides a unified interface to a set of interfaces in a subsystem.

Suppose there is an application with a set of interfaces to use Oracle or MySql database. We need to generate different types of reports, such as PDF reports, HTML reports. We will have a different set of interfaces to work with different types of databases.

Now, to get the required database connection and generate reports, a client application can use these interfaces. But when the complexity increases or when the interface behavior names are confusing, the client application will find it difficult to manage it.

So we can apply the Facade pattern here and provide a wrapper interface on top of the existing interface to help client application.

#### **f. Bridge Design Pattern in Java**

We use this pattern When there is an interface hierarchy in both interfaces as well as implementations. The bridge design pattern decouples the interfaces from implementation. It also hides the implementation details from the client programs.

A bridge design pattern usually decouples a class into two parts – abstraction and it’s implementation. This separation is such that both parts can evolve in the future without affecting each other.

Therefore, the loose coupling between class abstraction and its implementation increases.

#### **g. Decorator Design Pattern in Java**

The decorator design pattern modifies the functionality of an object at runtime. A decorator design pattern adds additional features or behaviors to a particular instance of a class.

It adds the while not modifying the other instances of the same class. The decorator design pattern uses abstract classes or interfaces.

We use composition or inheritance to extend the object’s behavior. But we can do it only at compile-time and it applies to all the objects of the class.

We can not add any new functionality to remove any existing behavior at runtime – this is when the Decorator pattern comes into the picture.

### Behavioral Design Patterns in Java

Behavioral patterns provide a solution for achieving a better interaction between objects. They also tell how to provide loose coupling and flexibility to extend easily.

Behavioral patterns deal with the behavior of objects, interactions between objects. How does an object communicate with other objects?

#### **a. Java Template Method Pattern**

Template Method Design Pattern is a behavioral design pattern. It creates a method stub and defers some of the steps of implementation to the subclasses.

The template pattern defines the steps to execute an algorithm. It also provides a default implementation common to all or some of the subclasses.

Let’s take an example, we want to provide an algorithm to build a house. The steps that we need to perform to build a house are: building a plan, building pillars, walls, windows, doors.

The important point is that we can not change the order of execution. This is because it is not possible to build windows before building the foundation. So, we can create a template method that will use different methods to build the house.

#### **b. Java Mediator Pattern**

The mediator design pattern provides a centralized communication medium between different objects of a system. This pattern is helpful in an enterprise application where multiple objects interact with each other.

When the objects interact with each other directly, then the system components couple with each other. This makes maintainability cost higher and not flexible to extend easily.

The mediator pattern focuses on providing a mediator between objects for communication. It also helps to implement loose-coupling between objects.

An example of a mediator pattern is the air traffic controller. The airport control room acts as a mediator for communication between different flights. The mediator pattern works as a router between objects and it has its own logic to provide a way of communication.

#### **c. Java Chain of Responsibility Pattern**

The chain of responsibility pattern achieves loose coupling in software design. Here, a client passes a request to a sequence or chain of objects.

After that, the object in the chain decides which object will process the request and whether there is a need to send the request to the next object or not.

We know that there can be multiple catch blocks in a try-catch block code. In this, every catch block acts as a processor that processes a particular exception. So, when there is an exception in the try block, the first catch block processes it.

If that catch block is unable to process it, it forwards the request to the next catch block or simply the next object in the chain. Even if the last catch block is not able to process it, then there is an exception outside of the chain to the calling program.

#### **d. Java Observer Pattern**

An observer design pattern deals with the state of an object and notifies us whenever there is any change in the object’s state.

In the observer pattern, the object that keeps a watch on the state of another object is the Observer and the object that it watches is the Subject.

Java provides an inbuilt platform to implement the Observer design pattern with the help of java.util.Observable class and java.util.Observer interface. We can not use them widely because the implementation is really simple.

Most of the time we don’t want to extend a class just for implementing the Observer pattern. This is because we know that Java has no support for multiple inheritance with classes.

#### **e. Java Strategy Pattern**

A strategy pattern is useful when there are multiple algorithms for a particular task. And, also when the client decides the actual implementation during runtime.

A strategy pattern is also known as Policy Pattern. We define multiple algorithms and allow client applications to pass the algorithm and use them as parameters.

Collections.sort() method is one of the best examples of this pattern that takes the Comparator parameter. It sorts the objects in different ways based on the different implementations of Comparator interfaces.

#### **f. Command Design Pattern in Java**

Command Pattern is useful when we need to implement loose coupling in a request-response model. In the command pattern, the request is sent to the invoker, and the invoker passes it to the encapsulated command object.

The command object passes the request to the appropriate method of Receiver to perform the specific action.

#### **g. State Pattern in Java**

A state design pattern is useful when an Object changes its behavior based on its internal state. We can use it if we want to change the behavior of an object based on its state.

We can create a state variable in the object and use the if-else condition block to perform different actions based on the state of the object.

The state pattern is useful when there is a need to provide a systematic and loosely coupled way to achieve this through Context and State implementations.

#### **h. Java Visitor Design Pattern**

The visitor pattern is helpful when we have to perform an operation on a group of similar kinds of Objects. With the help of a visitor pattern, we can move the operational logic from the objects to another class.

For example, there is a shopping cart where we can add different types of items or elements. When we click the checkout button, it calculates the total payable amount.

Now, using the Visitor design pattern, we can get calculation logic in item classes or we can put this logic in another class.

#### **i. Interpreter Design Pattern in Java**

The Interpreter Design pattern defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

Java compiler is the best example of this pattern that interprets the Java source code into bytecode that JVM understands. Another example of the Interpreter pattern is Google Translator.

Here, we can give the inputs in any language and get the output interpreted in the desired language.

#### **j. Iterator Design Pattern in Java**

Iterator pattern in one of the behavioral patterns and it provides a standard way to traverse through a group of objects. It finds its major use in the Java Collection Framework.

In the Collection framework, the methods of the Iterator interface helps to traverse through a Collection. Using the Iterator pattern, we can also provide different kinds of iterators based on our requirements.

#### **k. Memento Pattern in Java**

The memento design pattern saves the state of an object so that we can restore it later. Memento pattern implements this in such a way that there is no access to the saved state data of the object outside of the object. And, this protects the integrity of saved state data.

Originator and Caretaker are the two objects that implement the Memento pattern. The Originator uses an inner class to save the object’s state.

We need to save and restore this object. This inner class is nothing but Memento which is private so that no other object can access it.

Q: Design patterns in Spring and Spring Boot?

Q: How can we break singleton design pattern?

Singleton.Java

**package** demo1**;**

**public final** **class** Singleton **{**

**private** **static volatile** Singleton instance **=** **null;**

**private** Singleton**()** **{**

**}**

**public** **static** Singleton getInstance**()** **{**

**if** **(**instance **==** **null)** **{**

**synchronized** **(**Singleton**.**class**)** **{**

**if** **(**instance **==** **null)** **{**

instance **=** **new** Singleton**();**

**}**

**}**

**}**

**return** instance**;**

**}**

**}**

SingletonTest.java

**package** demo1**;**

**public** **class** SingletonTest **{**

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

Singleton object1 **=** Singleton**.**getInstance**();**

Singleton object2 **=** Singleton**.**getInstance**();**

System**.**out**.**println**(**"Hashcode of Object 1 - " **+** object1**.**hashCode**());**

System**.**out**.**println**(**"Hashcode of Object 2 - " **+** object2**.**hashCode**());**

**}**

**}**

### Reflection

Java  Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime. Using Reflection API we can create multiple objects in singleton class. Consider the following example.

ReflectionSingleton.java

**package** demo1**;**

**import** java.lang.reflect.Constructor**;**

**public** **class** ReflectionSingleton **{**

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

Singleton objOne **=** Singleton**.**getInstance**();**

Singleton objTwo **=** **null;**

**try** **{**

Constructor constructor **=** Singleton**.**class**.**getDeclaredConstructor**();**

constructor**.**setAccessible**(true);**

objTwo **=** **(**Singleton**)** constructor**.**newInstance**();**

**}** **catch** **(**Exception ex**)** **{**

System**.**out**.**println**(**ex**);**

**}**

System**.**out**.**println**(**"Hashcode of Object 1 - "**+**objOne**.**hashCode**());**

System**.**out**.**println**(**"Hashcode of Object 2 - "**+**objTwo**.**hashCode**());**

**}**

**}**

Example to show how reflection can break the singleton pattern with Java reflect. You will get two hash code as below. It has a break on the singleton pattern.

Hashcode of Object **1** **-** **1836019240**

Hashcode of Object **2** **-** **325040804**

**Prevent Singleton pattern from Reflection**  
  
There are many ways to prevent Singleton pattern from Reflection API, but one of the best solutions is to throw run time exception in the constructor if the instance already exists. In this, we can not able to create a second instance.

**private** Singleton**()** **{**

**if(** instance **!=** **null** **)** **{**

**throw** **new** InstantiationError**(** "Creating of this object is not allowed." **);**

**}**

**}**

### Deserialization

In serialization, we can save the object of a byte stream into a file or send over a network. Suppose if you serialize the Singleton class and then again de-serialize that object will create a new instance, hence deserialization will break the Singleton pattern.  
  
Below code is to illustrate how the Singleton pattern breaks with deserialization.  
  
Implements Serializable interface for Singleton Class.  
  
DeserializationSingleton.Java

**package** demo1**;**

**import** java.io.\***;**

**public** **class** DeserializationSingleton **{**

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

Singleton instanceOne **=** Singleton**.**getInstance**();**

ObjectOutput out **=** **new** ObjectOutputStream**(new** FileOutputStream**(**"file.text"**));**

out**.**writeObject**(**instanceOne**);**

out**.**close**();**

ObjectInput in **=** **new** ObjectInputStream**(new** FileInputStream**(**"file.text"**));**

Singleton instanceTwo **=** **(**Singleton**)** in**.**readObject**();**

in**.**close**();**

System**.**out**.**println**(**"hashCode of instance 1 is - " **+** instanceOne**.**hashCode**());**

System**.**out**.**println**(**"hashCode of instance 2 is - " **+** instanceTwo**.**hashCode**());**

**}**

**}**

The output is below and you can see two hashcodes.

hashCode of instance **1** is **-** **2125039532**

hashCode of instance **2** is **-** **381259350**

**Prevent Singleton Pattern from Deserialization**  
  
To overcome this issue, we need to override readResolve() method in Singleton class and return same Singleton instance. Update Singleton.java, with below method.

**protected** Object readResolve**()** **{**

**return** instance**;**

**}**

Now run above DeserializationDemo class and see the output.

hashCode of instance **1** is **-** **2125039532**

hashCode of instance **2** is **-** **2125039532**

### Cloning

Using the "clone" method we can create a copy of original object, samething if we applied clone in singleton pattern, it will create two instances one original and another one cloned object. In this case will break Singleton principle as shown in below code.  
  
Implement the "Cloneable" interface and override the clone method in the above Singleton class.  
  
Singleton.java

**@Override**

**protected** Object clone**()** **throws** CloneNotSupportedException **{**

**return** **super.**clone**();**

**}**

Then Test with cloning for breaking the singleton  
CloningSingleton.java

**public** **class** CloningSingleton **{**

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

Singleton instanceOne **=** Singleton**.**getInstance**();**

Singleton instanceTwo **=** **(**Singleton**)** instanceOne**.**clone**();**

System**.**out**.**println**(**"hashCode of instance 1 - " **+** instanceOne**.**hashCode**());**

System**.**out**.**println**(**"hashCode of instance 2 - " **+** instanceTwo**.**hashCode**());**

**}**

**}**

Here is the output

hashCode of instance **1** **-** **1836019240**

hashCode of instance **2** **-** **325040804**

If we see the above output, two instances have different hashcodes means these instances are not the same.  
  
  
**Prevent Singleton Pattern from Cloning**  
  
In the above code, breaks the Singleton principle i. e created two instances. To overcome the above issue we need to implement/override clone() method and throw an exception CloneNotSupportedException from clone method. If anyone try to create clone object of Singleton, it will throw an exception as see below code.

**@Override**

**protected** Object clone**()** **throws** CloneNotSupportedException **{**

**throw** **new** CloneNotSupportedException**();**

**}**

Now we can run the CloningSingleton class, it will throw CloneNotSupportedException while creating a clone object of Singleton object.