## ABOUT ROHIT JOSHI

Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

# Java Design Patterns Tutorial

Posted by: Rohit Joshi in Core Java September 30th, 2015

## Course Overview

A design pattern in architecture and computer science is a formal way of documenting a solution to a design problem in a particular field of expertise. The idea was introduced by the architect Christopher Alexander in the field of architecture and has been adapted for various other disciplines, including computer science.

A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them.



## About the Author

Rohit Joshi works as a Software Engineer in the Consumer Product Sector. He is a Sun Certified Java Programmer. He had worked in projects related to different domains. He is also involved in system analysis and system designing. He mainly works in Core Java and J2EE technologies but also have good experience in front-end technologies like Javascript and Jquery.

## Lessons

### Introduction to Design Patterns

In this lesson, you will get introduced to Design Patterns. You will learn what Design Patterns are, why they should be used in our code and how to select and use one. Finally, the categorization of the existing patterns is described.

## Adapter Design Pattern

Via a real life example, you will learn how and when the Adapter pattern should be used and how to structure your code in order to implement it. You will see how it can lead to elegant solutions to code problems.

The Facade Pattern makes a complex interface easier to use, using a Facade class. The Facade Pattern provides a unified interface to a set of interface in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

## Composite Design Pattern

The Composite Pattern allows you to compose objects into a tree structure to represent the part-whole hierarchy which means you can create a tree of objects that is made of different parts, but that can be treated as a whole one big thing. Composite lets clients to treat individual objects and compositions of objects uniformly, that's the intent of the Composite Pattern.

## Bridge Design Pattern

The Bridge Pattern's intent is to decouple an abstraction from its implementation so that the two can vary independently. It puts the abstraction and implementation into two different class hierarchies so that both can be extend independently.

## Singleton Design Pattern

The Singleton pattern is used when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point or when the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

## Observer Design Pattern

The Observer Pattern is a kind of behavior pattern which is concerned with the assignment of responsibilities between objects. It should be used when an abstraction has two aspects, one dependent on the other, when a change to one object requires changing others, and you don't know how many objects need to be changed or when an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

## Mediator Design Pattern

The Mediator Pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. Rather than interacting directly with each other, objects ask the Mediator to interact on their behalf which results in reusability and loose coupling. You will learn how and when the Mediator design pattern should be used and how to structure your code in order to implement it.

## Proxy Design Pattern

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it. It comes up with many different variations. Some of the important variations are, Remote Proxy, Virtual Proxy, and Protection Proxy. In this lesson, we will know more about these variations and we will implement each of them in Java. But before we do that, let's get to know more about the Proxy Pattern in general. You will learn how and when the Proxy design pattern should be used and how to structure your code in order to implement it.

## Chain of Responsibility Design Pattern

The Chain of Responsibility pattern is a behavior pattern in which a group of objects is chained together in a sequence and a responsibility (a request) is provided in order to be handled by the group. If an object in the group can process the particular request, it does so and returns the corresponding response. Otherwise, it forwards the request to the subsequent object in the group.

## Flyweight Design Pattern

The Flyweight Pattern is designed to control object creation where objects in an application have great similarities and are of a similar kind, and provides you with a basic caching mechanism. It allows you to create one object per type (the type here differs by a property of that object), and if you ask for an object with the same property (already created), it will return you the same object instead of creating a new one.

## Builder Design Pattern

The intent of the Builder Pattern is to separate the construction of a complex object from its representation, so that the same construction process can create different representations. This type of separation reduces the object size. The design turns out to be more modular with

each implementation contained in a different builder object. Adding a new implementation (i.e., adding a new builder) becomes easier.

## Factory Method Design Pattern

The Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method. The Factory Method selects an appropriate class from a class hierarchy based on the application context and other influencing factors. It then instantiates

## Abstract Factory Design Pattern

The Abstract Factory (A.K.A. Kit) is a design pattern which provides an interface for creating families of related or dependent objects without specifying their concrete classes. The Abstract Factory pattern takes the concept of the Factory Method Pattern to the next level. An abstract factory is a class that provides an interface to produce a family of objects.

## Prototype Design Pattern

The Prototype design pattern is used to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object.

## Memento Design Pattern

Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and "undo" mechanisms that let users back out of tentative operations or recover from errors. You must save state information somewhere, so that you can restore objects to their previous conditions. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally. Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility. The Memento pattern can be used to accomplish this without exposing the object's internal structure.

## Template Design Pattern

The Template Design Pattern is a behavior pattern and, as the name suggests, it provides a template or a structure of an algorithm which is used by users. A user provides its own implementation without changing the algorithm's structure. The Template Pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses to redefine certain steps of an algorithm without changing the algorithm's structure.

## State Design Pattern

The State Design Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class. The state of an object can be defined as its exact condition at any given point of time, depending on the values of its properties or attributes. The set of methods implemented by a class constitutes the behavior of its instances. Whenever there is a change in the values of its attributes, we say that the state of an object has changed.

## Strategy Design Pattern

The Strategy Design Pattern seems to be the simplest of all design patterns, yet it provides great flexibility to your code. This pattern is used almost everywhere, even in conjunction with the other design patterns. The Strategy Design Pattern defines a family of algorithms, encapsulating each one, and making them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

## Command Design Pattern

The Command Design Pattern is a behavioral design pattern and helps to decouples the invoker from the receiver of a request. The intent of the Command Design Pattern is to encapsulate a request as an object, thereby letting the developer to parameterize clients with different requests, queue or log requests, and support undoable operations.

## Interpreter Design Pattern

The Interpreter Design Pattern is a heavy-duty pattern. It's all about putting together your own programming language, or handling an existing one, by creating an interpreter for that language. Given a language, we can define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

## Decorator Design Pattern

The intent of the Decorator Design Pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality. The pattern is used to extend the functionality of an object dynamically without having to change the original class source or using inheritance. This is accomplished by creating an object wrapper referred to as a Decorator around the actual object.

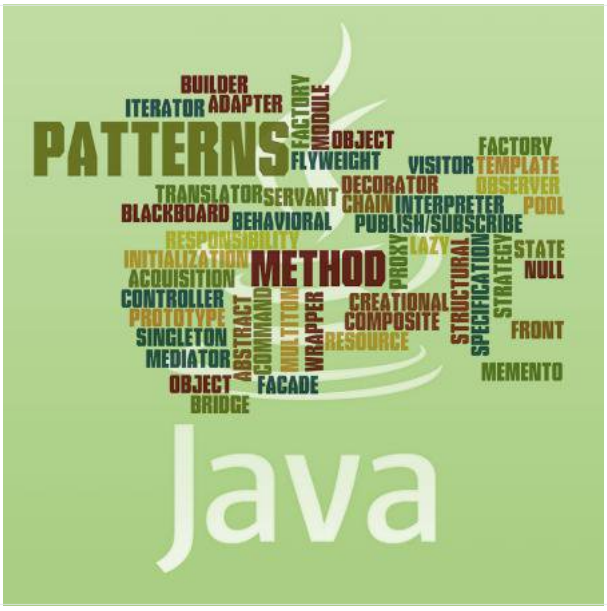
## Iterator Design Pattern

The intent of the Iterator Design Pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. he Iterator pattern allows a client object to access the contents of a container in a sequential manner, without having any knowledge about the internal representation of its contents.

## Visitor Design Pattern

the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Make sure to retweet this, let your social followers know!




**Java Code Geeks**
@javacodegeeks

Follow

#Java Design Patterns Tutorial - FREE Mega-Course!
buff.ly/1Lc4ISv
7:53 PM - 15 Oct 2015
24 9

Tagged with: DESIGN PATTERNS

## Do you want to know how to develop your skillset to become a Java Rockstar?

Subscribe to our newsletter to start Rocking right now!
To get you started we give you our best selling eBooks for FREE!

**1.** JPA Mini Book
**2.** JVM Troubleshooting Guide
**3.** JUnit Tutorial for Unit Testing
**4.** Java Annotations Tutorial
**5.** Java Interview Questions
**6.** Spring Interview Questions
**7.** Android UI Design

and many more ....

**Email address:**

Your email address

Sign up