**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)

<https://www.javatpoint.com/singleton-design-pattern-in-java>

<https://www.journaldev.com/1827/java-design-patterns-example-tutorial>

**Singleton Pattern**

Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.

There are two forms of singleton design pattern

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

#### Advantage of Singleton design pattern

* Saves memory because object is not created at each request. Only single instance is reused again and again.

#### Usage of Singleton design pattern

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

### 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

### Abstract Factory Pattern

Abstract Factory pattern is similar to Factory pattern and it’s factory of factories. 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 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.

### 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.

### Prototype 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.

### 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.

An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".**

In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

### 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.

### 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.

A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".**

# Facade Pattern

A Facade Pattern says that just **"just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".**

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.

### 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.

### 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.

### 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**.

### 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.

### 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.

### 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.

### Interpreter Pattern

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